MPLAB® XC32 C/C++ Compiler User's Guide

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

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

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

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in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates.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

Trademarks

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

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

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

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

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

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

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

Printed on recycled paper.

ISBN: 978-1-62076-726-9

**QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
═ ISO/TS 16949 ═**

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

Table of Contents

Preface 7

Chapter 1. Compiler Overview

1.1 Introduction 13
1.2 Device Description 13
1.3 Compiler Description and Documentation 13
1.4 Compiler and Other Development Tools 15

Chapter 2. Common C Interface

2.1 Introduction 17
2.2 Background – The Desire for Portable Code 17
2.3 Using the CCI 20
2.4 ANSI Standard Refinement 21
2.5 ANSI Standard Extensions 29
2.6 Compiler Features 43

Chapter 3. Compiler Command Line Driver

3.1 Introduction 45
3.2 Invoking the Compiler 45
3.3 The C Compilation Sequence 49
3.4 The C++ Compilation Sequence 51
3.5 Runtime Files 55
3.6 Start-up and Initialization 58
3.7 Compiler Output 58
3.8 Compiler Messages 60
3.9 Driver Option Descriptions 60

Chapter 4. Device-Related Features

4.1 Introduction 85
4.2 Device Support 85
4.3 Device Header Files 85
4.4 Stack 86
4.5 Using SFRs From C Code 88

Chapter 5. ANSI C Standard Issues

5.1 Divergence from the ANSI C Standard 91
5.2 Extensions to the ANSI C Standard 91
5.3 Implementation-defined behavior 92

Chapter 6. Supported Data Types and Variables

6.1 Introduction 93
6.2 Identifiers 93
6.3 Data Representation 93
6.4 Integer Data Types 94
6.5 Floating-Point Data Types 96
6.6 Structures and Unions 98
6.7 Pointer Types 100
6.8 Complex Data Types 102
6.9 Constant Types and Formats 102
6.10 Standard Type Qualifiers 104
6.11 Compiler-Specific Qualifiers 105
6.12 Variable Attributes 105

Chapter 7. Memory Allocation and Access

7.1 Introduction 109
7.2 Address Spaces 109
7.3 Variables in Data Memory 110
7.4 Auto Variable Allocation and Access 112
7.5 Variables in Program Memory 113
7.6 Variables in Registers 114
7.7 Dynamic Memory Allocation 114
7.8 Memory Models 114

Chapter 8. Operators and Statements

8.1 Introduction 117
8.2 Integral Promotion 117
8.3 Type References 118
8.4 Labels as Values 119
8.5 Conditional Operator Operands 120
8.6 Case Ranges 120

Chapter 9. Register Usage

9.1 Introduction 121
9.2 Register Usage 121
9.3 Register Conventions 121

Chapter 10. Functions

10.1 Writing Functions 123
10.2 Function Attributes and Specifiers 123
10.3 Allocation of Function Code 127
10.4 Changing the Default Function Allocation 128
10.5 Function Size Limits 128
10.6 Function Parameters 128
10.7 Function Return Values 131
10.8 Calling Functions 131
10.9 Inline Functions 131

Chapter 11. Interrupts

11.1 Introduction 133
11.2 Interrupt Operation 133
11.3 Writing an Interrupt Service Routine 134
11.4 Associating a Handler Function with an Exception Vector 140
11.5 Exception Handlers 142
11.6 Interrupt Service Routine Context Switching 142
11.7 Latency 143
11.8 Nesting Interrupts 143
11.9 Enabling/Disabling Interrupts 143
11.10 ISR Considerations 143

Chapter 12. Main, Runtime Start-up and Reset

12.1 Introduction 145
12.2 The Main Function 145
12.3 Runtime Start-up Code 145
12.4 The On Reset Routine 159

Chapter 13. Library Routines

13.1 Using Library Routines 161

Chapter 14. Mixing C/C++ and Assembly Language

14.1 Introduction 163
14.2 Mixing Assembly Language and C Variables and Functions 163
14.3 Using Inline Assembly Language 166
14.4 Predefined Assembly Macros 170

Chapter 15. Optimizations

15.1 Introduction 173

Chapter 16. Preprocessing

16.1 Introduction 175
16.2 C/C++ Language Comments 175
16.3 Preprocessor Directives 175
16.4 Pragma Directives 177
16.5 Predefined Macros 178

Chapter 17. Linking Programs

17.1 Introduction 181
17.2 Replacing Library Symbols 181
17.3 Linker-Defined Symbols 181
17.4 Default Linker Script 182

Appendix A. Implementation-Defined Behavior

A.1 Introduction 197
A.2 Highlights 197
A.3 Overview 197
A.4 Translation 198
A.5 Environment 198
A.6 Identifiers 199

A.7 Characters 199
A.8 Integers 200
A.9 Floating-Point 200
A.10 Arrays and Pointers 202
A.11 Hints 202
A.12 Structures, Unions, Enumerations, and Bit fields 203
A.13 Qualifiers 203
A.14 Declarators 204
A.15 Statements 204
A.16 Pre-Processing Directives 204
A.17 Library Functions 205
A.18 Architecture 208

Appendix B. ASCII Character Set

Appendix C. Deprecated Features

C.1 Introduction 211
C.2 Variables in Specified Registers 211

Appendix D. Built-In Functions

D.1 Introduction 213
D.2 Built-In Function Descriptions 214

Appendix E. Embedded Compiler Compatibility Mode

E.1 Introduction 217
E.2 Compiling in Compatibility Mode 218
E.3 Syntax Compatibility 218
E.4 Data Type 219
E.5 Operator 219
E.6 Extended Keywords 220
E.7 Intrinsic Functions 221
E.8 Pragmas 221

Appendix F. Document Revision History

Support 225

Glossary 227

Index 245

Worldwide Sales and Service 246

Preface

NOTICE TO CUSTOMERS

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

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

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

MPLAB® XC32 C/C++ Compiler documentation and support information is discussed in the sections below:

- Document Layout
- Conventions Used
- Recommended Reading

DOCUMENT LAYOUT

This document describes how to use GNU language tools to write code for 32-bit applications. The document layout is as follows:

- **Chapter 1. Compiler Overview** – describes the compiler, development tools and feature set.
- **Chapter 2. Common C Interface** – explains what you need to know about making code portable.
- **Chapter 3. Compiler Command Line Driver** – describes how to use the compiler from the command line.
- **Chapter 4. Device-Related Features** – describes the compiler header and register definition files, as well as how to use with the SFRs.
- **Chapter 5. ANSI C Standard Issues** – describes the differences between the C/C++ language supported by the compiler syntax and the standard ANSI-89 C.
- **Chapter 6. Supported Data Types and Variables** – describes the compiler integer and pointer data types.
- **Chapter 7. Memory Allocation and Access** – describes the compiler run-time model, including information on sections, initialization, memory models, the software stack and much more.
- **Chapter 8. Operators and Statements** – discusses operators and statements.
- **Chapter 9. Register Usage** – explains how to access and use SFRs.
- **Chapter 10. Functions** – details available functions.
- **Chapter 11. Interrupts** – describes how to use interrupts.
- **Chapter 12. Main, Runtime Start-up and Reset** – describes important elements of C/C++ code.
- **Chapter 13. Library Routines** – explains how to use libraries.
- **Chapter 14. Mixing C/C++ and Assembly Language** – provides guidelines for using the compiler with 32-bit assembly language modules.
- **Chapter 15. Optimizations** – describes optimization options.
- **Chapter 16. Preprocessing** – details the preprocessing operation.
- **Chapter 17. Linking Programs** – explains how linking works.
- **Appendix A. Implementation-Defined Behavior** – details compiler-specific parameters described as implementation-defined in the ANSI standard.
- **Appendix B. ASCII Character Set"** – contains the ASCII character set.
- **Appendix C. Deprecated Features –** details features that are considered obsolete.
- **Appendix D. Built-In Functions –** lists the built-in functions of the C compiler.
- **Appendix E. Embedded Compiler Compatibility Mode –** discusses using the compiler in compatibility mode.
- **Appendix F. Document Revision History –** information on previous and current revisions of this document.

CONVENTIONS USED

The following conventions may appear in this documentation:

DOCUMENTATION CONVENTIONS

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	*MPLAB® IDE User's Guide*
	Emphasized text	...is the *only* compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	select Enable Programmer
Quotes	A field name in a window or dialog	"Save project before build"
Underlined, italic text with right angle bracket	A menu path	<u>*File>Save*</u>
Bold characters	A dialog button	Click **OK**
	A tab	Click the **Power** tab
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
Courier font:		
Plain Courier	Sample source code	`#define START`
	Filenames	`autoexec.bat`
	File paths	`c:\mcc18\h`
	Keywords	`_asm, _endasm, static`
	Command-line options	`-Opa+, -Opa-`
	Bit values	`0, 1`
	Constants	`0xFF, 'A'`
Italic Courier	A variable argument	*`file`*`.o`, where *`file`* can be any valid filename
Square brackets [ ]	Optional arguments	`mpasmwin [options] file [options]`
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; an OR selection	`errorlevel {0|1}`
Ellipses...	Replaces repeated text	`var_name [, var_name...]`

	Represents code supplied by user	`void main (void)`
		`{ ...`
		`}`

Sidebar Text		
DD	Device Dependent. This feature is not supported on all devices. Devices supported will be listed in the title or text.	`xmemory` attribute

RECOMMENDED READING

This documentation describes how to use the MPLAB XC32 C Compiler. Other useful documents are listed below. The following Microchip documents are available and recommended as supplemental reference resources.

Release Notes (Readme Files)

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

MPLAB® Assembler, Linker and Utilities for PIC32 MCUs User's Guide (DS51833)

A guide to using the 32-bit assembler, object linker, object archiver/librarian and various utilities.

32-Bit Language Tools Libraries (DS51685)

Lists all library functions provided with the MPLAB XC32 C Compiler with detailed descriptions of their use.

Dinkum Compleat Libraries

The Dinkum Compleat Libraries are organized into a number of headers, files that you include in your program to declare or define library facilities. A link to the Dinkum libraries is available in the MPLAB X IDE application, on the My MPLAB X IDE tab, References & Featured Links section.

PIC32MX Configuration Settings

Lists the Configuration Bit settings for the Microchip PIC32MX devices supported by the MPLAB XC32 C Compiler's `#pragma config`.

Device-Specific Documentation

The Microchip website contains many documents that describe 32-bit device functions and features. Among these are:

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

C Standards Information

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

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

C++ Standards Information

Stroustrup, Bjarne, *C++ Programming Language: Special Edition*, 3rd Edition. Addison-Wesley Professional; Indianapolis, Indiana, 46240.

ISO/IEC 14882 C++ Standard. The ISO C++ Standard is standardized by ISO (The International Standards Organization) in collaboration with ANSI (The American National Standards Institute), BSI (The British Standards Institute) and DIN (The German national standards organization).

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.

GCC Documents

http://gcc.gnu.org/onlinedocs/

http://sourceware.org/binutils/

NOTES:

Chapter 1. Compiler Overview

1.1 INTRODUCTION

The MPLAB XC32 C Compiler is defined and described in the following topics:

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

1.2 DEVICE DESCRIPTION

The MPLAB XC32 C Compiler fully supports all Microchip 32-bit devices.

1.3 COMPILER DESCRIPTION AND DOCUMENTATION

The MPLAB XC32 C Compiler is a full-featured, optimizing compiler that translates standard ANSI C programs into 32-bit device assembly language source. The compiler also supports many command-line options and language extensions that allow full access to the 32-bit device hardware capabilities, and affords fine control of the compiler code generator.

The compiler is a port of the GCC compiler from the Free Software Foundation.

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

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

1.3.1 Conventions

Throughout this manual, the term "the compiler" is often used. It can refer to either all, or some subset of, the collection of applications that form the MPLAB XC32 C Compiler. Often it is not important to know, for example, whether an action is performed by the parser or code generator application, and it is sufficient to say it was performed by "the compiler".

It is also reasonable for "the compiler" to refer to the command-line driver (or just driver) as this is the application that is always executed to invoke the compilation process. The driver for the MPLAB XC32 C Compiler package is called `xc32-gcc`. The driver for the C/ASM projects is also `xc32-gcc`. The driver for C/C++/ASM projects is `xc32-g++`. The drivers and their options are discussed in **Section 3.9 "Driver Option Descriptions"**. Following this view, "compiler options" should be considered command-line driver options, unless otherwise specified in this manual.

Similarly "compilation" refers to all, or some part of, the steps involved in generating source code into an executable binary image.

1.3.2 ANSI C Standards

The compiler is a fully validated compiler that conforms to the ANSI C standard as defined by the ANSI specification (ANSI x3.159-1989) and described in Kernighan and Ritchie's *The C Programming Language* (second edition). The ANSI standard includes extensions to the original C definition that are now standard features of the language. These extensions enhance portability and offer increased capability. In addition, language extensions for PIC32 MCU embedded-control applications are included.

1.3.3 Optimization

The compiler uses a set of sophisticated optimization passes that employ many advanced techniques for generating efficient, compact code from C/C++ source. The optimization passes include high-level optimizations that are applicable to any C/C++ code, as well as PIC32 MCU-specific optimizations that take advantage of the particular features of the device architecture.

For more on optimizations, see **Chapter 15. "Optimizations"**.

1.3.4 ANSI Standard Library Support

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

1.3.5 ISO/IEC C++ Standard

The compiler is distributed with the 2003 Standard C++ Library.

Note: Do not specify an MPLAB XC32 system include directory (e.g. `/pic32mx/include/`) in your project properties. The `xc32-gcc` and `xc32-g++` compilation drivers automatically select the XC libc or the Dinkumware libc and their respective include-file directory for you. Manually adding a system include file path may disrupt this mechanism and cause the incorrect libc include files to be compiled into your project, causing a conflict between the include files and the library. Note that adding a system include path to your project properties has never been a recommended practice.

1.3.6 Compiler Driver

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

1.3.7 Documentation

The C compiler is supported under both the MPLAB IDE v8.xx or higher, and the MPLAB X IDE. For C++, MPLAB X IDE v1.40 or higher is required. For simplicity, both IDEs are referred to throughout the book as simply MPLAB IDE.

Features that are unique to specific devices, and therefore specific compilers, are noted with "DD" text the column (see the Preface) and text identifying the devices to which the information applies.

1.4 COMPILER AND OTHER DEVELOPMENT TOOLS

The compiler works with many other Microchip tools including:

- MPLAB XC32 assembler and linker - see the “*MPLAB® Assembler, Linker and Utilities for PIC32 MCUs User’s Guide*”.
- MPLAB IDE v8.xx and MPLAB X IDE (C++ required MPLAB X IDE v1.30 or higher)
- The MPLAB Simulator
- All Microchip debug tools and programmers
- Demo boards and starter kits that support 32-bit devices

NOTES:

Chapter 2. Common C Interface

2.1 INTRODUCTION

The Common C Interface (CCI) is available with all MPLAB XC C compilers and is designed to enhance code portability between these compilers. For example, CCI-conforming code would make it easier to port from a PIC18 MCU using the MPLAB XC8 C compiler to a PIC32 MCU using the MPLAB XC32 C Compiler.

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

The following topics are examined in this chapter:

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

2.2 BACKGROUND – THE DESIRE FOR PORTABLE CODE

All programmers want to write portable source code.

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

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

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

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

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

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

2.2.1 The ANSI Standard

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

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

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

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

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

If the standard is too strict, device architectures may not allow the compiler to conform[1]. But, if it is too weak, programmers would see wildly differing results within different compilers and architectures, and the standard would loose its effectiveness.

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

Implementation-defined behavior

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

Unspecified behavior

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

Undefined behavior

This is behavior for which the standard imposes no requirements.

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

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

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

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

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

2.2.2 The Common C Interface

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

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

Refinement of the ANSI C Standard

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

Consistent syntax for non-standard extensions

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

Coding guidelines

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

2.3 USING THE CCI

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

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

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

Enable the CCI

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

Include <xc.h> in every module

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

Ensure ANSI compliance

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

Observe refinements to ANSI by the CCI

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

Use the CCI extensions to the language

Use the CCI extensions rather than the native language extensions

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

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

2.4 ANSI STANDARD REFINEMENT

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

2.4.1 Source File Encoding

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

2.4.1.1 EXAMPLE

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

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

2.4.1.2 DIFFERENCES

All compilers have used this character set.

2.4.1.3 MIGRATION TO THE CCI

No action required.

2.4.2 The Prototype for `main`

The prototype for the `main()` function is

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

2.4.2.1 EXAMPLE

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

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

2.4.2.2 DIFFERENCES

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

2.4.2.3 MIGRATION TO THE CCI

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

2.4.3 Header File Specification

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

2.4.3.1 EXAMPLE

The following example shows two conforming include directives.

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

2.4.3.2 DIFFERENCES

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

2.4.3.3 MIGRATION TO THE CCI

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

For example, the following code:

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

should be changed to:

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

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

```
-Ilcd
```

2.4.4 Include Search Paths

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

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

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

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

2.4.4.1 EXAMPLE

If including a header file as in the following directive

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

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

2.4.4.2 DIFFERENCES

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

2.4.4.3 MIGRATION TO THE CCI

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

2.4.5 The Number of Significant Initial Characters in an Identifier

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

2.4.5.1 EXAMPLE

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

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

2.4.5.2 DIFFERENCES

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

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

2.4.5.3 MIGRATION TO THE CCI

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

2.4.6 Sizes of Types

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

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

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

2.4.6.1 EXAMPLE

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

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

2.4.6.2 DIFFERENCES

This is consistent with previous types implemented by the compiler.

2.4.6.3 MIGRATION TO THE CCI

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

2.4.7 Plain `char` Types

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

2.4.7.1 EXAMPLE

The following example

```
char foobar;
```

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

2.4.7.2 DIFFERENCES

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

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

2.4.7.3 MIGRATION TO THE CCI

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

```
signed char foobar;
```

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

2.4.8 Signed Integer Representation

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

2.4.8.1 EXAMPLE

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

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

2.4.8.2 DIFFERENCES

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

2.4.8.3 MIGRATION TO THE CCI

No action required.

2.4.9 Integer conversion

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

2.4.9.1 EXAMPLE

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

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

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

2.4.9.2 DIFFERENCES

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

2.4.9.3 MIGRATION TO THE CCI

No action required.

2.4.10 Bit-wise Operations on Signed Values

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

2.4.10.1 EXAMPLE

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

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

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

2.4.10.2 DIFFERENCES

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

2.4.10.3 MIGRATION TO THE CCI

No action required.

2.4.11 Right-shifting Signed Values

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

2.4.11.1 EXAMPLE

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

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

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

2.4.11.2 DIFFERENCES

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

2.4.11.3 MIGRATION TO THE CCI

No action required.

2.4.12 Conversion of Union Member Accessed Using Member With Different Type

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

2.4.12.1 EXAMPLE

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

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

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

```
float result;
result = foobbar.data;
```

2.4.12.2 DIFFERENCES

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

2.4.12.3 MIGRATION TO THE CCI

No action required.

2.4.13 Default Bit-field int Type

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

2.4.13.1 EXAMPLE

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

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

2.4.13.2 DIFFERENCES

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

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

2.4.13.3 MIGRATION TO THE CCI

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

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

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

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

2.4.14 Bit-fields Straddling a Storage Unit Boundary

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

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

2.4.14.1 EXAMPLE

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

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

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

2.4.14.2 DIFFERENCES

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

2.4.14.3 MIGRATION TO THE CCI

No action required.

2.4.15 The Allocation Order of Bits-field

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

2.4.15.1 EXAMPLE

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

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

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

2.4.15.2 DIFFERENCES

This is identical with the previous operation of all compilers.

2.4.15.3 MIGRATION TO THE CCI

No action required.

2.4.16 The NULL macro

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

2.4.16.1 EXAMPLE

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

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

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

2.4.16.2 DIFFERENCES

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

2.4.16.3 MIGRATION TO THE CCI

No action required.

2.4.17 Floating-point sizes

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

2.4.17.1 EXAMPLE

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

```
float outY;
```

2.4.17.2 DIFFERENCES

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

2.4.17.3 MIGRATION TO THE CCI

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

No migration is required for other compilers.

2.5 ANSI STANDARD EXTENSIONS

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

2.5.1 Generic Header File

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

2.5.1.1 EXAMPLE

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

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

2.5.1.2 DIFFERENCES

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

2.5.1.3 MIGRATION TO THE CCI

Change:

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

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

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

to:

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

2.5.2 Absolute addressing

Variables and functions can be placed at an absolute address by using the `__at()` construct.qualifier Note that XC16/32 may require the variable or function to be placed in a special section for absolute addressing to work. Stack-based (`auto` and parameter) variables cannot use the `__at()` specifier.

2.5.2.1 EXAMPLE

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

```
int scanMode __at(0x200);
const char keys[] __at(123) = { 'r', 's', 'u', 'd'};

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

2.5.2.2 DIFFERENCES

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

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

2.5.2.3 MIGRATION TO THE CCI

Avoid making objects and functions absolute if possible.

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

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

to:

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

In XC16/32, change code such as:

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

to:

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

2.5.2.4 CAVEATS

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

2.5.3 Far Objects and Functions

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

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

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

2.5.3.1 EXAMPLE

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

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

2.5.3.2 DIFFERENCES

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

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

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

2.5.3.3 MIGRATION TO THE CCI

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

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

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

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

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

to

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

2.5.3.4 CAVEATS

None.

2.5.4 Near Objects

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

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

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

2.5.4.1 EXAMPLE

The following shows a variable and function qualified using __near.

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

2.5.4.2 DIFFERENCES

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

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

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

2.5.4.3 MIGRATION TO THE CCI

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

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

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

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

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

to

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

2.5.4.4 CAVEATS

None.

2.5.5 Persistent Objects

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

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

2.5.5.1 EXAMPLE

The following shows a variable qualified using `__persistent`.

```
__persistent int serialNo;
```

2.5.5.2 DIFFERENCES

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

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

2.5.5.3 MIGRATION TO THE CCI

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

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

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

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

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

to

```
int __persistent tblIdx;
```

2.5.5.4 CAVEATS

None.

2.5.6 X and Y Data Objects

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

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

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

2.5.6.1 EXAMPLE

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

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

2.5.6.2 DIFFERENCES

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

Equivalent specifiers have never been defined for any other compiler.

2.5.6.3 MIGRATION TO THE CCI

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

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

to `__xdata`, or `__ydata`, i.e., `__xdata char template[20];`

2.5.6.4 CAVEATS

None.

2.5.7 Banked Data Objects

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

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

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

2.5.7.1 EXAMPLE

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

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

2.5.7.2 DIFFERENCES

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

Equivalent specifiers have never been defined for any other compiler.

2.5.7.3 MIGRATION TO THE CCI

For 8-bit compilers, change any occurrence of the bank*x* qualifiers, as in the following example:

```
bank2 int logEntry;
```

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

2.5.7.4 CAVEATS

None.

2.5.8 Alignment of Objects

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

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

2.5.8.1 EXAMPLE

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

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

2.5.8.2 DIFFERENCES

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

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

2.5.8.3 MIGRATION TO THE CCI

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

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

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

2.5.8.4 CAVEATS

This feature is not yet implemented on XC8.

2.5.9 EEPROM Objects

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

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

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

2.5.9.1 EXAMPLE

The following shows a variable qualified using `__eeprom`.

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

2.5.9.2 DIFFERENCES

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

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

2.5.9.3 MIGRATION TO THE CCI

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

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

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

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

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

to

```
int __eeprom mainSw;
```

2.5.9.4 CAVEATS

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

2.5.10 Interrupt Functions

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

The current interrupt types are:

<empty>

Implement the default interrupt function

low_priority

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

high_priority

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

save(*symbol-list*)

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

irq(*irqid*)

Specify the interrupt vector associated with this interrupt (XC16)

altirq(*altirqid*)

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

preprologue(*asm*)

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

shadow

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

auto_psv

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

no_auto_psv

The ISR will not set the PSVPAG register (XC16)

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

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

2.5.10.1 EXAMPLE

The following shows a function qualified using `__interrupt`.

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

2.5.10.2 DIFFERENCES

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

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

2.5.10.3 MIGRATION TO THE CCI

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

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

to the following, respectively

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

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

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

to

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

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

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

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

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

to

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

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

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

2.5.10.4 CAVEATS

None.

2.5.11 Packing Objects

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

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

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

2.5.11.1 EXAMPLE

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

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

2.5.11.2 DIFFERENCES

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

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

2.5.11.3 MIGRATION TO THE CCI

No migration is required for XC8.

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

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

to:

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

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

2.5.11.4 CAVEATS

None.

2.5.12 Indicating Antiquated Objects

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

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

2.5.12.1 EXAMPLE

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

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

2.5.12.2 DIFFERENCES

No deprecate feature was implemented on 8-bit compilers.

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

2.5.12.3 MIGRATION TO THE CCI

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

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

to:

```
int __deprecate intMask;
```

2.5.12.4 CAVEATS

None.

2.5.13 Assigning Objects to Sections

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

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

2.5.13.1 EXAMPLE

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

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

2.5.13.2 DIFFERENCES

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

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

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

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

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

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

2.5.13.3 MIGRATION TO THE CCI

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

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

to the `__section()` specifier, as in

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

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

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

to:

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

2.5.13.4 CAVEATS

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

2.5.14 Specifying Configuration Bits

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

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

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

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

2.5.14.1 EXAMPLE

The following shows configuration bits being specified using this pragma.

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

2.5.14.2 DIFFERENCES

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

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

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

2.5.14.3 MIGRATION TO THE CCI

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

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

to the `#pragma config` directive, as in

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

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

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

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

to:

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

No migration is required for 32-bit code.

2.5.14.4 CAVEATS

None.

2.5.15 Manifest Macros

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

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

TABLE 2-1: MANIFEST MACROS DEFINED BY THE CCI

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

2.5.15.1 EXAMPLE

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

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

2.5.15.2 DIFFERENCES

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

2.5.15.3 MIGRATION TO THE CCI

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

2.5.15.4 CAVEATS

None.

2.5.16 In-line Assembly

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

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

2.5.16.1 EXAMPLE

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

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

2.5.16.2 DIFFERENCES

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

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

2.5.16.3 MIGRATION TO THE CCI

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

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

to

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

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

2.5.16.4 CAVEATS

None.

2.6 COMPILER FEATURES

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

2.6.1 Enabling the CCI

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

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

2.6.1.1 DIFFERENCES

This option has never been implemented previously.

2.6.1.2 MIGRATION TO THE CCI

Enable the option.

2.6.1.3 CAVEATS

None.

NOTES:

Chapter 3. Compiler Command Line Driver

3.1 INTRODUCTION

The command line driver (`xc32-gcc` or `xc32-g++`) is the application that can be invoked to perform all aspects of compilation, including C/C++ code generation, assembly and link steps. Even if you use an IDE to assist with compilation, the IDE will ultimately call `xc32-gcc` for C projects or `xc32-g++` for C++ projects.

Although the internal compiler applications can be called explicitly from the command line, using the `xc32-gcc` or `xc32-g++` driver is the recommended way to use the compiler as it hides the complexity of all the internal applications used and provides a consistent interface for all compilation steps.

This chapter describes the steps the driver takes during compilation, files that the driver can accept and produce, as well as the command line options that control the compiler's operation. It also shows the relationship between these command line options and the controls in the MPLAB IDE *Build Options* dialog.

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

- Invoking the Compiler
- The C Compilation Sequence
- The C++ Compilation Sequence
- Runtime Files
- Start-up and Initialization
- Compiler Output
- Compiler Messages
- Driver Option Descriptions

3.2 INVOKING THE COMPILER

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

3.2.1 Driver Command Line Format

The compilation driver program (`xc32-gcc`) compiles, assembles and links C and assembly language modules and library archives. The `xc32-g++` driver must be used when the module source is written in C++. Most of the compiler command line options are common to all implementations of the GCC toolset (MPLAB XC16 uses the GCC toolset; XC8 does not). A few are specific to the compiler.

The basic form of the compiler command line is:

```
xc32-gcc [options] files
xc32-g++ [options] files
```

For example, to compile, assemble and link the C source file `hello.c`, creating the absolute executable `hello.elf`,execute this command:

```
xc32-gcc -o hello.elf hello.c
```

Or, to compile, assemble and link the C++ source file hello.cpp, creating the absolute executable hello.elf, execute:

```
xc32-g++ -o hello.elf hello.cpp
```

The available options are described in **Section 3.9 "Driver Option Descriptions"**. It is conventional to supply *options* (identified by a leading *dash* "-" before the filenames), although this is not mandatory.

The *files* may be any mixture of C/C++ and assembler source files, relocatable object files (.o) or archive files. The order of the files is important. It may affect the order in which code or data appears in memory or the search order for symbols. Typically archive files are specified after source files. The file types are described in **Section 3.2.2 "Input File Types"**.

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

Libraries is a list of user-defined object code library files that will be searched by the linker, in addition to the standard C libraries. The order of these files will determine the order in which they are searched. They are typically placed after the source filenames, but this is not mandatory.

It is assumed in this manual that the compiler applications are either in the console's search path, the appropriate environment variables have been specified, or the full path is specified when executing any application.

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 not set. The driver, or other subprogram, takes advantage of internal knowledge about the installation of the compiler. As long as the installation structure remains intact, with all subdirectories and executables remaining in the same relative position, the driver or subprogram will be able to determine a usable value. The "XC32" variables should be used for new projects; however, the "PIC32" variables may be used for legacy projects.

TABLE 3-1: COMPILER-RELATED ENVIRONMENTAL VARIABLES

Option	Definition
XC32_C_INCLUDE_PATH PIC32_C_INCLUDE_PATH	This variable's value is a semicolon-separated list of directories, much like PATH. When the compiler searches for header files, it tries the directories listed in the variable, after the directories specified with -I but before the standard header file directories. If the environment variable is undefined, the preprocessor chooses an appropriate value based on the standard installation. By default, the following directories are searched for include files: <install-path>\pic32mx\include
XC32_COMPILER_PATH PIC32_COMPILER_PATH	The value of PIC32_COMPILER_PATH is a semicolon-separated list of directories, much like PATH. The compiler tries the directories thus specified when searching for subprograms, if it can't find the subprograms using PIC32_EXEC_PREFIX.

TABLE 3-1: COMPILER-RELATED ENVIRONMENTAL VARIABLES

Option	Definition
XC32_EXEC_PREFIX PIC32_EXEC_PREFIX	If PIC32_EXEC_PREFIX is set, it specifies a prefix to use in the names of subprograms executed by the compiler. No directory delimiter is added when this prefix is combined with the name of a subprogram, but you can specify a prefix that ends with a slash if you wish. If the compiler cannot find the subprogram using the specified prefix, it tries looking in your PATH environment variable. If the PIC32_EXEC_PREFIX environment variable is unset or set to an empty value, the compiler driver chooses an appropriate value based on the standard installation. If the installation has not been modified, this will result in the driver being able to locate the required subprograms. Other prefixes specified with the -B command line option take precedence over the user- or driver-defined value of PIC32_EXEC_PREFIX. Under normal circumstances it is best to leave this value undefined and let the driver locate subprograms itself.
XC32_LIBRARY_PATH PIC32_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>\pic32mx\lib.
TMPDIR	If TMPDIR is set, it specifies the directory to use for temporary files. The compiler uses temporary files to hold the output of one stage of compilation that is to be used as input to the next stage: for example, the output of the preprocessor, which is the input to the compiler proper.

3.2.2 Input File Types

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

TABLE 3-2: FILE NAMES

Extensions	Definition
`file.c`	A C source file that must be preprocessed.
`file.cpp`	A C++ source file that must be preprocessed.
`file.h`	A header file (not to be compiled or linked).
`file.i`	A C source file that has already been pre-processed.
`file.o`	An object file.
`file.ii`	A C++ source file that has already been pre-processed.
`file.s`	An assembly language source file.
`file.S`	An assembly language source file that must be preprocessed.
other	A file to be passed to the linker.

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

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

A source file is a file that contains all or part of a program. They may contain C/C++ code, as well as preprocessor directives and commands. Source files are initially passed to the preprocessor by the driver.

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

3.3 THE C COMPILATION SEQUENCE

3.3.1 Single-step C Compilation

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

3.3.1.1 COMPILING A SINGLE C FILE

This section demonstrates how to compile and link a single file. For the purpose of this discussion, it is assumed the compiler's `<install-dir>/bin` directory has been added to your PATH variable. The following are other directories of note:

- `<install-dir>/pic32mx/include` the directory for standard C header files.
- `<install-dir>/pic32mx/include/proc` the directory for PIC32MX device-specific header files.
- `<install-dir>/pic32mx/lib` the directory structure for standard libraries and start-up files.
- `<install-dir>/pic32mx/include/peripheral` the directory for PIC32MX peripheral library include files.
- `<install-dir>/pic32mx/lib/proc` the directory for device-specific linker script fragments, register definition files and configuration data may be found.

The following is a simple C program that adds two numbers. Create the following program with any text editor and save it as `ex1.c`.

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

// Device-Specific Configuration-Bit settings
// SYSCLK = 80 MHz (8MHz Crystal/ FPLLIDIV * FPLLMUL / FPLLODIV)
// PBCLK = 40 MHz
// Primary Osc w/PLL (XT+,HS+,EC+PLL)
// WDT OFF
// Other options are don't care
//
#pragma config FPLLMUL = MUL_20, FPLLIDIV = DIV_2, FPLLODIV = DIV_1,
FWDTEN = OFF
#pragma config POSCMOD = HS, FNOSC = PRIPLL, FPBDIV = DIV_8

unsigned int x, y, z;

unsigned int
add(unsigned int a, unsigned int b)
{
  return(a+b);
}

int
main(void)
{
  /* Configure the target for maximum performance at 80 MHz. */
  SYSTEMConfigPerformance(80000000UL);
  x = 2;
  y = 5;
  z = add(x,y);
  return 0;
}
```

The first line of the program includes the header file `xc.h,` which provides definitions for all Special Function Registers (SFRs) on that part.

Compile the program by typing the following at the prompt:

```
xc32-gcc –mprocessor=32MX795F512L -o ex1.out ex1.c
```

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

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

```
xc32-bin2hex ex1.out
```

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

3.3.1.2 COMPILING MULTIPLE C FILES

This section demonstrates how to compile and link multiple files in a single step. Move the `Add()` function into a file called `add.c` to demonstrate the use of multiple files in an application. That is:

File 1

```
/* ex1.c */
#include <xc.h>
#include <plib.h>

// Device-Specific Configuration-Bit settings
// SYSCLK = 80 MHz (8MHz Crystal/ FPLLIDIV * FPLLMUL / FPLLODIV)
// PBCLK = 40 MHz
// Primary Osc w/PLL (XT+,HS+,EC+PLL)
// WDT OFF
// Other options are don't care
//
#pragma config FPLLMUL = MUL_20, FPLLIDIV = DIV_2, FPLLODIV = DIV_1,
FWDTEN = OFF
#pragma config POSCMOD = HS, FNOSC = PRIPLL, FPBDIV = DIV_8

int main(void);
unsigned int add(unsigned int a, unsigned int b);
unsigned int x, y, z;
int main(void)
{
  /* Configure the target for maximum performance at 80 MHz. */
  SYSTEMConfigPerformance(80000000UL);
  x = 2;
  y = 5;
  z = Add(x,y);
  return 0;
}
```

File 2

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

Compile both files by typing the following at the prompt:

```
xc32-gcc -mprocessor=32MX795F512L -o ex1.out 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.out` is created.

3.3.2 Multi-step C Compilation

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

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

If the compiler is being invoked using a make utility, the make file will need to be configured to use the intermediate files (`.o` files) and the options used to generate the intermediate files (`-c`, see **Section 3.9.2 “Options for Controlling the Kind of Output”**). Make utilities typically call the compiler multiple times: once for each source file to generate an intermediate file, and once to perform the second stage compilation.

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

```
xc32-gcc -mprocessor=32MX795F512L -c ex1.c
xc32-gcc -mprocessor=32MX795F512L -c add.c
xc32-gcc -mprocessor=32MX795F512L -o ex1.out ex1.o add.o
```

3.4 THE C++ COMPILATION SEQUENCE

3.4.1 Single-step C++ Compilation

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

3.4.1.1 COMPILING A SINGLE C++ FILE

This section demonstrates how to compile and link a single file. For the purpose of this discussion, it is assumed the compiler's `<install-dir>/bin` directory has been added to your PATH variable. The following are other directories of note:

- `<install-dir>/pic32mx/include/cpp` the directory for standard C++ header files.
- `<install-dir>/pic32mx/include/proc` the directory for PIC32MXdevice-specific header files.
- `<install-dir>/pic32mx/lib` the directory structure for standard libraries and start-up files.
- `<install-dir>/pic32mx/include/peripheral` the directory for PIC32 peripheral library include files.
- `<install-dir>/pic32mx/lib/proc` the directory for device-specific linker script fragments, register definition files, and configuration data may be found.

The following is a simple C++ program. Create the following program with any plain-text editor and save it as `ex2.cpp`.

```
/* ex2.cpp */
#include <xc.h>     // __XC_UART
#include <plib.h>   // SYSTEMConfigPerformance()

#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <set>
#include <map>
#include <string>
#include <algorithm>
#include <iterator>
#include <functional>
#include <numeric>
using namespace std;

// Device-Specific Configuration-bit settings
#pragma config FPLLMUL=MUL_20, FPLLIDIV=DIV_2, FPLLODIV=DIV_1,
FWDTEN=OFF
#pragma config POSCMOD=HS, FNOSC=PRIPLL, FPBDIV=DIV_8

template <class T>
inline void print_elements (const T& coll, const char* optcstr="")
{
    typename T::const_iterator pos;

    std::cout << optcstr;
    for (pos=coll.begin(); pos!=coll.end(); ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}

template <class T>
inline void insert_elements (T& coll, int first, int last)
{
    for (int i=first; i<=last; ++i)
    {
        coll.insert(coll.end(),i);
    }
}

int main(void) {

    // Configure the target for max performance at 80 MHz.
    SYSTEMConfigPerformance (80000000UL);

    // Direct stdout to UART 1 for use with the simulator
    __XC_UART = 1;

    deque<int> coll;
    insert_elements(coll,1,9);
    insert_elements(coll,1,9);

    print_elements(coll, "on entry: ");
    //sort elements
    sort (coll.begin(), coll.end());

    print_elements(coll, "sorted:  ");

    //sorted reverse
    sort (coll.begin(), coll.end(), greater<int>());

    print_elements(coll, "sorted >: ");

    while(1);
}
```

The first line of the program includes the header file `xc.h`, which provides definitions for all Special Function Registers (SFRs) on the target device. The second file of the program includes the header file `plib.h`, which provides the necessary prototypes for the peripheral library.

Compile the program by typing the following at a command prompt.

```
xc32-g++ -mprocessor=32MX795F512L -Wl,--defsym=_min_heap_size=0xF000
-o ex2.elf ex2.cpp
```

The option `-o ex2.elf` names the output executable file. This elf file may be loaded into MPLAB X IDE.

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

```
xc32-bin2hex ex2.elf
```

This creates an Intel hex file named `ex2.hex`.

3.4.2 Compiling Multiple C and C++ files

This section demonstrates how to compile and link multiple C and C++ files in a single step.

File 1

```
/* main.cpp */
#include <xc.h>      // __XC_UART
#include <plib.h>    // SYSTEMConfigPerformance()

#include <iostream>
using namespace std;

// Device-Specific Configuration-bit settings
#pragma config FPLLMUL=MUL_20, FPLLIDIV=DIV_2, FPLLODIV=DIV_1,
FWDTEN=OFF
#pragma config POSCMOD=HS, FNOSC=PRIPLL, FPBDIV=DIV_8

// add() must have C linkage
extern "C" {
extern unsigned int add(unsigned int a, unsigned int b);
}

int main(void) {
    int myvalue = 6;

    // Configure the target for max performance at 80 MHz.
    SYSTEMConfigPerformance (80000000UL);

    // Direct stdout to UART 1 for use with the simulator
    __XC_UART = 1;

    std::cout << "original value: " << myvalue << endl;
    myvalue = add (myvalue, 3);
    std::cout << "new value:      " << myvalue << endl;

    while(1);
}
```

File 2

```
/* ex3.c */
unsigned int
add(unsigned int a, unsigned int b)
{
  return(a+b);
}
```

Compile both files by typing the following at the prompt:

```
xc32-g++ -mprocessor=32MX795F512L -o ex3.elf main.cpp ex3.c
```

The command compiles the modules `main.cpp` and `ex3.c`. The compiled modules are linked with the compiler libraries for C++ and the executable file `ex3.elf` is created.

Note: Use the xc32-g++ driver (as opposed to the xc32-gcc driver) in order to link the project with the C++ support libraries necessary for the C++ source file in the project.

3.5 RUNTIME FILES

In addition to the C/C++ and assembly source files specified on the command line, there are also compiler-generated source files and pre-compiled library files which might be compiled into the project by the driver. These files contain:

- C/C++ Standard library routines
- Implicitly called arithmetic routines
- User-defined library routines
- The runtime start-up code

3.5.1 Library Files

The names of the C/C++ standard library files appropriate for the selected target device, and other driver options, are determined by the driver.

The target libraries, called multilibs, are built multiple times with a permuted set of options. When the compiler driver is called to compile and link an application, the driver chooses the version of the target library that has been built with the same options.

By default, the 32-bit language tools use the directory `<install-directory>/lib/gcc/` to store the specific libraries and the directory `<install-directory>/<pic32mx>/lib` to store the target-specific libraries. Both of these directory structures contain subdirectories for each of the multilib combinations specified above. These subdirectories, respectively, are as follows:

```
1. .
2. ./size
3. ./speed
4. ./mips32
5. ./no-float
6. ./mips32/no-float
7. ./size/mips32
8. ./size/no-float
9. ./size/mips32/no-float
10../speed/mips32
11../speed/no-float
12../speed/mips32/no-float
```

The target libraries that are distributed with the compiler are built for the following options:

- Size versus speed (`-Os` vs. `-O3`)
- 16-bit versus 32-bit (`-mips16` vs. `-mno-mips16`)

By default the 32-bit language tools compile for `-O0`, `-mno-mips16`, and `-msoft-float`. Therefore, the options that we are concerned with are `-Os` or `-O3`, `-mips16`, and `-mno-float`. Libraries built with the following command line options are made available:

1. Default command line options
2. `-Os`
3. `-O3`
4. `-mips16`
5. `-mno-float`
6. `-mips16 -mno-float`
7. `-Os -mips16`
8. `-Os -mno-float`
9. `-Os -mips16 -mno-float`
10. `-O3 -mips16`
11. `-O3 -mno-float`
12. `-O3 -mips16 -mno-float`

The following examples provide details on which of the multilibs subdirectories are chosen.

1.
   ```
   xc32-gcc foo.c
   xc32-g++ foo.cpp
   ```
   For this example, no command line options have been specified (i.e., the default command line options are being used). In this case, the `.` subdirectories are used.
2.
   ```
   xc32-gcc -Os foo.c
   xc32-g++ -Os foo.cpp
   ```
   For this example, the command line option for optimizing for size has been specified (i.e., `-Os` is being used). In this case, the `./size` subdirectories are used.
3.
   ```
   xc32-gcc -O2 foo.c
   xc32-g++ -O2 foo.cpp
   ```
   For this example, the command line option for optimizing has been specified; however, this command line option optimizes for neither size nor space (i.e., `-O2` is being used). In this case, the `.` subdirectories are used.
4.
   ```
   xc32-gcc -Os -mips16 foo.c
   xc32-g++ -Os -mips16 foo.cpp
   ```
   For this example, the command line options for optimizing for size and for MIPS16 code have been specified (i.e., `-Os` and `-mips16` are being used). In this case, the `./size/mips16` subdirectories are used.

3.5.1.1 STANDARD LIBRARIES

The C/C++ standard libraries contain a standardized collection of functions, such as string, math and input/output routines. The range of these functions are described in **Chapter 13. "Library Routines"**.

These libraries also contain C/C++ routines that are implicitly called by the output code of the code generator. These are routines that perform tasks such as floating-point operations and that may not directly correspond to a C/C++ function call in the source code.

3.5.1.2 USER-DEFINED LIBRARIES

User-defined libraries may be created and linked in with programs as required. Library files are more easy to manage and may result in faster compilation times, but must be compatible with the target device and options for a particular project. Several versions of a library may need to be created to allow it to be used for different projects.

User-created libraries that should be searched when building a project can be listed on the command line along with the source files.

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

3.5.2 Peripheral Library Functions

Many of the peripherals of the PIC32MX devices are supported by the peripheral library functions provided with the compiler tools. See the “*32-Bit Language Tools Libraries”* (DS51685) for details on the functions provided.

3.6 START-UP AND INITIALIZATION

For C:

There is only one start-up module, which initializes the C runtime environment. The source code for this is found in `<install-directory>/pic32-libs/libpi32c/startup/crt0.S` and it is precompiled into the library `<install-directory>/pic32mx/lib/crt0.o`. Multilib versions of these modules exist in order to support architectural differences between device families.

For C++:

Code from five object files link sequentially to create a single initialization routine, which initializes the C++ runtime environment.

The source code for this is found in `<install-directory>/pic32-libs/libpic32/startup`.

The PIC32 precompiled startup objects are located in `<install-directory>/pic32mx/lib/` and the filenames are `cpprt0.o`, `crti.o`, and `crtn.o`.

The GCC precompiled startup objects are located in `<install-directory>/lib/gcc/pic32mx/<gcc-version>/` and the filenames are `crtbegin.o` and `crtend.o`. Multilib variations of these modules exist in order to support architectural differences between device families and also optimization settings.

For more information about what the code in these start-up modules actual does, see **Section 12.3 "Runtime Start-up Code"**.

3.7 COMPILER OUTPUT

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

3.7.1 Output Files

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

TABLE 3-3: FILE NAMES

Extensions	Definition
`file.hex`	Executable file
`file.elf`	ELF debug file
`file.o`	Object file (intermediate file)
`file.s`	Assembly code file (intermediate file)
`file.i`	Preprocessed C file (intermediate file)
`file.ii`	Preprocessed C++ file (intermediate file)
`file.map`	Map file

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

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

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

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

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

The default behavior of `xc32-gcc` and xc32-g++ is to produce an ELF output. To make changes to the file's output or the file names, see **Section 3.9 "Driver Option Descriptions"**.

3.7.2 Diagnostic Files

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

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

The option to create a listing file in the assembler is `-a` (or `-Wa,-a` if passed to the driver). There are many variants to this option, which may be found in the "*MPLAB Assembler, Linker and Utilities for PIC32 MCUs User's Guide"*(DS51833). To pass the option from the compiler, see **Section 3.9.9 "Options for Assembling"**.

There is one list file produced for each build. There is one assembler listing file for each translation unit. This is a pre-link assembler listing so it will not show final addresses. Thus, if you require a list file for each source file, these files must be compiled separately, see **Section 3.3.2 "Multi-step C Compilation"**. This is the case if you build using MPLAB IDE. Each list file will be assigned the module name and extension `.lst`.

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

The option to create a map file in the linker is `-Map` *`file`* (or `-Wl,-Map=file` if passed to the driver), which may be found in the "*MPLAB Assembler, Linker and Utilities for PIC32 User's Guide"*. To pass the option from the compiler, see **Section 3.9.10 "Options for Linking"**.

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

3.8 COMPILER MESSAGES

There are three types of messages. These are described below along with the compiler’s behavior when encountering a message of each type.

Warning Messages indicate source code or some other situation that can be compiled, but is unusual and may lead to a runtime failure of the code. The code or situation that triggered the warning should be investigated; however, compilation of the current module will continue, as will compilation of any remaining modules.

Error Messages indicate source code that is illegal or that compilation of this code cannot take place. Compilation will be attempted for the remaining source code in the current module, but no additional modules will be compiled and the compilation process will then conclude.

Fatal Error Messages indicate a situation that cannot allow compilation to proceed and which requires the compilation process to stop immediately.

For information on options that control compiler output of errors, warnings or comments, see **Section 3.9.4 “Options for Controlling the C++ Dialect”**.

3.9 DRIVER OPTION DESCRIPTIONS

All single letter options are identified by a leading *dash* character, “-”, e.g. `-c`. Some single letter options specify an additional data field which follows the option name immediately and without any *whitespace*, e.g. `-Idir`. Options are case sensitive, so `-c` is a different option to `-C`.

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

- Options Specific to PIC32MX Devices
- Options for Controlling the Kind of Output
- Options for Controlling the C Dialect
- Options for Controlling the C++ Dialect
- 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

3.9.1 Options Specific to PIC32MX Devices

These options are specific to the device, not the compiler.

TABLE 3-4: PIC32MX DEVICE-SPECIFIC OPTIONS

Option	Definition
`-G` *`num`*	Put global and static items less than or equal to *`num`* bytes into the small data or bss section instead of the normal data or bss section. This allows the data to be accessed using a single instruction. All modules should be compiled with the same `-G` *`num`* value.
`-mappio-debug`	Enable the APPIN/APPOUT debugging library functions for the MPLAB® ICD 3 debugger and MPLAB REAL ICE™ in-circuit emulator. This feature allows you to use the DBPRINTF and related functions and macros as described in the *“32-bit Language Tool Libraries”* document (DS51685). Enable this option only when using a target PIC32 device that supports the APPIN/APPOUT feature.
`-mcci`	Enables the Microchip Common C Interface compilation mode.
`-mcheck-zero-division` `-mno-check-zero-division`	Trap (do not trap) on integer division by zero. The default is `-mcheck-zero-division`.
`-membedded-data` `-mno-embedded-data`	Allocate variables to the read-only data section first if possible, then next in the small data section if possible, otherwise in data. This gives slightly slower code than the default, but reduces the amount of RAM required when executing, and thus may be preferred for some embedded systems.
`-mframe-header-opt`	Allows the compiler to omit a few instructions for each function that does not use its incoming frame header. This feature usually improves both execution speed and code size.
`-mips16` `-mno-mips16`	Generate (do not generate) MIPS16 code. This is only available in the PRO edition.
`-mlong-calls` `-mno-long-calls`	Disable (do not disable) use of the `jal` instruction. Calling functions using `jal` is more efficient but requires the caller and callee to be in the same 256 megabyte segment. This option has no effect on abicalls code. The default is `-mno-long-calls`.
`-mmemcpy` `-mno-memcpy`	Force (do not force) the use of `memcpy()` for non-trivial block moves. The default is `-mno-memcpy`, which allows GCC to inline most constant-sized copies.
`-mno-float`	Do not use software floating-point libraries.
`-mno-peripheral-libs`	`-mno-peripheral-libs` is now the default. `-mperipheral-libs` is optional. By default, the peripheral libraries are linked specified via the device-specific linker script. Do not use the standard peripheral libraries when linking.
`-mprocessor`	Selects the device for which to compile. (e.g., `-mprocessor=32MX360F512L`)

TABLE 3-4: PIC32MX DEVICE-SPECIFIC OPTIONS (CONTINUED)

Option	Definition
-msmart-io=[0|1|2]	This option attempts to statically analyze format strings passed to printf, scanf and the 'f' and 'v' variations of these functions. Uses of nonfloating-point format arguments will be converted to use an integer-only variation of the library function. For many applications, this feature can reduce program-memory usage. -msmart-io=0 disables this option, while -msmart-io=2 causes the compiler to be optimistic and convert function calls with variable or unknown format arguments. -msmart-io=1 is the default and will convert only when the compiler can prove that floating-point support is not required.
-muninit-const-in-rodata -mno-uninit-const-in-rodata	Put uninitialized const variables in the read-only data section. This option is only meaningful in conjunction with -membedded-data.

3.9.2 Options for Controlling the Kind of Output

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

TABLE 3-5: 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.
-fexceptions	Enable exception handling. You may need to enable this option when compiling C code that needs to interoperate properly with exception handlers written in C++.
-o file	Place the output in *file*.
-S	Stop after compilation proper (i.e., before invoking the assembler). The default output file extension is .s.
-v	Print the commands executed during each stage of compilation.
-x	You can specify the input language explicitly with the -x option: -x *language* Specify explicitly the language for the following input files (rather than letting the compiler choose a default based on the file name suffix). This option applies to all following input files until the next -x option. The following values are supported by the compiler: c c++ c-header cpp-output assembler assembler-with-cpp -x none Turn off any specification of a language, so that subsequent files are handled according to their file name suffixes. This is the default behavior but is needed if another -x option has been used. For example: xc32-gcc -x assembler foo.asm bar.asm -x none main.c mabonga.s Without the -x none, the compiler assumes all the input files are for the assembler.

TABLE 3-5: KIND-OF-OUTPUT CONTROL OPTIONS (CONTINUED)

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

3.9.3 Options for Controlling the C Dialect

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

TABLE 3-6: C DIALECT CONTROL OPTIONS

Option	Definition
`-ansi`	Support all (and only) ANSI-standard C programs.
`-aux-info filename`	Output to the given filename prototyped declarations for all functions declared and/or defined in a translation unit, including those in header files. This option is silently ignored in any language other than C. Besides declarations, the file indicates, in comments, the origin of each declaration (source file and line), whether the declaration was implicit, prototyped or unprototyped (`I`, `N` for new or `O` for old, respectively, in the first character after the line number and the colon), and whether it came from a declaration or a definition (`C` or `F`, respectively, in the following character). In the case of function definitions, a K&R-style list of arguments followed by their declarations is also provided, inside comments, after the declaration.

`-fcheck-new /`
`-fno-check-new`
`(default)`
	Check that the pointer returned by operator new is non-null.

`-fenforce-eh-specs`
`(default) /`
`-fno-enforce-eh-specs`
	Generate/Do not generate code to check for violation of exception specifications at runtime. The `-fno-enforce-eh-specs` option violates the C++ standard, but may be useful for reducing code size in production builds, much like defining `NDEBUG'. This does not give user code permission to throw exceptions in violation of the exception specifications; the compiler will still optimize based on the specifications, so throwing an unexpected exception will result in undefined behavior.

`-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 start-up 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.
`-fno-exceptions`	Disable C++ exception handling. This option disables the generation of extra code needed to propagate exceptions.

TABLE 3-6: C DIALECT CONTROL OPTIONS (CONTINUED)

Option	Definition
-fno-rtti	Enable/Disable runtime type-identification features. The -fno-rtti option disables generation of information about every class with virtual functions for use by the C++ runtime type identification features ('dynamic_cast' and 'typeid'). If you don't use those parts of the language, you can save some space by using this flag. Note that exception handling uses the same information, but it will generate it as needed. The 'dynamic_cast' operator can still be used for casts that do not require runtime type information, i.e. casts to void * or to unambiguous base classes.
-fsigned-char	Let the type char be signed, like signed char. **(This is the default.)**
-fsigned-bitfields -funsigned-bitfields -fno-signed-bitfields -fno-unsigned-bitfields	These options control whether a bit field is signed or unsigned, when the declaration does not use either signed or unsigned. By default, such a bit field is signed, unless -traditional is used, in which case bit fields are always unsigned.
-funsigned-char	Let the type char be unsigned, like unsigned char.
-fwritable-strings	Store strings in the writable data segment and do not make them unique.

3.9.4 Options for Controlling the C++ Dialect

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

TABLE 3-7: C++ DIALECT CONTROL OPTIONS

Option	Definition
-ansi	Support all (and only) ANSI-standard C++ programs.
-aux-info filename	Output to the given filename prototyped declarations for all functions declared and/or defined in a translation unit, including those in header files. This option is silently ignored in any language other than C++. Besides declarations, the file indicates, in comments, the origin of each declaration (source file and line), whether the declaration was implicit, prototyped or unprototyped (I, N for new or O for old, respectively, in the first character after the line number and the colon), and whether it came from a declaration or a definition (C or F, respectively, in the following character). In the case of function definitions, a K&R-style list of arguments followed by their declarations is also provided, inside comments, after the declaration.
-ffreestanding	Assert that compilation takes place in a freestanding environment. This implies -fno-builtin. A freestanding environment is one in which the standard library may not exist, and program start-up 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.

TABLE 3-7: C++ DIALECT CONTROL OPTIONS (CONTINUED)

Option	Definition
`-fsigned-char`	Let the type `char` be signed, like `signed char`. **(This is the default.)**

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

`-funsigned-char`	Let the type `char` be unsigned, like `unsigned char`.
`-fwritable-strings`	Store strings in the writable data segment and do not make them unique.

3.9.5 Options for Controlling Warnings and Errors

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

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

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

TABLE 3-8: WARNING AND ERROR OPTIONS IMPLIED BY ALL WARNINGS

Option	Definition
`-fsyntax-only`	Check the code for syntax, but don’t do anything beyond that.
`-pedantic`	Issue all the warnings demanded by strict ANSI C. Reject all programs that use forbidden extensions.
`-pedantic-errors`	Like `-pedantic`, except that errors are produced rather than warnings.
`-w`	Inhibit all warning messages.
`-Wall`	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. Note that some warning flags 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. Some of them are enabled by `-Wextra` but many of them must be enabled individually.
`-Waddress`	Warn about suspicious uses of memory addresses. These include using the address of a function in a conditional expression, such as `void func(void)`; if `(func)`, and comparisons against the memory address of a string literal, such as if `(x == "abc")`. Such uses typically indicate a programmer error: the address of a function always evaluates to true, so their use in a conditional usually indicates that the programmer forgot the parentheses in a function call; and comparisons against string literals result in unspecified behavior and are not portable in C, so they usually indicate that the programmer intended to use `strcmp`.

TABLE 3-8: WARNING AND ERROR OPTIONS IMPLIED BY ALL WARNINGS

Option	Definition
`-Wchar-subscripts`	Warn if an array subscript has type `char`.
`-Wcomment`	Warn whenever a comment-start sequence `/*` appears in a `/*` comment, or whenever a Backslash-Newline appears in a `//` comment.
`-Wdiv-by-zero`	Warn about compile-time integer division by zero. To inhibit the warning messages, use `-Wno-div-by-zero`. Floating-point division by zero is not warned about, as it can be a legitimate way of obtaining infinities and NaNs. **(This is the default.)**
`-Wformat`	Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified.
`-Wimplicit`	Equivalent to specifying both `-Wimplicit-int` and `-Wimplicit-function-declaration`.
`-Wimplicit-function-declaration`	Give a warning whenever a function is used before being declared.
`-Wimplicit-int`	Warn when a declaration does not specify a type.
`-Wmain`	Warn if the type of `main` is suspicious. `main` should be a function with external linkage, returning `int`, taking either zero, two or three arguments of appropriate types.

`-Wmissing-braces`	Warn if an aggregate or union initializer is not fully bracketed. In the following example, the initializer for `a` is not fully bracketed, but that for `b` is fully bracketed.
	`int a[2][2] = { 0, 1, 2, 3 };`
	`int b[2][2] = { { 0, 1 }, { 2, 3 } };`

`-Wno-multichar`	Warn if a multi-character `character` constant is used. Usually, such constants are typographical errors. Since they have implementation-defined values, they should not be used in portable code. The following example illustrates the use of a multi-character `character` constant:
	`char`
	`xx(void)`
	`{`
	`return('xx');`
	`}`

`-Wparentheses`	Warn if parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, or when operators are nested whose precedence people often find confusing.
`-Wreturn-type`	Warn whenever a function is defined with a return-type that defaults to `int`. Also warn about any `return` statement with no return-value in a function whose return-type is not `void`.

TABLE 3-8: WARNING AND ERROR OPTIONS IMPLIED BY ALL WARNINGS

Option	Definition
`-Wsequence-point`	Warn about code that may have undefined semantics because of violations of sequence point rules in the C standard. The C standard defines the order in which expressions in a C program are evaluated in terms of sequence points, which represent a partial ordering between the execution of parts of the program: those executed before the sequence point and those executed after it. These occur after the evaluation of a full expression (one which is not part of a larger expression), after the evaluation of the first operand of a `&&`, `||`, `? :` or `,` (comma) operator, before a function is called (but after the evaluation of its arguments and the expression denoting the called function), and in certain other places. Other than as expressed by the sequence point rules, the order of evaluation of subexpressions of an expression is not specified. All these rules describe only a partial order rather than a total order, since, for example, if two functions are called within one expression with no sequence point between them, the order in which the functions are called is not specified. However, the standards committee has ruled that function calls do not overlap. It is not specified when between sequence points modifications to the values of objects take effect. Programs whose behavior depends on this have undefined behavior. The C standard specifies that “Between the previous and next sequence point, an object shall have its stored value modified, at most once, by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.” If a program breaks these rules, the results on any particular implementation are entirely unpredictable. Examples of code with undefined behavior are `a = a++;`, `a[n] = b[n++]` and `a[i++] = i;`. Some more complicated cases are not diagnosed by this option, and it may give an occasional false positive result, but in general it has been found fairly effective at detecting this sort of problem in programs.
`-Wswitch`	Warn whenever a `switch` statement has an index of enumeral type and lacks a case for one or more of the named codes of that enumeration. (The presence of a default label prevents this warning.) `case` labels outside the enumeration range also provoke warnings when this option is used.
`-Wsystem-headers`	Print warning messages for constructs found in system header files. Warnings from system headers are normally suppressed on the assumption that they usually do not indicate real problems and would only make the compiler output harder to read. Using this command line option tells the compiler to emit warnings from system headers as if they occurred in user code. However, note that using `-Wall` in conjunction with this option does not warn about unknown pragmas in system headers. For that, `-Wunknown-pragmas` must also be used.
`-Wtrigraphs`	Warn if any trigraphs are encountered (assuming they are enabled).

TABLE 3-8: WARNING AND ERROR OPTIONS IMPLIED BY ALL WARNINGS

Option	Definition
`-Wuninitialized`	Warn if an automatic variable is used without first being initialized. These warnings are possible only when optimization is enabled, because they require data flow information that is computed only when optimizing. These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared `volatile`, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers. Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.
`-Wunknown-pragmas`	Warn when a `#pragma` directive is encountered which is not understood by the compiler. If this command line option is used, warnings will even be issued for unknown pragmas in system header files. This is not the case if the warnings were only enabled by the `-Wall` command line option.
`-Wunused`	Warn whenever a variable is unused aside from its declaration, whenever a function is declared static but never defined, whenever a label is declared but not used, and whenever a statement computes a result that is explicitly not used. In order to get a warning about an unused function parameter, both `-W` and `-Wunused` must be specified. Casting an expression to void suppresses this warning for an expression. Similarly, the `unused` attribute suppresses this warning for unused variables, parameters and labels.
`-Wunused-function`	Warn whenever a static function is declared but not defined or a non-inline static function is unused.
`-Wunused-label`	Warn whenever a label is declared but not used. To suppress this warning, use the `unused` attribute.
`-Wunused-parameter`	Warn whenever a function parameter is unused aside from its declaration. To suppress this warning, use the `unused` attribute.
`-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.
`-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 you might occasionally 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 3-9: WARNING AND ERROR OPTIONS NOT IMPLIED BY ALL WARNINGS

Option	Definition
-W	Print extra warning messages for these events: • A nonvolatile automatic variable might be changed by a call to longjmp. These warnings are possible only in optimizing compilation. The compiler sees only the calls to setjmp. It cannot know where longjmp will be called. In fact, a signal handler could call it at any point in the code. As a result, a warning may be generated even when there is in fact no problem, because longjmp cannot in fact be called at the place that would cause a problem. • A function could exit both via return value; and return;. Completing the function body without passing any return statement is treated as return;. • An expression-statement or the left-hand side of a comma expression contains no side effects. To suppress the warning, cast the unused expression to void. For example, an expression such as x[i,j] causes a warning, but x[(void)i,j] does not. • An unsigned value is compared against zero with < or <=. • A comparison like x<=y<=z appears, This is equivalent to (x<=y ? 1 : 0) <= z, which is a different interpretation from that of ordinary mathematical notation. • Storage-class specifiers like static are not the first things in a declaration. According to the C Standard, this usage is obsolescent. • If -Wall or -Wunused is also specified, warn about unused arguments. • A comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. (But don't warn if -Wno-sign-compare is also specified.) • An aggregate has a partly bracketed initializer. For example, the following code would evoke such a warning, because braces are missing around the initializer for x.h: struct s { int f, g; }; struct t { struct s h; int i; }; struct t x = { 1, 2, 3 }; • An aggregate has an initializer that does not initialize all members. For example, the following code would cause such a warning, because x.h would be implicitly initialized to zero: struct s { int f, g, h; }; struct s x = { 3, 4 };
-Waggregate-return	Warn if any functions that return structures or unions are defined or called.
-Wbad-function-cast	Warn whenever a function call is cast to a non-matching type. For example, warn if int foof() is cast to anything *.
-Wcast-align	Warn whenever a pointer is cast, such that the required alignment of the target is increased. For example, warn if a char * is cast to an int *.
-Wcast-qual	Warn whenever a pointer is cast, so as to remove a type qualifier from the target type. For example, warn if a const char * is cast to an ordinary char *.

TABLE 3-9: WARNING AND ERROR OPTIONS NOT IMPLIED BY ALL WARNINGS (CONTINUED)

Option	Definition
`-Wconversion`	Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed point argument, except when the same as the default promotion. Also, warn if a negative integer constant expression is implicitly converted to an unsigned type. For example, warn about the assignment `x = -1` if `x` is unsigned. But do not warn about explicit casts like `(unsigned) -1`.
`-Werror`	Make all warnings into errors.
`-Winline`	Warn if a function can not be inlined, and either it was declared as inline, or else the `-finline-functions` option was given.
`-Wlarger-than-len`	Warn whenever an object of larger than `len` bytes is defined.
`-Wlong-long` `-Wno-long-long`	Warn if `long long` type is used. This is default. To inhibit the warning messages, use `-Wno-long-long`. Flags `-Wlong-long` and `-Wno-long-long` are taken into account only when `-pedantic` flag is used.
`-Wmissing-declarations`	Warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a prototype.
`-Wmissing-format-attribute`	If `-Wformat` is enabled, also warn about functions that might be candidates for format attributes. Note these are only possible candidates, not absolute ones. This option has no effect unless `-Wformat` is enabled.
`-Wmissing-noreturn`	Warn about functions that might be candidates for attribute `noreturn`. These are only possible candidates, not absolute ones. Care should be taken to manually verify functions. In fact, do not ever return before adding the `noreturn` attribute, otherwise subtle code generation bugs could be introduced.
`-Wmissing-prototypes`	Warn if a global function is defined without a previous prototype declaration. This warning is issued even if the definition itself provides a prototype. (This option can be used to detect global functions that are not declared in header files.)
`-Wnested-externs`	Warn if an `extern` declaration is encountered within a function.
`-Wno-deprecated-declarations`	Do not warn about uses of functions, variables and types marked as deprecated by using the `deprecated` attribute.
`-Wpadded`	Warn if padding is included in a structure, either to align an element of the structure or to align the whole structure.
`-Wpointer-arith`	Warn about anything that depends on the size of a function type or of `void`. The compiler assigns these types a size of 1, for convenience in calculations with `void *` pointers and pointers to functions.
`-Wredundant-decls`	Warn if anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.
`-Wshadow`	Warn whenever a local variable shadows another local variable.

TABLE 3-9: WARNING AND ERROR OPTIONS NOT IMPLIED BY ALL WARNINGS (CONTINUED)

Option	Definition
`-Wsign-compare` `-Wno-sign-compare`	Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. This warning is also enabled by `-W`. To get the other warnings of `-W` without this warning, use `-W -Wno-sign-compare`.
`-Wstrict-prototypes`	Warn if a function is declared or defined without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration which specifies the argument types.)
`-Wtraditional`	Warn about certain constructs that behave differently in traditional and ANSI C. • Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C. • A function declared external in one block and then used after the end of the block. • A switch statement has an operand of type `long`. • A nonstatic function declaration follows a static one. This construct is not accepted by some traditional C compilers.
`-Wundef`	Warn if an undefined identifier is evaluated in an `#if` directive.
`-Wunreachable-code`	Warn if the compiler detects that code will never be executed. It is possible for this option to produce a warning even though there are circumstances under which part of the affected line can be executed, so care should be taken when removing apparently unreachable code. For instance, when a function is inlined, a warning may mean that the line is unreachable in only one inlined copy of the function.
`-Wwrite-strings`	Give string constants the type `const char[length]` so that copying the address of one into a non-const `char *` pointer gets a warning. At compile time, these warnings help you find code that you can try to write into a string constant, but only if you have been very careful about using `const` in declarations and prototypes. Otherwise, it’s just a nuisance, which is why `-Wall` does not request these warnings.

3.9.6 Options for Debugging

The following options are used for debugging.

TABLE 3-10: DEBUGGING OPTIONS

Option	Definition
`-g`	Produce debugging information. The compiler supports the use of `-g` with `-O` making it possible to debug optimized code. The shortcuts taken by optimized code may occasionally produce surprising results: • Some declared variables may not exist at all • Flow of control may briefly move unexpectedly • Some statements may not be executed because they compute constant results or their values were already at hand • Some statements may execute in different places because they were moved out of loops Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.
`-Q`	Makes the compiler print out each function name as it is compiled, and print some statistics about each pass when it finishes.

`-save-temps`
`-save-temps=cwd`
	Don't delete intermediate files. Place them in the current directory and name them based on the source file. Thus, compiling `foo.c` with `-c -save-temps` would produce the following files:
	`foo.i` (preprocessed file)
	`foo.s` (assembly language file)
	`foo.o` (object file)

`-save-temps=obj`	Similar to `-save-temps=cwd`, but if the `-o` option is specified, the temporary files are based on the object file. If the `-o` option is not specified, the `-save-temps=obj` switch behaves like `-save-temps`.
	For example:
	`xc32-gcc -save-temps=obj -c foo.c`
	`xc32-gcc -save-temps=obj -c bar.c -o dir/xbar.o`
	`xc32-gcc -save-temps=obj foobar.c -o dir2/yfoobar`
	would create `foo.i`, `foo.s`, `dir/xbar.i`, `dir/xbar.s`, `dir2/yfoobar.i`, `dir2/yfoobar.s`, and `dir2/yfoobar.o`.

3.9.7 Options for Controlling Optimization

The following options control compiler optimizations.

TABLE 3-11: GENERAL OPTIMIZATION OPTIONS

Option	Edition	Definition
`-O0`	All	Do not optimize. **(This is the default.)** Without `-O`, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code. The compiler only allocates variables declared `register` in registers.
`-O` `-O1`	All	Optimization level 1. Optimizing compilation takes somewhat longer, and a lot more host memory for a large function. With `-O`, the compiler tries to reduce code size and execution time. When `-O` is specified, the compiler turns on `-fthread-jumps` and `-fdefer-pop`. The compiler turns on `-fomit-frame-pointer`.

TABLE 3-11: GENERAL OPTIMIZATION OPTIONS (CONTINUED)

Option	Edition	Definition
-O2	STD, PRO	Optimization level 2. The compiler performs nearly all supported optimizations that do not involve a space-speed trade-off. -O2 turns on all optional optimizations except for loop unrolling (-funroll-loops), function inlining (-finline-functions), and strict aliasing optimizations (-fstrict-aliasing). It also turns on force copy of memory operands (-fforce-mem) and Frame Pointer elimination (-fomit-frame-pointer). As compared to -O, this option increases both compilation time and the performance of the generated code.
-O3	PRO	Optimization level 3. -O3 turns on all optimizations specified by -O2 and also turns on the inline-functions option.
-Os	PRO	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 3-12: SPECIFIC OPTIMIZATION OPTIONS

Option	Definition
-falign-functions -falign-functions=n	Align the start of functions to the next power-of-two greater than n, skipping up to n bytes. For instance, -falign-functions=32 aligns functions to the next 32-byte boundary, but -falign-functions=24 would align to the next 32-byte boundary only if this can be done by skipping 23 bytes or less. -fno-align-functions and -falign-functions=1 are equivalent and mean that functions are not aligned. The assembler only supports this flag when n is a power of two, so n is rounded up. If n is not specified, use a machine-dependent default.
-falign-labels -falign-labels=n	Align all branch targets to a power-of-two boundary, skipping up to n bytes like -falign-functions. This option can easily make code slower, because it must insert dummy operations for when the branch target is reached in the usual flow of the code. If -falign-loops or -falign-jumps are applicable and are greater than this value, then their values are used instead. If n is not specified, use a machine-dependent default which is very likely to be 1, meaning no alignment.
-falign-loops -falign-loops=n	Align loops to a power-of-two boundary, skipping up to n bytes like -falign-functions. The hope is that the loop is executed many times, which makes up for any execution of the dummy operations. If n is not specified, use a machine-dependent default.
-fcaller-saves	Enable values to be allocated in registers that are 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 follows the jump when the condition tested is false.

TABLE 3-12: SPECIFIC OPTIMIZATION OPTIONS (CONTINUED)

Option	Definition
`-fcse-skip-blocks`	This is similar to `-fcse-follow-jumps`, but causes CSE to follow jumps which conditionally skip over blocks. When CSE encounters a simple `if` statement with no `else` clause, `-fcse-skip-blocks` causes CSE to follow the jump around the body of the `if`.
`-fexpensive-optimizations`	Perform a number of minor optimizations that are relatively expensive.
`-ffunction-sections` `-fdata-sections`	Place each function or data item into its own section in the output file. The name of the function or the name of the data item determines the section's name in the output file. Only use these options when there are significant benefits for doing so. When you specify these options, the assembler and linker may create larger object and executable files and is also slower.
`-fgcse`	Perform a global common subexpression elimination pass. This pass also performs global constant and copy propagation.
`-fgcse-lm`	When `-fgcse-lm` is enabled, global common subexpression elimination attempts to move loads which are only killed by stores into themselves. This allows a loop containing a load/store sequence to change 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 attempts to move stores out of loops. When used in conjunction with `-fgcse-lm`, loops containing a load/store sequence can change 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 machine specific peephole optimizations. Peephole optimizations occur at various points during the compilation. `-fno-peephole` disables peephole optimization on machine instructions, while `-fno-peephole2` disables high level peephole optimizations. To disable peephole entirely, use both options.
`-foptimize-register-move` `-fregmove`	Attempt to reassign register numbers in move instructions and as operands of other simple instructions in order to maximize the amount of register tying. `-fregmove` and `-foptimize-register-moves` are the same optimization.
`-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 most benefits processors with lots of registers. It can, however, make debugging impossible, since variables 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.

TABLE 3-12: SPECIFIC OPTIMIZATION OPTIONS (CONTINUED)

Option	Definition
`-fschedule-insns`	Attempt to reorder instructions to eliminate instruction 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.

`-fstrict-aliasing`	Allows the compiler to assume the strictest aliasing rules applicable to the language being compiled. For C, this activates optimizations based on the type of expressions. In particular, an object of one type is assumed never to reside at the same address as an object of a different type, unless the types are almost the same. For example, an `unsigned int` can alias an `int`, but not a `void*` or a `double`. A character type may alias any other type.
	Pay special attention to code like this:
	`union a_union {`
	`  int i;`
	`  double d;`
	`};`
	
	`int f() {`
	`  union a_union t;`
	`  t.d = 3.0;`
	`  return t.i;`
	`}`
	The practice of reading from a different union member than the one most recently written to (called “type-punning”) is common. Even with `-fstrict-aliasing`, type-punning is allowed, provided the memory is accessed through the union type. So, the code above works as expected. However, this code might not:
	`int f() {`
	`  a_union t;`
	`  int* ip;`
	`  t.d = 3.0;`
	`  ip = &t.i;`
	`  return *ip;`
	`}`

`-fthread-jumps`	Perform optimizations where a check is made to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.
`-funroll-loops`	Perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or run time. `-funroll-loops` implies both `-fstrength-reduce` and `-frerun-cse-after-loop`.
`-funroll-all-loops`	Perform the optimization of loop unrolling. This is done for all loops and usually makes programs run more slowly. `-funroll-all-loops` implies `-fstrength-reduce`, as well as `-frerun-cse-after-loop`.
`-fuse-caller-save`	Allows the compiler to use the caller-save register model. When combined with interprocedural optimizations, the compiler can generate more efficient code.

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

TABLE 3-13: MACHINE-INDEPENDENT OPTIMIZATION OPTIONS

Option	Definition
-fforce-mem	Force memory operands to be copied into registers before doing arithmetic on them. This produces better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register load. The -O2 option turns on this option.
-finline-functions	Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated, and the function is declared static, then the function is normally not output as assembler code in its own right.
-finline-limit=n	By default, the compiler limits the size of functions that can be inlined. This flag allows the control of this limit for functions that are explicitly marked as inline (i.e., marked with the inline keyword). *n* is the size of functions that can be inlined in number of pseudo instructions (not counting parameter handling). The default value of n is 10000. Increasing this value can result in more inlined code at the cost of compilation time and memory consumption. Decreasing usually makes the compilation faster and less code is inlined (which presumably means slower programs). This option is particularly useful for programs that use inlining. **Note:** Pseudo instruction represents, in this particular context, an abstract measurement of function's size. In no way does it represent a count of assembly instructions and as such, its exact meaning might change from one release of the compiler to an another.
-fkeep-inline-functions	Even if all calls to a given function are integrated, and the function is declared static, output a separate run time callable version of the function. This switch does not affect extern inline functions.
-fkeep-static-consts	Emit variables are declared static const when optimization isn't turned on, even if the variables are not referenced. The compiler enables this option by default. If you want to force the compiler to check if the variable was referenced, regardless of whether or not optimization is turned on, use the -fno-keep-static-consts option.
-fno-function-cse	Do not put function addresses in registers. Make each instruction that calls a constant function contain the function's address explicitly. This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

TABLE 3-13: MACHINE-INDEPENDENT OPTIMIZATION OPTIONS

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

3.9.8 Options for Controlling the Preprocessor

The following options control the compiler preprocessor.

TABLE 3-14: PREPROCESSOR OPTIONS

Option	Definition
-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 *macro* with string 1 as its definition.
-Dmacro=defn	Define macro *macro* as *defn*. All instances of -D on the command line are processed before any -U options.
-dM	Tell the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing. Used with the -E option.
-dN	Like -dD except that the macro arguments and contents are omitted. Only #define name is included in the output.
-fno-show-column	Do not print column numbers in diagnostics. This may be necessary if diagnostics are being scanned by a program that does not understand the column numbers, such as DejaGnu.
-H	Print the name of each header file used, in addition to other normal activities.

TABLE 3-14: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
-I-	Any directories you specify with -I options before the -I- options are searched only for the case of #include "file". They are not searched for #include <*file*>. If additional directories are specified with -I options after the -I-, these directories are searched for all #include directives. (Ordinarily all -I directories are used this way.) In addition, the -I- option inhibits the use of the current directory (where the current input file came from) as the first search directory for #include "file". There is no way to override this effect of -I-. With -I. you can specify searching the directory that was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory. -I- does not inhibit the use of the standard system directories for header files. Thus, -I- and -nostdinc are independent. **NOTE:** Do not specify an MPLAB XC32 system include directory (e.g. /pic32mx/include/) in your project properties. The xc32-gcc and xc32-g++ compilation drivers automatically select the default C libc or the C++ libc and their respective include-file directory for you. Manually adding a system include file path may disrupt this mechanism and cause the incorrect libc include files to be compiled into your project, causing a conflict between the include files and the library. Note that adding a system include path to your project properties has never been a recommended practice.
-Idir	Add the directory *dir* to the head of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one -I option, the directories are scanned in left-to-right order. The standard system directories come after.
-idirafter dir	Add the directory dir to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that -I adds to).
-imacros file	Process file as input, discarding the resulting output, before processing the regular input file. Because the output generated from the file is discarded, the only effect of -imacros *file* is to make the macros defined in file available for use in the main input. Any -D and -U options on the command line are always processed before -imacros *file*, regardless of the order in which they are written. All the -include and -imacros options are processed in the order in which they are written.
-include file	Process file as input before processing the regular input file. In effect, the contents of file are compiled first. Any –D and -U options on the command line are always processed before -include *file*, regardless of the order in which they are written. All the -include and -imacros options are processed in the order in which they are written.
-M	Tell the preprocessor to output a rule suitable for make describing the dependencies of each object file. For each source file, the preprocessor outputs one make-rule whose target is the object file name for that source file and whose dependencies are all the #include header files it uses. This rule may be a single line or may be continued with \-newline if it is long. The list of rules is printed on standard output instead of the preprocessed C program. -M implies -E (see **Section 3.9.2 "Options for Controlling the Kind of Output"**).

TABLE 3-14: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
`-MD`	Like `-M` but the dependency information is written to a file and compilation continues. The file containing the dependency information is given the same name as the source file with a `.d` extension.
`-MF file`	When used with `-M` or `-MM`, specifies a file in which to write the dependencies. If no `-MF` switch is given, the preprocessor sends the rules to the same place it would have sent preprocessed output. When used with the driver options, `-MD` or `-MMD`, `-MF,` overrides the default dependency output file.
`-MG`	Treat missing header files as generated files and assume they live in the same directory as the source file. If `-MG` is specified, then either `-M` or `-MM` must also be specified. `-MG` is not supported with `-MD` or `-MMD`.
`-MM`	Like `−M` but the output mentions only the user header files included with `#include "file"`. System header files included with `#include <file>` are omitted.
`-MMD`	Like `-MD` except mention only user header files, not system header files.

`-MP`	This option instructs CPP to add a phony target for each dependency other than the main file, causing each to depend on nothing. These dummy rules work around errors `make` gives if you remove header files without updating the make-file to match.
	This is typical output:
	`test.o: test.c test.h`
	`test.h:`

`-MQ`	Same as `-MT`, but it quotes any characters which are special to `make`. `-MQ '$(objpfx)foo.o'` gives `$$(objpfx)foo.o: foo.c` The default target is automatically quoted, as if it were given with `-MQ`.
`-MT target`	Change the target of the rule emitted by dependency generation. By default, CPP takes the name of the main input file, including any path, deletes any file suffix such as `.c`, and appends the platform's usual object suffix. The result is the target. An `-MT` option sets the target to be exactly the string you specify. If you want multiple targets, you can specify them as a single argument to `-MT`, or use multiple `-MT` options. For example: `-MT '$(objpfx)foo.o'` might give `$(objpfx)foo.o: foo.c`
`-nostdinc`	Do not search the standard system directories for header files. Only the directories you have specified with `-I` options (and the current directory, if appropriate) are searched. (See **Section 3.9.11 "Options for Directory Search"**) for information on `-I`. By using both `-nostdinc` and `-I-`, the include-file search path can be limited to only those directories explicitly specified.
`-P`	Tell the preprocessor not to generate `#line` directives. Used with the `-E` option (see **Section 3.9.2 "Options for Controlling the Kind of Output"**).
`-trigraphs`	Support ANSI C trigraphs. The `-ansi` option also has this effect.
`-Umacro`	Undefine macro *`macro`*. `-U` options are evaluated after all `-D` options, but before any `-include` and `-imacros` options.
`-undef`	Do not predefine any nonstandard macros (including architecture flags).

3.9.9 Options for Assembling

The following options control assembler operations.

TABLE 3-15: ASSEMBLY OPTIONS

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

3.9.10 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 3-16: LINKING OPTIONS

Option	Definition
`-fill=<options`	A memory-fill option to be passed on to the linker.
`-Ldir`	Add directory *dir* to the list of directories to be searched for libraries specified by the command line option `-l`.

`-llibrary`	Search the library named *library* when linking.
	The linker searches a standard list of directories for the library, which is actually a file named `lib`*library*`.a`. The linker then uses this file as if it had been specified precisely by name.
	It makes a difference where in the command you write this option. The linker processes libraries and object files in the order they are specified. Thus, `foo.o -lz bar.o` searches library `z` after file `foo.o` but before `bar.o`. If `bar.o` refers to functions in `libz.a`, those functions may not be loaded.
	The directories searched include several standard system directories, plus any that you specify with `-L`.
	Normally the files found this way are library files (archive files whose members are object files). The linker handles an archive file by scanning through it for members which define symbols that have been referenced but not defined yet. But if the file found is an ordinary object file, it is linked in the usual fashion. The only difference between using an `-l` option (e.g., `-lmylib`) and specifying a file name (e.g., `libmylib.a`) is that `-l` searches several directories, as specified.
	By default the linker is directed to search:
	`<install-path>\lib`
	for libraries specified with the `-l` option. For a compiler installed into the default location, this would be:
	`Program Files\Microchip\mplab32\<version>\lib`
	This behavior can be overridden using the environment variables.
	See also the `INPUT` and `OPTIONAL` linker script directives.

`-nodefaultlibs`	Do not use the standard system libraries when linking. Only the libraries you specify are passed to the linker. The compiler may generate calls to `memcmp`, `memset` and `memcpy`. These entries are usually resolved by entries in the standard compiler libraries. These entry points should be supplied through some other mechanism when this option is specified.
`-nostdlib`	Do not use the standard system start-up files or libraries when linking. No start-up files and only the libraries you specify are passed to the linker. The compiler may generate calls to `memcmp`, `memset` and `memcpy`. These entries are usually resolved by entries in standard compiler libraries. These entry points should be supplied through some other mechanism when this option is specified.

TABLE 3-16: LINKING OPTIONS (CONTINUED)

Option	Definition
`-s`	Remove all symbol table and relocation information from the executable.
`-u symbol`	Pretend *symbol* is undefined to force linking of library modules to define the symbol. It is legitimate to use `-u` multiple times with different symbols to force loading of additional library modules.
`-Wl,option`	Pass *option* as an option to the linker. If *option* contains commas, it is split into multiple options at the commas.
`-Xlinker option`	Pass *option* as an option to the linker. You can use this to supply system-specific linker options that the compiler does not know how to recognize.

3.9.11 Options for Directory Search

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

TABLE 3-17: DIRECTORY SEARCH OPTIONS

Option	Definition
`-Bprefix`	This option specifies where to find the executables, libraries, include files and data files of the compiler itself. The compiler driver program runs one or more of the sub-programs `xc32-cpp`, `xc32-as` and `xc32-ld`. It tries *prefix* as a prefix for each program it tries to run. For each sub-program to be run, the compiler driver first tries the `-B` prefix, if any. Lastly, the driver searches the current `PATH` environment variable for the subprogram. `-B` prefixes that effectively specify directory names also apply to libraries in the linker, because the compiler translates these options into `-L` options for the linker. They also apply to include files in the preprocessor, because the compiler translates these options into `-isystem` options for the preprocessor. In this case, the compiler appends `include` to the prefix.
`-specs=file`	Process file after the compiler reads in the standard `specs` file, in order to override the defaults that the `xc32-gcc` driver program uses when determining what switches to pass to `xc32-as`, `xc32-ld`, etc. More than one `-specs=`*`file`* can be specified on the command line, and they are processed in order, from left to right.

3.9.12 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 3-18: CODE GENERATION CONVENTION OPTIONS

Option	Definition

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

`-fcall-saved-reg`	Treat the register named *reg* as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way saves and restores the register *reg* if they use it. It is an error to use this flag with the Frame Pointer or Stack Pointer. Use of this flag for other registers that have fixed pervasive roles in the machine's execution model produces disastrous results. A different sort of disaster results from the use of this flag for a register in which function values are returned. This flag should be used consistently through all modules.
`-fcall-used-reg`	Treat the register named *reg* as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way do not save and restore the register *reg*. It is an error to use this flag with the Frame Pointer or Stack Pointer. Use of this flag for other registers that have fixed pervasive roles in the machine's execution model produces disastrous results. This flag should be used consistently through all modules.
`-ffixed-reg`	Treat the register named *reg* as a fixed register. Generated code should never refer to it (except perhaps as a Stack Pointer, Frame Pointer or in some other fixed role). *reg* must be the name of a register (e.g., `-ffixed-$0`).

TABLE 3-18: CODE GENERATION CONVENTION OPTIONS (CONTINUED)

Option	Definition

`-finstrument-functions`	Generate instrumentation calls for entry and exit to functions. Just after function entry and just before function exit, the following profiling functions are called with the address of the current function and its call site.
	`void __cyg_profile_func_enter (void *this_fn, void *call_site);`
	`void __cyg_profile_func_exit (void *this_fn, void *call_site);`
	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.
	Function instrumentation requires the use of a Frame Pointer. Some optimization levels disable the use of the Frame Pointer. Using `-fno-omit-frame-pointer` prevents this.
	This instrumentation is also done for functions expanded inline in other functions. The profiling calls indicates 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 `extern inline` in your C code, an addressable version of such functions must be provided.
	A function may be given the attribute `no_instrument_function`, in which case this instrumentation is not done.

`-fno-ident`	Ignore the `#ident` directive.
`-fpack-struct`	Pack all structure members together without holes. Usually you would not want to use this option, since it makes the code sub-optimal, and the offsets of structure members won't agree with system libraries.
`-fpcc-struct-return`	Return short `struct` and `union` values in memory like longer ones, rather than in registers. This convention is less efficient, but it has the advantage of allowing capability between 32-bit compiled files and files compiled with other compilers. Short structures and unions are those whose size and alignment match that of an integer type.
`-fno-short-double`	By default, the compiler uses a `double` type equivalent to `float`. This option makes `double` equivalent to `long double`. Mixing this option across modules can have unexpected results if modules share double data either directly through argument passage or indirectly through shared buffer space. Libraries provided with the product function with either switch setting.
`-fshort-enums`	Allocate to an `enum` type only as many bytes as it needs for the declared range of possible values. Specifically, the `enum` type is equivalent to the smallest integer type that 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.

Chapter 4. Device-Related Features

4.1 INTRODUCTION

The MPLAB XC32 C Compiler supports a number of special features and extensions to the C/C++ language which are designed to ease the task of producing ROM-based applications. This chapter documents the special language features which are specific to these devices.

- Device Support
- Device Header Files
- Stack
- Using SFRs From C Code

4.2 DEVICE SUPPORT

MPLAB XC32 C Compiler aims to support all PIC32 devices. However, new devices in these families are frequently released. Check the readme document for a full list of all available devices.

4.3 DEVICE HEADER FILES

There is one header file that is recommended be included into each source file you write. The file is `<xc.h>` and is a generic file that will include other device-specific header files when you build your project.

Inclusion of this file will allow access to SFRs via special variables, as well as `#define`s which allow the use of conventional register names from within assembly language files.

4.3.1 CP0 Register Definitions Header File

The CP0 register definitions header file (`cp0defs.h`) is a file that contains definitions for the CP0 registers and their fields. In addition, it contains macros for accessing the CP0 registers.

The CP0 register definitions header file is located in the `pic32mx/include` directory of your compiler installation directory. The CP0 register definitions header file is automatically included when you include the generic device header file, `xc.h`.

The CP0 register definitions header file was designed to work with either Assembly or C/C++ files. The CP0 register definitions header file is dependent on macros defined within the processor generic header file).

4.4 STACK

The PIC32 devices use what is referred to in this user's guide as a "software stack". This is the typical stack arrangement employed by most computers and is ordinary data memory accessed by a push-and-pop type instruction and a stack pointer register. The term "hardware stack" is used to describe the stack employed by Microchip 8-bit devices, which is only used for storing function return addresses.

The PIC32 devices use a dedicated stack pointer register `sp` (register 29) for use as a software Stack Pointer. All processor stack operations, including function calls, interrupts and exceptions, use the software stack. It points to the next free location on the stack. The stack grows downward, towards lower memory addresses.

By default, the size of the stack is 1024 bytes. The size of the stack may be changed by specifying the size on the linker command line using the `--defsym_min_stack_size` linker command line option. An example of allocating a stack of 2048 bytes using the command line is:

```
xc32-gcc foo.c -Wl,--defsym,_min_stack_size=2048
```

The run-time stack grows downward from higher addresses to lower addresses. Two working registers are used to manage the stack:

- Register 29 (`sp`) – This is the Stack Pointer. It points to the next free location on the stack.
- Register 30 (`fp`) – This is the Frame Pointer. It points to the current function's frame.

No stack overflow detection is supplied.

The C/C++ run-time start-up module initializes the stack pointer during the start-up and initialization sequence, see **Section 12.3.2 "Initialize Stack Pointer and Heap"**.

4.4.1 Configuration Bit Access

The PIC32 devices have several locations which contain the Configuration bits or fuses. These bits specify fundamental device operation, such as the oscillator mode, watchdog timer, programming mode and code protection. Failure to correctly set these bits may result in code failure, or a non-running device.

The `#pragma config` directive specifies the processor-specific configuration settings (i.e., Configuration bits) to be used by the application. Refer to the "PIC32MX Configuration Settings" online help (found under *MPLAB IDE>Help>Topics>Language Tools)* for more information. (If using the compiler from the command line, this help file is located at the default location at: `Program Files/Microchip/<install-dir>/doc/hlpPIC32MXConfigSet.chm`.)

Configuration settings may be specified with multiple `#pragma config` directives. The compiler verifies that the configuration settings specified are valid for the processor for which it is compiling. If a given setting in the Configuration word has not been specified in any `#pragma config` directive, the bits associated with that setting default to the unprogrammed value. Configuration settings should be specified in only a single translation unit (a C/C++ file with all of its include files after preprocessing).

For each Configuration word for which a setting is specified with the `#pragma config` directive, the compiler generates a read-only data section named `.config_`*`address`*, where *`address`* is the hexadecimal representation of the address of the Configuration word. For example, if a configuration setting was specified for the Configuration word located at address `0xBFC02FFC`, a read-only data section named `.config_BFC02FFC` would be created.

- Syntax
- Example

4.4.1.1 SYNTAX

The following shows the meta syntax notation for the different forms the pragma may take.

pragma-config-directive:
    `# pragma config` *setting-list*
setting-list:
    setting
  | *setting-list*, *setting*
setting:
    setting-name = *value-name*

The *setting-name* and *value-name* are device specific and can be determined by utilizing the *PIC32MX Configuration Settings* document.

All `#pragma config` directives should be placed outside of a function definition as they do not define executable code.

4.4.1.2 EXAMPLE

The following example shows how the `#pragma config` directive might be utilized. The example does the following:

- Enables the Watchdog Timer
- Sets the Watchdog Postscaler to 1:128
- Selects the HS Oscillator for the Primary Oscillator

```
#pragma config FWDTEN = ON, WDTPS = PS128
#pragma config POSCMOD = HS
...
int main (void)
{
...
}
```

4.5 USING SFRS FROM C CODE

The Special Function Registers (SFRs) are registers which control aspects of the MCU operation or that of peripheral modules on the device. These registers are memory mapped, which means that they appear at specific addresses in the device memory map. With some registers, the bits within the register control independent features.

Memory-mapped SFRs are accessed by special C variables that are placed at the addresses of the registers and use special attributes. These variables can be accessed like any ordinary C variable so that no special syntax is required to access SFRs.

The SFR variables are predefined in header files and will be accessible once the `<xc.h>` header file (see **Section 4.3 "Device Header Files"**) has been included into your source code. Structures are also defined by these header files to allow access to bits within the SFR.

The names given to the C variables, which map over the registers and bit variables, or bit fields, within the registers are based on the names specified in the device data sheet. The names of the structures that hold the bit fields will typically be those of the corresponding register followed by `bits`. For example, the following shows code that includes the generic header file, clears PORTB as a whole and sets bit 2 of PORTB using the structure/bit field definitions.

> **Note:** The symbols `PORTB` and `PORTBbits` refer to the same register and resolve to the same address. Writing to one register will change the values held by both.

```
#include <xc.h>
int main(void)
{
    PORTB = 0x00;
    PORTBbits.RB2 = 1;
}
```

For use with assembly, the `PORTB` register is declared as: `.extern PORTB`.

To confirm the names that are relevant for the device you are using, check the device specific header file that `<xc.h>` will include for the definitions of each variable. These files will be located in the `pic32mx/include/proc` directory of the compiler and will have a name that represents the device. There is a one-to-one correlation between device and header file name that will be included by `<xc.h>`, e.g. when compiling for a PIC32MX360F512L device, the `<xc.h>` header file will include `<p32mx360f512l.h>`. Remember that you do not need to include this chip-specific file into your source code; it is automatically included by `<xc.h>`.

Some of the PIC32 SFRs have associated registers that allow the bits within the SFR to be set, cleared or toggled atomically. For example, the `PORTB` SFR has the write-only registers `PORTBSET`, `PORTBCLR` and `PORTBINV` associated with it. Writing a '`1`' to a bit location in these registers sets, clears or toggles, respectively, the corresponding bit in the `PORTB` SFR. So to set bit 1 in `PORTB`, you can use the following code:

```
PORTBSET = 0x2;
```

or alternatively, using macros provided in the device header files:

```
PORTBSET = _PORTB_RB1_MASK;
```

The same operation can also be achieved using the peripheral library functions, for example

```
mPORTBSetBits(BIT_1);
```

Always ensure that you confirm the operation of peripheral modules from the device data sheet.

4.5.1 CP0 Register Definitions

When the CP0 register definitions header file is included from an Assembly file, the CP0 registers are defined as:

```
#define _CP0_register_name $register_number, select_number
```

For example, the `IntCtl` register is defined as:

```
#define _CP0_INTCTL $12, 1
```

When the CP0 register definitions header file is included from a C file, the CP0 registers and selects are defined as:

```
#define _CP0_register_name register_number
#define _CP0_register_name_SELECT select_number
```

For example, the `IntCtl` register is defined as:

```
#define _CP0_INTCTL 12
#define _CP0_INTCTL_SELECT 1
```

4.5.2 CP0 Register Field Definitions

When the CP0 register definitions header file is included from either an Assembly or a C/C++ file, three `#defines` exist for each of the CP0 register fields.

`_CP0_register_name_field_name_POSITION` – the starting bit location

`_CP0_register_name_field_name_MASK` – the bits that are part of this field are set

`_CP0_register_name_field_name_LENGTH` – the number of bits that this field occupies

For example, the vector spacing field of the `IntCtl` register has the following defines:

```
#define _CP0_INTCTL_VS_POSITION 0x00000005
#define _CP0_INTCTL_VS_MASK     0x000003E0
#define _CP0_INTCTL_VS_LENGTH   0x00000005
```

4.5.3 CP0 Access Macros

When the CP0 register definitions header file is included from a C file, CP0 access macros are defined. Each CP0 register may have up to six different access macros defined:

`_CP0_GET_register_name ()`	Returns the value for register, `register_name`.
`_CP0_SET_register_name (val)`	Sets the register, `register_name`, to val, and returns void. Only defined for registers that contain a writable field.
`_CP0_XCH_register_name (val)`	Sets the register, `register_name`, to val, and returns the previous register value. Only defined for registers that contain a writable field.
`_CP0_BIS_register_name (set)`	Sets the register, `register_name`, to (reg |= set), and returns the previous register value. Only defined for registers that contain writable bit fields.
`_CP0_BIC_register_name (clr)`	Sets the register, `register_name`, to (reg &= ~clr), and returns the previous register value. Only defined for registers that contain writable bit fields.
`_CP0_BCS_register_name (clr, set)`	Sets the register, `register_name`, to (reg = (reg & ~clr) | set), and returns the previous register value. Only defined for registers that contain writable bit fields.

4.5.4 Address Translation Macros

System code may need to translate between virtual and physical addresses, as well as between kernel segment addresses. Macros are provided to make these translations easier and to determine the segment an address is in.

`KVA_TO_PA(v)`	Translate a kernel virtual address to a physical address.
`PA_TO_KVA0(pa)`	Translate a physical address to a KSEG0 virtual address.
`PA_TO_KVA1(pa)`	Translate a physical address to a KSEG1 virtual address.
`KVA0_TO_KVA1(v)`	Translate a KSEG0 virtual address to a KSEG1 virtual address.
`KVA1_TO_KVA0(v)`	Translate a KSEG1 virtual address to a KSEG0 virtual address.
`IS_KVA(v)`	Evaluates to 1 if the address is a kernel segment virtual address, zero otherwise.
`IS_KVA0(v)`	Evaluate to 1 if the address is a KSEG0 virtual address, zero otherwise.
`IS_KVA1(v)`	Evaluate to 1 if the address is a KSEG1 virtual address, zero otherwise.
`IS_KVA01(v)`	Evaluate to 1 if the address is either a KSEG0 or a KSEG1 virtual address, zero otherwise.

Chapter 5. ANSI C Standard Issues

This compiler conforms to the ANS X3.159-1989 Standard for programming languages. This is commonly called the C89 Standard. It is referred to as the ANSI C Standard in this manual. Some features from the later standard C99 are also supported.

- Divergence from the ANSI C Standard
- Extensions to the ANSI C Standard
- Implementation-defined behavior

5.1 DIVERGENCE FROM THE ANSI C STANDARD

There are no divergences from the ANSI C standard.

5.2 EXTENSIONS TO THE ANSI C STANDARD

C/C++ code for the MPLAB XC32 C Compiler differs from the ANSI C standard in these areas: keywords, statements and expressions.

5.2.1 Keyword Differences

The new keywords are part of the base GCC implementation and the discussions in the referenced sections are based on the standard GCC documentation, tailored for the specific syntax and semantics of the 32-bit compiler port of GCC.

- Specifying Attributes of Variables – **Section 6.12 "Variable Attributes"**
- Specifying Attributes of Functions – **Section 10.2 "Function Attributes and Specifiers"**
- Inline Functions – **Section 10.9 "Inline Functions"**
- Variables in Specified Registers – **Section 6.12 "Variable Attributes"**
- Complex Numbers – **Section 6.8 "Complex Data Types"**
- Referring to a Type with `typeof` – **Section 6.10 "Standard Type Qualifiers"**

5.2.2 Statement Differences

The statement differences are part of the base GCC implementation, and the discussions in the referenced sections are based on the standard GCC documentation, tailored for the specific syntax and semantics of the 32-bit compiler port of GCC.

- Labels as Values – **Section 8.4 "Labels as Values"**
- Conditionals with Omitted Operands – **Section 8.5 "Conditional Operator Operands"**
- Case Ranges – **Section 8.6 "Case Ranges"**

5.2.3 Expression Differences

Expression differences are:

- Binary constants – **Section 6.9 "Constant Types and Formats"**.

5.3 IMPLEMENTATION-DEFINED BEHAVIOR

Certain features of the ANSI C standard have implementation-defined behavior. This means that the exact behavior of some C code can vary from compiler to compiler. The exact behavior of the MPLAB XC32 C Compiler is detailed throughout this documentation, and is fully summarized in **Appendix A. "Implementation-Defined Behavior"**.

Chapter 6. Supported Data Types and Variables

6.1 INTRODUCTION

The MPLAB XC32 C Compiler supports a variety of data types and attributes. These data types and variables are discussed here. For information on where variables are stored in memory, see **Chapter 7. "Memory Allocation and Access"**.

- Data Representation
- Integer Data Types
- Floating-Point Data Types
- Structures and Unions
- Pointer Types
- Complex Data Types
- Constant Types and Formats
- Standard Type Qualifiers
- Compiler-Specific Qualifiers
- Variable Attributes

6.2 IDENTIFIERS

A C/C++ variable identifier (the following is also true for function identifiers) is a sequence of letters and digits, where the underscore character "_" counts as a letter. Identifiers cannot start with a digit. Although they may start with an underscore, such identifiers are reserved for the compiler's use and should not be defined by your programs. Such is not the case for assembly domain identifiers, which often begin with an underscore

Identifiers are case sensitive, so `main` is different than `Main`.

All characters are significant in an identifier, although identifiers longer than 31 characters in length are less portable.

6.3 DATA REPRESENTATION

The compiler stores multibyte values in little-endian format. That is, the Least Significant Byte is stored at the lowest address.

For example, the 32-bit value `0x12345678` would be stored at address `0x100` as:

Address	0x100	0x101	0x102	0x103
Data	0x78	0x56	0x34	0x12

6.4 INTEGER DATA TYPES

Integer values in the compiler are represented in 2's complement and vary in size from 8 to 64 bits. These values are available in compiled code via `limits.h`.

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`, `long`, `signed long`	32	-2^{31}	2^{31}-1
`unsigned int`, `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

6.4.1 Signed and Unsigned Character Types

By default, values of type plain `char` are signed values. This behavior is implementation-defined by the C standard, and some environments[1] define a plain C/C++ `char` value to be unsigned. The command line option `-funsigned-char` can be used to set the default type to unsigned for a given translation unit.

6.4.2 `limits.h`

The `limits.h` header file defines the ranges of values which can be represented by the integer types.

Macro name	Value	Description
`CHAR_BIT`	8	The size, in bits, of the smallest non-bit field object.
`SCHAR_MIN`	-128	The minimum value possible for an object of type `signed char`.
`SCHAR_MAX`	127	The maximum value possible for an object of type `signed char`.
`UCHAR_MAX`	255	The maximum value possible for an object of type `unsigned char`.
`CHAR_MIN`	-128 (or 0, see **Section 6.4.1 "Signed and Unsigned Character Types"**)	The minimum value possible for an object of type `char`.
`CHAR_MAX`	127 (or 255, see **Section 6.4.1 "Signed and Unsigned Character Types"**)	The maximum value possible for an object of type `char`.
`MB_LEN_MAX`	16	The maximum length of multibyte character in any locale.
`SHRT_MIN`	-32768	The minimum value possible for an object of type `short int`.
`SHRT_MAX`	32767	The maximum value possible for an object of type `short int`.
`USHRT_MAX`	65535	The maximum value possible for an object of type `unsigned short int`.

[1] Notably, PowerPC and ARM.

Macro name	Value	Description
INT_MIN	-2^{31}	The minimum value possible for an object of type int.
INT_MAX	$2^{31}-1$	The maximum value possible for an object of type int.
UINT_MAX	$2^{32}-1$	The maximum value possible for an object of type unsigned int.
LONG_MIN	-2^{31}	The minimum value possible for an object of type long.
LONG_MAX	$2^{31}-1$	The maximum value possible for an object of type long.
ULONG_MAX	$2^{32}-1$	The maximum value possible for an object of type unsigned long.
LLONG_MIN	-2^{63}	The minimum value possible for an object of type long long.
LLONG_MAX	$2^{63}-1$	The maximum value possible for an object of type long long.
ULLONG_MAX	$2^{64}-1$	The maximum value possible for an object of type unsigned long long.

6.5 FLOATING-POINT DATA TYPES

The compiler uses the IEEE-754 floating-point format. Detail regarding the implementation limits is available to a translation unit in `float.h`.

Type	Bits
`float`	32
`double`	32
`long double`	64

Variables may be declared using the `float`, `double` and `long double` keywords, respectively, to hold values of these types. Floating-point types are always signed and the `unsigned` keyword is illegal when specifying a floating-point type. All floating-point values are represented in little endian format with the Least Significant Byte at the lower address.

This format is described in Table 6-1, where:

- Sign is the sign bit which indicates if the number is positive or negative
- For 32-bit floating point values, the exponent is 8 bits which is stored as excess 127 (i.e. an exponent of 0 is stored as 127).
- For 64-bit floating point values, the exponent is 11 bits which is stored as excess 1023 (i.e. an exponent of 0 is stored as 1023).
- Mantissa is the mantissa, which is to the right of the radix point. There is an implied bit to the left of the radix point which is always 1 except for a zero value, where the implied bit is zero. A zero value is indicated by a zero exponent.

The value of this number for 32-bit floating point values is:

$(-1)^{sign} \times 2^{(exponent-127)} \times 1.\text{ mantissa}$

and for 64-bit values

$(-1)^{sign} \times 2^{(exponent-1023)} \times 1.\text{ mantissa}.$

Here is an example of the IEEE 754 32-bit format shown in Table 6-1. Note that the Most Significant bit of the mantissa column (i.e. the bit to the left of the radix point) is the implied bit, which is assumed to be 1 unless the exponent is zero (in which case the float is zero).

TABLE 6-1: FLOATING-POINT FORMAT EXAMPLE IEEE 754

Format	Number	Biased exponent	1.mantissa	Decimal
32-bit	7DA6B69Bh	11111011b	1.0100110101101101 0011011b	2.77000e+37
		(251)	(1.302447676659)	—

The example in Table 6-1 can be calculated manually as follows.

The sign bit is zero; the biased exponent is 251, so the exponent is 251-127=124. Take the binary number to the right of the decimal point in the mantissa. Convert this to decimal and divide it by 2^{23} where 23 is the number of bits taken up by the mantissa, to give 0.302447676659. Add 1 to this fraction. The floating-point number is then given by:

$-1^{0} \times 2^{124} \times 1.302447676659$

which becomes:

$1 \times 2.126764793256e+37 \times 1.302447676659$

which is approximately equal to:

2.77000*e*+37

Binary floating-point values are sometimes misunderstood. It is important to remember that not every floating-point value can be represented by a finite sized floating-point number. The size of the exponent in the number dictates the range of values that the number can hold, and the size of the mantissa relates to the spacing of each value that can be represented exactly. Thus the 64-bit floating-point format allows for values with a larger range of values and that can be more accurately represented.

So, for example, if you are using a 32-bit wide floating-point type, it can exactly store the value 95000.0. However, the next highest number it can represent is (approximately) 95000.00781 and it is impossible to represent any value in between these two in such a type as it will be rounded. This implies that C/C++ code which compares floating-point type may not behave as expected. For example:

```
volatile float myFloat;
myFloat = 95000.006;
if(myFloat == 95000.007)      // value will be rounded
   LATA++;                   // this line will be executed!
```

in which the result of the `if()` expression will be true, even though it appears the two values being compared are different.

The characteristics of the floating-point formats are summarized in Table 6-2. The symbols in this table are preprocessor macros which are available after including `<float.h>` in your source code. Two sets of macros are available for `float` and `double` types, where *`XXX`* represents `FLT` and `DBL`, respectively. So, for example, `FLT_MAX` represents the maximum floating-point value of the `float` type. `DBL_MAX` represents the same values for the `double` type. As the size and format of floating-point data types are not fully specified by the ANSI Standard, these macros allow for more portable code which can check the limits of the range of values held by the type on this implementation.

TABLE 6-2: RANGES OF FLOATING-POINT TYPE VALUES

Symbol	Meaning	32-bit Value	64-bit Value
*XXX*_RADIX	Radix of exponent representation	2	2
*XXX*_ROUNDS	Rounding mode for addition	1	
*XXX*_MIN_EXP	Min. *n* such that FLT_RADIX^{n-1} is a normalized float value	-125	-1021
*XXX*_MIN_10_EXP	Min. *n* such that 10^{n} is a normalized float value	-37	-307
*XXX*_MAX_EXP	Max. *n* such that FLT_RADIX^{n-1} is a normalized float value	128	1024
*XXX*_MAX_10_EXP	Max. *n* such that 10^{n} is a normalized float value	38	308
*XXX*_MANT_DIG	Number of FLT_RADIX mantissa digits	24	53
*XXX*_EPSILON	The smallest number which added to 1.0 does not yield 1.0	1.1920929e-07	2.2204460492503131e-16

6.6 STRUCTURES AND UNIONS

MPLAB XC32 C Compiler supports `struct` and `union` types. Structures and unions only differ in the memory offset applied to each member.

These types will be at least 1 byte wide. Bit fields are fully supported.

No padding of structure members is added.

Structures and unions may be passed freely as function arguments and function return values. Pointers to structures and unions are fully supported.

6.6.1 Structure and Union Qualifiers

The MPLAB XC32 C Compiler supports the use of type qualifiers on structures. When a qualifier is applied to a structure, all of its members will inherit this qualification. In the following example the structure is qualified `const`.

```
const struct {
        int number;
        int *ptr;
} record = { 0x55, &i };
```

In this case, the entire structure will be placed into the program memory and each member will be read-only. Remember that all members are usually initialized if a structure is `const` as they cannot be initialized at runtime.

If the members of the structure were individually qualified `const`, but the structure was not, then the structure would be positioned into RAM, but each member would be read-only. Compare the following structure with the above.

```
struct {
        const int number;
        int * const ptr;
} record = { 0x55, &i};
```

6.6.2 Bit Fields in Structures

MPLAB XC32 C Compiler fully supports bit fields in structures.

Bit fields are always allocated within 8-bit storage units, even though it is usual to use the type `unsigned int` in the definition. Storage units are aligned on a 32-bit boundary, although this can be changed using the `packed` attribute.

The first bit defined will be the Least Significant bit of the word in which it will be stored. When a bit field is declared, it is allocated within the current 8-bit unit if it will fit; otherwise, a new byte is allocated within the structure. Bit fields can never cross the boundary between 8-bit allocation units. For example, the declaration:

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

will produce a structure occupying 1 byte. If `foo` was ultimately linked at address 10H, the field `lo` will be bit 0 of address 10H; `hi` will be bit 7 of address 10H. The Least Significant bit of `dummy` will be bit 1 of address 10H and the Most Significant bit of `dummy` will be bit 6 of address 10h.

Unnamed bit fields may be declared to pad out unused space between active bits in control registers. For example, if `dummy` is never referenced, the structure above could have been declared as:

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

A structure with bit fields may be initialized by supplying a *comma*-separated list of initial values for each field. For example:

```
struct {
        unsigned        lo  : 1;
        unsigned        mid : 6;
        unsigned        hi  : 1;
} foo = {1, 8, 0};
```

Structures with unnamed bit fields may be initialized. No initial value should be supplied for the unnamed members, for example:

```
struct {
        unsigned        lo  : 1;
        unsigned            : 6;
        unsigned        hi  : 1;
} foo = {1, 0};
```

will initialize the members `lo` and `hi` correctly.

The MPLAB XC compiler supports anonymous unions. These are unions with no identifier and whose members can be accessed without referencing the enclosing union. These unions can be used when placing inside structures. For example:

```
struct {
      union {
      int x;
      double y;
   };
} aaa;

int main(void)
{
   aaa.x = 99;
   // ...}
```

Here, the union is not named and its members accessed as if they are part of the structure. Anonymous unions are not part of any C Standard and so their use limits the portability of any code.

6.7 POINTER TYPES

There are two basic pointer types supported by the MPLAB XC32 C Compiler: data pointers and function pointers. Data pointers hold the addresses of variables which can be indirectly read, and possible indirectly written, by the program. Function pointers hold the address of an executable function which can be called indirectly via the pointer.

6.7.1 Combining Type Qualifiers and Pointers

It is helpful to first review the ANSI C/C++ standard conventions for definitions of pointer types.

Pointers can be qualified like any other C/C++ object, but care must be taken when doing so as there are two quantities associated with pointers. The first is the actual pointer itself, which is treated like any ordinary C/C++ variable and has memory reserved for it. The second is the target, or targets, that the pointer references, or to which the pointer points. The general form of a pointer definition looks like the following:

*target_type_&_qualifiers * pointer's_qualifiers pointer's_name;*

Any qualifiers to the right of the `*` (i.e. next to the pointer's name) relate to the pointer variable itself. The type and any qualifiers to the left of the `*` relate to the pointer's targets. This makes sense since it is also the * operator that dereferences a pointer, which allows you to get from the pointer variable to its current target.

Here are three examples of pointer definitions using the `volatile` qualifier. The fields in the definitions have been highlighted with spacing:

```
volatile int *            vip ;
int          * volatile  ivp ;
volatile int * volatile  vivp ;
```

The first example is a pointer called `vip`. It contains the address of `int` objects that are qualified `volatile`. The pointer itself — the variable that holds the address — is *not* `volatile`; however, the objects that are accessed when the pointer is dereferenced are treated as being `volatile`. In other words, the target objects accessible via the pointer may be externally modified.

The second example is a pointer called `ivp` which also contains the address of `int` objects. In this example, the pointer itself is `volatile`, that is, the address the pointer contains may be externally modified; however, the objects that can be accessed when dereferencing the pointer are not `volatile`.

The last example is of a pointer called `vivp` which is itself qualified `volatile`, and which also holds the address of `volatile` objects.

Bear in mind that one pointer can be assigned the addresses of many objects; for example, a pointer that is a parameter to a function is assigned a new object address every time the function is called. The definition of the pointer must be valid for every target address assigned.

> **Note:** Care must be taken when describing pointers. Is a "const pointer" a pointer that points to `const` objects, or a pointer that is `const` itself? You can talk about "pointers to const" and "const pointers" to help clarify the definition, but such terms may not be universally understood.

6.7.2 Data Pointers

Pointers in the compiler are all 32 bits in size. These can hold an address which can reach all memory locations.

6.7.3 Function Pointers

The MPLAB XC compiler fully supports pointers to functions, which allows functions to be called indirectly. These are often used to call one of several function addresses stored in a user-defined C/C++ array, which acts like a lookup table.

Function pointers are always 32 bits in size and hold the address of the function to be called.

Any attempt to call a function with a function pointer containing NULL will result in an ifetch Bus Error.

6.7.3.1 SPECIAL POINTER TARGETS

Pointers and integers are not interchangeable. Assigning an integer constant to a pointer will generate a warning to this effect. For example:

```
const char * cp = 0x123;  // the compiler will flag this as bad code
```

There is no information in the integer constant, 0x123, relating to the type or size of the destination. This code is also not portable and there is a very good chance of code failure if pointers are assigned integer addresses and dereferenced, particularly for PIC® devices that have more than one memory space.

Always take the address of a C/C++ object when assigning an address to a pointer. If there is no C/C++ object defined at the destination address, then define or declare an object at this address which can be used for this purpose. Make sure the size of the object matches the range of the memory locations that can be accessed.

For example, a checksum for 1000 memory locations starting at address 0xA0001000 is to be generated. A pointer is used to read this data. You may be tempted to write code such as:

```
int * cp;
cp = 0xA0001000;  // what resides at 0xA0001000???
```

and increment the pointer over the data. A much better solution is this:

```
int * cp;
int __attribute__((address(0xA0001000))) inputData [1000];
cp = &inputData;
// cp is incremented over inputData and used to read values there
```

In this case, the compiler can determine the size of the target and the memory space. The array size and type indicates the size of the pointer target.

Take care when comparing (subtracting) pointers. For example:

```
if(cp1 == cp2)
  ; take appropriate action
```

The ANSI C standard only allows pointer comparisons when the two pointer targets are the same object. The address may extend to one element past the end of an array.

Comparisons of pointers to integer constants are even more risky, for example:

```
if(cp1 == 0xA0000100)
  ; take appropriate action
```

A NULL pointer is the one instance where a constant value can be assigned to a pointer and this is handled correctly by the compiler. A NULL pointer is numerically equal to 0 (zero), but this is a special case imposed by the ANSI C standard. Comparisons with the macro NULL are also allowed.

6.8 COMPLEX DATA TYPES

Complex data types are currently not implemented in MPLAB XC32 C Compiler.

6.9 CONSTANT TYPES AND FORMATS

A constant is used to represent a numerical value in the source code, for example 123 is a constant. Like any value, a constant must have a C/C++ type. In addition to a constant's type, the actual value can be specified in one of several formats. The format of integral constants specifies their radix. `MPLAB XC32 C` supports the ANSI standard radix specifiers as well as ones which enables binary constants to be specified in C code.

The formats used to specify the radices are given in Table 6-3. The letters used to specify binary or hexadecimal radices are case insensitive, as are the letters used to specify the hexadecimal digits.

TABLE 6-3: RADIX FORMATS

Radix	Format	Example
binary	`0b` *number* or `0B` *number*	0b10011010
octal	`0` *number*	0763
decimal	*number*	129
hexadecimal	`0x` *number* or `0X` *number*	0x2F

Any integral constant will have a type of `int`, `long int` or `long long int`, so that the type can hold the value without overflow. Constants specified in octal or hexadecimal may also be assigned a type of `unsigned int`, `unsigned long int` or `unsigned long long int` if the signed counterparts are too small to hold the value.

The default types of constants may be changed by the addition of a suffix after the digits, e.g. `23U`, where `U` is the suffix. Table 6-4 shows the possible combination of suffixes and the types that are considered when assigning a type. So, for example, if the suffix `l` is specified and the value is a decimal constant, the compiler will assign the type `long int`, if that type will hold the constant; otherwise, it will assigned `long long int`. If the constant was specified as an octal or hexadecimal constant, then unsigned types are also considered.

TABLE 6-4: SUFFIXES AND ASSIGNED TYPES

Suffix	Decimal	Octal or Hexadecimal

`u` or `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` or `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` or `U`, and `l` or `L`	`unsigned long int`
	`unsigned long long int`
		`unsigned long int`
		`unsigned long long int`

`ll` or `LL`	`long long int`	`long long int`
		`unsigned long long int`

`u` or `U`, and `ll` or `LL`	`unsigned long long int`	`unsigned long long int`

Here is an example of code that may fail because the default type assigned to a constant is not appropriate:

```
unsigned long int result;
unsigned char shifter;

int main(void)
{
    shifter = 40;
    result = 1 << shifter;
    // code that uses result
}
```

The constant 1 will be assigned an int type hence the result of the shift operation will be an int and the upper bits of the long variable, result, can never be set, regardless of how much the constant is shifted. In this case, the value 1 shifted left 40 bits will yield the result 0, not 0x10000000000.

The following uses a suffix to change the type of the constant, hence ensure the shift result has an unsigned long type.

```
    result = 1UL << shifter;
```

Floating-point constants have double type unless suffixed by f or F, in which case it is a float constant. The suffixes l or L specify a long double type.

Character constants are enclosed by single quote characters, ’, for example ’a’. A character constant has int type, although this may be optimized to a char type later in the compilation.

Multi-byte character constants are accepted by the compiler but are not supported by the standard libraries.

String constants, or string literals, are enclosed by double quote characters “, for example “hello world”. The type of string constants is const char * and the character that make up the string are stored in the program memory, as are all objects qualified const.

To comply with the ANSI C standard, the compiler does not support the extended character set in characters or character arrays. Instead, they need to be escaped using the backslash character, as in the following example:

```
const char name[] = "Bj\370rk";
printf(“%s's Resum\351”, name); \\ prints “Bjørk's Resumé”
```

Assigning a string literal to a pointer to a non-const char will generate a warning from the compiler. This code is legal, but the behavior if the pointer attempts to write to the string will fail. For example:

```
char * cp= “one”;         // “one” in ROM, produces warning
const char * ccp= “two”;  // “two”  in ROM, correct
```

Defining and initializing a non-const array (i.e. not a pointer definition) with a string, for example:

```
char ca[]= “two”;         // “two”  different to the above
```

is a special case and produces an array in data space which is initialized at start-up with the string “two” (copied from program space), whereas a string constant used in other contexts represents an unnamed const -qualified array, accessed directly in program space.

The MPLAB XC32 C Compiler will use the same storage location and label for strings that have identical character sequences. For example, in the code snippet

```
if(strncmp(scp, “hello world”, 6) == 0)
    fred = 0;
if(strcmp(scp, “hello world”) == 0)
    fred++;
```

the two identical character string greetings will share the same memory locations. The link-time optimization must be enabled to allow this optimization when the strings may be located in different modules.

Two adjacent string constants (i.e. two strings separated *only* by white space) are concatenated by the compiler. Thus:

```
const char * cp = "hello"  " world";
```

will assign the pointer with the address of the string "`hello world`".

6.10 STANDARD TYPE QUALIFIERS

Type qualifiers provide additional information regarding how an object may be used. The MPLAB XC32 C Compiler supports both ANSI C qualifiers and additional special qualifiers which are useful for embedded applications and which take advantage of the PIC MCU architecture.

6.10.1 Const Type Qualifier

The MPLAB XC32 C Compiler supports the use of the ANSI type qualifiers `const` and `volatile`.

The `const` type qualifier is used to tell the compiler that an object is read only and will not be modified. If any attempt is made to modify an object declared `const`, the compiler will issue a warning or error.

Usually a `const` object must be initialized when it is declared, as it cannot be assigned a value at any point at runtime. For example:

```
const int  version = 3;
```

will define `version` as being an `int` variable that will be placed in the program memory, will always contain the value 3, and which can never be modified by the program.

Objects qualified const are placed into the program memory unless the `-mno-embedded-data` option is used.

6.10.2 Volatile Type Qualifier

The `volatile` type qualifier is used to tell the compiler that an object cannot be guaranteed to retain its value between successive accesses. This prevents the optimizer from eliminating apparently redundant references to objects declared `volatile` because it may alter the behavior of the program to do so.

Any SFR which can be modified by hardware or which drives hardware is qualified as `volatile`, and any variables which may be modified by interrupt routines should use this qualifier as well. For example:

```
extern volatile unsigned int WDTCON __attribute__((section("sfrs")));
```

The `volatile` qualifier does not guarantee that any access will be atomic, but the compiler will try to implement this.

The code produced by the compiler to access `volatile` objects may be different than that to access ordinary variables, and typically the code will be longer and slower for `volatile` objects, so only use this qualifier if it is necessary. However failure to use this qualifier when it is required may lead to code failure.

Another use of the `volatile` keyword is to prevent variables from being removed if they are not used in the C/C++ source. If a non-`volatile` variable is never used, or used in a way that has no effect on the program's function, then it may be removed before code is generated by the compiler.

A C/C++ statement that consists only of a `volatile` variable's name will produce code that reads the variable's memory location and discards the result. For example the entire statement:

```
PORTB;
```

will produce assembly code the reads `PORTB`, but does nothing with this value. This is useful for some peripheral registers that require reading to reset the state of interrupt flags. Normally such a statement is not encoded as it has no effect.

6.11 COMPILER-SPECIFIC QUALIFIERS

There are no non-standard qualifiers implemented in MPLAB XC32 C Compiler. Attributes are used to control variables and functions.

6.12 VARIABLE ATTRIBUTES

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

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

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

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

`address (addr)`

Specify an absolute virtual address for the variable. This attribute can be used in conjunction with a section attribute. For data variables, the address is typically in the range [0xA0000000,0xA00FFFFC], as defined in the linker script as the 'kseg1_data_mem' region. This attribute can be used to start a group of variables at a specific address:

```
  int foo __attribute__((section("mysection"),address(0xA0001000)));
  int bar __attribute__((section("mysection")));
  int baz __attribute__((section("mysection")));
```

Keep in mind that the compiler performs no error checking on the specified address. The section will be located at the specified address regardless of the memory-region ranges listed in the linker script or the actual ranges on the target device. This application code is responsible for ensuring that the address is valid for the target device and application.

Also, be aware that variables attributed with an absolute address are not accessed via GP-relative addressing. This means that they may be more expensive to access than non-address attributed variables.

In addition, to make effective use of absolute sections and the new best-fit allocator, standard program-memory and data-memory sections should not be mapped in the linker script. The built-in linker script does not map most standard sections such as the `.text`, `.data`, `.bss`, or `.ramfunc` section. By not mapping these sections in the linker script, we allow these sections to be allocated using the best-fit allocator rather than the sequential allocator. Sections that are unmapped in the linker script can flow around absolute sections whereas sections that are linker-script mapped are grouped together and allocated sequentially, potentially causing conflicts with absolute sections.

Finally, note that "small" data and bss (.sdata, .sbss, etc.) sections are still mapped in the built-in default linker script. This is because "small" data variables must be grouped together so that they are within range of the more efficient GP-relative addressing mode. *To avoid conflict with these linker-script mapped sections, choose high addresses for your absolute-address variables.*

Note: In almost all cases, you will want to combine the address attribute with the space attribute to indicate code or data.

aligned (n)

The attributed variable will be aligned on the next n byte boundary.

The aligned attribute can also be used on a structure member. Such a member will be aligned to the indicated boundary within the structure.

If the alignment value n is omitted, the alignment of the variable is set 8 (the largest alignment value for a basic data type).

Note that the aligned attribute is used to increase the alignment of a variable, not reduce it. To decrease the alignment value of a variable, use the packed attribute.

cleanup (function)

Indicate a function to call when the attributed automatic function scope variable goes out of scope.

The indicated function should take a single parameter, a pointer to a type compatible with the attributed variable, and have void return type.

deprecated
deprecated (*msg*)

When a variable specified as deprecated is used, a warning is generated. The optional msg argument, which must be a string, will be printed in the warning, if present.

packed

The attributed variable or structure member will have the smallest possible alignment. That is, no alignment padding storage will be allocated for the declaration. Used in combination with the aligned attribute, packed can be used to set an arbitrary alignment restriction greater or lesser than the default alignment for the type of the variable or structure member.

section ("section-name")

Place the variable into the named section.

For example,

```
unsigned int dan __attribute__ ((section (".quixote")))
```

Variable dan will be placed in section .quixote.

The -fdata-sections command line option has no effect on variables defined with a section attribute unless unique_section is also specified.

space(memory-space)

The space attribute can be used to direct the compiler to allocate a variable in a specific memory space. Valid memory spaces are prog for program memory and data for data memory. The data space is the default for variables. This attribute also controls how initialized data is handled. The linker generates an entry in the data initialization template for the default space(data), but it does not generate an entry for space(prog) since the variable is located in non-volatile memory.

For example,

```
const unsigned int __attribute__((space(prog)) jack = 10;
signed int __attribute__((space(data))) oz = 5;
```

unique_section

Place the variable in a uniquely named section, just as if -fdata-sections had been specified. If the variable also has a section attribute, use that section name as the prefix for generating the unique section name.

For example,

```
int tin __attribute__ ((section (“.ofcatfood”), unique_section)
```

Variable tin will be placed in section .ofcatfood.

unused

Indicate to the compiler that the variable may not be used. The compiler will not issue a warning for this variable if it is not used.

weak

The weak attribute causes the declaration to be emitted as a weak symbol. A weak symbol indicates that if a global version of the same symbol is available, that version should be used instead.

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.

NOTES:

Chapter 7. Memory Allocation and Access

7.1 INTRODUCTION

There are two broad groups of RAM-based variables: auto/parameter variables, which are allocated to some form of stack, and global/static variables, which are positioned freely throughout the data memory space. The memory allocation of these two groups is discussed separately in the following sections.

- Address Spaces
- Variables in Data Memory
- Auto Variable Allocation and Access
- Variables in Program Memory
- Variables in Registers
- Dynamic Memory Allocation
- Memory Models

7.2 ADDRESS SPACES

Unlike the 8- and 16-bit PIC devices, the PIC32 has a unified programming model. PIC32 devices provide a single 32-bit wide address space for all code, data, peripherals and Configuration bits.

Memory regions within this single address space are designated for different purposes; for example, as memory for instruction code or memory for data. Internally the device uses separate buses[1] to access the instructions and data in these regions, thus allowing for parallel access. The terms program memory and data memory, which are used on the 8- and 16-bit PIC devices, are still relevant on PIC32 devices, but the smaller parts implement these in different address spaces.

All addresses used by the CPU within the device are virtual addresses. These are mapped to physical addresses by the system control processor (CP0).

[1] The device can be considered a Harvard architecture in terms of its internal bus arrangement.

7.3 VARIABLES IN DATA MEMORY

Most variables are ultimately positioned into the data memory. The exceptions are non-`auto` variables which are qualified as `const`, which are placed in the program memory space, see **Section 6.10.1 "Const Type Qualifier"**.

Due to the fundamentally different way in which `auto` variables and non-`auto` variables are allocated memory, they are discussed separately. To use the C/C++ language terminology, these two groups of variables are those with automatic storage duration and those with permanent storage duration, respectively.

> **Note:** The terms "local" and "global" are commonly used to describe variables, but are not ones defined by the language standard. The term "local variable" is often taken to mean a variable which has scope inside a function, and "global variable" is one which has scope throughout the entire program. However, the C/C++ language has three common scopes: block, file (i.e. internal linkage) and program (i.e. external linkage), so using only two terms to describe these can be confusing. For example, a `static` variable defined outside a function has scope only in that file, so it is not globally accessible, but it can be accessed by more than one function inside that file, so it is not local to any one function either. In terms of memory allocation, variables are allocated space based on whether it is an `auto` or not, hence the grouping in the following sections.

7.3.1 Non-auto Variable Allocation

Non-`auto` variables (those with permanent storage duration) are located by the compiler into any of the available data banks. This is done in a two-stage process: placing each variable into an appropriate section and later linking that section into data memory.

The compiler considers three categories of non-`auto` variable, which all relate to the value the variable should contain by the time the program begins. The following sections are used for the categories described.

`.pbss` These sections are used to store variables which use the `persistent` attribute, whose values should not be altered by the runtime start-up code. They are not cleared or otherwise modified at start-up.

`.bss` These sections (also .sbss) contain any uninitialized variables, which are not assigned a value when they are defined, or variables which should be cleared by the runtime start-up code.

`.data` These sections (also .sdata) contain the RAM image of any initialized variables, which are assigned a non-zero initial value when they are defined and which must have a value copied to them by the runtime start-up code.

Note that the data section used to hold initialized variables is the section that holds the RAM variables themselves. There is a corresponding section (called `.dinit`) that is placed into program memory (so it is non-volatile) and which is used to hold the initial values that are copied to the RAM variables by the runtime start-up code.

7.3.2 Static Variables

All `static` variables have permanent storage duration, even those defined inside a function which are "local static" variables. Local `static` variables only have scope in the function or block in which they are defined, but unlike `auto` variables, their memory is reserved for the entire duration of the program. Thus, they are allocated memory like other non-`auto` variables. Static variables may be accessed by other functions via pointers since they have permanent duration.

Variables which are static are guaranteed to retain their value between calls to a function, unless explicitly modified via a pointer.

Variables which are static and which are initialized only have their initial value assigned once during the program's execution. Thus, they may be preferable over initialized auto objects which are assigned a value every time the block they are defined in begins execution. Any initialized static variables are initialized in the same way as other non-auto initialized objects by the runtime start-up code, see **Section 3.5.2 "Peripheral Library Functions"**. Static variables are located in the same sections as their non-static counterparts.

7.3.3 Non-auto Variable Size Limits

Arrays of any type (including arrays of aggregate types) are fully supported by the compiler. So too are the structure and union aggregate types, see **Section 6.6 "Structures and Unions"**. There are no theoretical limits as to how large these objects can be made.

7.3.4 Changing the Default Non-auto Variable Allocation

There are several ways in which non-auto variables can be located in locations other than the default.

Variables can be placed in other device memory spaces by the use of qualifiers. For example if you wish to place variables in the program memory space, then the const specifier should be used (see **Section 6.10.1 "Const Type Qualifier"**).

If you wish to prevent all variables from using one or more data memory locations so that these locations can be used for some other purpose, you are best defining a variable (or array) using the address attribute so that it consumes the memory space, see **Section 6.12 "Variable Attributes"**.

If only a few non-auto variables are to be located at specific addresses in data space memory, then the variables can be located using the address attribute. This attribute is described in **Section 6.12 "Variable Attributes"**.

7.3.5 Data Memory Allocation Macros

Macros are provided for many commonly used attributes in order to enhance user code readability.

`__section__(s)`	Apply the section attribute with section name s.
`__unique_section__`	Apply the unique_section attribute.
`__ramfunc__`	Locate the attributed function in the RAM function code section.
`__longramfunc__`	Locate the attributed function in the RAM function code section and apply the longcall attribute.
`__longcall__`	Apply the longcall attribute.
`__ISR(v,ipl)`	Apply the interrupt attribute with priority level ipl and the vector attribute with vector number v.
`__ISR_AT_VECTOR(v,ipl)`	Apply the interrupt attribute with priority level ipl and the at_vector attribute with vector number v.
`__ISR_SINGLE__`	Specifies a function as an Interrupt Service Routine in single-vector mode. This places a jump at the single-vector location to the interrupt handler.
`__ISR_SINGLE_AT_VECTOR__`	Places the entire single-vector interrupt handler at the vector 0 location. When used, ensure that the vector spacing is set to accommodate the size of the handler.

7.4 AUTO VARIABLE ALLOCATION AND ACCESS

This section discusses allocation of auto variables (those with automatic storage duration). This also includes function parameter variables, which behave like auto variables, as well as temporary variables defined by the compiler.

The auto (short for *automatic*) variables are the default type of local variable. Unless explicitly declared to be static, a local variable will be made auto. The auto keyword may be used if desired.

The auto variables, as their name suggests, automatically come into existence when a function is executed, then disappear once the function returns. Since they are not in existence for the entire duration of the program, there is the possibility to reclaim memory they use when the variables are not in existence and allocate it to other variables in the program.

The PIC32's software stack is used to store all auto variables. Functions are reentrant and each instance of the function has its own area of memory on the stack for its auto and parameter variables, as described below. See **Section 4.4 "Stack"** and **Section 12.3.2 "Initialize Stack Pointer and Heap"** for more information on the stack.

The compiler dedicates General Purpose Register 29 as the software Stack Pointer. All processor stack operations, including function call, interrupts and exceptions use the software stack. The stack grows downward from high addresses to low addresses.

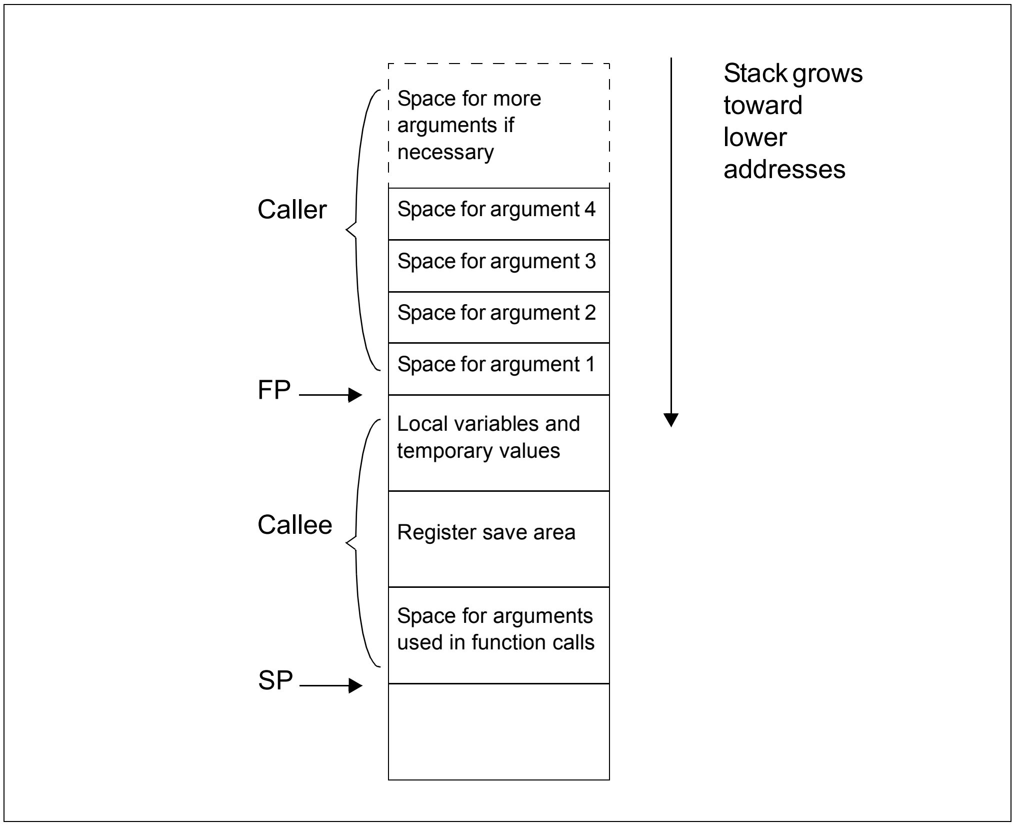
By default, the size of the stack is 1024 bytes. The size of the stack may be changed by specifying the size on the linker command line using the --defsym_min_stack_size linker command line option. An example of allocating a stack of 2048 bytes using the command line is:

```
xc32-gcc foo.c -Wl,--defsym,_min_stack_size=2048
```

The run-time stack grows downward from higher addresses to lower addresses (see Figure 7-1). The compiler uses two working registers to manage the stack:

- Register 29 (sp) – This is the Stack Pointer. It points to the next free location on the stack.
- Register 30 (fp) – This is the Frame Pointer. It points to the current function's frame. Each function, if required, creates a new frame from which automatic and temporary variables are allocated. Compiler optimization may eliminate Stack Pointer references via the Frame Pointer to equivalent references via the Stack Pointer. This optimization allows the Frame Pointer to be used as a General Purpose Register.

FIGURE 7-1: STACK FRAME


Space for more arguments if necessary
Space for argument 4
Space for argument 3
Space for argument 2
Space for argument 1
Caller
FP
Local variables and temporary values
Register save area
Space for arguments used in function calls
Callee
SP
Stack grows toward lower addresses

The the standard qualifiers `const` and `volatile` may both be used with `auto` variables and these do not affect how they are positioned in memory. This implies that a local `const`-qualified object is still an `auto` object and, as such, will be allocated memory on the stack in the data memory, not in the program memory like with non-`auto` `const` objects.

7.4.1 Local Variable Size Limits

There is no theoretical maximum size for auto variables.

7.5 VARIABLES IN PROGRAM MEMORY

The only variables that are placed into program memory are those that are not `auto` and which have been qualified `const`. If the `-mno-embedded-data` option is used, then even `const` objects are placed in RAM rather than the program memory. Any `auto` variables qualified `const` are placed on the stack along with other `auto` variables.

Any `const`-qualified (`auto` or non-`auto`) variable will always be read-only and any attempt to write to these in your source code will result in an error being issued by the compiler.

A `const` object is usually defined with initial values, as the program cannot write to these objects at runtime. However this is not a requirement. An uninitialized `const` object is allocated space in the bss section, along with other uninitialized RAM variables, but is still read-only.

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

7.5.1 Size Limitations of `const` Variables

There is no theoretical maximum size for `const` variables.

7.5.2 Changing the Default Allocation

If you only intend to prevent all variables from using one or more program memory locations so that you can use those locations for some other purpose, you are best reserving the memory using the memory adjust options.

If only a few non-auto `const` variables are to be located at specific addresses in program space memory, then the variables should use the address attribute to locate them at the desired location. This attribute is described in **Section 6.12 "Variable Attributes"**.

7.6 VARIABLES IN REGISTERS

Allocating variables to registers, rather than to a memory location, can make code more efficient. With MPLAB XC32 C Compiler, variables may be allocated to registers as part of code optimizations. For optimization levels 1 and higher, the values assigned to variables may cached in a register. During this time, the memory location associated with the variable may not hold a valid value.

The `register` keyword may be used to indicate your preference for the variable to be allocated a register, but this is just a recommendation and may not be honored. The specific register may be indicated as well, but this is not recommended as your register choice may conflict with the needs of the compiler. For example:

```
register unsigned int foo __asm__("at");
```

will attempt to allocate foo to the `at` register. As indicated in **Section 10.6 "Function Parameters"**, parameters may be passed to a function via a register.

7.7 DYNAMIC MEMORY ALLOCATION

The run-time heap is an uninitialized area of data memory that is used for dynamic memory allocation using the standard C library dynamic memory management functions, `calloc`, `malloc` and `realloc` along with the C++ new operator. Most C++ applications will require a heap.

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 function that uses one of these functions, then a heap must be created. A heap is created by specifying its size on the linker command line using the `--defsym_min_heap_size` linker command line option. An example of allocating a heap of 512 bytes using the command line is:

```
xc32-gcc foo.c -Wl,--defsym,_min_heap_size=512
```

An example of allocating a heap of `0xF000` bytes using the xc32-g++ driver is:

```
xc32-g++ vector.cpp -Wl,--defsym,_min_heap_size=0xF000
```

The linker allocates the heap immediately before the stack.

7.8 MEMORY MODELS

MPLAB XC32 C Compiler does not use fixed memory models to alter allocation of variables to memory.

The -G option (see **Section 3.9.1 "Options Specific to PIC32MX Devices"**), which controls the gp-relative addressing threshold, is similar to the small-data/large-data/scalar-data memory models offered by the Microchip MPLAB XC16 compiler. The value specified with this option indicates the maximum size of objects that will be allocated to the small data sections, e.g. sbss, sdata, etc.

NOTES:

Chapter 8. Operators and Statements

8.1 INTRODUCTION

The MPLAB XC32 C Compiler supports all ANSI operators. The exact results of some of these are implementation defined. Implementation-defined behavior is fully documented in **Appendix A. "Implementation-Defined Behavior"**. The following sections illustrate code operations that are often misunderstood, as well as additional operations that the compiler is capable of performing.

- Integral Promotion
- Type References
- Labels as Values
- Conditional Operator Operands
- Case Ranges

8.2 INTEGRAL PROMOTION

When there is more than one operand to an operator, they typically must be of exactly the same type. The compiler will automatically convert the operands, if necessary, so they do have the same type. The conversion is to a "larger" type so there is no loss of information; however, the change in type can cause different code behavior to what is sometimes expected. These form the standard type conversions.

Prior to these type conversions, some operands are unconditionally converted to a larger type, even if both operands to an operator have the same type. This conversion is called *integral promotion* and is part of Standard C behavior. The MPLAB XC32 C Compiler performs these integral promotions where required, and there are no options that can control or disable this operation. If you are not aware that the type has changed, the results of some expressions are not what would normally be expected.

Integral promotion is the implicit conversion of enumerated types, `signed` or `unsigned` varieties of `char`, `short int` or bit field types to either `signed int` or `unsigned int`. If the result of the conversion can be represented by an `signed int`, then that is the destination type, otherwise the conversion is to `unsigned int`.

Consider the following example:

```
unsigned char count, a=0, b=50;
if(a - b < 10)
  count++;
```

The `unsigned char` result of `a - b` is 206 (which is not less than 10), but both `a` and `b` are converted to `signed int` via integral promotion before the subtraction takes place. The result of the subtraction with these data types is -50 (which *is* less than 10) and hence the body of the `if()` statement is executed.

If the result of the subtraction is to be an `unsigned` quantity, then apply a cast. For example:

```
if((unsigned int)(a - b) < 10)
  count++;
```

The comparison is then done using `unsigned int`, in this case, and the body of the `if()` would not be executed.

Another problem that frequently occurs is with the bitwise compliment operator, ~. This operator toggles each bit within a value. Consider the following code:

```
unsigned char count, c;
c = 0x55;
if( ~c == 0xAA)
  count++;
```

If c contains the value 0x55, it often assumed that ~c will produce 0xAA, however the result is 0xFFFFFFAA and so the comparison in the above example would fail. The compiler may be able to issue a mismatched comparison error to this effect in some circumstances. Again, a cast could be used to change this behavior.

The consequence of integral promotion as illustrated above is that operations are not performed with char -type operands, but with int -type operands. However there are circumstances when the result of an operation is identical regardless of whether the operands are of type char or int. In these cases, the MPLAB XC32 C Compiler will not perform the integral promotion so as to increase the code efficiency. Consider the following example:

```
unsigned char a, b, c;
a = b + c;
```

Strictly speaking, this statement requires that the values of b and c should be promoted to unsigned int, the addition performed, the result of the addition cast to the type of a, and then the assignment can take place. Even if the result of the unsigned int addition of the promoted values of b and c was different to the result of the unsigned char addition of these values without promotion, after the unsigned int result was converted back to unsigned char, the final result would be the same. If an 8-bit addition is more efficient than a 32-bit addition, the compiler will encode the former.

If, in the above example, the type of a was unsigned int, then integral promotion would have to be performed to comply with the ANSI C standard.

8.3 TYPE REFERENCES

Another way to refer to the type of an expression is with the typeof keyword. This is a non-standard extension to the language. Using this feature reduces your code portability.

The syntax for using this keyword looks like sizeof, but the construct acts semantically like a type name defined with typedef.

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

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

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

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is a pointer to int.

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

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

- This declares y with the type of what x points to:
  `typeof (*x) y;`
- This declares y as an array of such values:
  `typeof (*x) y[4];`

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

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

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

Now the declaration can be rewritten this way:

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

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

8.4 LABELS AS VALUES

You can get the address of a label defined in the current function (or a containing function) with the unary operator '&&'. This is a non-standard extension to the language. Using this feature reduces your code portability.

The value returned has type void *. This value is a constant and can be used wherever a constant of that type is valid. For example:

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

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

```
goto *ptr;
```

Any expression of type void * is allowed.

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

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

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

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

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

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

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

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

8.5 CONDITIONAL OPERATOR 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. This is a non-standard extension to the language. Using this feature reduces your code portability.

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.

8.6 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 is a non-standard extension to the language. Using this feature reduces your code portability.

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:
```

Chapter 9. Register Usage

9.1 INTRODUCTION

This chapter examines registers used by the compiler to generate assembly from C/C++ source code.

- Register Usage
- Register Conventions

9.2 REGISTER USAGE

The assembly generated from C/C++ source code by the compiler will use certain registers that are present on the PIC MCU device. The compiler assumes that nothing other than code it generates can alter the contents of these registers, but an extended assembly language format can be used to indicate to the compiler registers used in assembly code so that code can be adjusted accordingly.

9.3 REGISTER CONVENTIONS

The 32 general purpose registers contained in the PIC32 are shown in Table 9-1. Some of these registers are assigned a dedicated task by the compiler. The name used in assembly code and the usage is indicated.

TABLE 9-1: REGISTER CONVENTIONS

Register Number	Software Name	Use
$0	`zero`	Always 0 when read.
$1	`at`	Assembler temporary variable.
$2-$3	`v0-v1`	Return value from functions.
$4-$7	`a0-a3`	Used for passing arguments to functions.
$8-$15	`t0-t7`	Temporary registers used by compiler for expression evaluation. Values not saved across function calls.
$16-$23	`s0-s7`	Temporary registers whose values are saved across function calls.
$24-$25	`t8-t9`	Temporary registers used by compiler for expression evaluation. Values not saved across function calls.
$26-$27	`k0-k1`	Reserved for interrupt/trap handler.
$28	`gp`	Global Pointer.
$29	`sp`	Stack Pointer.
$30	`fp or s8`	Frame Pointer if needed. Additional temporary saved register if not.
$31	`ra`	Return address for functions.

NOTES:

Chapter 10. Functions

The following sections describe how function definitions are written, and specifically how they can be customized to suit your application. The conventions used for parameters and return values, as well as the assembly call sequences are also discussed.

- Writing Functions
- Function Attributes and Specifiers
- Allocation of Function Code
- Changing the Default Function Allocation
- Function Size Limits
- Function Parameters
- Function Return Values
- Calling Functions
- Inline Functions

10.1 WRITING FUNCTIONS

Functions may be written in the usual way in accordance with the C/C++ language.

The only specifier that has any effect on function is `static`. Interrupt functions are defined with the use of the interrupt attribute, see **Section 10.2 "Function Attributes and Specifiers"**.

A function defined using the `static` specifier only affects the scope of the function, i.e. limits the places in the source code where the function may be called. Functions that are `static` may only be directly called from code in the file in which the function is defined. The equivalent symbol used in assembly code to represent the function may change if the function is `static`, see **Section 7.3.2 "Static Variables"**. This specifier does not change the way the function is encoded.

10.2 FUNCTION ATTRIBUTES AND SPECIFIERS

10.2.1 Function Attributes

`address(addr)`

The address attribute specifies an absolute virtual address for the function. Be sure to specify the address attribute using an appropriate virtual address for the target device. The address is typically in the range [0x9D000000,0x9D0FFFFC], as defined in the linker script as the 'kseg0_program_mem' memory region. For example,

```
    __attribute__((address(0x9D008000))) void bar (void);
```

The compiler performs no error checking on the address. The section containing the function will be located at the specified address regardless of the memory-regions specified in the linker script or the actual memory ranges on the target device. The application code must ensure that the address is valid for the target device.

To make effective use of absolute sections and the new best-fit allocator, standard program-memory and data-memory sections should not be mapped in the linker script. The built-in linker script does not map most standard sections such as the `.text`, `.data`, `.bss`, or `.ramfunc` sections. By not mapping these sections in the linker script, we allow these sections to be allocated using the best-fit allocator rather than the sequential allocator. Sections that are unmapped in the linker script can flow around absolute sections, whereas sections that are linker-script mapped are grouped together and allocated sequentially, potentially causing conflicts with absolute sections.

`alias ("symbol")`

Indicates that the function is an alias for another symbol. For example:

```
void foo (void) { /* stuff */ }
__attribute__ ((alias("foo"))) void bar (void);
```

Symbol `bar` is considered to be an alias for the symbol `foo`.

`always_inline`

If the function is declared `inline`, always inline the function, even if no optimization level was specified.

`at_vector`

Place the body of the function at the indicated exception vector address.
See **Chapter 11. "Interrupts"** and **Section 11.5 "Exception Handlers"**.

`const`

If a pure function determines its return value exclusively from its parameters (i.e., does not examine any global variables), it may be declared `const`, allowing for even more aggressive optimization. Note that a function which de-references a pointer argument is not `const` since the pointer de-reference uses a value which is not a parameter, even though the pointer itself is a parameter.

`deprecated`
`deprecated (msg)`

When a function specified as `deprecated` is used, a warning is generated. The optional *msg* argument, which must be a string, will be printed in the warning if present. The `deprecated` attribute may also be used for variables and types.

`far`

Always invoke the function by first loading its address into a register and then using the contents of that register. This allows calling a function located beyond the 28-bit addressing range of the direct `CALL` instruction.

`format (type, format_index, first_to_check)`

The `format` attribute indicates that the function takes a `printf`, `scanf`, `strftime`, or `strfmon` style format string and arguments and that the compiler should type check those arguments against the format string, just as it does for the standard library functions.

The `type` parameter is one of `printf`, `scanf`, `strftime` or `strfmon` (optionally with surrounding double underscores, e.g., `__printf__`) and determines how the format string will be interpreted.

The `format_index` parameter specifies which function parameter is the format string. Function parameters are numbered from the left-most parameter, starting from 1.

The first_to_check parameter specifies which parameter is the first to check against the format string. If first_to_check is zero, type checking is not performed, and the compiler only checks the format string for consistency (e.g., vfprintf).

format_arg (index)

The format_arg attribute specifies that a function manipulates a printf style format string and that the compiler should check the format string for consistency. The function attribute which is a format string is identified by index.

interrupt (priority)

Generate prologue and epilogue code for the function as an interrupt handler function. See **Chapter 11. "Interrupts"**. The argument specifies the interrupt priority using the symbols ip1 to ip7 to represent the 7 levels of priority.

keep

The __attribute__((keep)) may be applied to a function. The keep attribute will prevent the linker from removing the function with --gc-sections, even if it is unused.

longcall

Functionally equivalent to far.

malloc

Any non-Null Pointer return value from the indicated function will not alias any other pointer which is live at the point when the function returns. This allows the compiler to improve optimization.

mips16

Generate code for the function in the MIPS16 instruction set.

naked

Generate no prologue or epilogue code for the function.

near

Always invoke the function with an absolute CALL instruction, even when the -mlong-calls command line option is specified.

noinline

The function will never be considered for inlining.

nomips16

Always generate code for the function in the MIPS32® instruction set, even when compiling the translation unit with the -mips16 command line option.

nonnull (index, ...)

Indicate to the compiler that one or more pointer arguments to the function must be non-null. If the compiler determines that a Null Pointer is passed as a value to a non-null argument, and the -Wnonnull command line option was specified, a warning diagnostic is issued.

If no arguments are given to the nonnull attribute, all pointer arguments of the function are marked as non-null.

noreturn

Indicate to the compiler that the function will never return. In some situations, this can allow the compiler to generate more efficient code in the calling function since optimizations can be performed without regard to behavior if the function ever did return. Functions declared as `noreturn` should always have a return type of `void`.

optimize

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

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

pure

If a function has no side effects other than its return value, and the return value is dependent only on parameters and/or (nonvolatile) global variables, the compiler can perform more aggressive optimizations around invocations of that function. Such functions can be indicated with the `pure` attribute.

ramfunc

Treat the function as if it was in data memory. Allocate the function at the highest appropriately aligned address for executable code. Note that due to `ramfunc` alignment and placement requirements, the address attribute should not be used with the `ramfunc` attribute. The presence of the `ramfunc` section causes the linker to emit the symbols necessary for the crt0.S start-up code to initialize the bus matrix appropriately for executing code out of data memory. Use this attribute along with the `far/longcall` attribute and the `section` attribute. For example:

```
  __attribute__((ramfunc,section(“.ramfunc”),far,unique_section))
  unsigned int myramfunct (void_
    { /* code */ }
```

A macro in the sys/attribs.h header file makes the `ramfunc` attribute simple to use:

```
  #include <sys/attribs.h>
  __longramfunc__ unsigned int  myramfunct (void)
    { /* code */ }
```

section(“name”)

Place the function into the named section.

For example:

```
void __attribute__ ((section (“.wilma”))) baz () {return;}
```

Function `baz` will be placed in section `.wilma`.

The `-ffunction-sections` command line option has no effect on functions defined with a `section` attribute.

unique_section

Place the function in a uniquely named section, as if -ffunction-sections had been specified. If the function also has a section attribute, use that section name as the prefix for generating the unique section name.

For example:

```
void __attribute__ ((section (“.fred”), unique_section) foo (void)
{return;}
```

Function foo will be placed in section .fred.foo.

unused

Indicate to the compiler that the function may not be used. The compiler will not issue a warning for this function if it is not used.

used

Indicate to the compiler that the function is always used and code must be generated for the function even if the compiler cannot see a reference to the function. For example, if inline assembly is the only reference to a static function.

vector

Generate a branch instruction at the indicated exception vector which targets the function. See **Chapter 11. “Interrupts”** and **Section 11.5 “Exception Handlers”**.

warn_unused_result

A warning will be issued if the return value of the indicated function is unused by a caller.

weak

A weak symbol indicates that if another version of the same symbol is available, that version should be used instead. For example, this is useful when a library function is implemented such that it can be overridden by a user written function.

10.3 ALLOCATION OF FUNCTION CODE

Code associated with C/C++ functions is normally always placed in the program Flash memory of the target device.

Functions may be located in and executed from RAM rather than Flash by using the __ramfunc__ and __longramfunc__ macros.

Functions specified as a RAM function will be copied to RAM by the start-up code and all calls to those functions will reference the RAM location. Functions located in RAM will be in a different 512MB memory segment than functions located in program memory, so the longcall attribute should be applied to any RAM function, which will be called from a function not in RAM. The __longramfunc__ macro will apply the longcall attribute as well as place the function in RAM[1].

```
#include <sys/attribs.h>
/* function ‘foo’ will be placed in RAM */
void __ramfunc__ foo (void)
{
}
```

[1] Specifying __longramfunc__ is functionally equivalent to specifying both __ramfunc__ and __longcall__.

```
/* function 'bar' will be placed in RAM and will be invoked
   using the full 32 bit address */
void __longramfunc__ bar (void)
{
}
```

10.4 CHANGING THE DEFAULT FUNCTION ALLOCATION

The assembly code associated with a C/C++ function can be placed at an absolute address. This can be accomplished by using the address attribute and specifying the virtual address of the function, see **Section 6.12 "Variable Attributes"**.

Functions can also be placed at specific positions by placing them in a user-defined section and then linking this section at an appropriate address, see **Section 6.12 "Variable Attributes"**.

10.5 FUNCTION SIZE LIMITS

There are no theoretical limits as to how large functions can be made.

10.6 FUNCTION PARAMETERS

MPLAB XC uses a fixed convention to pass arguments to a function. The method used to pass the arguments depends on the size and number of arguments involved.

> **Note:** The names "argument" and "parameter" are often used interchangeably, but typically an argument is the actual value that is passed to the function and a parameter is the variable defined by the function to store the argument.

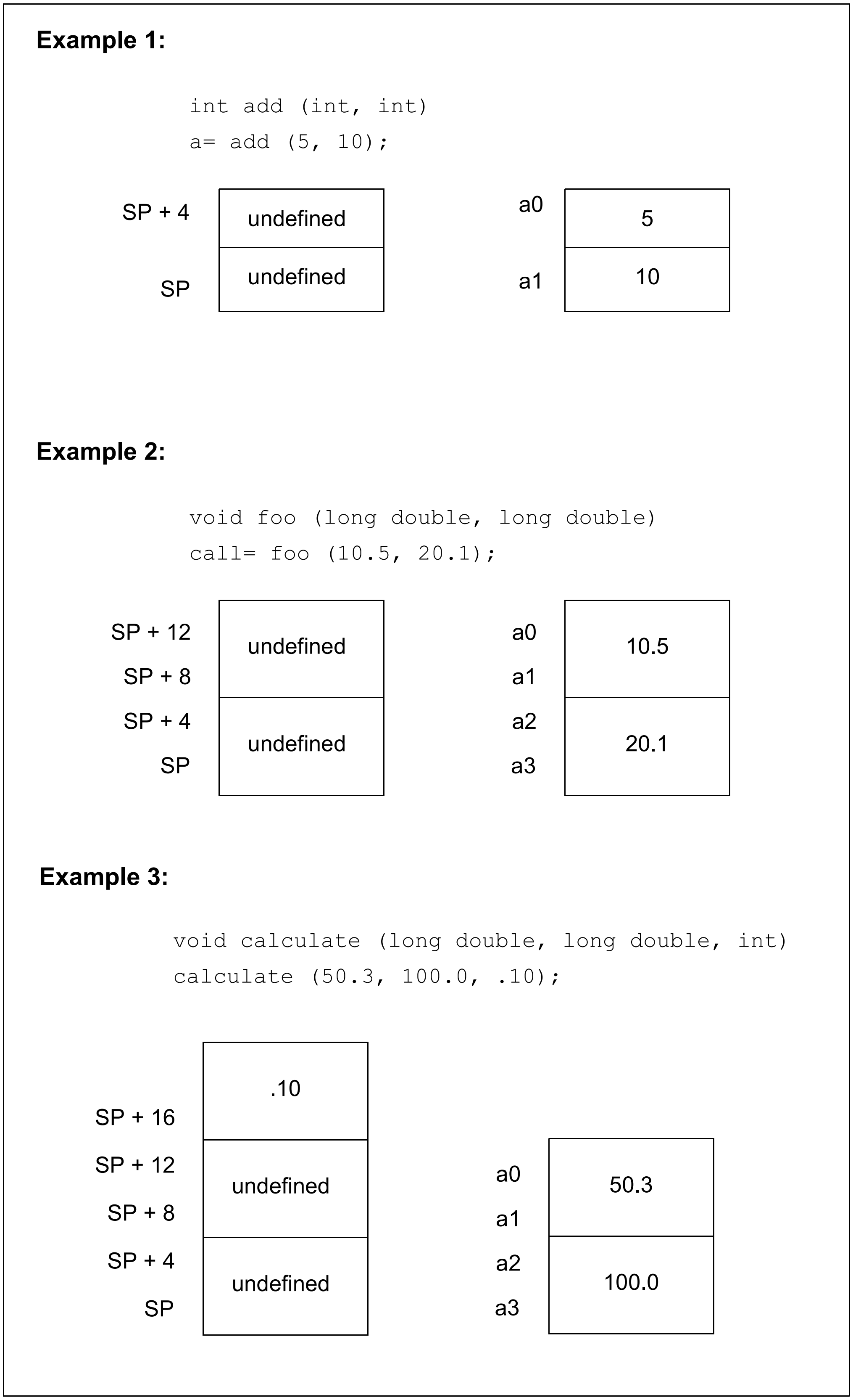
The Stack Pointer is always aligned on an 8-byte boundary.

- All integer types smaller than a 32-bit integer are first converted to a 32-bit value. The first four 32 bits of arguments are passed via registers `a0`-`a3` (see Table 10-1 for how many registers are required for each data type).
- Although some arguments may be passed in registers, space is still allocated on the stack for all arguments to be passed to a function (see Figure 10-1). Application code should not assume that the current argument value is on the stack, even when space is allocated.
- When calling a function:
  - Registers `a0`-`a3` are used for passing arguments to functions. Values in these registers are not preserved across function calls.
  - Registers `t0`-`t7` and `t8`-`t9` are caller saved registers. The calling function must push these values onto the stack for the registers' values to be saved.
  - Registers `s0`-`s7` are called saved registers. The function being called must save any of these registers it modifies.
  - Register `s8` is a saved register if the optimizer eliminates its use as the Frame Pointer. `s8` is a reserved register otherwise.
  - Register `ra` contains the return address of a function call.

TABLE 10-1: REGISTERS REQUIRED

Data Type	Number of Registers Required
char	1
short	1
int	1
long	1
long long	2
float	1
double	1
long double	2
structure	Up to 4, depending on the size of the struct.

FIGURE 10-1: PASSING ARGUMENTS



Example 1:

```
int add (int, int)
a= add (5, 10);
```

Example 2:

```
void foo (long double, long double)
call= foo (10.5, 20.1);
```

Example 3:

```
void calculate (long double, long double, int)
calculate (50.3, 100.0, .10);
```

10.7 FUNCTION RETURN VALUES

Function return values are returned in registers.

If a function needs to return an actual structure or union — not a pointer to such an object — the called function copies this object to an area of memory that is reserved by the caller. The caller passes the address of this memory area in register $4 when the function is called. The function also returns a pointer to the returned object in register `v0`. Having the caller supply the return object's space allows re-entrance.

10.8 CALLING FUNCTIONS

By default, functions are called using the direct form of the call (`jal`) instruction. This allows calls to destinations within a 256 MB segment. This operation can be changed through the use of attributes applied to functions or command-line options so that a longer, but unrestricted, call is made.

The `-mlong-calls` option, see **Section 3.9.1 "Options Specific to PIC32MX Devices"**, forces a register form of the call to be employed by default. Generated code is longer, but calls are not limited in terms of the destination address.

The attributes `longcall` or `far` can be used with a function definition to always enforce the longer call sequence for that function. The `near` attribute can be used with a function so that calls to it use the shorter direct call, even if the `-mlong-calls` option is in force.

10.9 INLINE FUNCTIONS

By declaring a function `inline`, you can direct the compiler to integrate that function's code into the code for its callers. This usually makes execution faster by eliminating the function-call overhead. In addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time, so that not all of the inline function's code needs to be included. The effect on code size is less predictable. Machine code may be larger or smaller with inline functions, depending on the particular case.

Note: Function inlining will only take place when the function's definition is visible (not just the prototype). In order to have a function inlined into more than one source file, the function definition may be placed into a header file that is included by each of the source files.

To declare a function inline, use the `inline` keyword in its declaration, like this:

```
inline int
inc (int *a)
{
  (*a)++;
}
```

(If you are using the `-traditional` option or the `-ansi` option, write `__inline__` instead of `inline`.) You can also make all "simple enough" functions inline with the command-line option `-finline-functions`. The compiler heuristically decides which functions are simple enough to be worth integrating in this way, based on an estimate of the function's size.

Note: The `inline` keyword will only be recognized with `-finline` or optimizations enabled.

Certain usages in a function definition can make it unsuitable for inline substitution. Among these usages are: use of `varargs`, use of `alloca`, use of variable-sized data, use of computed `goto` and use of nonlocal `goto`. Using the command-line option `-Winline` will warn when a function marked `inline` could not be substituted, and will give the reason for the failure.

In compiler syntax, the `inline` keyword does not affect the linkage of the function.

When a function is both `inline` and `static`, if all calls to the function are integrated into the caller and the function's address is never used, then the function's own assembler code is never referenced. In this case, the compiler does not actually output assembler code for the function, unless you specify the command-line option `-fkeep-inline-functions`. Some calls cannot be integrated for various reasons (in particular, calls that precede the function's definition cannot be integrated and neither can recursive calls within the definition). If there is a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that can't be inlined. The compiler will only eliminate `inline` functions if they are declared to be `static` and if the function definition precedes all uses of the function.

When an `inline` function is not `static`, then the compiler must assume that there may be calls from other source files. Since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-`static` inline function is always compiled on its own in the usual fashion.

If you specify both `inline` and `extern` in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function and had not defined it.

This combination of `inline` and `extern` has a similar effect to a macro. Put a function definition in a header file with these keywords and put another copy of the definition (lacking `inline` and `extern`) in a library file. The definition in the header file will cause most calls to the function to be inlined. If any uses of the function remain, they will refer to the single copy in the library.

Chapter 11. Interrupts

11.1 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.

PIC32MX devices support multiple interrupts, from both internal and external sources. The devices allow high-priority interrupts to override any lower priority interrupts that may be in progress.

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

- Interrupt Operation
- Writing an Interrupt Service Routine
- Associating a Handler Function with an Exception Vector
- Exception Handlers
- Interrupt Service Routine Context Switching
- Latency
- Nesting Interrupts
- Enabling/Disabling Interrupts
- ISR Considerations

11.2 INTERRUPT OPERATION

The compiler incorporates features allowing interrupts to be fully handled from C/C++ code. Interrupt functions are often called interrupt handlers or *Interrupt Service Routines* (ISRs).

Each interrupt source typically has a control bit in an SFR which can disable that interrupt source. Check your device data sheet for full information how your device handles interrupts.

Interrupt code is the name given to any code that executes as a result of an interrupt occurring. Interrupt code completes at the point where the corresponding return from interrupt instruction is executed. This contrasts with *main-line code*, which, for a freestanding application, is usually the main part of the program that executes after reset.

11.3 WRITING AN INTERRUPT SERVICE ROUTINE

An interrupt handler function is different to an ordinary function in that it handles the context save and restore to ensure that upon return from interrupt, the program context is maintained.A different code sequence is used to return from these functions as well.

Several attributes can be used to ensure that the compiler generates the correct code for an ISR. Macros are provided so that this is easier to accomplish, see the following sections.

There are several actions that the compiler needs to take to generate an interrupt service routine. The compiler has to be told to use an alternate form of return code. The function also needs to be linked to the interrupt vector. Only the mip32 instruction set can be used in ISRs, so the compiler must be told to generate code using this instruction set, even if the option to generate mip16 instructions has been used.

An interrupt function must be declared as type `void` and may not have parameters. This is the only function prototype that makes sense for an interrupt function since they are never directly called in the source code.

Interrupt functions must not be called directly from C/C++ code (due to the different return instruction that is used), but they themselves may call other functions, both user-defined and library functions, but be aware that this may use additional registers which will need to be saved and restored by the context switch code.

A function is marked as an interrupt handler function (also known as an Interrupt Service Routine or ISR) via either the interrupt attribute or the interrupt pragma[1]. While each method is functionally equivalent to the other, the interrupt attribute is more commonly used and therefore the recommended method. The interrupt is specified as handling interrupts of a specific priority level or for operating in single vector mode.

For all interrupt vectors without specific handlers, a default interrupt handler will be installed. The default interrupt handler is supplied by the libpic32.a library and will cause a debug breakpoint and reset the device. An application may override the default handler and provide an application-specific default interrupt handler by declaring an interrupt function with the name `_DefaultInterrupt`.

11.3.1 Interrupt Attribute

```
__attribute__((interrupt([IPLn[SRS|SOFT|AUTO]])))
```

Where *n* is in the range of 0..7, inclusive.

Use the interrupt attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present. The generated code preserves context by either using a shadow register set (SRS) or using generated software instructions (SOFT) to push context onto the stack. See Example 11-1 for an interrupt attribute.

EXAMPLE 11-1: INTERRUPT ATTRIBUTE

```
void __attribute__((interrupt(IPL7SRS))) bambam (void);
```

Many PIC32 devices allow us to specify, via configuration-bit settings, which interrupt priority level will use the shadow register set (e.g., `#pragma config` FSRSSEL=PRIORITY_7). Refer to the device data sheet to determine if your PIC32 target device supports this feature. This means we must specify which context-saving mechanism to use for each interrupt handler. The compiler will generate interrupt

[1] Note that pre-processor macros are not expanded in pragma directives.

function prologue and epilogue code utilizing shadow register context saving for the `IPLnSRS` Interrupt Priority Level (IPL) specifier. It will use software context saving for the `IPLnSOFT` IPL specifier.

Note: Application code is responsible for applying the correct value to the matching handler routine.

The compiler also supports an `IPLnAUTO` IPL specifier that uses the run-time value in `SRSCTL` to determine whether it should use software or SRS context-saving code. The compiler defaults to using `IPLnAUTO` when the IPL specifier is omitted from the `interrupt()` attribute.

For devices that do not support a shadow register set for interrupt context saving, use IPLnSOFT for all interrupt handlers.

Note: `SRS` has the shortest latency and `SOFT` has a longer latency due to registers saved on the stack. `AUTO` adds a few cycles to test if `SRS` or `SOFT` should be used.

11.3.2 Interrupt Pragma

Note: The interrupt pragma is provided only for compatibility when porting code from other compilers. The interrupt function attribute is the preferred and more common way to write an interrupt service routine.

```
# pragma interrupt function-name IPLn[AUTO|SOFT|SRS] [vector
[@]vector-number [, vector-number-list]]
# pragma interrupt function-name  single [vector [@] 0
```

Where *n* is in the range of 0..7, inclusive.

The `IPLn [AUTO|SOFT|SRS]` IPL specifier may be all uppercase or all lowercase.

The function definition for a handler function indicated by an interrupt pragma must follow in the same translation unit as the pragma itself.

The `interrupt` attribute will also indicate that a function definition is an interrupt handler. It is functionally equivalent to the interrupt pragma.

For example, the definitions of `foo` below both indicate that it is an interrupt handler function for an interrupt of priority 4 that uses software context saving.

```
#pragma interrupt foo IPL4SOFT
void foo (void)
```

is functionally equivalent to

```
void __attribute__ ((interrupt(IPL4SOFT))) foo (void)
```

11.3.3 __ISR Macros

The <`sys/attribs.h`> header file provides macros intended to simplify the application of attributes to interrupt functions. There are also vector macros defined in the processor header files. (See the appropriate header file in the compiler's `/pic32mx/include/proc` directory.)

- __ISR(V, IPL)
- __ISR_AT_VECTOR(v, IPL)
- Interrupt-Vector Macros

11.3.3.1 __ISR(V, IPL)

Use the `__ISR(v, IPL)` macro to assign the vector-number location and associate it with the specified IPL. This will place a jump to the interrupt handler at the associated vector location. This macro also applies the `nomips16` attribute since PIC32MX devices require that interrupt handlers must use the MIPS32 instruction set.

EXAMPLE 11-2: CORE TIMER VECTOR, IPL2SOFT

```
#include <xc.h>
#include <sys/attribs.h>
void __ISR(_CORE_TIMER_VECTOR, IPL2SOFT) CoreTimerHandler(void);
```

Example 11-2 creates an interrupt handler function for the core timer interrupt that has an interrupt priority level of two. The compiler places a dispatch function at the associated vector location. To reach this function, the core timer interrupt flag and enable bits must be set, and the interrupt priority should be set to a level of two. The compiler generates software context-saving code for this handler function.

EXAMPLE 11-3: CORE SOFTWARE 0 VECTOR, IPL3SRS

```
#include <xc.h>
#include <sys/attribs.h>
void __ISR(_CORE_SOFTWARE_0_VECTOR,IPL3SRS)
CoreSoftwareInt0Handler(void);
```

Example 11-3 creates an interrupt handler function for the core software interrupt 0 that has an interrupt priority level of three. The compiler places a dispatch function at the associated vector location. To reach this function, the core software interrupt flag and enable bits must be set, and the interrupt priority should be set to a level of three. The device configuration fuses must assign Shadow Register Set 1 to interrupt priority level three. The compiler generates code that assumes that register context will be saved in SRS1.

EXAMPLE 11-4: CORE SOFTWARE 1 VECTOR, IPL0AUTO

```
#include <xc.h>
#include <sys/attribs.h>
void __ISR(_CORE_SOFTWARE_1_VECTOR, IPL0AUTO)
CoreSoftwareInt1Handler(void);
```

Example 11-4 creates an interrupt handler function for the core software interrupt 1 that has an interrupt priority level of zero. The compiler places a dispatch function at the associated vector location. To reach this function, the core software interrupt 1 flag and enable bits must be set, and the interrupt priority should be set to a level of zero. The compiler generates code that determines at run time whether software context saving is required.

EXAMPLE 11-5: CORE SOFTWARE 1 VECTOR, DEFAULT

```
#include <xc.h>
#include <sys/attribs.h>
void __ISR(_CORE_SOFTWARE_1_VECTOR) _CoreSoftwareInt1Handler(void);
```

Example 11-5 is functionally equivalent to Example 3. Because the IPL specifier is omitted, the compiler assumes `IPL0AUTO`.

11.3.3.2 __ISR_AT_VECTOR(v, IPL)

Use the `__ISR_AT_VECTOR(v, IPL)` to place the entire interrupt handler at the vector location and associate it with the software-assigned interrupt priority. Application code is responsible for making sure that the vector spacing is set to accommodate the size of the handler. This macro also applies the nomips16 attribute since ISR functions are required to be MIPS32.

EXAMPLE 11-6: CORE TIMER VECTOR, IPL2SOFT

```
#include <xc.h>
#include <sys/attribs.h>
void __ISR_AT_VECTOR(_CORE_TIMER_VECTOR, IPL2SOFT)
CoreTimerHandler(void);
```

Example 11-6 creates an interrupt handler function for the core timer interrupt that has an interrupt priority level of two. The compiler places the entire interrupt handler at the vector location. It does not use a dispatch function. To reach this function, the core timer interrupt flag and enable bits must be set, and the interrupt priority should be set to a level of two. The compiler generates software context-saving code for this handler function.

11.3.3.3 INTERRUPT-VECTOR MACROS

Each processor-support header file provides a macro for each interrupt-vector number (for example, `/pic32mx/include/proc/p32mx360f512l.h`. See the appropriate header file in the compiler install directory). When used in conjunction with the `__ISR()` macro provided by the `sys\attribs.h` header file, these macros help make an Interrupt Service Routine easier to write and maintain.

EXAMPLE 11-7: INTERRUPT-VECTOR WITH HANDLER

```
#include <xc.h>
#include <sys/attribs.h>
void __ISR (_TIMER_1_VECTOR, IPL7SRS) Timer1Handler (void);
```

Example 11-7 creates an interrupt handler function for the Timer 1 interrupt that has an interrupt priority level of seven. The compiler places a dispatch function at the vector location associated with macro `_TIMER_1_VECTOR` as defined in the device-specific header file. To reach this function, the Timer 1 interrupt flag and enable bits must be set, and the interrupt priority should be set to a level of seven. For devices that allow assignment of shadow registers to specific IPL values, the device Configuration bit settings must assign Shadow Register Set 1 to interrupt priority level seven. The compiler generates code that assumes that register context will be saved in SRS1.

Example 11-8 uses the peripheral library provided with the compiler to set up Timer 1 for an interrupt using priority level 7. The code is written to toggle pin RD0.

EXAMPLE 11-8: FULL TIMER 1 EXAMPLE WITH PERIPHERAL LIBRARY

```
/*  Blink an LED on the PIC32MX Ethernet Starter Kit using the
 *  PIC32MX795F512L target device.
 */
#include <xc.h>
#include <plib.h>
#include <sys/attribs.h>

// Configuration Bit settings using the config pragma
// SYSCLK = 80 MHz (8MHz Crystal/ FPLLIDIV * FPLLMUL / FPLLODIV)
// PBCLK = 10 MHz
// Primary Osc w/PLL (XT+,HS+,EC+PLL)
// WDT OFF
// Other options are “do not care”
//
#pragma config FPLLMUL=MUL_20, FPLLIDIV=DIV_2, FPLLODIV=DIV_1, FWDTEN=OFF
#pragma config POSCMOD=HS, FNOSC=PRIPLL, FPBDIV=DIV_8

// Calculate the PR1 (period) at compile time
#define SYS_FREQ                (80000000L)
#define PB_DIV                  8
#define PRESCALE                256
#define TOGGLES_PER_SEC         1
#define T1_TICK                 (SYS_FREQ/PB_DIV/PRESCALE/TOGGLES_PER_SEC)

int main(void)
{
  // STEP 1
  // Configure the device for maximum performance but do not change the PBDIV
  // Given the options, this function will change the flash wait states, RAM
  // wait state and enable prefetch cache but will not change the PBDIV.
  // The PBDIV value is already set via the config pragma FPBDIV option above.
  SYSTEMConfig(SYS_FREQ, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);

  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  // STEP 2. configure Timer 1 using internal clock, 1:256 prescale
  OpenTimer1(T1_ON | T1_SOURCE_INT | T1_PS_1_256, T1_TICK);
  // set up the timer interrupt with a priority of 7
  ConfigIntTimer1(T1_INT_ON | T1_INT_PRIOR_7);

  // enable multi-vector interrupts
  INTEnableSystemMultiVectoredInt();

  // configure PORTDbits.RD0 = output
  mPORTDSetPinsDigitalOut(BIT_0);
  while(1);
}

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// STEP 3. configure the Timer 1 interrupt handler
// Determine shadow-register or software-stack context saving at
// runtime by using the IPL7AUTO priority specifier.
// Note that the n value in IPLnAUTO _must_ match the priority
// of the timer1 interrupt source configured above.

#ifdef __cplusplus
  // For C linkage when compiling for C++
  extern "C" {
#endif /* __cplusplus */
```

```
void __ISR(_TIMER_1_VECTOR, IPL7AUTO) Timer1Handler(void)
{
  // clear the interrupt flag
  mT1ClearIntFlag();
  // .. things to do
  // .. in this case, toggle the LED
  mPORTDToggleBits(BIT_0);
}

#ifdef __cplusplus
 }
#endif
```

11.4 ASSOCIATING A HANDLER FUNCTION WITH AN EXCEPTION VECTOR

For PIC32MX devices, there are 64 exception vectors, numbered 0..63 inclusive. Each interrupt source is mapped to an exception vector as specified in the device data sheet. By default, four words of space are reserved at each vector address for a dispatch to the handler function for that exception source.

An interrupt handler function can be associated with an interrupt vector either as the target of a dispatch function located at the exception vector address, or as being located directly at the exception vector address. A single handler function can be the target of multiple dispatch functions.

The association of a handler function to one or more exception vector addresses is specified via a vector attribute on the function declaration. For compatibility purposes, you may also associate a handler function to a vector address using a clause of the interrupt pragma, a separate vector pragma, or a vector attribute on the function declaration.

11.4.1 Vector Attribute

A handler function can be associated with one or more exception vector addresses via an attribute. The `at_vector` attribute indicates that the handler function should itself be placed at the exception vector address. The `vector` attribute indicates that a dispatch function should be created at the exception vector address(es) which will transfer control to the handler function.

For example, the following declaration specifies that function `foo` will be created as an interrupt handler function of priority four. `foo` will be located at the address of exception vector 54.

```
void __attribute__ ((interrupt(IPL4SOFT))) __attribute__
((at_vector(54))) foo (void)
```

The following declaration specifies that function `foo` will be created as an interrupt handler function of priority four. Define dispatch functions targeting `foo` at exception vector addresses 52 and 53.

```
void __attribute__ ((interrupt(IPL4SOFT))) __attribute__
((vector(53, 52))) foo (void)
```

Handler functions that are linked directly to the vector will be executed faster. Although the vector spacing can be adjusted, there is limited space between vectors and linking a substantial handler function directly at a vector may cause it to overlap the higher vector locations, preventing their use. In such situations, using a dispatch function is a safer option.

11.4.2 Interrupt Pragma Clause

Note:	The interrupt pragma and its vector clause are provided only for compatibility when porting code from other compilers. The vector function attribute is the preferred way to associate a handler function to an exception vector address.

The interrupt pragma has an optional `vector` clause following the priority specifier.

# pragma interrupt *function-name IPL-specifier* [**vector** [@]*vector-number* [, *vector-number-list*]]

A dispatch function targeting the specified handler function will be created at the exception vector address for the specified vector numbers. If the first vector number is specified with a preceding “@” symbol, the handler function itself will be located there directly.

For example, the following pragma specifies that function `foo` will be created as an interrupt handler function of priority four. `foo` will be located at the address of exception vector 54. A dispatch function targeting `foo` will be created at exception vector address 34.

```
#pragma interrupt foo IPL4AUTO vector @54, 34
```

The following pragma specifies that function `bar` will be created as an interrupt handler function of priority five. `bar` will be located in general purpose program memory (.text section). A dispatch function targeting `bar` will be created at exception vector address 23.

```
#pragma interrupt bar IPL5SOFT vector 23
```

11.4.3 Vector Pragma

Note:	The vector pragma is provided only for compatibility when porting code from other compilers. The vector function attribute is the preferred way to associate a handler function to an exception vector address.

The `vector` pragma creates one or more dispatch functions targeting the indicated function. For target functions specified with the `interrupt` pragma, this functions as if the vector clause had been used. The target function of a `vector` pragma can be any function, including external functions implemented in assembly or by other means.

# pragma vector *function-name* **vector** *vector-number* [, *vector-number-list*]

The following pragma defines a dispatch function targeting `foo` at exception vector address 54.

```
#pragma vector foo 54
```

11.5 EXCEPTION HANDLERS

The PIC32MX devices also have two exception vectors for non-interrupt exceptions. These exceptions are grouped into bootstrap exceptions and general exceptions.

11.5.1 Bootstrap Exception

A reset exception is any exception which occurs while bootstrap code is running ($\texttt{Status}_{\texttt{BEV}}$=1). All reset exceptions are vectored to 0xBFC00380.

At this location, the 32-bit toolchain places a branch instruction targeting a function named _bootstrap_exception_handler(). In the standard library, a default weak version of this function is provided which merely goes into an infinite loop. If the user application provides an implementation of _bootstrap_exception_handler(), that implementation will be used instead.

11.5.2 General Exception

A general exception is any non-interrupt exception which occurs during program execution outside of bootstrap code ($\texttt{Status}_{\texttt{BEV}}$=0). General exceptions are vectored to offset 0x180 from EBase.

At this location, the 32-bit toolchain places a branch instruction targeting a function named _general_exception_context(). The provided implementation of this function saves context, calls an application handler function, restores context and performs a return from the exception instruction. The context saved is the hi and lo registers and all General Purpose Registers except s0-s8, which are defined to be preserved by all called functions and so are not necessary to actively save here again. The values of the Cause and Status registers are passed to the application handler function (_general_exception_handler()). If the user application provides an implementation of _general_exception_context(), that implementation will be used instead.

```
void _general_exception_handler (unsigned cause, unsigned status);
```

A weak default implementation of _general_exception_handler() is provided in the standard library which merely goes into an infinite loop. If the user application provides an implementation of _general_exception_handler(), that implementation will be used instead.

11.6 INTERRUPT SERVICE ROUTINE CONTEXT SWITCHING

The standard calling convention for C/C++ functions will already preserve zero, s0-s7, gp, sp, and fp. k0 and k1 are used by the compiler to access and preserve non-GPR context, but are always accessed atomically (i.e., in sequences with global interrupts disabled), so they need not be preserved actively. A handler function will actively preserve the a0-a3, t0-t9, v0, v1 and ra registers in addition to the standard registers.

An interrupt handler function will also actively save and restore processor status registers that are utilized by the handler function. Specifically, the EPC, SR, hi and lo registers are preserved as context.

Handler functions may use a shadow register set to preserve the General Purpose Registers, enabling lower latency entry into the application code of the handler function. On some devices, the shadow register set is assigned to an interrupt priority level (IPL) using the device Configuration bit settings (e.g., #pragma config FSRSSEL=PRIORITY_6). While on other devices, the shadow register set may be hard wired to IPL7. Consult the target device's data sheet for more information on the shadow register set.

11.6.1 Context Restoration

Any objects saved by software are automatically restored by software before the interrupt function returns. The order of restoration is the reverse to that used when context is saved.

11.7 LATENCY

There are two elements that affect the number of cycles between the time the interrupt source occurs and the execution of the first instruction of your ISR code. These are:

- **Processor Servicing of Interrupt** – The amount of time it takes the processor to recognize the interrupt and branch to the first address of the interrupt vector. To determine this value, refer to the processor data sheet for the specific processor and interrupt source being used.
- **ISR Code** – The compiler saves the registers that were used by the ISR. This includes the TODO registers. Moreover, if the ISR calls an ordinary function, then the compiler will save all the working registers, 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.

11.8 NESTING INTERRUPTS

Interrupts may be nested. The interrupt priority scheme implemented in the PIC32 architecture allows you to specify which interrupt sources may be interruptible by others. See your device data sheet for explicit details on interrupt operation.

11.9 ENABLING/DISABLING INTERRUPTS

Macros are available in the PIC32 peripheral library to control aspects of interrupt operation. See the Microchip PIC32MX Peripheral Library documentation for more information.

11.10 ISR CONSIDERATIONS

There are few issues arising with interrupt functions.

As with all compilers, limiting the number of registers used by the interrupt function, or any functions called by the interrupt function, may result in less context switch code being generated and executed by the compiler, see **Section 11.7 “Latency”**. Keeping interrupt functions small and simple will help you achieve this.

Chapter 12. Main, Runtime Start-up and Reset

12.1 INTRODUCTION

When creating C/C++ code, there are elements that are required to ensure proper program operation: a `main` function must be present; start-up code will be needed to initialize and clear variables and setup registers and the processor; and reset conditions will need to be handled.

- The Main Function
- Runtime Start-up Code
- The On Reset Routine

12.2 THE MAIN FUNCTION

The identifier `main` is special. It must be used as the name of a function that will be the first function to execute in a program. You must always have one and only one function called `main` in your programs. Code associated with `main`, however, is not the first code to execute after reset. Additional code provided by the compiler and known as the runtime start-up code is executed first and is responsible for transferring control to the `main()` function.

12.3 RUNTIME START-UP CODE

A C/C++ program requires certain objects to be initialized and the processor to be in a particular state before it can begin execution of its function `main()`. It is the job of the runtime start-up code to perform these tasks. The runtime start-up code is executed before `main()`, but if you require any special initialization to be performed immediately after reset, you should use on reset feature described in **Section 12.4 "The On Reset Routine"**

The PIC32MX start-up code must perform the following:

1. Jump to NMI Handler if an NMI Occurred
2. Initialize Stack Pointer and Heap
3. Initialize Global Pointer
4. Initialize or Clear Variables and RAM Functions Using the Data-Initialization Template
5. Initialize Bus Matrix Registers
6. Call "On Bootstrap" Procedure
7. Change Location of Exception Vectors
8. For C++, call the C++ initialization code to invoke all constructors for file-scope static storage objects
9. Call Main

The following provisions are made regarding the run-time model:

- Kernel mode only
- KSEG1 only
- RAM functions are attributed with `__ramfunc__` or `__longramfunc__`, (defined in `sys/attribs.h`) meaning that all RAM functions end up in the `.ramfunc` section and the function is `ramfunc` attributed.

12.3.1 Jump to NMI Handler if an NMI Occurred

If an NMI caused entry to the Reset vector, a jump to an NMI handler procedure (`_nmi_handler`) occurs. A weak version of the NMI handler procedure is provided that performs an `ERET`. The `_nmi_handler` function must be attributed with `nomips16` [e.g., `__attribute__((nomips16))`] since the start-up code jumps to this function.

12.3.2 Initialize Stack Pointer and Heap

The Stack Pointer (`sp`) register must be initialized in the start-up code. To enable the start-up code to initialize the `sp` register, the linker script must initialize a variable which points to the end of KSEG1 data memory[1]. This variable is named `_stack`. The user can change the minimum amount of stack space allocated by providing the command line option `--defsym _min_stack_size=`*N* to the linker. `_min_stack_size` is provided by the linker script with a default value of `1024`.

On a similar note, the user may wish to utilize a heap with their application. While the start-up code does not need to initialize the heap, the standard C libraries (`sbrk`) must be made aware of the heap location and its size. The linker script creates a variable to identify the beginning of the heap. The location of the heap is the end of the utilized KSEG1 data memory. This variable is named `_heap`. The user can change the minimum amount of heap space allocated by providing the command line option `--defsym _min_heap_size=`*M* to the linker. `_min_heap_size` is provided by the linker script with a default value of `0`. If the heap is used when the heap size is set to zero, the behavior is the same as when the heap usage exceeds the minimum heap size. Namely, it overflows into the space allocated for the stack.

The heap and the stack use the unallocated KSEG1 data memory, with the heap starting from a low address in KSEG1 data memory, and growing upwards towards the stack while the stack starts at a higher address in KSEG1 data memory and grows downwards towards the heap. The linker attempt to allocate the heap and stack together in the largest gap of memory available in the KSEG1 data memory region. If enough space is not available based on the minimum amount of heap size and stack size requested, the linker issues an error.

[1] The end of data memory is different based on whether RAM functions exist. If RAM functions exist, then part of the DRM must be configured for the kernel program to contain the RAM functions, and the Stack Pointer is located one word prior to the beginning of the DRM kernel program boundary address. If RAM functions do not exist, then the Stack Pointer is located at the true end of DRM.

FIGURE 12-1: STACK AND HEAP LAYOUT

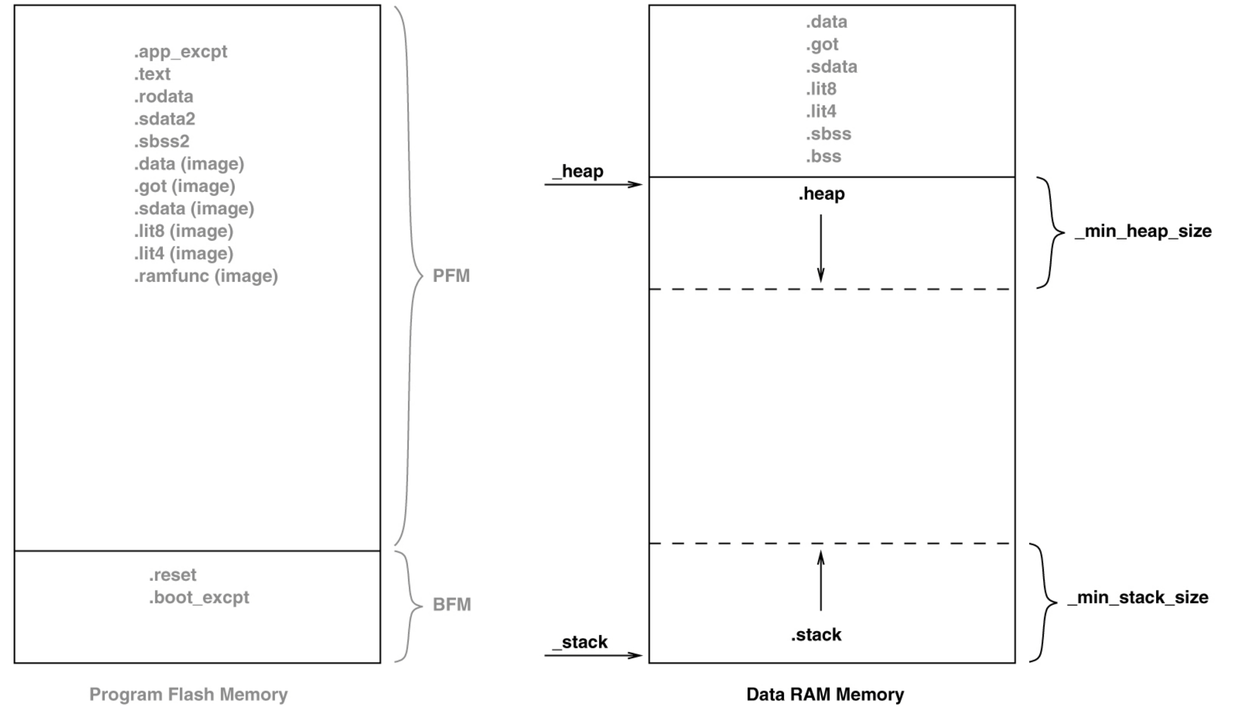
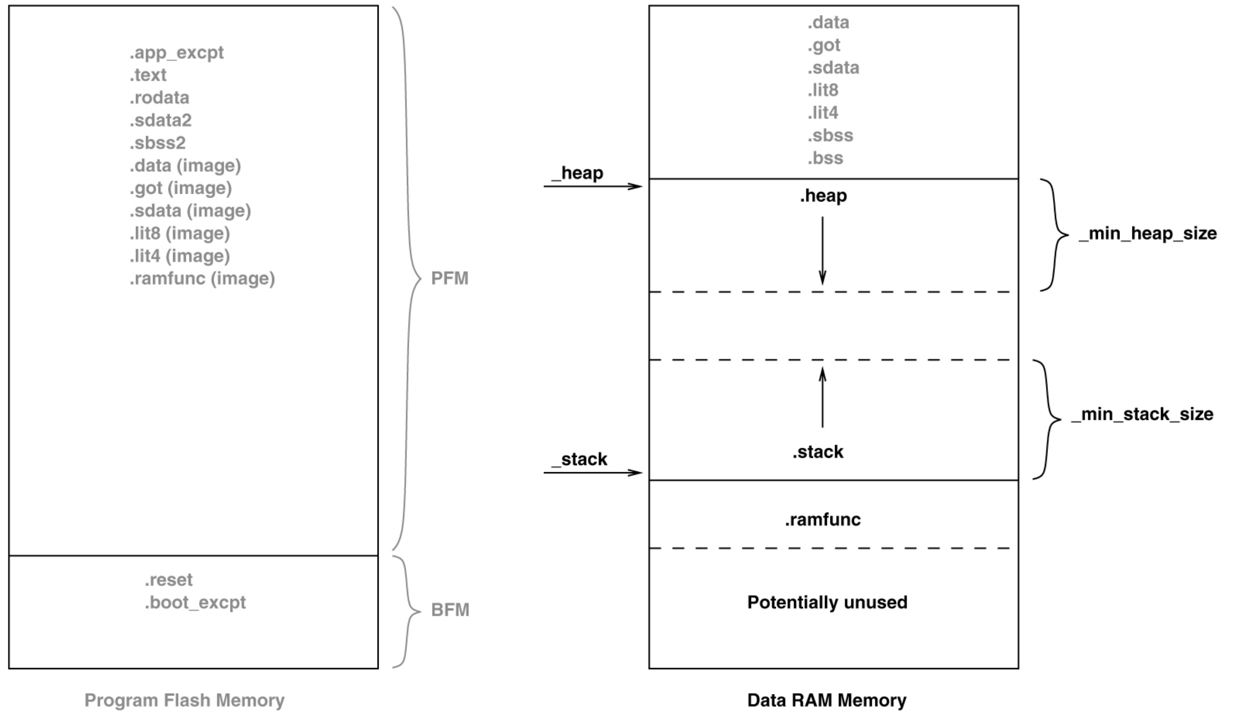


FIGURE 12-2: STACK AND HEAP LAYOUT WITH RAM FUNCTIONS



12.3.3 Initialize Global Pointer

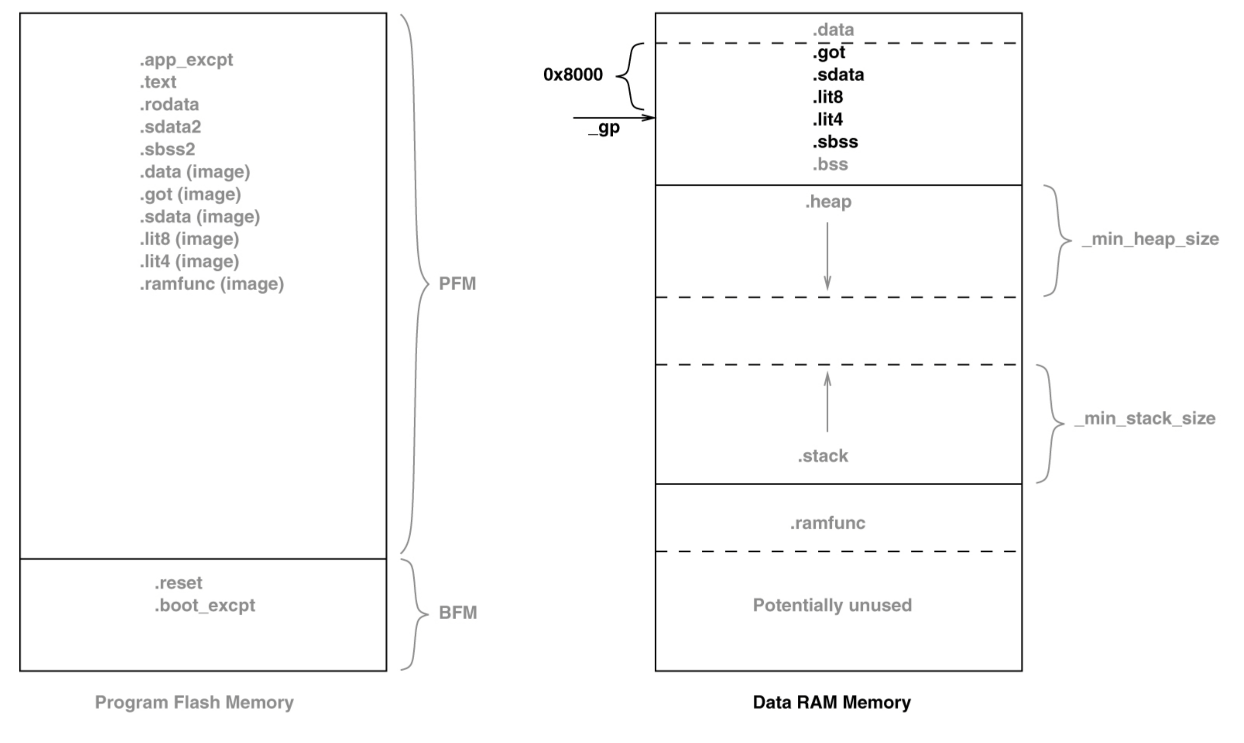
The compiler toolchain supports Global Pointer (gp) relative addressing. Loads and stores to data residing within 32KB of either side of the address stored in the gp register can be performed in a single instruction using the gp register as the base register. Without the Global Pointer, loading data from a static memory area takes two instructions – one to load the Most Significant bits of the 32-bit constant address computed by the compiler/linker and one to do the data load.

To utilize gp-relative addressing, the compiler and assembler must group all of the "small" variables and constants into one of the following sections:

- .lit4.
- .sdata.
- .sdata.*
- .gnu.linkonce.s.*
- lit8
- sbss
- sbss.*
- .gnu.linkonce.sb.*

The linker must then group all of the above input sections together. This grouping is handled by the default linker script. The run-time start-up code must initialize the gp register to point to the "middle" of this output section. To enable the start-up code to initialize the gp register, the linker script must initialize a variable which is 32 KB from the start of the output section containing the "small" variables and constants. This variable is named _gp (to match core linker scripts). Besides being initialized in the standard GPR set, the Global Pointer must also be initialized in the register shadow set.

FIGURE 12-3: GLOBAL POINTER LOCATION



12.3.4 Initialize or Clear Variables and RAM Functions Using the Data-Initialization Template

Those non-auto objects which are not initialized must be cleared before execution of the program begins. This task is also performed by the runtime start-up code.

Uninitialized variables are those which are not auto objects and which are not assigned a value in their definition, for example output in the following example:

```
int output;
int main(void) { ...
```

Such uninitialized objects will only require space to be reserved in RAM where they will reside and be accessed during program execution (runtime).

There are two uninitialized data sections—.sbss and .bss. The .sbss section is a data segment containing uninitialized variables less than or equal to *n* bytes where *n* is determined by the -G*n* command line option. The .bss section is a data segment containing uninitialized variables not included in .sbss.

Another task of the runtime start-up code is to ensure that any initialized variables contain their initial value before the program begins execution. Initialized variables are those which are not auto objects and which are assigned an initial value in their definition, for example input in the following example:

```
int input = 88;
int main(void) { ...
```

Such initialized objects have two components: their initial value (0x0088 in the above example) stored in program memory (i.e. placed in the HEX file), and space for the variable reserved in RAM, which will reside and be accessed during program execution (runtime).

The runtime start-up code will copy all the blocks of initial values from program memory to RAM so the variables will contain the correct values before main is executed.

Since auto objects are dynamically created, they require code to be positioned in the function in which they are defined to perform their initialization. It is possible that the initial value of an auto object may change on each instance of the function and so the initial values cannot be stored in program memory and copied. As a result, initialized auto objects are not considered by the runtime start-up code, but are instead initialized by assembly code in each function output.

Note: Initialized auto variables can impact code performance, particularly if the objects are large in size. Consider using global or static objects instead.

Variables whose contents should be preserved over a reset, or even power off, should be qualified with the persistent attribute, see **Section 6.10 “Standard Type Qualifiers”**. Such variables are linked at a different area of memory and are not altered by the runtime start-up code in any way.

Four initialized data sections exist:.sdata, .data, .lit4, and .lit8. The .sdata section is a data segment containing initialized variables less than or equal to *n* bytes where *n* is determined by the -G*n* command line option. The .data section is a data segment containing initialized variables not included in .sdata. The .lit4 and .lit8 sections contain constants, which the assembler stores in memory rather than in the instruction stream.

In order to clear or initialize these sections, the linker creates a data-initialization template, which is loaded into an output section named `.dinit`. The linker creates this special `.dinit` section, allocated in program memory, to hold the template for the run-time initialization of data. The C/C++ start-up module, `crt0.o`, interprets this template and copies initial data values into initialized data sections. This includes sections containing `ramfunc` attributed functions. Other data sections (such as `.bss`) are cleared before the `main()` function is called. The persistent data section (`.pbss`) is not affected. When the application's main program takes control, all variables and RAM functions in data memory have been initialized.

The data initialization template contains one record for each output section in data memory. The template is terminated by a null instruction word. The format of a data initialization record is:

```
/* data init record */
struct data_record {
  char *dst;            /* destination address  */
  unsigned int len;     /* length in bytes      */
  unsigned int format;  /* format code          */
  char dat[0];          /* variable-length data */
};
```

The first element of the record is a pointer to the section in data memory. The second and third elements are the section length and format code, respectively. The last element is an optional array of data bytes. For bss-type sections, no data bytes are required.

Currently supported format codes are:

- 0 – Fill the output section with zeros
- 1 – Copy each byte of data from the data array

12.3.5 Initialize Bus Matrix Registers

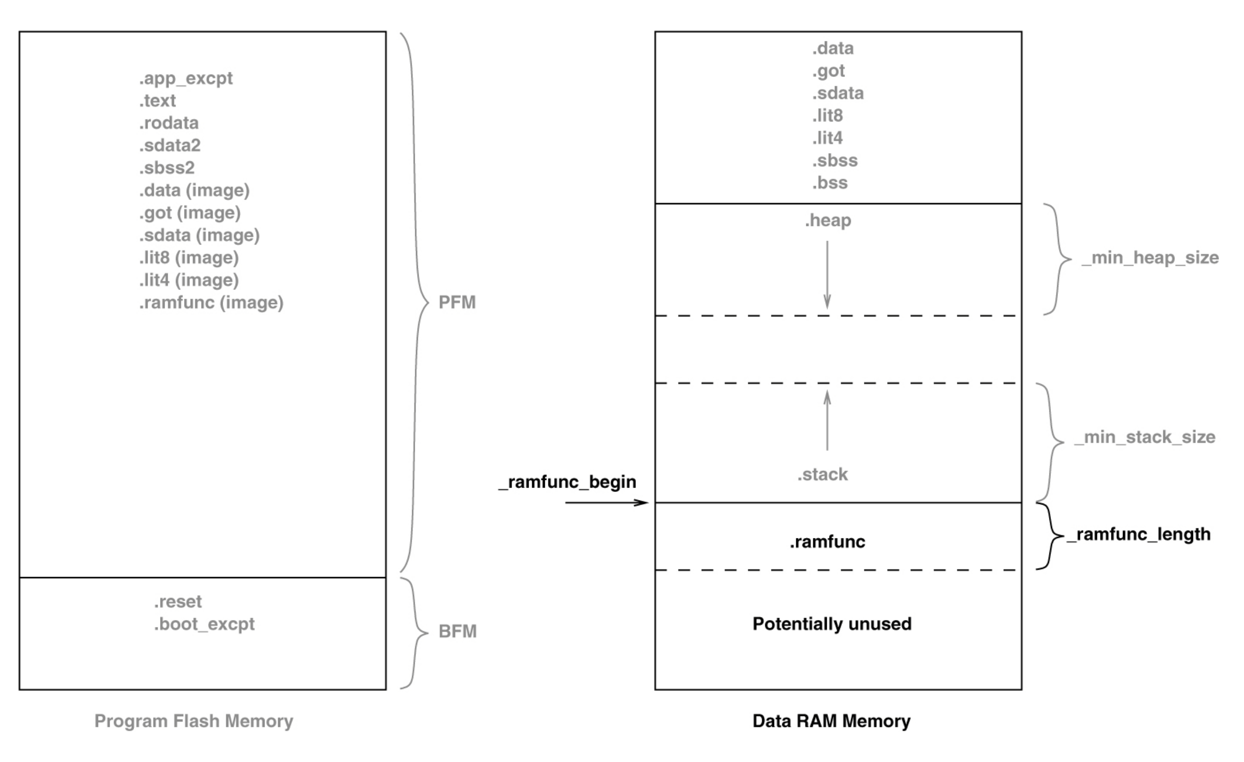
The bus matrix registers (`BMXDKPBA`, `BMXDUDBA`, `BMXDUPBA`) should be initialized by the start-up code if any RAM functions exist; otherwise, these registers should not be modified. To determine whether any RAM functions exist in the application, the linker provides a variable that contains the length of the `.ramfunc` section[1]. This variable is named `_ramfunc_length`. In addition, the linker provides a 2K-aligned variable required for the boundary register (BMXDKPBA). The variable is named `_bmxdkpba_address`. The default linker script also provides two variables that contain the address of the bus matrix registers. These variables are named `_bmxdkpba_address`, `_bmxdudba_address`, and `_bmxdupba_address`. The following calculations are used to calculate these addresses:

```
_bmxdudba_address = LENGTH(${DATA_MEMORY_LOCATION}) ;
_bmxdupba_address = LENGTH(${DATA_MEMORY_LOCATION}) ;
```

The linker ensures that RAM functions are aligned to a 2K alignment boundary as is required by the `BMXDKPBA` register.

[1] All functions attributed with `__ramfunc__` or `__longramfunc__` are placed in the `.ramfunc` section.

FIGURE 12-4: BUS MATRIX INITIALIZATION



12.3.5.1 INITIALIZE CP0 REGISTERS

The `CP0` registers are initialized in the following order:

1. `Count` register
2. `Compare` register
3. `EBase` register
4. `IntCtl` register
5. `Cause` register
6. `Status` register

12.3.5.2 HARDWARE ENABLE REGISTER (`HWREna` – CP0 REGISTER 7, SELECT 0)

This register contains a bit mask that determines which hardware registers are accessible via the `RDHWR` instruction. Privileged software may determine which of the hardware registers are accessible by the `RDHWR` instruction. In doing so, a register may be virtualized at the cost of handling a Reserved Instruction Exception, interpreting the instruction, and returning the virtualized value. For example, if it is not desirable to provide direct access to the `Count` register, access to the register may be individually disabled, and the return value can be virtualized by the operating system.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.3 BAD VIRTUAL ADDRESS REGISTER (`BadVAddr` – CP0 REGISTER 8, SELECT 0)

This register is a read-only register that captures the most recent virtual address that caused an Address Error exception (`AdEL` or `AdES`).

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.4 COUNT REGISTER (Count – CP0 REGISTER 9, SELECT 0)

This register acts as a timer, incrementing at a constant rate, whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. The counter increments every other clock if the DC bit in the Cause register is '0'. The Count register can be written for functional or diagnostic purposes, including at Reset or to synchronize processors. By writing the $\texttt{Count}_{DM}$ bit in the Debug register, it is possible to control whether the Count register continues incrementing while the processor is in Debug mode.

This register is cleared in the PIC32MX start-up code.

12.3.5.5 COMPARE REGISTER (Compare – CP0 REGISTER 11, SELECT 0)

This register acts in conjunction with the Count register to implement a timer and timer interrupt function. The timer interrupt is an output of the core. The Compare register maintains a stable value and does not change on its own. When the value of the Count register equals the value of the Compare register, the SI_TimerInt pin is asserted. This pin remains asserted until the Compare register is written. The SI_TimerInt pin can be fed back into the core on one of the interrupt pins to generate an interrupt. For diagnostic purposes, the Compare register is a read/write register. In normal use, however, the Compare register is write-only. Writing a value to the Compare register, as a side effect, clears the timer interrupt.

This register is set to 0xFFFFFFFF in the PIC32MX start-up code.

12.3.5.6 STATUS REGISTER (Status – CP0 REGISTER 12, SELECT 0)

This register is a read/write register that contains the operating mode, Interrupt Enabling, and the diagnostic states of the processor. Fields of this register combine to create operating modes for the processor.

The following settings are initialized by the PIC32MX start-up code (0b000000000x0xx0?0000000000000000):

- Access to Coprocessor 0 not allowed in User mode (CU0 = 0)
- User mode uses configured endianess (RE = 0)
- No change to exception vectors location (BEV = no change)
- No change to flag bits that indicate reason for entry to the Reset exception vector (SR, NMI = no change)
- Interrupt masks are cleared to disable any pending interrupt requests (IM7..IM2 = 0, IM1..IM0 = 0)
- Interrupt priority level is 0 (IPL = 0)
- Base mode is Kernel mode (UM = 0)
- Error level is normal (ERL = 0)
- Exception level is normal (EXL = 0)
- Interrupts are disabled (IE = 0)

12.3.5.7 INTERRUPT CONTROL REGISTER (IntCtl – CP0 REGISTER 12, SELECT 1)

This register controls the expanded interrupt capability added in Release 2 of the Architecture, including vectored interrupts and support for an external interrupt controller.

This register contains the vector spacing for interrupt handling. The vector spacing portion of this register (bits 9..5) is initialized with the value of the _vector_spacing symbol by the PIC32MX start-up code. All other bits are set to '1'.

12.3.5.8 SHADOW REGISTER CONTROL REGISTER (SRSCtl – CP0 REGISTER 12, SELECT 2)

This register controls the operation of the GPR shadow sets in the processor.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.9 SHADOW REGISTER MAP REGISTER (SRSMap – CP0 REGISTER 12, SELECT 3)

This register contains eight 4-bit fields that provide the mapping from a vector number to the shadow set number to use when servicing such an interrupt. The values from this register are not used for a non-interrupt exception, or a non-vectored interrupt ($\mathtt{Cause}_{\mathtt{IV}}$ `= 0` or $\mathtt{IntCtl}_{\mathtt{VS}}$ `= 0`). In such cases, the shadow set number comes from `SRSCtlESS`. If `SRSCtlHSS` is zero, the results of a software read or write of this register are UNPREDICTABLE. The operation of the processor is UNDEFINED if a value is written to any field in this register that is greater than the value of `SRSCtlHSS`. The `SRSMap` register contains the shadow register set numbers for vector numbers 7..0. The same shadow set number can be established for multiple interrupt vectors, creating a many-to-one mapping from a vector to a single shadow register set number.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.10 CAUSE REGISTER (Cause – CP0 REGISTER 13, SELECT 0)

This register primarily describes the cause of the most recent exception. In addition, fields also control software interrupt requests and the vector through which interrupts are dispatched. With the exception of the `DC`, `IV`, and `IP1..IP0` fields, all fields in the `Cause` register are read-only. Release 2 of the Architecture added optional support for an External Interrupt Controller (EIC) interrupt mode, in which `IP7..IP2` are interpreted as the Requested Interrupt Priority Level (`RIPL`).

The following settings are initialized by the PIC32MX start-up code:

- Enable counting of `Count` register (`DC = no change`)
- Use the special exception vector (`16#200`) (`IV = 1`)
- Disable software interrupt requests (`IP1..IP0 = 0`)

12.3.5.11 EXCEPTION PROGRAM COUNTER (EPC – CP0 REGISTER 14, SELECT 0)

This register is a read/write register that contains the address at which processing resumes after an exception has been serviced. All bits of the `EPC` register are significant and must be writable. For synchronous (precise) exceptions, the `EPC` contains one of the following:

- The virtual address of the instruction that was the direct cause of the exception
- The virtual address of the immediately preceding branch or jump instruction, when the exception causing instruction is a branch delay slot and the `Branch Delay` bit in the `Cause` register is set.

On new exceptions, the processor does not write to the `EPC` register when the `EXL` bit in the `Status` register is set; however, the register can still be written via the `MTC0` instruction.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.12 PROCESSOR IDENTIFICATION REGISTER (PRId – CP0 REGISTER 15, SELECT 0)

This register is a 32-bit read-only register that contains information identifying the manufacturer, manufacturer options, processor identification, and revision level of the processor.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.13 EXCEPTION BASE REGISTER (EBase – CP0 REGISTER 15, SELECT 1)

This register is a read/write register containing the base address of the exception vectors used when $\mathrm{Status}_{\mathrm{BEV}}$ equals 0, and a read-only CPU number value that may be used by software to distinguish different processors in a multi-processor system. The EBase register provides the ability for software to identify the specific processor within a multi-processor system, and allows the exception vectors for each processor to be different, especially in systems composed of heterogeneous processors. Bits 31..12 of the EBase register are concatenated with zeros to form the base of the exception vectors when $\mathrm{Status}_{\mathrm{BEV}}$ is 0. The exception vector base address comes from fixed defaults when $\mathrm{Status}_{\mathrm{BEV}}$ is 1, or for any EJTAG Debug exception. The reset state of bits 31..12 of the EBase register initialize the exception base register to 16#80000000, providing backward compatibility with Release 1 implementations. Bits 31..30 of the EBase register are fixed with the value 2#10 to force the exception base address to be in KSEG0 or KSEG1 unmapped virtual address segments.

If the value of the exception base register is to be changed, this must be done with $\mathrm{Status}_{\mathrm{BEV}}$ equal 1. The operation of the processor is UNDEFINED if the Exception Base field is written with a different value when $\mathrm{Status}_{\mathrm{BEV}}$ is 0.

Combining bits 31..30 with the Exception Base field allows the base address of the exception vectors to be placed at any 4K byte page boundary. If vectored interrupts are used, a vector offset greater than 4K byte can be generated. In this case, bit 12 of the Exception Base field must be zero. The operation of the processor is UNDEFINED if software writes bit 12 of the Exception Base field with a 1 and enables the use of a vectored interrupt whose offset is greater than 4K bytes from the exception base address.

This register is initialized with the value of the _ebase_address symbol by the PIC32MX start-up code. _ebase_address is provided by the linker script with a default value of the start of KSEG1 program memory. The user can change this value by providing the command line option `--defsym _ebase_address=A` to the linker.

12.3.5.13.1 Config Register (Config – CP0 Register 16, Select 0)

This register specifies various configuration and capabilities information. Most of the fields in the Config register are initialized by hardware during the Reset exception process, or are constant.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.13.2 Config1 Register (Config1 – CP0 Register 16, Select 1)

This register is an adjunct to the Config register and encodes additional information about the capabilities present on the core. All fields in the Config1 register are read-only.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.13.3 Config2 Register (`Config2` – CP0 Register 16, Select 2)

This register is an adjunct to the `Config` register and is reserved to encode additional capabilities information. `Config2` is allocated for showing the configuration of level 2/3 caches. These fields are reset to 0 because L2/L3 caches are not supported on the core. All fields in the `Config2` register are read-only.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.13.4 Config3 Register (`Config3` – CP0 Register 16, Select 3)

This register encodes additional capabilities. All fields in the `Config3` register are read-only.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.14 DEBUG REGISTER (`Debug` – CP0 REGISTER 23, SELECT 0)

This register is used to control the debug exception and provide information about the cause of the debug exception, and when re-entering at the debug exception vector due to a normal exception in Debug mode. The read-only information bits are updated every time the debug exception is taken, or when a normal exception is taken when already in Debug mode. Only the `DM` bit and the $\texttt{EJTAG}_{\texttt{ver}}$ field are valid when read from non-Debug mode. The values of all other bits and fields are UNPREDICTABLE. Operation of the processor is UNDEFINED if the `Debug` register is written from non-Debug mode.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.15 TRACE CONTROL REGISTER (`TraceControl` – CP0 REGISTER 23, SELECT 1)

This register provides control and status information. The `TraceControl` register is only implemented if the EJTAG Trace capability is present.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.16 TRACE CONTROL 2 REGISTER (`TraceControl2` – CP0 REGISTER 23, SELECT 2)

This register provides additional control and status information. The `TraceControl2` register is only implemented if the EJTAG Trace capability is present.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.17 USER TRACE DATA REGISTER (`UserTraceData` – CP0 REGISTER 23, SELECT 3)

When this register is written to, a trace record is written indicating a type 1 or type 2 user format. This type is based on the `UT` bit in the `TraceControl` register. This register cannot be written in consecutive cycles. The trace output data is UNPREDICTABLE if this register is written in consecutive cycles. The `UserTraceData` register is only implemented if the EJTAG Trace capability is present.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.18 TRACEBPC REGISTER (`TraceBPC` – CP0 REGISTER 23, SELECT 4)

This register is used to control start and stop of tracing using an EJTAG hardware breakpoint. The hardware breakpoint would then be set as a triggered source and optionally also as a Debug exception breakpoint. The `TraceBPC` register is only implemented if both the hardware breakpoints and the EJTAG Trace cap are present.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.19 DEBUG2 REGISTER (Debug2 – CP0 REGISTER 23, SELECT 5)

This register holds additional information about complex breakpoint exceptions. The Debug2 register is only implemented if complex hardware breakpoints are present.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.20 DEBUG EXCEPTION PROGRAM COUNTER (DEPC – CP0 REGISTER 24, SELECT 0)

This register is a read/write register that contains the address at which processing resumes after a debug exception or Debug mode exception has been serviced. For synchronous (precise) debug and Debug mode exceptions, the DEPC contains either:

- The virtual address of the instruction that was the direct cause of the debug exception, or
- The virtual address of the immediately preceding branch or jump instruction, when the debug exception causing instruction is in a branch delay slot, and the Debug Branch Delay (DBD) bit in the Debug register is set.

For asynchronous debug exceptions (debug interrupt, complex break), the DEPC contains the virtual address of the instruction where execution should resume after the debug handler code is executed.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.21 ERROR EXCEPTION PROGRAM COUNTER (ErrorEPC – CP0 REGISTER 30, SELECT 0)

This register is a read/write register, similar to the EPC register, except that it is used on error exceptions. All bits of the ErrorEPC are significant and must be writable. It is also used to store the program counter on Reset, Soft Reset, and Non-Maskable Interrupt (NMI) exceptions. The ErrorEPC register contains the virtual address at which instruction processing can resume after servicing an error. This address can be:

- The virtual address of the instruction that caused the exception, or
- The virtual address of the immediately preceding branch or jump instruction when the error causing instruction is a branch delay slot.

Unlike the EPC register, there is no corresponding branch delay slot indication for the ErrorEPC register.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.5.22 DEBUG EXCEPTION SAVE REGISTER (DeSave – CP0 REGISTER 31, SELECT 0)

This register is a read/write register that functions as a simple memory location. This register is used by the debug exception handler to save one of the GPRs that is then used to save the rest of the context to a pre-determined memory area (such as in the EJTAG Probe). This register allows the safe debugging of exception handlers and other types of code where the existence of a valid stack for context saving cannot be assumed.

No initialization is performed on this register in the PIC32MX start-up code.

12.3.6 Call "On Bootstrap" Procedure

A procedure is called after initializing the CP0 registers. This procedure allows users to perform actions during bootstrap (i.e., while StatusBEV is set) and before entering into the main routine. An empty weak version of this procedure (_on_bootstrap) is provided with the start-up code. This procedure may be used for performing hardware initialization and/or for initializing the environment required by an RTOS.

12.3.7 Change Location of Exception Vectors

Immediately before executing any application code, the `StatusBEV` is cleared to change the location of the exception vectors from the bootstrap location to the normal location.

12.3.8 Call the C++ initialization code

Invoke all constructors for C++ file-scope static-storage objects. The startup code must call the constructors last because the low-level initialization must be done before executing application code.

12.3.9 Call Main

The last thing that the start-up code performs is a call to the main routine. If the user returns from main, the start-up code goes into an infinite loop.

12.3.10 Symbols Required by Start-up Code and C/C++ Library

This section details the symbols that are required by the start-up code and C/C++ library. Currently the default linker script defines these symbols. If an application provides a custom linker script, the user must ensure that all of the following symbols are provided in order for the start-up code and C library to function:

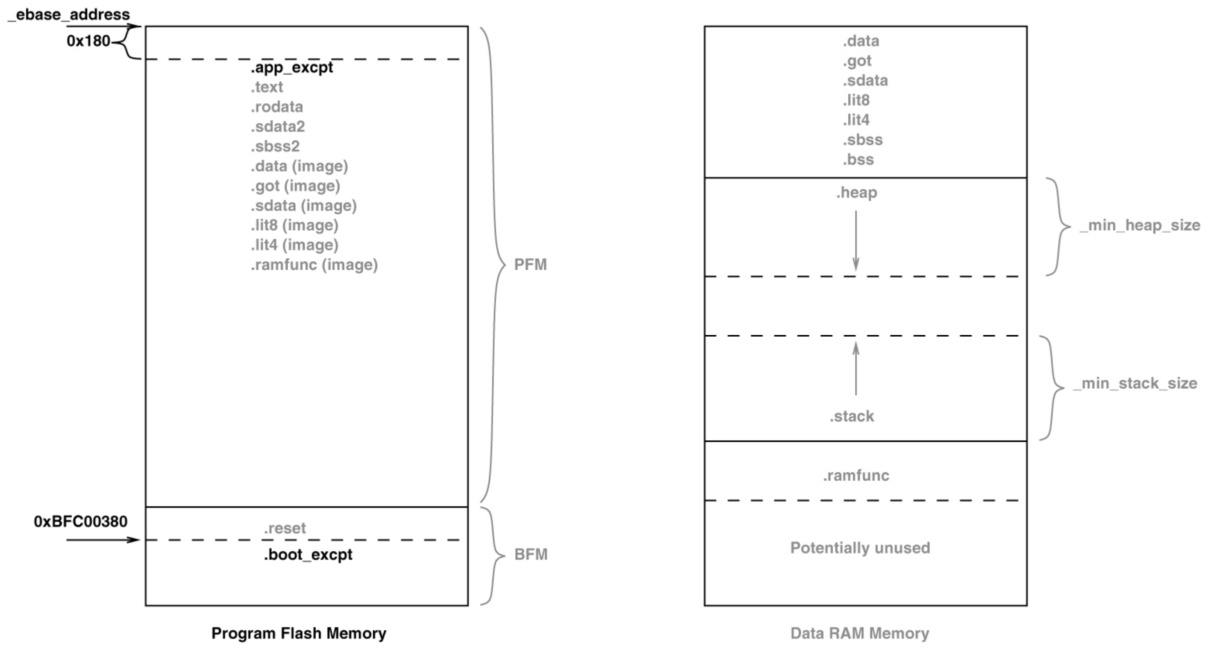
Symbol Name	Description
`_bmxdkpba_address`	The address to place into the `BMXDKPBA` register if `_ramfunc_length` is greater than 0.
`_bmxdudba_address`	The address to place into the `BMXDUDBA` register if `_ramfunc_length` is greater than 0.
`_bmxdupba_address`	The address to place into the `BMXDUPBA` register if `_ramfunc_length` is greater than 0.
`_ebase_address`	The location of `EBASE`.
`_end`	The end of data allocation.
`_gp`	Points to the “middle” of the small variables region. By convention this is 0x8000 bytes from the first location used for small variables.
`_heap`	The starting location of the heap in DRM.
`_ramfunc_begin`	The starting location of the RAM functions. This should be located at a 2K boundary as it is used to initialize the `BMXDKPBA` register.
`_ramfunc_length`	The length of the `.ramfunc` section.
`_stack`	The starting location of the stack in DRM. Remember that the stack grows from the bottom of data memory so this symbol should point to the bottom of the section allocated for the stack.
`_vector_spacing`	The initialization value for the vector spacing field in the `IntCtl` register.

12.3.11 Exceptions

In addition, two weak general exception handlers are provided that can be overridden by the application — one to handle exceptions when `StatusBEV` is 1 (`_bootstrap_exception_handler`), and one to handle exceptions when `StatusBEV` is 0 (`_general_exception_handler`). Both the weak Reset exception handler and the weak general exception handler provided with the start-up code enters an infinite loop. The start-up code arranges for a jump to the reset exception handler to be located at `0xBFC00380`, and a jump to the general exception handler to be located at `EBASE + 0x180`.

Both handlers must be attributed with the `nomips16` [e.g., `__attribute__ ((nomips16))`], since the start-up code jumps to these functions.

FIGURE 12-5: EXCEPTIONS



12.4 THE ON RESET ROUTINE

Some hardware configurations require special initialization, often within the first few instruction cycles after reset. To achieve this, there is a hook provided via the on reset routine.

This routine is called after initializing a minimum ‘C’ context. An empty weak version of this procedure (`_on_reset`) is provided with the start-up code. A stub for this routine can be found in `pic32-libs/libc/stubs` in the installation directory of your compiler.

Special consideration needs to be taken by the user if this procedure is written in ‘C’. Most importantly, statically allocated variables are not initialized (with either the specified initializer or a zero as required for uninitialized variables).The stack pointer has been initialized when this routine is called.

12.4.1 Clearing Objects

The runtime start-up code will clear all memory locations occupied by uninitialized variables so they will contain zero before `main()` is executed.

Variables whose contents should be preserved over a reset should use the `persistent` attribute, see **Section 6.10 “Standard Type Qualifiers”** for more information. Such variables are linked in a different area of memory and are not altered by the runtime start-up code in any way.

NOTES:

Chapter 13. Library Routines

13.1 USING LIBRARY ROUTINES

Library functions or routines (and any associated variables) will be automatically linked into a program once they have been referenced in your source code. The use of a function from one library file will not include any other functions from that library. Only used library functions will be linked into the program output and consume memory.

Your program will require declarations for any functions or symbols used from libraries. These are contained in the standard C header (.h) files. Header files are not library files and the two files types should not be confused. Library files contain precompiled code, typically functions and variable definitions; the header files provide declarations (as opposed to definitions) for functions, variables and types in the library files, as well as other preprocessor macros.

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

int main(void)
{
  double i;

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

NOTES:

Chapter 14. Mixing C/C++ and Assembly Language

14.1 INTRODUCTION

Assembly language code can be mixed with C/C++ code using two different techniques: writing assembly code and placing it into a separate assembler module, or including it as in-line assembly in a C/C++ module.This section describes how to use assembly language and C/C++ modules together. It gives examples of using C/C++ variables and functions in assembly code, and examples of using assembly language variables and functions in C/C++.

The more assembly code a project contains, the more difficult and time consuming its maintenance will be. As the project is developed, the compiler may work in different ways as some optimizations look at the entire program. The assembly code is more likely to fail if the compiler is updated due to differences in the way the updated compiler may work. These factors do not affect code written in C/C++

Note:	If assembly must be added, it is preferable to write this as self-contained routine in a separate assembly module rather than in-lining it in C code.

- Mixing Assembly Language and C Variables and Functions
- Using Inline Assembly Language
- Predefined Assembly Macros

14.2 MIXING ASSEMBLY LANGUAGE AND C VARIABLES AND FUNCTIONS

The following guidelines indicate how to interface separate assembly language modules with C modules.

- Follow the register conventions described in **Section 9.3 "Register Conventions"**. In particular, registers $4-$7 are used for parameter passing. An assembly -language function will receive parameters, and should pass arguments to called functions, in these registers.
- Table 9-1 "Register Conventions describes which registers must be saved across non-interrupt function calls
- 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. 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 14-1: MIXING C AND ASSEMBLY

```
/*
** file: ex1.S
*/
#include <xc.h>

    /* define which section (for example "text")
     * does this portion of code resides in. Typically,
     * all your code will reside in .text section as
     * shown below.
     */
   .text

    /*  This is important for an assembly programmer. This
     * directive tells the assembler that don't optimize
     * the order of the instructions as well as don't insert
     * 'nop' instructions after jumps and branches.
     */
   .set noreorder

/*********************************************************************
* asmFunction(int bits)
* This function clears the specified bites in IOPORT A.
********************************************************************/
.global asmFunction
.ent asmFunction
asmFunction:
   /* function prologue - save registers used in this function
     * on stack and adjust stack-pointer
     */
   addiu   sp, sp, -4
   sw      s0, 0(sp)

   la      s0, LATACLR
   sw      a0, 0(s0)      /* clear specified bits */

   la      s0, PORTA
   lw      s1, 0(s0)
   la      s0, cVariable
   sw      s1, 0(s0)      /* keep a copy */

   /* function epilogue - restore registers used in this function
     * from stack and adjust stack-pointer
     */
   lw      s0, 0(sp)
   addiu   sp, sp,

   addu    s1, ra, zero
   jal     foo
   nop
   addu    ra, s1, zero
   nop
   /* return to caller */
   jr      ra
   nop
.end asmFunction

  .bss
  .global asmVariable
  .align 2
asmVariable: .space 4
```

The file `ex1.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.c
;
#include <xc.h>
#include <plib.h>

extern void asmFunction(int bits);
extern unsigned int asmVariable;
volatile unsigned int cVariable = 0;
volatile unsigned int jak = 0;

int main(void) {
    SYSTEMConfigPerformance(80000000ull);
    TRISA = 0;
    LATA = 0xC6FFul;

    asmFunction(0xA55Au);
    while (1)
    {
        asmVariable++;
    }
}

void foo (void)
{
    jak++;
}
```

In the C file, `ex2.c`, external references to symbols declared in an assembly file are declared using the standard `extern` keyword; note that `asmFunction` is a `void` function and is declared accordingly.

In the assembly file, `ex1.S`, the symbols `asmFunction` and `asmVariable` are made globally visible through the use of the `.global` assembler directive and can be accessed by any other source file.

14.3 USING INLINE ASSEMBLY LANGUAGE

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

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

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

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

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

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

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

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

Constraint letters and modifiers supported by the compiler are listed in Table 14-1 through Table 14-4.

TABLE 14-1: REGISTER CONSTRAINT LETTERS SUPPORTED BY THE COMPILER

Letter	Constraint
`c`	A register suitable for use in an indirect jump
`d`	An address register. This is equivalent to `@code{r}` unless generating MIPS16 code
`ka`	Registers that can be used as the target of multiply-accumulate instructions
`l`	The `@code{lo}` register. Use this register to store values that are no bigger than a word
`x`	The concatenated `@code{hi}` and `@code{lo}` registers. Use this register to store double-word values

TABLE 14-2: INTEGER CONSTRAINT LETTERS SUPPORTED BY THE COMPILER

Letter	Constraint
`I`	A signed 32-bit constant (for arithmetic instructions)
`J`	Integer zero
`K`	An unsigned 32-bit constant (for logic instructions)
`L`	A signed 32-bit constant in which the lower 32 bits are zero. Such constants can be loaded using `@code{lui}`
`M`	A constant that cannot be loaded using `@code{lui}`, `@code{addiu}`or `@code{ori}`
`N`	A constant in the range -65535 to -1 (inclusive)
`O`	A signed 15-bit constant
`P`	A constant in the range 1 to 65535 (inclusive)

TABLE 14-3: GENERAL CONSTRAINT LETTERS SUPPORTED BY THE COMPILER

Letter	Constraint
`R`	An address that can be used in a non-macro load or store.

TABLE 14-4: CONSTRAINT MODIFIERS SUPPORTED BY THE COMPILER

Letter	Constraint
`=`	Means that this operand is write-only for this instruction: the previous value is discarded and replaced by output data
`+`	Means that this operand is both read and written by the instruction
`&`	Means that this operand is an `earlyclobber` operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address
`d`	Second register for operand number *n*, i.e., `%d`*n*..
`q`	Fourth register for operand number *n*, i.e., `%q`*n*..
`t`	Third register for operand number *n*, i.e., `%t`*n*..

Examples:

- Insert Bit Field
- Multiple Assembler Instructions

Insert Bit Field

This example demonstrates how to use the INS instruction to insert a bit field into a 32-bit wide variable. This function-like macro uses inline assembly to emit the INS instruction, which is not commonly generated from C/C++ code.

```
/* MIPS32r2 insert bits */
#define _ins(tgt,val,pos,sz) __extension__({                   \
    unsigned int __t = (tgt), __v = (val);                     \
    __asm__ ("ins %0,%z1,%2,%3"                 /* template */ \
             : "+d" (__t)                       /* output   */ \
             : "dJ" (__v), "I" (pos), "I" (sz)); /* input    */ \
    __t;                                                       \
})
```

Here __v, pos, and sz are input operands. The __v operand is constrained to be of type 'd' (an address register) or 'J' (integer zero). The pos and sz operands are constrained to be of type 'I' (a signed 32-bit constant).

The __t output operand is constrained to be of type 'd' (an address register). The '+' modifier means that this operand is both read and written by the instruction and so the operand is both an input and an output.

The following example shows this macro in use.

```
unsigned int result;
void example (void)
{
    unsigned int insertval = 0x12;
    result = 0xAAAAAAAAu;
    result = _ins(result, insertval, 4, 8);
    /* result is now 0xAAAAA12A */
}
```

For this example, the compiler may generate assembly code similar to the following.

```
    li     $2,-1431699456      # 0xaaaa0000
    ori    $2,$2,0xaaaa        # 0xaaaa0000 | 0xaaaa

    li     $3,18               # 0x12
    ins  $2,$3,4,8             # inline assembly

    lui    $3,%hi(result)      # assign the result back
    j      $31                 # return
    sw     $2,%lo(result)($3)
```

Multiple Assembler Instructions

This example demonstrates how to use the WSBH and ROTR instructions together for a byte swap. The WSBH instruction is a 32-bit byte swap within each of the two halfwords. The ROTR instruction is a rotate right by immediate. This function-like macro uses inline assembly to create a “byte-swap word” using instructions that are not commonly generated from C/C++ code.

The following shows the definition of the function-like macro, _bswapw.

```
/* MIPS32r2 byte-swap word */
#define _bswapw(x) __extension__({        \
    unsigned int __x = (x), __v;          \
    __asm__ ("wsbh %0,%1;\n\t"            \
               "rotr %0,16" /* template */ \
              : "=d" (__v)  /* output */   \
              : "d" (__x))  /* input*/ ;   \
    __v;                                   \
})
```

Here `__x` is the C expression for the input operand. The operand is constrained to be of type 'd', which denotes an address register.

The C expression `__v` is the output operand. This operand is also constrained to be of type 'd'. The '=' means that this operand is write-only for this instruction: the previous value is discarded and replaced by output data.

The function-like macro is shown in the following example assigning to `result` the content of `value`, swapped.

```
unsigned int result;
int example (void)
{
   unsigned int value = 0x12345678u;
   result = _bswapw(value);
   /* result == 0x78563412 */
}
```

The compiler may generate assembly code similar to the following for this example:

```
  li$2,305397760          # 0x12340000
  addiu$2,$2,22136        # 0x12340000 + 0x5678
  wsbh $2,$2;             # From inline asm
  rotr $2,16              # From inline asm
  lui$2,%hi(result)       # assign back to result
  j $31                   # return
  sw$3,%lo(result)($2)
```

14.3.1 Equivalent Assembly Symbols

C/C++ symbols can be accessed directly with no modification in extended assembly code.

14.4 PREDEFINED ASSEMBLY MACROS

Several predefined macros are available once you include <xc.h>. The exact operation of these macros is dependent on the instruction set employed. Table 14-5 shows general purpose predefined macros and their operation.

TABLE 14-5: PREDEFINED MACROS

Macro	Description
`_nop()`	Insert a No Operation instruction
`_ehb()`	Insert Execution Hazard Barrier instruction
`_sync()`	Insert Synchronize Shared Memory instruction
`_wait()`	Insert instruction to enter Standby mode
`_mfc0(rn, sel)`	See `<xc.h>` file
`_mtc0(rn, sel, v)`	See `<xc.h>` file
`_mxc0(rn, sel, v)`	See `<xc.h>` file
`_bcc0(rn, sel, clr)`	For the CP0 register specified by `rn` and `sel`, clear bits corresponding to those bits in `clr` which are non-zero
`_bsc0(rn, sel, set)`	For the CP0 register specified by `rn` and `sel`, clear bits corresponding to those bits in `clr` which are non-zero
`_bcsc0(rn, sel, clr, set)`	For the CP0 register specified by `rn` and `sel`, clear bits corresponding to those bits in `clr` which are non-zero, and set bits corresponding to those bits in set which are non-zero for the CP0 register specified by `rn` and `sel`, clear bits corresponding to those bits in `clr` which are non-zero, and set bits corresponding to those bits in set which are non-zero
`_clz(x)`	Count leading zeroes in `x`
`_ctz(x)`	Count trailing zeroes in x
`_clo(x)`	Count leading ones in x
`_dclz(x)`	Simulate 64-bit count leading zeroes in x
`_dclo(x)`	Simulate 64-bit count leading ones in x
`_dctz(x`	Simulate 64-bit count trailing zeroes in x
`_wsbh(x)`	See `<xc.h>` file
`_bswapw(x)`	See `<xc.h>` file
`_ins(tgt,val,pos,sz)`	See `<xc.h>` file
`_ext(x,pos,sz)`	See `<xc.h>` file
`_jr_hb()`	See `<xc.h>` file
`_wrpgpr(regno, val)`	See `<xc.h>` file
`_rdpgpr(regno)`	See `<xc.h>` file
`_get_byte(addr, errp)`	Return the least significant byte of addr
`_get_half(addr, errp)`	Return the least significant 16-bit word of addr
`_get_word(addr, errp)`	Return the least significant 32-bit word of addr
`_get_dword(addr, errp)`	Return the least significant 64-bit of addr

TABLE 14-5: PREDEFINED MACROS (CONTINUED)

Macro	Description
`_put_byte(addr, v)`	Write the least significant byte of addr with v
`_put_half(addr, v)`	Write the least significant 16-bit word of addr with v
`_put_word(addr, v)`	Write the least significant 32-bit word of addr with v
`_put_dword(addr, v)`	Write the least significant 64-bit word of addr with v

NOTES:

Chapter 15. Optimizations

15.1 INTRODUCTION

Different MPLAB XC32 C Compiler editions support different levels of optimization. Some editions are free to download and others must be purchased. Visit http://www.microchip.com/MPLABXCcompilers for more information on C and C++ licenses.

The compiler editions are:

Edition	Cost	Description
Professional (PRO)	Yes	Implemented with the highest optimizations and performance levels.
Standard (STD)	Yes	Implemented with ample optimizations levels and high performance levels.
Free	No	Implemented with the most code optimizations restrictions.
Evaluation (EVAL)	No	PRO edition enabled for 60 days and then reverts to Free edition.

Setting Optimization Levels

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

For details on compiler options used to set optimizations, see **Section 3.9.7 "Options for Controlling Optimization"**.

NOTES:

Chapter 16. Preprocessing

16.1 INTRODUCTION

All C/C++ source files are preprocessed before compilation. Assembly source files that use the .S extension (upper case) are also preprocessed. A large number of options control the operation of the preprocessor and preprocessed code, see **Section 3.9.8 "Options for Controlling the Preprocessor"**.

- C/C++ Language Comments
- Preprocessor Directives
- Pragma Directives
- Predefined Macros

16.2 C/C++ LANGUAGE COMMENTS

A C/C++ comment is ignored by the compiler and can be used to provide information to someone reading the source code. They should be used freely.

Comments may be added by enclosing the desired characters within `/*` and `*/`. The comment can run over multiple lines, but comments cannot be nested. Comments can be placed anywhere in C/C++ code, even in the middle of expressions, but cannot be placed in character constants or string literals.

Since comments cannot be nested, it may be desirable to use the `#if` preprocessor directive to comment out code that already contains comments, for example:

```
#if 0
   result = read();  /* TODO: Jim, check this function is right */
#endif
```

Single-line, C++ style comments may also be specified. Any characters following `//` to the end of the line are taken to be a comment and will be ignored by the compiler, as shown below:

```
   result = read();  // TODO: Jim, check this function is right
```

16.3 PREPROCESSOR DIRECTIVES

MPLAB XC32 C Compiler accepts all the standard preprocessor directives, which are listed in Table 16-1.

TABLE 16-1: PREPROCESSOR DIRECTIVES

Directive	Meaning	Example
`#`	Preprocessor null directive, do nothing	`#`
`#assert`	Generate error if condition false	`#assert SIZE > 10`

`#define`	Define preprocessor macro	`#define SIZE 5`
		`#define FLAG`
		`#define add(a,b) ((a)+(b))`

`#elif`	Short for `#else #if`	`see #ifdef`
`#else`	Conditionally include source lines	`see #if`

TABLE 16-1: PREPROCESSOR DIRECTIVES (CONTINUED)

Directive	Meaning	Example
`#endif`	Terminate conditional source inclusion	`see #if`
`#error`	Generate an error message	`#error Size too big`

`#if`	Include source lines if constant expression true	`#if SIZE < 10`
		`  c = process(10)`
		`#else`
		`  skip();`
		`#endif`

`#ifdef`	Include source lines if preprocessor symbol defined	`#ifdef FLAG`
		`  do_loop();`
		`#elif SIZE == 5`
		`  skip_loop();`
		`#endif`

`#ifndef`	Include source lines if preprocessor symbol not defined	`#ifndef FLAG`
		`  jump();`
		`#endif`

`#include`	Include text file into source	`#include <stdio.h>`
		`#include "project.h"`

`#line`	Specify line number and filename for listing	`#line 3 final`
#*nn*	(Where *nn* is a number) short for `#line` *nn*	`#20`
`#pragma`	Compiler specific options	`Refer to` **Section 16.4 "Pragma Directives"**
`#undef`	Undefines preprocessor symbol	`#undef FLAG`
`#warning`	Generate a warning message	`#warning Length not set`

Macro expansion using arguments can use the `#` character to convert an argument to a string, and the `##` sequence to concatenate arguments. If two expressions are being concatenated, consider using two macros in case either expression requires substitution itself, so for example,

```
#define   paste1(a,b)   a##b
#define   paste(a,b)    paste1(a,b)
```

lets you use the `paste` macro to concatenate two expressions that themselves may require further expansion. The replacement token is rescanned for more macro identifiers, but remember that once a particular macro identifier has been expanded, it will not be expanded again if it appears after concatenation.

The type and conversion of numeric values in the preprocessor domain is the same as in the C domain. Preprocessor values do not have a type, but acquire one as soon as they are converted by the preprocessor. Expressions may overflow their allocated type in the same way that C expressions may overflow.

Overflow may be avoided by using a constant suffix. For example, an `L` after the number indicates it should be interpreted as a long once converted.

So for example:

```
#define MAX 100000*100000
```

and

```
#define MAX 100000*100000L
```

(note the `L` suffix) will define the values 0x540be400 and 0x2540be400, respectively.

16.4 PRAGMA DIRECTIVES

There are certain compile-time directives that can be used to modify the behavior of the compiler. These are implemented through the use of the ANSI standard #`pragma` facility. Any pragma which is not understood by the compiler will be ignored.

The format of a pragma is:

```
#pragma keyword options
```

where *`keyword`* is one of a set of keywords, some of which are followed by certain *`options`*. A description of the keywords is given below.

`#pragma interrupt`

Mark a function as an interrupt handler. The prologue and epilogue code for the function will perform more extensive context preservation. Note that the `interrupt` attribute (rather than this pragma) is the recommended mechanism for marking a function as an interrupt handler. The interrupt pragma is provided for compatibility with other compilers. See **Chapter 11. “Interrupts”** and **Section 11.5 “Exception Handlers”**.

`#pragma vector`

Generate a branch instruction at the indicated exception vector which targets the function. Note that the vector attribute (rather than this pragma) is the recommended mechanism for generating an exception/interrupt vector. See **Chapter 11. “Interrupts”** and **Section 11.5 “Exception Handlers”**.

`#pragma config`

The #pragma config directive specifies the processor-specific configuration settings (i.e., Configuration bits) to be used by the application. See **Section 11.3.2 “Interrupt Pragma”**.

16.5 PREDEFINED MACROS

These predefined macros are available for use with the compiler:

- 32-Bit C/C++ Compiler Macros
- SDE Compatibility Macros

16.5.1 32-Bit C/C++ Compiler Macros

The compiler defines a number of macros, most with the prefix “_MCHP_,” which characterize the various target specific options, the target processor and other aspects of the host environment.C/C++

_MCHP_SZINT	32 or 64, depending on command line options to set the size of an integer (-mint32 -mint64)
_MCHP_SZLONG	32 or 64, depending on command line options to set the size of an integer (-mlong32 -mlong64)
_MCHP_SZPTR	32 always since all pointers are 32 bits
__mchp_no_float	Defined if -mno-float specified
__NO_FLOAT	Defined if -mno-float specified
__PIC__ __pic__	The translation unit is being compiled for position independent code
__PIC32MX __PIC32MX__	Always defined
__PIC32_FEATURE_SET__	The compiler predefines a macro based on the features available for the selected device. These macros are intended to be used when writing code to take advantage of features available on newer devices while maintaining compatibility with older devices. Examples: PIC32MX795F512L would use __PIC32_FEATURE_SET__ == 795, and PIC32MX340F128H would use __PIC32_FEATURE_SET__ == 340
PIC32MX	Defined if -ansi is not specified
__LANGUAGE_ASSEMBLY __LANGUAGE_ASSEMBLY__ _LANGUAGE_ASSEMBLY	Defined if compiling a pre-processed assembly file (.S files)
LANGUAGE_ASSEMBLY	Defined if compiling a pre-processed assembly file (.S files) and -ansi is not specified
__LANGUAGE_C __LANGUAGE_C__ _LANGUAGE_C	Defined if compiling a C file
LANGUAGE_C	Defined if compiling a C file and -ansi is not specified
__LANGUAGE_C_PLUS_PLUS __cplusplus _LANGUAGE_C_PLUS_PLUS__	Defined if compiling a C++ file
__EXCEPTIONS	Defined if X++ exceptions are enabled
__GXX_RTTI	Defined if runtime type information is enabled

__processor__	Where “processor” is the capitalized argument to the −mprocessor option. e.g., -mprocessor=32mx12f3456 will define __32MX12F3456__
__XC	Always defined to indicate this is a Microchip XC compiler
__XC32	Always defined to indicate this the XC32 compiler
__VERSION__	The __VERSION__ macro expands to a string constant describing the compiler in use. Do not rely on its contents having any particular form, but it should contain at least the release number. Use the __XC32_VERSION macro for a numeric version number
__XC32_VERSION or __C32_VERSION__	The C compiler defines the constant __XC32_VERSION, giving a numeric value to the version identifier. This macro can be used to construct applications that take advantage of new compiler features while still remaining backward compatible with older versions. The value is based upon the major and minor version numbers of the current release. For example, release version 1.03 will have a __XC32_VERSION definition of 1030. This macro can be used, in conjunction with standard preprocessor comparison statements, to conditionally include/exclude various code constructs

16.5.2 SDE Compatibility Macros

The MIPS® SDE (Software Development Environment) defines a number of macros, most with the prefix “_MIPS_,” which characterize various target specific options, some determined by command line options (e.g., `-mint64`). Where applicable, these macros will be defined by the compiler in order to ease porting applications and middleware from the SDE to the compiler.

`_MIPS_SZINT`	32 or 64, depending on command line options to set the size of an integer (`-mint32 -mint64`)
`_MIPS_SZLONG`	32 or 64, depending on command line options to set the size of an integer (`-mlong32 -mlong64`)
`_MIPS_SZPTR`	32 always since all pointers are 32 bits
`__mips_no_float`	Defined if `-mno-float` specified

`__mips__`
`_mips`
`_MIPS_ARCH_PIC32MX`
`_MIPS_TUNE_PIC32MX`
`_R3000`
`__R3000`
`__R3000__`
`__mips_soft_float`
`__MIPSEL`
`__MIPSEL__`
`_MIPSEL`
	Always defined

`R3000` `MIPSEL`	Defined if `-ansi` is not specified
`_mips_fpr`	Defined as 32
`__mips16`	Defined if `-mips16` specified
`__mips`	Defined as 32
`__mips_isa_rev`	Defined as 2
`_MIPS_ISA`	Defined as `_MIPS_ISA_MIPS32`

Chapter 17. Linking Programs

17.1 INTRODUCTION

The compiler will automatically invoke the linker unless the compiler has been requested to stop after producing an intermediate file.

Linker scripts are used to specify the available memory regions and where sections should be positioned in those regions.

The linker creates a map file which details the memory assigned to sections. The map file is the best place to look for memory information.

- Replacing Library Symbols
- Linker-Defined Symbols
- Default Linker Script

17.2 REPLACING LIBRARY SYMBOLS

Unlike with the Microchip MPLAB XC8 compiler, not all library functions can be replaced with user-defined routines using MPLAB XC32 C Compiler. Only weak library functions (see **Section 6.12 "Variable Attributes"**) can be replaced in this way. For those that are weak, any function you write in your code will replace an identically named function in the library files.

17.3 LINKER-DEFINED SYMBOLS

The 32-bit linker defines several symbols that may be used in your C code development. Please see the "*MPLAB® Assembler, Linker and Utilities for PIC32 MCUs User's Guide"*(DS51833) for more information.

The linker defines the symbols `_ramfunc_begin` and `_bmxdkpba_address`, which represent the starting address in RAM where ram functions will be accessed, and the corresponding address in the program memory from which the functions will be copied. They are used by the default runtime start-up code to initialize the bus matrix if ram functions exist in the project, see **Section 10.3 "Allocation of Function Code"**.

The linker also defines the symbol `_stack`, which is used by the runtime start-up code to initialize the stack pointer. This symbol represents the starting address for the software stack.

All the above symbols are rarely required for more programs, but may assist you if you are writing your own runtime start-up code.

17.4 DEFAULT LINKER SCRIPT

The default linker script is located in the `<install-directory>/pic32mx/lib/ldscripts/elf32pic32mx.x` file. When compiling with the xc32-gcc or xc32-g++ compilation driver, the linker uses this file as the default linker script. The driver passes the path to the default linker script using the `-T` linker option.

The default linker script contains the following categories of information:

- Output Format and Entry Points
- Default Values for Minimum Stack and Heap Sizes
- Processor Definitions Include File
  - Inclusion of Processor-Specific Object File(s)
  - OPTIONAL Inclusion of Processor-Specific Peripheral Libraries
  - Base Exception Vector Address and Vector Spacing Symbols
  - Memory Address Equates
  - Memory Regions
  - Configuration Words Input/Output Section Map
- Input/Output Section Map

Note: All addresses specified in the linker scripts should be specified as virtual addresses, not physical addresses.

17.4.1 Output Format and Entry Points

The first several lines of the default linker script define the output format and the entry point for the application. Copies of the default linker scripts are provided in `program files/.../<install-dir>/pic32mx/lib/ldscripts`.

```
OUTPUT_FORMAT("elf32-tradlittlemips")
OUTPUT_ARCH(pic32mx)
ENTRY(_reset)
```

The `OUTPUT_FORMAT` line selects the object file format for the output file. The output object file format generated by the 32-bit language tools is a traditional, little-endian, MIPS, ELF32 format.

The `OUTPUT_ARCH` line selects the specific machine architecture for the output file. The output files generated by the 32-bit language tools contain information that identifies the file was generated for the PIC32MX architecture.

The `ENTRY` line selects the entry point of the application. This is the symbol identifying the location of the first instruction to execute. The 32-bit language tools begins execution at the instruction identified by the `_reset` label.

17.4.2 Default Values for Minimum Stack and Heap Sizes

The next section of the default linker script provides default values for the minimum stack and heap sizes.

```
/*
 * Provide for a minimum stack and heap size
 * - _min_stack_size - represents the minimum space that must
 *                     be made available for the stack. Can
 *                     be overridden from the command line
 *                     using the linker's --defsym option.
 * - _min_heap_size  - represents the minimum space that must
 *                     be made available for the heap. Can
 *                     be overridden from the command line
 *                     using the linker's --defsym option.
 */
EXTERN (_min_stack_size _min_heap_size)
PROVIDE(_min_stack_size = 0x400) ;
PROVIDE(_min_heap_size = 0) ;
```

The EXTERN line ensures that the rest of the linker script has access to the default values of _min_stack_size and _min_heap_size assuming that the user does not override these values using the linker's --defsym command line option.

The two PROVIDE lines ensure that a default value is provided for both _min_stack_size and _min_heap_size. The default value for the minimum stack size is 1024 bytes (0x400). The default value for the minimum heap size is 0 bytes.

17.4.3 Processor Definitions Include File

The next line in the default linker script pulls in information specific to the processor.

```
INCLUDE procdefs.ld
```

The file procdefs.ld is included in the linker script at this point. The file is searched for in the current directory and in any directory specified with the -L command line option. The compiler shell ensures that the correct directory is passed to the linker with the -L command line option based on the processor selected with the -mprocessor command line option.

The processor definitions linker script contains the following pieces of information:

- Inclusion of Processor-Specific Object File(s)
- Base Exception Vector Address and Vector Spacing Symbols
- Memory Address Equates
- Memory Regions
- Configuration Words Input/Output Section Map

17.4.3.1 INCLUSION OF PROCESSOR-SPECIFIC OBJECT FILE(S)

This section of the processor definitions linker script ensures that the processor-specific object file(s) get included in the link.

```
/*************************************************************
 * Processor-specific object file. Contains SFR definitions.
 *************************************************************/
INPUT(“processor.o”)
```

The INPUT line specifies that processor.o should be included in the link as if this file were named on the command line. The linker attempts to find this file in the current directory. If it is not found, the linker searches through the library search paths (i.e., the paths specified with the -L command line option).

17.4.3.2 OPTIONAL INCLUSION OF PROCESSOR-SPECIFIC PERIPHERAL LIBRARIES

This section of the processor definitions linker script ensures that the processor-specific peripheral libraries get included, but only if the files exist.

```
/***********************************************************************
 * Processor-specific peripheral libraries are optional

***********************************************************************/
OPTIONAL("libmchp_peripheral.a")
OPTIONAL("libmchp_peripheral_32MX795F512L.a")
```

The OPTIONAL lines specify that `libmchp_peripheral.a` and `libmchp_peripheral_32MX795F512L.a` should be included in the link as if the files were named on the command line. The linker attempts to find these files in the current directory. If they are not found in the current directory, the linker searches through the library search paths. If they are not found in the library search paths, the link process continues without error. The linker will error only when a symbol from the peripheral library is required but not found elsewhere.

17.4.3.3 BASE EXCEPTION VECTOR ADDRESS AND VECTOR SPACING SYMBOLS

This section of the processor definitions linker script defines values for the base exception vector address and vector spacing.

```
/****************************************************************
 * For interrupt vector handling
 ****************************************************************/
_vector_spacing= 0x00000001;
_ebase_address= 0x9FC01000;
```

The first line defines a value of `1` for `_vector_spacing`. The available memory for exceptions only supports a vector spacing of `1`. The second line defines the location of the base exception vector address (`EBASE`).

On some devices, the base exception vector address is located in the KSEG0 boot segment. On other devices, the size of the KSEG0 boot segment is not sufficient for the vector table, so the base exception vector address is located in the KSEG0 program segment. In general, devices with at least 12 KB in the KSEG0 boot segment use the boot flash for the exception vector table. Devices with less than 12 KB in the KSEG0 boot segment use the KSEG0 program segment for the exception vector table. Be sure to check the `procdefs.ld` include file for the default address for your target device.

17.4.3.4 MEMORY ADDRESS EQUATES

This section of the processor definitions linker script provides information about certain memory addresses required by the default linker script.

```
/**************************************************************
 * Memory Address Equates
 **************************************************************/
_RESET_ADDR      = 0xBFC00000;
_BEV_EXCPT_ADDR  = 0xBFC00380;
_DBG_EXCPT_ADDR  = 0xBFC00480;
_DBG_CODE_ADDR   = 0xBFC02000;
_GEN_EXCPT_ADDR  = _ebase_address + 0x180;
```

The `_RESET_ADDR` defines the processor's Reset address. This is the virtual begin address of the IFM boot section in Kernel mode.

The `_BEV_EXCPT_ADDR` defines the address that the processor jumps to when an exception is encountered and $\mathtt{Status}_{BEV} = 1$.

The `_DBG_EXCPT_ADDR` defines the address that the processor jumps to when a debug exception is encountered.

The `_DBG_CODE_ADDR` defines the address that is the start address of the debug executive. Note that this address may vary depending on the size of the `KSEG0` boot segment on your target device.

The `_GEN_EXCPT_ADDR` defines the address that the processor jumps to when an exception is encountered and $\mathtt{Status}_{BEV} = 0$.

17.4.3.5 MEMORY REGIONS

This section of the processor definitions linker script provides information about the memory regions that are available on the device.

```
/**************************************************************
 * Memory Regions
 *
 * Memory regions without attributes cannot be used for
 * orphaned sections. Only sections specifically assigned to
 * these regions can be allocated into these regions.
 **************************************************************/
MEMORY
{
  kseg0_program_mem (rx) : ORIGIN = 0x9D000000, LENGTH = 0x8000
  kseg0_boot_mem         : ORIGIN = 0x9FC00490, LENGTH = 0x970
  exception_mem          : ORIGIN = 0x9FC01000, LENGTH = 0x1000
  kseg1_boot_mem         : ORIGIN = 0xBFC00000, LENGTH = 0x490
  debug_exec_mem         : ORIGIN = 0xBFC02000, LENGTH = 0xFF0
  config3                : ORIGIN = 0xBFC02FF0, LENGTH = 0x4
  config2                : ORIGIN = 0xBFC02FF4, LENGTH = 0x4
  config1                : ORIGIN = 0xBFC02FF8, LENGTH = 0x4
  config0                : ORIGIN = 0xBFC02FFC, LENGTH = 0x4
  kseg1_data_mem (w!x)   : ORIGIN = 0xA0000000, LENGTH = 0x2000
  sfrs                   : ORIGIN = 0xBF800000, LENGTH = 0x10000
}
```

Eleven memory regions are defined with an associated start address and length:

1. Program memory region (`kseg0_program_mem`) for application code
2. Boot memory regions (`kseg0_boot_mem` and `kseg1_boot_mem`)
3. Exception memory region (`exception_mem`)
4. Debug executive memory region (`debug_exec_mem`)
5. Configuration memory regions (`config3`, `config2`, `config1`, and `config0`)

6. Data memory region (`kseg1_data_mem`)
7. SFR memory region (`sfrs`)

The default linker script uses these names to locate sections into the correct regions. Sections which are non-standard become orphaned sections. The attributes of the memory regions are used to locate these orphaned sections. The attributes (`rx`) specify that read-only sections or executable sections can be located into the program memory regions. Similarly, the attributes (`w!x`) specify that sections that are not read-only and not executable can be located in the data memory region. Since no attributes are specified for the boot memory region, the configuration memory regions, or the SFR memory region, only specified sections may be located in these regions (i.e., orphaned sections may not be located in the boot memory regions, the exception memory region, the configuration memory regions, the debug executive memory region, or the SFR memory region).

17.4.3.6 CONFIGURATION WORDS INPUT/OUTPUT SECTION MAP

The last section in the processor definitions linker script is the input/output section map for Configuration Words. This section map is additive to the Input/Output Section Map found in the default linker script (see **Section 17.4.4 "Input/Output Section Map"**). It defines how input sections for Configuration Words are mapped to output sections for Configuration Words. Note that input sections are portions of an application that are defined in source code, while output sections are created by the linker. Generally, several input sections may be combined into a single output section. All output sections are specified within a `SECTIONS` command in the linker script.

For each Configuration Word that exists on the specific processor, a distinct output section named `.config_`*`address`* exists where address is the location of the Configuration Word in memory. Each of these sections contains the data created by the `#pragma config` directive (see **Section 16.4 "Pragma Directives"**) for that Configuration Word. Each section is assigned to their respective memory region (`config`*`n`*).

```
SECTIONS
{
  .config_BFC02FF0 : {
    *(.config_BFC02FF0)
  } > config3
  .config_BFC02FF4 : {
    *(.config_BFC02FF4)
  } > config2
  .config_BFC02FF8 : {
    *(.config_BFC02FF8)
  } > config1
  .config_BFC02FFC : {
    *(.config_BFC02FFC)
  } > config0
}
```

17.4.4 Input/Output Section Map

The last section in the default linker script is the input/output section map. The section map is the heart of the linker script. It defines how input sections are mapped to output sections. Note that input sections are portions of an application that are defined in source code, while output sections are created by the linker. Generally, several input sections may be combined into a single output section. All output sections are specified within a `SECTIONS` command in the linker script.

The following output sections may be created by the linker:

- .reset Section
- .bev_excpt Section
- .dbg_excpt Section
- .dbg_code Section
- .app_excpt Section
- .vector_0 .. .vector_63 Sections
- .start-up Section
- .text Section
- .rodata Sectionn
- .sdata2 Section
- .sbss2 Section
- .dbg_data Section
- .data Section
- .got Section
- .sdata Section
- .lit8 Section
- .lit4 Section
- .sbss Section
- .bss Section
- .heap Section
- .stack Section
- .ramfunc Section
- Stack Location
- Debug Sections

17.4.4.1 .RESET SECTION

This section contains the code that is executed when the processor performs a Reset. This section is located at the Reset address (`_RESET_ADDR`), as specified in the processor definitions linker script and is assigned to the boot memory region (`kseg1_boot_mem`). The `.reset` output section also contains the C start-up code from the `.reset.startup` input section.

```
.reset _RESET_ADDR :
{
  KEEP(*(.reset))
  KEEP(*(.reset.startup))
 } > kseg1_boot_mem
```

17.4.4.2 .BEV_EXCPT SECTION

This section contains the handler for exceptions that occur when $\mathtt{Status}_{\mathtt{BEV}}$ = 1. This section is located at the BEV exception address (_BEV_EXCPT_ADDR) as specified in the processor definitions linker script and is assigned to the boot memory region (kseg1_boot_mem).

```
.bev_excpt _BEV_EXCPT_ADDR :
{
  (*(.bev_handler))
} > kseg1_boot_mem
```

17.4.4.3 .DBG_EXCPT SECTION

This section reserves space for the debug exception vector. This section is only allocated if the symbol _DEBUGGER has been defined. (This symbol is defined if the -mdebugger command line option is specified to the shell.) This section is located at the debug exception address (_DBG_EXCPT_ADDR) as specified in the processor definitions linker script and is assigned to the boot memory region (kseg1_boot_mem). The section is marked as NOLOAD as it is only intended to ensure that application code cannot be placed at locations reserved for the debug executive.

```
.dbg_excpt _DBG_EXCPT_ADDR (NOLOAD) :
{
  . += (DEFINED (_DEBUGGER) ? 0x8 : 0x0);
} > kseg1_boot_mem
```

17.4.4.4 .DBG_CODE SECTION

This section reserves space for the debug exception handler. This section is only allocated if the symbol _DEBUGGER has been defined. (This symbol is defined if the -mdebugger command line option is specified to the shell.) This section is located at the debug code address (_DBG_CODE_ADDR) as specified in the processor definitions linker script and is assigned to the debug executive memory region (debug_exec_mem). The section is marked as NOLOAD because it is only intended to ensure that application code cannot be placed at locations reserved for the debug executive.

```
.dbg_code _DBG_CODE_ADDR (NOLOAD) :
{
  . += (DEFINED (_DEBUGGER) ? 0xFF0 : 0x0);
} > debug_exec_mem
```

17.4.4.5 .APP_EXCPT SECTION

This section contains the handler for exceptions that occur when $\mathtt{Status}_{\mathtt{BEV}}$ = 0. This section is located at the general exception address (_GEN_EXCPT_ADDR) as specified in the processor definitions linker script and is assigned to the exception memory region (exception_mem).

```
.app_excpt _GEN_EXCPT_ADDR :
{
  KEEP(*(.gen_handler))
} > exception_mem
```

17.4.4.6 .VECTOR_0 .. .VECTOR_63 SECTIONS

These sections contain the handler for each of the interrupt vectors. These sections are located at the correct vectored addresses using the formula:

```
_ebase_address + 0x200 + (_vector_spacing << 5) * n
```

where `n` is the respective vector number.

Each of the sections is followed by an assert that ensures the code located at the vector does not exceed the vector spacing specified.

```
.vector_n _ebase_address + 0x200 + (_vector_spacing << 5) * n :
  {
    KEEP(*(.vector_n))
  } > exception_mem
ASSERT (SIZEOF(.vector_n) < (_vector_spacing << 5), "function at
exception vector n too large")
```

17.4.4.7 .START-UP SECTION

In XC32, the C and C++ startup code is located in the `.reset` section. We maintain the `.startup` output section in the default linker script for backwards compatibility purposes only.

```
.startup ORIGIN(kseg0_boot_mem) :
  {
    *(.startup)
  } > kseg0_boot_mem
```

17.4.4.8 .TEXT SECTION

The standard executable code sections are no longer mapped to the `.text` output section. However, a few special executable sections are still mapped here as shown below. This section is assigned to the program memory region (`kseg0_program_mem`) and has a fill value of `NOP (0)`.

The built-in linker script no longer maps standard `.text` executable code input sections. By not mapping these sections in the linker script, we allow these sections to be allocated using the best-fit allocator rather than the sequential allocator. Sections that are unmapped in the linker script, can flow around absolute sections specified in code whereas sections that are linker-script mapped are grouped together and allocated sequentially, potentially causing conflicts with absolute sections (using the address function attribute).

```
.text ORIGIN(kseg0_program_mem) :
{
*(.stub .gnu.linkonce.t.*)
  KEEP (*(.text.*personality*))
  *(.gnu.warning)
  *(.mips16.fn.*)
  *(.mips16.call.*)
} > kseg0_program_mem =0
```

17.4.4.9 C++ INITIALIZATION SECTIONS

The sections `.init`, `.preinit_array`, `.init_array`, `.fini_array`, `.ctors`, and `.dtors` are all used for the construction and destruction of file-scope static-storage C++ objects.

```
/* Global-namespace object initialization */
.init   :
{
  KEEP (*crti.o(.init))
  KEEP (*crtbegin.o(.init))
  KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o *crtn.o ).init))
  KEEP (*crtend.o(.init))
  KEEP (*crtn.o(.init))
  . = ALIGN(4) ;
} >kseg0_program_mem
.fini   :
{
  KEEP (*(.fini))
  . = ALIGN(4) ;
} >kseg0_program_mem
.preinit_array   :
{
  PROVIDE_HIDDEN (__preinit_array_start = .);
  KEEP (*(.preinit_array))
  PROVIDE_HIDDEN (__preinit_array_end = .);
  . = ALIGN(4) ;
} >kseg0_program_mem
.init_array   :
{
  PROVIDE_HIDDEN (__init_array_start = .);
  KEEP (*(SORT(.init_array.*)))
  KEEP (*(.init_array))
  PROVIDE_HIDDEN (__init_array_end = .);
  . = ALIGN(4) ;
} >kseg0_program_mem
.fini_array   :
{
  PROVIDE_HIDDEN (__fini_array_start = .);
  KEEP (*(SORT(.fini_array.*)))
  KEEP (*(.fini_array))
  PROVIDE_HIDDEN (__fini_array_end = .);
  . = ALIGN(4) ;
} >kseg0_program_mem
.ctors   :
{
  /* XC32 uses crtbegin.o to find the start of
     the constructors, so we make sure it is
     first.  Because this is a wildcard, it
     doesn't matter if the user does not
     actually link against crtbegin.o; the
     linker won't look for a file to match a
     wildcard.  The wildcard also means that it
     doesn't matter which directory crtbegin.o
     is in.  */
  KEEP (*crtbegin.o(.ctors))
  KEEP (*crtbegin?.o(.ctors))
  /* We don't want to include the .ctor section from
     the crtend.o file until after the sorted ctors.
     The .ctor section from the crtend file contains the
     end of ctors marker and it must be last */
```

```
    KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o ) .ctors))
    KEEP (*(SORT(.ctors.*)))
    KEEP (*(.ctors))
    . = ALIGN(4) ;
  } >kseg0_program_mem
  .dtors   :
  {
    KEEP (*crtbegin.o(.dtors))
    KEEP (*crtbegin?.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o *crtend?.o ) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*(.dtors))
    . = ALIGN(4) ;
  } >kseg0_program_mem
```

Note: The order of the input sections within each output section is significant.

17.4.4.10 .RODATA SECTION

Standard read-only sections are not mapped in the linker script. A few special read-only sections are still mapped in the linker script, but most sections are unmapped, allowing them to be handled by the best fit allocator. This section is assigned to the program memory region (`kseg0_program_mem`).

```
.rodata   :
{
  *(.gnu.linkonce.r.*)
  *(.rodata1)
} > kseg0_program_mem
```

17.4.4.11 .SDATA2 SECTION

This section collects the small initialized constant global and static data from all of the application’s input files. Because of the constant nature of the data, this section is also a read-only section. This section is assigned to the program memory region (`kseg0_program_mem`).

```
/*
 * Small initialized constant global and static data can be
 * placed in the .sdata2 section. This is different from
 * .sdata, which contains small initialized non-constant
 * global and static data.
 */
.sdata2   :
{
  *(.sdata2 .sdata2.* .gnu.linkonce.s2.*)
} > kseg0_program_mem
```

17.4.4.12 .SBSS2 SECTION

This section collects the small uninitialized constant global and static data from all of the application's input files. Because of the constant nature of the data, this section is also a read-only section. This section is assigned to the program memory region (kseg0_program_mem).

```
/*
 * Uninitialized constant global and static data (i.e.,
 * variables which will always be zero). Again, this is
 * different from .sbss, which contains small non-initialized,
 * non-constant global and static data.
 */
.sbss2   :
{
  *(.sbss2 .sbss2.* .gnu.linkonce.sb2.*)
} > kseg0_program_mem
```

17.4.4.13 .DBG_DATA SECTION

This section reserves space for the data required by the debug exception handler. This section is only allocated if the symbol _DEBUGGER has been defined. (This symbol is defined if the -mdebugger command line option is specified to the shell.) This section is assigned to the data memory region (kseg1_data_mem). The section is marked as NOLOAD as it is only intended to ensure that application data cannot be placed at locations reserved for the debug executive.

```
.dbg_data (NOLOAD) :
{
  . += (DEFINED (_DEBUGGER) ? 0x200 : 0x0);
} > kseg1_data_mem
```

17.4.4.14 .DATA SECTION

The linker generates a data-initialization template that the C start-up code uses to initialize variables.

17.4.4.15 .GOT SECTION

This section collects the global offset table from all of the application's input files. This section is assigned to the data memory region (kseg1_data_mem) with a load address located in the program memory region (kseg0_program_mem). A symbol is defined to represent the location of the Global Pointer (_gp).

```
  _gp = ALIGN(16) + 0x7FF0 ;
  .got   :
  {
     *(.got.plt) *(.got)
  } > kseg1_data_mem AT> kseg0_program_mem
```

17.4.4.16 .SDATA SECTION

This section collects the small initialized data from all of the application's input files. This section is assigned to the data memory region (`kseg1_data_mem`) with a load address located in the program memory region (`kseg0_program_mem`). Symbols are defined to represent the virtual begin (`_sdata_begin`) and end (`_sdata_end`) addresses of this section.

```
/*
 * We want the small data sections together, so
 * single-instruction offsets can access them all, and
 * initialized data all before uninitialized, so
 * we can shorten the on-disk segment size.
 */
.sdata   :
{
  _sdata_begin = . ;
  *(.sdata .sdata.* .gnu.linkonce.s.*)
  _sdata_end = . ;
} > kseg1_data_mem AT> kseg0_program_mem
```

17.4.4.17 .LIT8 SECTION

This section collects the 8-byte constants which the assembler decides to store in memory rather than in the instruction stream from all of the application's input files. This section is assigned to the data memory region (`kseg1_data_mem`) with a load address located in the program memory region (`kseg0_program_mem`).

```
.lit8           :
{
  *(.lit8)
} > kseg1_data_mem AT> kseg0_program_mem
```

17.4.4.18 .LIT4 SECTION

This section collects the 4-byte constants which the assembler decides to store in memory rather than in the instruction stream from all of the application's input files. This section is assigned to the data memory region (`kseg1_data_mem`) with a load address located in the program memory region (`kseg0_program_mem`). A symbol is defined to represent the virtual end address of the initialized data (`_data_end`).

```
  .lit4           :
  {
    *(.lit4)
  } > kseg1_data_mem AT> kseg0_program_mem
  _data_end = . ;
```

17.4.4.19 .SBSS SECTION

This section collects the small uninitialized data from all of the application's input files. This section is assigned to the data memory region (kseg1_data_mem). A symbol is defined to represent the virtual begin address of uninitialized data (_bss_begin). Symbols are also defined to represent the virtual begin (_sbss_begin) and end (_sbss_end) addresses of this section.

```
  _bss_begin = . ;
  .sbss   :
  {
    _sbss_begin = . ;
    *(.dynsbss)
    *(.sbss .sbss.* .gnu.linkonce.sb.*)
    *(.scommon)
    _sbss_end = . ;
  } > kseg1_data_mem
```

17.4.4.20 .BSS SECTION

This section collects the uninitialized data from all of the application's input files. This section is assigned to the data memory region (kseg1_data_mem). A symbol is defined to represent the virtual end address of uninitialized data (_bss_end). A symbol is also to represent the virtual end address of data memory (_end).

```
.bss   :
{
  *(.dynbss)
  *(.bss .bss.* .gnu.linkonce.b.*)
  *(COMMON)
  /*
   * Align here to ensure that the .bss section occupies
   * space up to _end.  Align after .bss to ensure correct
   * alignment even if the .bss section disappears because
   * there are no input sections.
   */
  . = ALIGN(32 / 8) ;
} > kseg1_data_mem
. = ALIGN(32 / 8) ;
_end = . ;
_bss_end = . ;
```

17.4.4.21 .HEAP SECTION

The linker now dynamically reserves an area of memory for the heap. The .heap section is no longer mapped in the linker script. The linker finds the largest unused gap of memory after all other sections are allocated and uses that gap for both the heap and the stack. The minimum amount of space reserved for the heap is determined by the symbol _min_heap_size.

17.4.4.22 .STACK SECTION

The linker now dynamically reserves an area of memory for the stack. The .stack section is no longer mapped in the linker script. The linker finds the largest unused gap of memory after all other sections are allocated and uses that gap for both the heap and the stack. The minimum amount of space reserved for the stack is determined by the symbol _min_stack_size.

17.4.4.23 .RAMFUNC SECTION

The linker now dynamically collects the 'ramfunc' attributed and "`.ramfunc`" named sections and allocates them sequentially in an appropriate range of memory. The first ramfunc attributed function is placed at the highest appropriately aligned address.

The presence of a ramfunc section causes the linker to emit the symbols necessary for the crt0.S start-up code to initialize the PIC32 bus matrix appropriately.

```
/*
 * RAM functions go at the end of our stack and heap allocation.
 * Alignment of 2K required by the boundary register (BMXDKPBA).
 *
 * RAM functions are now allocated by the linker. The linker generates
 * _ramfunc_begin and _bmxdkpba_address symbols depending on the
 * location of RAM functions.
 */

_bmxdudba_address = LENGTH(kseg1_data_mem) ;
_bmxdupba_address = LENGTH(kseg1_data_mem) ;
```

17.4.4.24 STACK LOCATION

A symbol is defined to represent the location of the Stack Pointer (`_stack`). As described previously, the heap and the stack are now allocated to the largest available gap of memory after other sections have been allocated.

For PIC32 devices with more than 64K of data memory, GP relative addressing mode should not be used. To avoid conflict of using GP-relative addressing to the linker generated symbols, allocate the symbols in section "`_linkergenerated`": `extern unsigned int __attribute__((section("_linkergenerated"))) _splim;`

17.4.4.25 DEBUG SECTIONS

The debug sections contain DWARF2 debugging information. They are not loaded into program Flash.

```
  /* Stabs debugging sections.  */
  .stab          0 : { *(.stab) }
  .stabstr       0 : { *(.stabstr) }
  .stab.excl     0 : { *(.stab.excl) }
  .stab.exclstr  0 : { *(.stab.exclstr) }
  .stab.index    0 : { *(.stab.index) }
  .stab.indexstr 0 : { *(.stab.indexstr) }
  .comment       0 : { *(.comment) }
  /* DWARF debug sections.
     Symbols in the DWARF debugging sections are relative to the 
beginning
     of the section so we begin them at 0.  */
  /* DWARF 1 */
  .debug          0 : { *(.debug) }
  .line           0 : { *(.line) }
  /* GNU DWARF 1 extensions */
  .debug_srcinfo  0 : { *(.debug_srcinfo) }
  .debug_sfnames  0 : { *(.debug_sfnames) }
  /* DWARF 1.1 and DWARF 2 */
  .debug_aranges  0 : { *(.debug_aranges) }
  .debug_pubnames 0 : { *(.debug_pubnames) }
  /* DWARF 2 */
  .debug_info     0 : { *(.debug_info .gnu.linkonce.wi.*) }
  .debug_abbrev   0 : { *(.debug_abbrev) }
  .debug_line     0 : { *(.debug_line) }
  .debug_frame    0 : { *(.debug_frame) }
  .debug_str      0 : { *(.debug_str) }
  .debug_loc      0 : { *(.debug_loc) }
  .debug_macinfo  0 : { *(.debug_macinfo) }
  /* SGI/MIPS DWARF 2 extensions */
  .debug_weaknames 0 : { *(.debug_weaknames) }
  .debug_funcnames 0 : { *(.debug_funcnames) }
  .debug_typenames 0 : { *(.debug_typenames) }
  .debug_varnames  0 : { *(.debug_varnames) }
  .debug_pubtypes 0 : { *(.debug_pubtypes) }
  .debug_ranges   0 : { *(.debug_ranges) }
  /DISCARD/ : { *(.rel.dyn) }
  .gnu.attributes 0 : { KEEP (*(.gnu.attributes)) }
  /DISCARD/ : { *(.note.GNU-stack) }
  /DISCARD/ : { *(.note.GNU-stack) *(.gnu_debuglink) *(.gnu.lto_*) 
*(.discard) }
```

Appendix A. Implementation-Defined Behavior

A.1 INTRODUCTION

This chapter discusses the choices for implementation defined behavior in compiler.

A.2 HIGHLIGHTS

Items discussed in this chapter are:

- Overview
- Translation
- Environment
- Identifiers
- Characters
- Integers
- Floating-Point
- Arrays and Pointers
- Hints
- Structures, Unions, Enumerations, and Bit fields
- Qualifiers
- Declarators
- Statements
- Pre-Processing Directives
- Library Functions
- Architecture

A.3 OVERVIEW

ISO C requires a conforming implementation to document the choices for behaviors defined in the standard as “implementation-defined.” The following sections list all such areas, the choices made for the compiler, and the corresponding section number from the ISO/IEC 9899:1999 standard.

A.4 TRANSLATION

ISO Standard: "How a diagnostic is identified (3.10, 5.1.1.3)."

Implementation: All output to `stderr` is a diagnostic.

ISO Standard: "Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2)."

Implementation: Each sequence of whitespace is replaced by a single character.

A.5 ENVIRONMENT

ISO Standard: "The name and type of the function called at program start-up in a freestanding environment (5.1.2.1)."

Implementation: `int main` (`void`);

ISO Standard: "The effect of program termination in a freestanding environment (5.1.2.1)."

Implementation: An infinite loop (branch to self) instruction will be executed.

ISO Standard: "An alternative manner in which the `main` function may be defined (5.1.2.2.1)."

Implementation: `int main` (void);

ISO Standard: "The values given to the strings pointed to by the `argv` argument to `main` (5.1.2.2.1)."

Implementation: No arguments are passed to `main`. Reference to `argc` or `argv` is undefined.

ISO Standard: "What constitutes an interactive device (5.1.2.3)."

Implementation: Application defined.

ISO Standard: "Signals for which the equivalent of `signal(sig, SIG_IGN);` is executed at program start-up (7.14.1.1)."

Implementation: Signals are application defined.

ISO Standard: "The form of the status returned to the host environment to indicate unsuccessful termination when the `SIGABRT` signal is raised and not caught (7.20.4.1)."

Implementation: The host environment is application defined.

ISO Standard: "The forms of the status returned to the host environment by the `exit` function to report successful and unsuccessful termination (7.20.4.3)."

Implementation: The host environment is application defined.

ISO Standard: "The status returned to the host environment by the `exit` function if the value of its argument is other than zero, `EXIT_SUCCESS`, or `EXIT_FAILURE` (7.20.4.3)."

Implementation: The host environment is application defined.

ISO Standard: "The set of environment names and the method for altering the environment list used by the `getenv` function (7.20.4.4)."

Implementation: The host environment is application defined.

ISO Standard: "The manner of execution of the string by the system function (7.20.4.5)."

Implementation: The host environment is application defined.

A.6 IDENTIFIERS

ISO Standard: “Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).”

Implementation: No.

ISO Standard: “The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).”

Implementation: All characters are significant.

A.7 CHARACTERS

ISO Standard: “The number of bits in a byte (C90 3.4, C99 3.6).”

Implementation: 8.

ISO Standard: “The values of the members of the execution character set (C90 and C99 5.2.1).”

ISO Standard: “The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (C90 and C99 5.2.2).”

Implementation: The execution character set is ASCII.

ISO Standard: “The value of a char object into which has been stored any character other than a member of the basic execution character set (C90 6.1.2.5, C99 6.2.5).”

Implementation: The value of the char object is the 8-bit binary representation of the character in the source character set. That is, no translation is done.

ISO Standard: “Which of signed char or unsigned char has the same range, representation, and behavior as “plain” char (C90 6.1.2.5, C90 6.2.1.1, C99 6.2.5, C99 6.3.1.1).”

Implementation: By default, signed char is functionally equivalent to plain char. The options `-funsigned-char` and `-fsigned-char` can be used to change the default.

ISO Standard: “The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (C90 6.1.3.4, C99 6.4.4.4, C90 and C99 5.1.1.2).”

Implementation: The binary representation of the source character set is preserved to the execution character set.

ISO Standard: “The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (C90 6.1.3.4, C99 6.4.4.4).”

Implementation: The compiler determines the value for a multi-character character constant one character at a time. The previous value is shifted left by eight, and the bit pattern of the next character is masked in. The final result is of type `int`. If the result is larger than can be represented by an `int`, a warning diagnostic is issued and the value truncated to `int` size.

ISO Standard: “The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (C90 6.1.3.4, C99 6.4.4.4).”

Implementation: See previous.

ISO Standard: “The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (C90 6.1.3.4, C99 6.4.4.4).”

Implementation: LC_ALL

ISO Standard: "The current locale used to convert a wide string literal into corresponding wide character codes (C90 6.1.4, C99 6.4.5)."

Implementation: LC_ALL

ISO Standard: "The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (C90 6.1.4, C99 6.4.5)."

Implementation: The binary representation of the characters is preserved from the source character set.

A.8 INTEGERS

ISO Standard: "Any extended integer types that exist in the implementation (C99 6.2.5)."

Implementation: There are no extended integer types.

ISO Standard: "Whether signed integer types are represented using sign and magnitude, two's complement, or one's complement, and whether the extraordinary value is a trap representation or an ordinary value (C99 6.2.6.2)."

Implementation: All integer types are represented as two's complement, and all bit patterns are ordinary values.

ISO Standard: "The rank of any extended integer type relative to another extended integer type with the same precision (C99 6.3.1.1)."

Implementation: No extended integer types are supported.

ISO Standard: "The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (C90 6.2.1.2, C99 6.3.1.3)."

Implementation: When converting value X to a type of width N, the value of the result is the Least Significant N bits of the 2's complement representation of X. That is, X is truncated to N bits. No signal is raised.

ISO Standard: "The results of some bitwise operations on signed integers (C90 6.3, C99 6.5)."

Implementation: Bitwise operations on signed values act on the 2's complement representation, including the sign bit. The result of a signed right shift expression is sign extended.
C99 allows some aspects of signed '<<' to be undefined. The compiler does not do so.

A.9 FLOATING-POINT

ISO Standard: "The accuracy of the floating-point operations and of the library functions in <math.h> and <complex.h> that return floating-point results (C90 and C99 5.2.4.2.2)."

Implementation: The accuracy is unknown.

ISO Standard: "The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in <stdio.h>, <stdlib.h>, and <wchar.h> (C90 and C99 5.2.4.2.2)."

Implementation: The accuracy is unknown.

ISO Standard: "The rounding behaviors characterized by non-standard values of FLT_ROUNDS (C90 and C99 5.2.4.2.2)."

Implementation: No such values are used.

ISO Standard: "The evaluation methods characterized by non-standard negative values of FLT_EVAL_METHOD (C90 and C99 5.2.4.2.2)."

Implementation: No such values are used.

ISO Standard: “The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (C90 6.2.1.3, C99 6.3.1.4).”

Implementation: C99 Annex F is followed.

ISO Standard: “The direction of rounding when a floating-point number is converted to a narrower floating-point number (C90 6.2.1.4, 6.3.1.5).”

Implementation: C99 Annex F is followed.

ISO Standard: “How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (C90 6.1.3.1, C99 6.4.4.2).”

Implementation: C99 Annex F is followed.

ISO Standard: “Whether and how floating expressions are contracted when not disallowed by the FP_CONTRACT pragma (C99 6.5).”

Implementation: The pragma is not implemented.

ISO Standard: “The default state for the FENV_ACCESS pragma (C99 7.6.1).”

Implementation: This pragma is not implemented.

ISO Standard: “Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (C99 7.6, 7.12).”

Implementation: None supported.

ISO Standard: “The default state for the FP_CONTRACT pragma (C99 7.12.2).”

Implementation: This pragma is not implemented.

ISO Standard: “Whether the “inexact” floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (C99 F.9).”

Implementation: Unknown.

ISO Standard: “Whether the “underflow” (and “inexact”) floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (C99 F.9).”

Implementation: Unknown.

A.10 ARRAYS AND POINTERS

ISO Standard: "The result of converting a pointer to an integer or vice versa (C90 6.3.4, C99 6.3.2.3)."

Implementation: A cast from an integer to a pointer or vice versa results uses the binary representation of the source type, reinterpreted as appropriate for the destination type.

If the source type is larger than the destination type, the Most Significant bits are discarded. When casting from a pointer to an integer, if the source type is smaller than the destination type, the result is sign extended. When casting from an integer to a pointer, if the source type is smaller than the destination type, the result is extended based on the signedness of the source type.

ISO Standard: "The size of the result of subtracting two pointers to elements of the same array (C90 6.3.6, C99 6.5.6)."

Implementation: 32-bit signed integer.

A.11 HINTS

ISO Standard: "The extent to which suggestions made by using the register storage-class specifier are effective (C90 6.5.1, C99 6.7.1)."

Implementation: The register storage class specifier generally has no effect.

ISO Standard: "The extent to which suggestions made by using the inline function specifier are effective (C99 6.7.4)."

Implementation: If `-fno-inline` or `-O0` are specified, no functions will be inlined, even if specified with the `inline` specifier. Otherwise, the function may or may not be inlined dependent on the optimization heuristics of the compiler.

A.12 STRUCTURES, UNIONS, ENUMERATIONS, AND BIT FIELDS

ISO Standard: "A member of a union object is accessed using a member of a different type (C90 6.3.2.3)."

Implementation: The corresponding bytes of the union object are interpreted as an object of the type of the member being accessed without regard for alignment or other possible invalid conditions.

ISO Standard: "Whether a "plain" `int` bit field is treated as a `signed int` bit field or as an `unsigned int` bit field (C90 6.5.2, C90 6.5.2.1, C99 6.7.2, C99 6.7.2.1)."

Implementation: By default, a plain `int` bit field is treated as a signed integer. This behavior can be altered by use of the `-funsigned-bitfields` command line option.

ISO Standard: "Allowable bit field types other than `_Bool`, `signed int`, and `unsigned int` (C99 6.7.2.1)."

Implementation: No other types are supported.

ISO Standard: "Whether a bit field can straddle a storage unit boundary (C90 6.5.2.1, C99 6.7.2.1)."

Implementation: No.

ISO Standard: "The order of allocation of bit fields within a unit (C90 6.5.2.1, C99 6.7.2.1)."

Implementation: Bit fields are allocated left to right.

ISO Standard: "The alignment of non-bit field members of structures (C90 6.5.2.1, C99 6.7.2.1)."

Implementation: Each member is located to the lowest available offset allowable according to the alignment restrictions of the member type.

ISO Standard: "The integer type compatible with each enumerated type (C90 6.5.2.2, C99 6.7.2.2)."

Implementation: If the enumeration values are all non-negative, the type is `unsigned int`, else it is `int`. The `-fshort-enums` command line option can change this.

A.13 QUALIFIERS

ISO Standard: "What constitutes an access to an object that has volatile-qualified type (C90 6.5.3, C99 6.7.3)."

Implementation: Any expression which uses the value of or stores a value to a volatile object is considered an access to that object. There is no guarantee that such an access is atomic.

If an expression contains a reference to a volatile object but neither uses the value nor stores to the object, the expression is considered an access to the volatile object or not depending on the type of the object. If the object is of scalar type, an aggregate type with a single member of scalar type, or a union with members of (only) scalar type, the expression is considered an access to the volatile object. Otherwise, the expression is evaluated for its side effects but is not considered an access to the volatile object.

For example:

```
volatile int a;
a; /* access to 'a' since 'a' is scalar */
```

A.14 DECLARATORS

ISO Standard: “The maximum number of declarators that may modify an arithmetic, structure or union type (C90 6.5.4).”

Implementation: No limit.

A.15 STATEMENTS

ISO Standard: “The maximum number of case values in a switch statement (C90 6.6.4.2).”

Implementation: No limit.

A.16 PRE-PROCESSING DIRECTIVES

ISO Standard: “How sequences in both forms of header names are mapped to headers or external source file names (C90 6.1.7, C99 6.4.7).”

Implementation: The character sequence between the delimiters is considered to be a string which is a file name for the host environment.

ISO Standard: “Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (C90 6.8.1, C99 6.10.1).”

Implementation: Yes.

ISO Standard: “Whether the value of a single-character `character` constant in a constant expression that controls conditional inclusion may have a negative value (C90 6.8.1, C99 6.10.1).”

Implementation: Yes.

ISO Standard: “The places that are searched for an included `< >` delimited header, and how the places are specified or the header is identified (C90 6.8.2, C99 6.10.2).”

Implementation:

```
<install
directory>/lib/gcc/pic32mx/3.4.4/include
<install directory>/pic32mx/include
```

ISO Standard: “How the named source file is searched for in an included `""` delimited header (C90 6.8.2, C99 6.10.2).”

Implementation: The compiler first searches for the named file in the directory containing the including file, the directories specified by the `-iquote` command line option (if any), then the directories which are searched for a `< >` delimited header.

ISO Standard: “The method by which preprocessing tokens are combined into a header name (C90 6.8.2, C99 6.10.2).”

Implementation: All tokens, including whitespace, are considered part of the header file name. Macro expansion is not performed on tokens inside the delimiters.

ISO Standard: “The nesting limit for `#include` processing (C90 6.8.2, C99 6.10.2).”

Implementation: No limit.

ISO Standard: “The behavior on each recognized non-`STDC #pragma` directive (C90 6.8.6, C99 6.10.6).”

Implementation: See **Section 6.12 “Variable Attributes”**.

ISO Standard: “The definitions for `__DATE_ _` and `__TIME_ _` when respectively, the date and time of translation are not available (C90 6.8.8, C99 6.10.8).”

Implementation: The date and time of translation are always available.

A.17 LIBRARY FUNCTIONS

ISO Standard: "The Null Pointer constant to which the macro NULL expands (C90 7.1.6, C99 7.17)."

Implementation: `(void *)0`

ISO Standard: "Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1)."

Implementation: See the "*32-Bit Language Tools Libraries"* (DS51685).

ISO Standard: "The format of the diagnostic printed by the `assert` macro (7.2.1.1)."

Implementation: "Failed assertion '*message*' at line *line* of '*filename*'.\n"

ISO Standard: "The default state for the `FENV_ACCESS` pragma (7.6.1)."

Implementation: Unimplemented.

ISO Standard: "The representation of floating-point exception flags stored by the `fegetexceptflag` function (7.6.2.2)."

Implementation: Unimplemented.

ISO Standard: "Whether the `feraiseexcept` function raises the inexact exception in addition to the overflow or underflow exception (7.6.2.3)."

Implementation: Unimplemented.

ISO Standard: "Floating environment macros other than `FE_DFL_ENV` that can be used as the argument to the `fesetenv` or `feupdateenv` function (7.6.4.3, 7.6.4.4)."

Implementation: Unimplemented.

ISO Standard: "Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function (7.11.1.1)."

Implementation: None.

ISO Standard: "The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 or greater than 2 (7.12)."

Implementation: Unimplemented.

ISO Standard: "The infinity to which the `INFINITY` macro expands, if any (7.12)."

Implementation: Unimplemented.

ISO Standard: "The quiet NaN to which the `NAN` macro expands, when it is defined (7.12)."

Implementation: Unimplemented.

ISO Standard: "Domain errors for the mathematics functions, other than those required by this International Standard (7.12.1)."

Implementation: None.

ISO Standard: "The values returned by the mathematics functions, and whether `errno` is set to the value of the macro `EDOM`, on domain errors (7.12.1)."

Implementation: `errno` is set to `EDOM` on domain errors.

ISO Standard: "Whether the mathematics functions set `errno` to the value of the macro `ERANGE` on overflow and/or underflow range errors (7.12.1)."

Implementation: Yes.

ISO Standard: "The default state for the `FP_CONTRACT` pragma (7.12.2)

Implementation: Unimplemented.

ISO Standard: "Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero (7.12.10.1)."

Implementation: NaN is returned.

ISO Standard: "The base-2 logarithm of the modulus used by the `remquo` function in reducing the quotient (7.12.10.3)."

Implementation: Unimplemented.

ISO Standard: “The set of signals, their semantics, and their default handling (7.14).”

Implementation: The default handling of signals is to always return failure. Actual signal handling is application defined.

ISO Standard: “If the equivalent of `signal(sig, SIG_DFL);` is not executed prior to the call of a signal handler, the blocking of the signal that is performed (7.14.1.1).”

Implementation: Application defined.

ISO Standard: “Whether the equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler for the signal `SIGILL` (7.14.1.1).”

Implementation: Application defined.

ISO Standard: “Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).”

Implementation: Application defined.

ISO Standard: “Whether the last line of a text stream requires a terminating new-line character (7.19.2).”

Implementation: Yes.

ISO Standard: “Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.19.2).”

Implementation: Yes.

ISO Standard: “The number of null characters that may be appended to data written to a binary stream (7.19.2).”

Implementation: No null characters are appended to a binary stream.

ISO Standard: “Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.19.3).”

Implementation: Application defined. The system level function `open` is called with the `O_APPEND` flag.

ISO Standard: “Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3).”

Implementation: Application defined.

ISO Standard: “The characteristics of file buffering (7.19.3).”

ISO Standard: “Whether a zero-length file actually exists (7.19.3).”

Implementation: Application defined.

ISO Standard: “The rules for composing valid file names (7.19.3).”

Implementation: Application defined.

ISO Standard: “Whether the same file can be open multiple times (7.19.3).”

Implementation: Application defined.

ISO Standard: “The nature and choice of encodings used for multibyte characters in files (7.19.3).”

Implementation: Encodings are the same for each file.

ISO Standard: “The effect of the `remove` function on an open file (7.19.4.1).”

Implementation: Application defined. The system function `unlink` is called.

ISO Standard: “The effect if a file with the new name exists prior to a call to the `rename` function (7.19.4.2).”

Implementation: Application defined. The system function `link` is called to create the new filename, then `unlink` is called to remove the old filename. Typically, `link` will fail if the new filename already exists.

ISO Standard: “Whether an open temporary file is removed upon abnormal program termination (7.19.4.3).”

Implementation: No.

ISO Standard: “What happens when the `tmpnam` function is called more than `TMP_MAX` times (7.19.4.4).”

Implementation: Temporary names will wrap around and be reused.

ISO Standard: “Which changes of mode are permitted (if any), and under what circumstances (7.19.5.4).”

Implementation: The file is closed via the system level `close` function and re-opened with the `open` function with the new mode. No additional restriction beyond those of the application defined `open` and `close` functions are imposed.

ISO Standard: “The style used to print an infinity or NaN, and the meaning of the *n-char-sequence* if that style is printed for a NaN (7.19.6.1, 7.24.2.1).”

Implementation: No char sequence is printed.
NaN is printed as “NaN”.
Infinity is printed as “[-/+]Inf”.

ISO Standard: “The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.19.6.1, 7.24.2.1).”

Implementation: Functionally equivalent to `%x`.

ISO Standard: “The interpretation of a `-` character that is neither the first nor the last character, nor the second where a `^` character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.1).”

Implementation: Unknown

ISO Standard: “The set of sequences matched by the `%p` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.2).”

Implementation: The same set of sequences matched by %x.

ISO Standard: “The interpretation of the input item corresponding to a `%p` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.2).”

Implementation: If the result is not a valid pointer, the behavior is undefined.

ISO Standard: “The value to which the macro `errno` is set by the `fgetpos`, `fsetpos`, or `ftell` functions on failure (7.19.9.1, 7.19.9.3, 7.19.9.4).”

Implementation: If the result exceeds `LONG_MAX`, `errno` is set to `ERANGE`.
Other errors are application defined according to the application definition of the `lseek` function.

ISO Standard: “The meaning of the *n-char-sequence* in a string converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function (7.20.1.3, 7.24.4.1.1).”

Implementation: No meaning is attached to the sequence.

ISO Standard: “Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.20.1.3, 7.24.4.1.1).”

Implementation: Yes.

ISO Standard: “Whether the `calloc`, `malloc`, and `realloc` functions return a Null Pointer or a pointer to an allocated object when the size requested is zero (7.20.3).”

Implementation: A pointer to a statically allocated object is returned.

ISO Standard: “Whether open output streams are flushed, open streams are closed, or temporary files are removed when the `abort` function is called (7.20.4.1).”

Implementation: No.

ISO Standard: “The termination status returned to the host environment by the `abort` function (7.20.4.1).”

Implementation: By default, there is no host environment.

ISO Standard: “The value returned by the `system` function when its argument is not a Null Pointer (7.20.4.5).”

Implementation: Application defined.

ISO Standard: "The local time zone and Daylight Saving Time (7.23.1)."

Implementation: Application defined.

ISO Standard: "The era for the `clock` function (7.23.2.1)."

Implementation: Application defined.

ISO Standard: "The positive value for `tm_isdst` in a normalized `tmx` structure (7.23.2.6)."

Implementation: 1.

ISO Standard: "The replacement string for the `%Z` specifier to the `strftime`, `strfxtime`, `wcsftime`, and `wcsfxtime` functions in the "`C`" locale (7.23.3.5, 7.23.3.6, 7.24.5.1, 7.24.5.2)."

Implementation: Unimplemented.

ISO Standard: "Whether or when the trigonometric, hyperbolic, base-*e* exponential, base-*e* logarithmic, error, and log gamma functions raise the inexact exception in an IEC 60559 conformant implementation (F.9)."

Implementation: No.

ISO Standard: "Whether the inexact exception may be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (F.9)."

Implementation: No.

ISO Standard: "Whether the underflow (and inexact) exception may be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (F.9)."

Implementation: No.

ISO Standard: "Whether the functions honor the Rounding Direction mode (F.9)."

Implementation: The Rounding mode is not forced.

A.18 ARCHITECTURE

ISO Standard: "The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (C90 and C99 5.2.4.2, C99 7.18.2, 7.18.3)."

Implementation: See **Section 6.4.2 "limits.h"**.

ISO Standard: "The number, order, and encoding of bytes in any object (when not explicitly specified in the standard) (C99 6.2.6.1)."

Implementation: Little endian, populated from Least Significant Byte first. See **Section 6.3 "Data Representation"**.

ISO Standard: "The value of the result of the size of operator (C90 6.3.3.4, C99 6.5.3.4)."

Implementation: See **Section 6.3 "Data Representation"**.

Appendix B. ASCII Character Set

TABLE B-1: ASCII CHARACTER SET

		Most Significant Character							
	Hex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Least Significant Character	**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 C. Deprecated Features

C.1 INTRODUCTION

The features described below are considered to be obsolete and have been replaced with more advanced functionality. Projects which depend on deprecated features will work properly with versions of the language tools cited. The use of a deprecated feature will result in a warning; programmers are encouraged to revise their projects in order to eliminate any dependency on deprecated features. Support for these features may be removed entirely in future versions of the language tools.

Deprecated features covered are:

Variables in Specified Registers

C.2 VARIABLES IN SPECIFIED REGISTERS

The compiler allows you to put a few global variables into specified hardware registers.

> **Note:** Using too many registers, in particular register W0, may impair the ability of the 32-bit compiler to compile. It is not recommended that registers be placed into fixed registers.

You can also specify the register in which an ordinary register variable should be allocated.

- Global register variables reserve registers throughout the program. This may be useful in programs such as programming language interpreters which have a couple of global variables that are accessed very often.
- Local register variables in specific registers do not reserve the registers. The compiler's data flow analysis is capable of determining where the specified registers contain live values, and where they are available for other uses. Stores into local register variables may be deleted when they appear to be unused. References to local register variables may be deleted, moved or simplified.

These local variables are sometimes convenient for use with the extended inline assembly (see **Chapter 14. "Mixing C/C++ and Assembly Language"**), 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).

C.2.1 Defining Global Register Variables

You can define a global register variable like this:

```
register int *foo asm ("w8");
```

Here `w8` is the name of the register which should be used. Choose a register that is normally saved and restored by function calls (W8-W13), so that library routines will not clobber it.

Defining a global register variable in a certain register reserves that register entirely for this use, at least within the current compilation. The register will not be allocated for any other purpose in the functions in the current compilation. The register will not be saved and restored by these functions. Stores into this register are never deleted even if they would appear to be dead, but references may be deleted, moved or simplified.

It is not safe to access the global register variables from signal handlers, or from more than one thread of control, because the system library routines may temporarily use the register for other things (unless you recompile them especially for the task at hand).

It is not safe for one function that uses a global register variable to call another such function `foo` by way of a third function `lose` that was compiled without knowledge of this variable (i.e., in a source file in which the variable wasn't declared). This is because `lose` might save the register and put some other value there. For example, you can't expect a global register variable to be available in the comparison-function that you pass to `qsort`, since `qsort` might have put something else in that register. This problem can be avoided by recompiling `qsort` with the same global register variable definition.

If you want to recompile `qsort` or other source files that do not actually use your global register variable, so that they will not use that register for any other purpose, then it suffices to specify the compiler command-line option `-ffixed-`*`reg`*. You need not actually add a global register declaration to their source code.

A function that can alter the value of a global register variable cannot safely be called from a function compiled without this variable, because it could clobber the value the caller expects to find there on return. Therefore, the function that is the entry point into the part of the program that uses the global register variable must explicitly save and restore the value that belongs to its caller.

The library function `longjmp` will restore 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.

C.2.2 Specifying Registers for Local Variables

You can define a local register variable with a specified register like this:

```
register int *foo asm ("w8");
```

Here `w8` is the name of the register that should be used. Note that this is the same syntax used for defining global register variables, but for a local variable it would appear within a function.

Defining such a register variable does not reserve the register; it remains available for other uses in places where flow control determines the variable's value is not live. Using this feature may leave the compiler too few available registers to compile certain functions.

This option does not ensure that the compiler will generate code that has this variable in the register you specify at all times. You may not code an explicit reference to this register in an `asm` statement and assume it will always refer to this variable.

Assignments to local register variables may be deleted when they appear to be unused. References to local register variables may be deleted, moved or simplified.

Appendix D. Built-In Functions

D.1 INTRODUCTION

This appendix lists the built-in functions that are specific to MPLAB XC32 C Compiler.

Built-in functions give the C programmer access to assembler operators or machine instructions that are currently only accessible using inline assembly, but are sufficiently useful that they are applicable to a broad range of applications. Built-in functions are coded in C source files syntactically like function calls, but they are compiled to assembly code that directly implements the function, and do not involve function calls or library routines.

There are a number of reasons why providing built-in functions is preferable to requiring programmers to use inline assembly. They include the following:

1. Providing built-in functions for specific purposes simplifies coding.
2. Certain optimizations are disabled when inline assembly is used. This is not the case for built-in functions.
3. For machine instructions that use dedicated registers, coding inline assembly while avoiding register allocation errors can require considerable care. The built-in functions make this process simpler as you do not need to be concerned with the particular register requirements for each individual machine instruction.

Built-In Function List

unsigned long__builtin_section_begin(quoted-section-name)

unsigned long __builtin_section_end(quoted-section-name)

unsigned long __builtin_section_size(quoted-section-name)

unsigned int __builtin_get_isr_state(void)

__builtin_set_isr_state(unsigned int)

void __builtin_disable_interrupts(void)

void __builtin_enable_interrupts(void)

D.2 BUILT-IN FUNCTION DESCRIPTIONS

This section describes the programmer interface to the compiler built-in functions. Since the functions are "built in", there are no header files associated with them. Similarly, there are no command-line switches associated with the built-in functions – they are always available. The built-in function names are chosen such that they belong to the compiler's namespace (they all have the prefix `__builtin_`), so they will not conflict with function or variable names in the programmer's namespace.

The following builtins get run-time information about section addresses and sizes:

unsigned long__builtin_section_begin(quoted-section-name)

Description: Return the beginning address of the quoted section name.

Prototype: `unsigned long __builtin_section_begin(quoted-section-name);`

Argument: `quoted-section-name` The name of the section.

Return Value: The address of the section.

Assembler Operator/ Machine Instruction: `.startof.`

Error Messages An "undefined reference" error message will be displayed if the quoted section name does not exist in the link.

unsigned long __builtin_section_end(quoted-section-name)

Description: Return the end address of the quoted section name + 1.

Prototype: `unsigned long __builtin_section_end(quoted-section-name);`

Argument: `quoted-section-name` The name of the section.

Return Value: The end address of the section + 1.

Assembler Operator/ Machine Instruction: `.endof.`

Error Messages An "undefined reference" error message will be displayed if the quoted section name does not exist in the link.

unsigned long __builtin_section_size(quoted-section-name)

Description: Return the size in bytes of the named quoted section.

Prototype: `unsigned long __builtin_section_size(quoted-section-name);`

Argument: `quoted-section-name` The name of the section.

Return Value: The size in bytes of the named section.

Assembler Operator/ Machine Instruction: `.sizeof.`

Error Messages An "undefined reference" error message will be displayed if the quoted section name does not exist in the link.

The following builtins inspect or manipulate the current CPU interrupt state:

unsigned int __builtin_get_isr_state(void)

Description: Get the current Interrupt Priority Level and Interrupt Enable bits.

Prototype: `unsigned int __builtin_get_isr_state(void);`

Argument: None.

Return Value: The current IPL and interrupt enable bits in a packed format. This value is to be used with the `__builtin_set_isr_state()` function.

unsigned int __builtin_get_isr_state(void) (Continued)

Assembler Operator/ Machine Instruction:

```
mfc0   $3, $12, 0
srl    $2,$3,10
ins    $2,$3,3,1
andi   $2,$2,0xf
sw     $2,0($fp)
```

Error Messages None.

__builtin_set_isr_state(unsigned int)

Description: Set the Interrupt Priority Level and Interrupt Enable bits using a value obtained from `__builtin_get_isr_state()`.

Prototype: `void __builtin_set_isr_state(unsigned int);`

Argument: An unsigned integer value obtained from `__builtin_get_isr_state()`.

Return Value: None.

Assembler Operator/ Machine Instruction:

```
di
ehb
mfc0   $2, $12, 0
ins    $2,$3,10,3
srl    $3,$3,3
ins    $2,$3,0,1
mtc0   $2, $12, 0
ehb
```

Error Messages None.

void __builtin_disable_interrupts(void)

Description: Disable interrupts.

Prototype: `void __builtin_disable_interrupts(void);`

Argument: None.

Return Value: None.

Assembler Operator/ Machine Instruction:

```
di     $2
ehb
```

Error Messages None.

void __builtin_enable_interrupts(void)

Description: Enable interrupts.

Prototype: `void __builtin_enable_interrupts(void);`

Argument: None.

Return Value: None.

Assembler Operator/ Machine Instruction:

```
ei     $2
```

Error Messages None.

NOTES:

Appendix E. Embedded Compiler Compatibility Mode

E.1 INTRODUCTION

All three MPLAB XC C compilers can be placed into a compatibility mode. In this mode, they are syntactically compatible with the non-standard C language extensions used by other non-Microchip embedded compiler vendors. This compatibility allows C source code written for other compilers to be compiled with minimum modification when using the MPLAB XC compilers.

Since very different device architectures may be targeted by other compilers, the semantics of the non-standard extensions may be different to that in the MPLAB XC compilers. This document indicates when the original C code may need to be reviewed.

The compatibility features offered by the MPLAB C compilers are discussed in the following topics:

- Compiling in Compatibility Mode
- Syntax Compatibility
- Data Type
- Operator
- Extended Keywords
- All assembly code specified by this construct is device-specific and will need review when porting to any Microchip device.
- Pragmas

E.2 COMPILING IN COMPATIBILITY MODE

An option is used to enable vendor-specific syntax compatibility. When using MPLAB XC8, this option is `--ext=`*`vendor`*; when using MPLAB XC16 or MPLAB XC32, the option is `-mext=`*`vendor`*. The argument *`vendor`* is a key that is used to represent the syntax. See Table E-1 for a list of all keys usable with the MPLAB XC compilers.

TABLE E-1: VENDOR KEYS

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

The Common Compiler Interface is a language standard that is common to all Microchip MPLAB XC compilers. The non-standard extensions associated with this syntax are already described in **Chapter 2. "Common C Interface"** and are not repeated here.

E.3 SYNTAX COMPATIBILITY

The goal of this syntax compatibility feature is to ease the migration process when porting source code from other C compilers to the native MPLAB XC compiler syntax.

Many non-standard extensions are not required when compiling for Microchip devices and, for these, there are no equivalent extensions offered by MPLAB XC compilers. These extensions are then simply ignored by the MPLAB XC compilers, although a warning message is usually produced to ensure thatyou are aware of the different compiler behavior. You should confirm that your project will still operate correctly with these features disabled.

Other non-standard extensions are not compatible with Microchip devices. Errors will be generated by the MPLAB XC compiler if these extensions are not removed from the source code. You should review the ramifications of removing the extension and decide whether changes are required to other source code in your project.

Table E-2 indicates the various levels of compatibility used in the tables that are presented throughout this guide.

TABLE E-2: LEVEL OF SUPPORT INDICATORS

Level	Explanation
support	The syntax is accepted in the specified compatibility mode, and its meaning will mimic its meaning when it is used with the original compiler.
support (no args)	In the case of pragmas, the base pragma is supported in the specified compatibility mode, but the arguments are ignored.
native support	The syntax is equivalent to that which is already accepted by the MPLAB XC compiler, and the semantics are compatible. You can use this feature without a vendor compatibility mode having been enabled.
ignore	The syntax is accepted in the specified compatibility mode, but the implied action is not required or performed. The extension is ignored and a warning will be issued by the compiler.
error	The syntax is not accepted in the specified compatibility mode. An error will be issued and compilation will be terminated.

Note that even if a C feature is supported by an MPLAB XC compiler, addresses, register names, assembly instructions, or any other device-specific argument is unlikely to be valid when compiling for a Microchip device. Always review code which uses these items in conjunction with the data sheet of your target Microchip device.

E.4 DATA TYPE

Some compilers allow use of the boolean type, `bool`, as well as associated values `true` and `false`, as specified by the C99 ANSI Standard. This type and these values may be used by all MPLAB XC compilers when in compatibility mode[1], as shown in Table E-3.

As indicated by the ANSI Standard, the `<stdbool.h>` header must be included for this feature to work as expected when it is used with MPLAB XC compilers.

TABLE E-3: SUPPORT FOR C99 BOOL TYPE

IAR Compatibility Mode			
Type	**XC8**	**XC16**	**XC32**
`bool`	support	support	support

Do not confuse the boolean type, `bool`, and the integer type, `bit`, implemented by MPLAB XC8.

E.5 OPERATOR

The `@` operator may be used with other compilers to indicate the desired memory location of an object. As Table E-4 indicates, support for this syntax in MPLAB C is limited to MPLAB XC8 only.

Any address specified with another device is unlikely to be correct on a new architecture. Review the address in conjunction with the data sheet for your target Microchip device.

Using *@* in a compatibility mode with MPLAB XC8 will work correctly, but will generate a warning. To prevent this warning from appearing again, use the reviewed address with the MPLAB C `__at()` specifier instead.

For MPLAB XC16/32, consider using the `address` attribute.

TABLE E-4: SUPPORT FOR NON-STANDARD OPERATOR

IAR Compatibility Mode			
Operator	**XC8**	**XC16**	**XC32**
`@`	native support	error	error

[1] Not all C99 features have been adopted by all Microchip MPLAB XC compilers.

E.6 EXTENDED KEYWORDS

Non-standard extensions often specify how objects are defined or accessed. Keywords are usually used to indicate the feature. The non-standard C keywords corresponding to other compilers are listed in Table E-5, as well as the level of compatibility offered by MPLAB XC compilers. The table notes offer more information about some extensions.

TABLE E-5: SUPPORT FOR NON-STANDARD KEYWORDS

IAR Compatibility Mode			
Keyword	**XC8**	**XC16**	**XC32**
`__section_begin`	ignore	support	support
`__section_end`	ignore	support	support
`__section_size`	ignore	support	support
`__segment_begin`	ignore	support	support
`__segment_end`	ignore	support	support
`__segment_size`	ignore	support	support
`__sfb`	ignore	support	support
`__sfe`	ignore	support	support
`__sfs`	ignore	support	support
`__asm or asm`(1)	support(2)	native support	native support
`__arm`	ignore	ignore	ignore
`__big_endian`	error	error	error
`__fiq`	support	error	error
`__intrinsic`	ignore	ignore	ignore
`__interwork`	ignore	ignore	ignore
`__irq`	support	error	error
`__little_endian`(3)	ignore	ignore	ignore
`__nested`	ignore	ignore	ignore
`__no_init`	support	support	support
`__noreturn`	ignore	support	support
`__ramfunc`	ignore	ignore	support(4)
`__packed`	ignore(5)	support	support
`__root`	ignore	support	support
`__swi`	ignore	ignore	ignore
`__task`	ignore	support	support
`__weak`	ignore	support	support
`__thumb`	ignore	ignore	ignore
`__farfunc`	ignore	ignore	ignore
`__huge`	ignore	ignore	ignore
`__nearfunc`	ignore	ignore	ignore
`__inline`	support	native support	native support

Note 1: All assembly code specified by this construct is device-specific and will need review when porting to any Microchip device.

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

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

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

5: Although this keyword is ignored, by default, all structures are packed when using

MPLAB XC8, so there is no loss of functionality.

E.7 INTRINSIC FUNCTIONS

Intrinsic functions can be used to perform common tasks in the source code. The MPLAB XC compilers' support for the intrinsic functions offered by other compilers is shown in Table E-6.

TABLE E-6: SUPPORT FOR NON-STANDARD INTRINSIC FUNCTIONS

IAR Compatibility Mode			
Function	**XC8**	**XC16**	**XC32**
__disable_fiq[1]	support	ignore	ignore
__disable_interrupt	support	support	support
__disable_irq[1]	support	ignore	ignore
__enable_fiq[1]	support	ignore	ignore
__enable_interrupt	support	support	support
__enable_irq[1]	support	ignore	ignore
__get_interrupt_state	ignore	support	support
__set_interrupt_state	ignore	support	support

Note 1: These intrinsic functions map to macros which disable or enable the global interrupt enable bit on 8-bit PIC® devices.

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

E.8 PRAGMAS

Pragmas may be used by a compiler to control code generation. Any compiler will ignore an unknown pragma, but many pragmas implemented by another compiler have also been implemented by the MPLAB XC compilers in compatibility mode. Table E-7 shows the pragmas and the level of support when using each of the MPLAB XC compilers.

Many of these pragmas take arguments. Even if a pragma is supported by an MPLAB XC compiler, this support may not apply to all of the pragma's arguments. This is indicated in the table.

TABLE E-7: SUPPORT FOR NON-STANDARD PRAGMAS

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

TABLE E-7: SUPPORT FOR NON-STANDARD PRAGMAS (CONTINUED)

IAR Compatibility Mode			
Pragma	**XC8**	**XC16**	**XC32**
`object_attribute`	ignore	ignore	ignore
`optimize`	ignore	native support	native support
`pack`	ignore	native support	native support
`__printf_args`	support	support	support
`required`	ignore	support	support
`rtmodel`	ignore	ignore	ignore
`__scanf__args`	ignore	support	support
`section`	ignore	support	support
`segment`	ignore	support	support
`swi_number`	ignore	ignore	ignore
`type_attribute`	ignore	ignore	ignore
`weak`	ignore	native support	native support

Appendix F. Document Revision History

DOCUMENT REVISION HISTORY

Revision D (January 2012)

- Changed product name from MPLAB C32 C Compiler to MPLAB XC32 C Compiler. Completely reorganized document to align with other Microchip compiler documentation.

Revision E (July 2012)

- Added information pertaining to C++ throughout the document.
- Added new section describing the Common Compiler Interface (CCI) Standard.

Revision F (December 2012)

- Added Edition column to Table 3-11: General Optimization Options.
- Added `keep` and `optimize` function attributes to Chapter 10 Functions.
- Added Section 14.2 Mixing Assembly Language and C Variables and Functions.
- Added Appendix D Built-In Functions.
- Added Appendix E Embedded Compiler Compatibility Mode.
- Added Appendix F Document Revision History. This information was previously located in the Preface.
- Added Support chapter. This information was previously located in the Preface.

NOTES:

Support

INTRODUCTION

Please refer to the items discussed here for support issues.

- myMicrochip Personalized Notification Service
- The Microchip Web Site
- Microchip Forums
- Customer Support
- Contact Microchip Technology

myMICROCHIP PERSONALIZED NOTIFICATION SERVICE

myMicrochip: http://www.microchip.com/pcn

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

Please visit myMicrochip to begin the registration process and select your preferences to receive personalized notifications. A FAQ and registration details are available on the page, which can be opened by selecting the link above.

When you are selecting your preferences, choosing "Development Systems" will populate the list with available development tools. The main categories of tools are listed below:

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

THE MICROCHIP WEB SITE

Web Site: http://www.microchip.com

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

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

MICROCHIP FORUMS

Forums: http://www.microchip.com/forums

Microchip provides additional online support via our web forums. Currently available forums are:

- Development Tools
- 8-bit PIC MCUs
- 16-bit PIC MCUs
- 32-bit PIC MCUs

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

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

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

Technical Support: http://support.microchip.com

Documentation errors or comments may be emailed to docerrors@microchip.com.

CONTACT MICROCHIP TECHNOLOGY

You may call or fax Microchip Corporate offices at the numbers below:

Voice: (480) 792-7200

Fax: (480) 792-7277

Glossary

A

Absolute Section

A section with a fixed (absolute) address that cannot be changed by the linker.

Access Memory

PIC18 Only – Special registers on PIC18 devices that allow access regardless of the setting of the Bank Select Register (BSR).

Access Entry Points

Access entry points provide a way to transfer control across segments to a function which may not be defined at link time. They support the separate linking of boot and secure application segments.

Address

Value that identifies a location in memory.

Alphabetic Character

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

Alphanumeric

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

ANDed Breakpoints

Set up an ANDed condition for breaking, i.e., breakpoint 1 AND breakpoint 2 must occur at the same time before a program halt. This can only be accomplished if a data breakpoint and a program memory breakpoint occur at the same time.

Anonymous Structure

32-bit C/C++ Compiler **–** An unnamed structure.

PIC18 C Compiler **–** An unnamed structure that is a member of a C union. The members of an anonymous structure may be accessed as if they were members of the enclosing union. For example, in the following code, `hi` and `lo` are members of an anonymous structure inside the union `caster`.

```
union castaway
 int intval;
 struct {
  char lo; //accessible as caster.lo
  char hi; //accessible as caster.hi
 };
} caster;
```

ANSI

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

Application

A set of software and hardware that may be controlled by a PIC® microcontroller.

Archive/Archiver

An archive/library is a collection of relocatable object modules. It is created by assembling multiple source files to object files, and then using the archiver/librarian to combine the object files into one archive/library file. An archive/library can be linked with object modules and other archives/libraries to create executable code.

ASCII

American Standard Code for Information Interchange is a character set encoding that uses 7 binary digits to represent each character. It includes upper and lower case letters, digits, symbols and control characters.

Assembly/Assembler

Assembly is a programming language that describes binary machine code in a symbolic form. An assembler is a language tool that translates assembly language source code into machine code.

Assigned Section

A section which has been assigned to a target memory block in the linker command file.

Asynchronously

Multiple events that do not occur at the same time. This is generally used to refer to interrupts that may occur at any time during processor execution.

Attribute

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

Attribute, Section

Characteristics of sections, such as “executable”, “readonly”, or “data” that can be specified as flags in the assembler `.section` directive.

B

Binary

The base two numbering system that uses the digits 0-1. The rightmost digit counts ones, the next counts multiples of 2, then $2^2 = 4$, etc.

Breakpoint

Hardware Breakpoint: An event whose execution will cause a halt.

Software Breakpoint: An address where execution of the firmware will halt. Usually achieved by a special break instruction.

Build

Compile and link all the source files for an application.

C

C\C++

C is a general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators. C++ is the object-oriented version of C.

Calibration Memory

A special function register or registers used to hold values for calibration of a PIC microcontroller on-board RC oscillator or other device peripherals.

Central Processing Unit

The part of a device that is responsible for fetching the correct instruction for execution, decoding that instruction, and then executing that instruction. When necessary, it works in conjunction with the arithmetic logic unit (ALU) to complete the execution of the instruction. It controls the program memory address bus, the data memory address bus, and accesses to the stack.

Clean

Clean removes all intermediary project files, such as object, hex and debug files, for the active project. These files are recreated from other files when a project is built.

COFF

Common Object File Format. An object file of this format contains machine code, debugging and other information.

Command Line Interface

A means of communication between a program and its user based solely on textual input and output.

Compiler

A program that translates a source file written in a high-level language into machine code.

Conditional Assembly

Assembly language code that is included or omitted based on the assembly-time value of a specified expression.

Conditional Compilation

The act of compiling a program fragment only if a certain constant expression, specified by a preprocessor directive, is true.

Configuration Bits

Special-purpose bits programmed to set PIC microcontroller modes of operation. A Configuration bit may or may not be preprogrammed.

Control Directives

Directives in assembly language code that cause code to be included or omitted based on the assembly-time value of a specified expression.

CPU

See Central Processing Unit.

Cross Reference File

A file that references a table of symbols and a list of files that references the symbol. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

D

Data Directives

Data directives are those that control the assembler’s allocation of program or data memory and provide a way to refer to data items symbolically; that is, by meaningful names.

Data Memory

On Microchip MCU and DSC devices, data memory (RAM) is comprised of General Purpose Registers (GPRs) and Special Function Registers (SFRs). Some devices also have EEPROM data memory.

Debug/Debugger

See ICE/ICD.

Debugging Information

Compiler and assembler options that, when selected, provide varying degrees of information used to debug application code. See compiler or assembler documentation for details on selecting debug options.

Deprecated Features

Features that are still supported for legacy reasons, but will eventually be phased out and no longer used.

Device Programmer

A tool used to program electrically programmable semiconductor devices such as microcontrollers.

Digital Signal Controller

A digital signal controller (DSC) is a microcontroller device with digital signal processing capability, i.e., Microchip dsPIC® DSC devices.

Digital Signal Processing/Digital Signal Processor

Digital Signal Processing (DSP) is the computer manipulation of digital signals, commonly analog signals (sound or image) which have been converted to digital form (sampled). A digital signal processor is a microprocessor that is designed for use in digital signal processing.

Directives

Statements in source code that provide control of the language tool's operation.

Download

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

DWARF

Debug With Arbitrary Record Format. DWARF is a debug information format for ELF files.

E

EEPROM

Electrically Erasable Programmable Read Only Memory. A special type of PROM that can be erased electrically. Data is written or erased one byte at a time. EEPROM retains its contents even when power is turned off.

ELF

Executable and Linking Format. An object file of this format contains machine code. Debugging and other information is specified in with DWARF. ELF/DWARF provide better debugging of optimized code than COFF.

Emulation/Emulator

See ICE/ICD.

Endianness

The ordering of bytes in a multi-byte object.

Environment

MPLAB PM3 **–** A folder containing files on how to program a device. This folder can be transferred to a SD/MMC card.

Epilogue

A portion of compiler-generated code that is responsible for deallocating stack space, restoring registers and performing any other machine-specific requirement specified in the runtime model. This code executes after any user code for a given function, immediately prior to the function return.

EPROM

Erasable Programmable Read Only Memory. A programmable read-only memory that can be erased usually by exposure to ultraviolet radiation.

Error/Error File

An error reports a problem that makes it impossible to continue processing your program. When possible, an error identifies the source file name and line number where the problem is apparent. An error file contains error messages and diagnostics generated by a language tool.

Event

A description of a bus cycle which may include address, data, pass count, external input, cycle type (fetch, R/W), and time stamp. Events are used to describe triggers, breakpoints and interrupts.

Executable Code

Software that is ready to be loaded for execution.

Export

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

Expressions

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

Extended Microcontroller Mode

In extended microcontroller mode, on-chip program memory as well as external memory is available. Execution automatically switches to external if the program memory address is greater than the internal memory space of the PIC18 device.

Extended Mode (PIC18 MCUs)

In Extended mode, the compiler will utilize the extended instructions (i.e., `ADDFSR`, `ADDULNK`, `CALLW`, `MOVSF`, `MOVSS`, `PUSHL`, `SUBFSR` and `SUBULNK`) and the indexed with literal offset addressing.

External Label

A label that has external linkage.

External Linkage

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

External Symbol

A symbol for an identifier which has external linkage. This may be a reference or a definition.

External Symbol Resolution

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

External Input Line

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

External RAM

Off-chip Read/Write memory.

F

Fatal Error

An error that will halt compilation immediately. No further messages will be produced.

File Registers

On-chip data memory, including General Purpose Registers (GPRs) and Special Function Registers (SFRs).

Filter

Determine by selection what data is included/excluded in a trace display or data file.

Flash

A type of EEPROM where data is written or erased in blocks instead of bytes.

FNOP

Forced No Operation. A forced NOP cycle is the second cycle of a two-cycle instruction. Since the PIC microcontroller architecture is pipelined, it prefetches the next instruction in the physical address space while it is executing the current instruction. However, if the current instruction changes the program counter, this prefetched instruction is explicitly ignored, causing a forced NOP cycle.

Frame Pointer

A pointer that references the location on the stack that separates the stack-based arguments from the stack-based local variables. Provides a convenient base from which to access local variables and other values for the current function.

Free-Standing

An implementation that accepts any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (ANSI '89 standard clause 7) is confined to the contents of the standard headers `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>` and `<stdint.h>`.

G

GPR

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

H

Halt

A stop of program execution. Executing Halt is the same as stopping at a breakpoint.

Heap

An area of memory used for dynamic memory allocation where blocks of memory are allocated and freed in an arbitrary order determined at runtime.

Hex Code/Hex File

Hex code is executable instructions stored in a hexadecimal format code. Hex code is contained in a hex file.

Hexadecimal

The base 16 numbering system that uses the digits 0-9 plus the letters A-F (or a-f). The digits A-F represent hexadecimal digits with values of (decimal) 10 to 15. The rightmost digit counts ones, the next counts multiples of 16, then 16^2 = 256, etc.

High Level Language

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

I

ICE/ICD

In-Circuit Emulator/In-Circuit Debugger: A hardware tool that debugs and programs a target device. An emulator has more features than an debugger, such as trace.

In-Circuit Emulation/In-Circuit Debug: The act of emulating or debugging with an in-circuit emulator or debugger.

-ICE/-ICD: A device (MCU or DSC) with on-board in-circuit emulation or debug circuitry. This device is always mounted on a header board and used to debug with an in-circuit emulator or debugger.

ICSP™ Programming Capability

In-Circuit Serial Programming™ programming capability. A method of programming Microchip embedded devices using serial communication and a minimum number of device pins.

IDE

Integrated Development Environment, as in MPLAB IDE.

Identifier

A function or variable name.

IEEE

Institute of Electrical and Electronics Engineers.

Import

Bring data into the MPLAB IDE from an outside source, such as from a hex file.

Initialized Data

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

```
int myVar=5;
```

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

Instruction Set

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

Instructions

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

Internal Linkage

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

International Organization for Standardization

An organization that sets standards in many businesses and technologies, including computing and communications. Also known as ISO.

Interrupt

A signal to the CPU that suspends the execution of a running application and transfers control to an Interrupt Service Routine (ISR) so that the event may be processed. Upon completion of the ISR, normal execution of the application resumes.

Interrupt Handler

A routine that processes special code when an interrupt occurs.

Interrupt Service Request (IRQ)

An event which causes the processor to temporarily suspend normal instruction execution and to start executing an interrupt handler routine. Some processors have several interrupt request events allowing different priority interrupts.

Interrupt Service Routine (ISR)

Language tools **–** A function that handles an interrupt.

MPLAB IDE **–** User-generated code that is entered when an interrupt occurs. The location of the code in program memory will usually depend on the type of interrupt that has occurred.

Interrupt Vector

Address of an interrupt service routine or interrupt handler.

L

L-value

An expression that refers to an object that can be examined and/or modified. An l-value expression is used on the left-hand side of an assignment.

Latency

The time between an event and its response.

Library/Librarian

See Archive/Archiver.

Linker

A language tool that combines object files and libraries to create executable code, resolving references from one module to another.

Linker Script Files

Linker script files are the command files of a linker. They define linker options and describe available memory on the target platform.

Listing Directives

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

Listing File

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

Little Endian

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

Local Label

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

Logic Probes

Up to 14 logic probes can be connected to some Microchip emulators. The logic probes provide external trace inputs, trigger output signal, +5V, and a common ground.

Loop-Back Test Board

Used to test the functionality of the MPLAB REAL ICE™ in-circuit emulator.

LVDS

Low Voltage Differential Signaling. A low noise, low-power, low amplitude method for high-speed (gigabits per second) data transmission over copper wire.

With standard I/0 signaling, data storage is contingent upon the actual voltage level. Voltage level can be affected by wire length (longer wires increase resistance, which lowers voltage). But with LVDS, data storage is distinguished only by positive and negative voltage values, not the voltage level. Therefore, data can travel over greater lengths of wire while maintaining a clear and consistent data stream.

Source: http://www.webopedia.com/TERM/L/LVDS.html.

M

Machine Code

The representation of a computer program that is actually read and interpreted by the processor. A program in binary machine code consists of a sequence of machine instructions (possibly interspersed with data). The collection of all possible instructions for a particular processor is known as its "instruction set".

Machine Language

A set of instructions for a specific central processing unit, designed to be usable by a processor without being translated.

Macro

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

Macro Directives

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

Makefile

Export to a file the instructions to Make the project. Use this file to Make your project outside of MPLAB IDE, i.e., with a `make`.

Make Project

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

MCU

Microcontroller Unit. An abbreviation for microcontroller. Also µC.

Memory Model

For C compilers, a representation of the memory available to the application. For the PIC18 C compiler, a description that specifies the size of pointers that point to program memory.

Message

Text displayed to alert you to potential problems in language tool operation. A message will not stop operation.

Microcontroller

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

Microcontroller Mode

One of the possible program memory configurations of PIC18 microcontrollers. In microcontroller mode, only internal execution is allowed. Thus, only the on-chip program memory is available in microcontroller mode.

Microprocessor Mode

One of the possible program memory configurations of PIC18 microcontrollers. In microprocessor mode, the on-chip program memory is not used. The entire program memory is mapped externally.

Mnemonics

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

MPASM™ Assembler

Microchip Technology's relocatable macro assembler for PIC microcontroller devices, KeeLoq® devices and Microchip memory devices.

MPLAB *Language Tool* for *Device*

Microchip's C compilers, assemblers and linkers for specified devices. Select the type of language tool based on the device you will be using for your application, e.g., if you will be creating C code on a PIC18 MCU, select the MPLAB C Compiler for PIC18 MCUs.

MPLAB ICD

Microchip's in-circuit debuggers that works with MPLAB IDE. *See* ICE/ICD.

MPLAB IDE

Microchip's Integrated Development Environment. MPLAB IDE comes with an editor, project manager and simulator.

MPLAB PM3

A device programmer from Microchip. Programs PIC18 microcontrollers and dsPIC digital signal controllers. Can be used with MPLAB IDE or stand-alone. Replaces PRO MATE II.

MPLAB REAL ICE In-Circuit Emulator

Microchip's next-generation in-circuit emulators that works with MPLAB IDE. *See* ICE/ICD.

MPLAB SIM

Microchip's simulator that works with MPLAB IDE in support of PIC MCU and dsPIC DSC devices.

MPLIB™ Object Librarian

Microchip's librarian that can work with MPLAB IDE. MPLIB librarian is an object librarian for use with COFF object modules created using either MPASM assembler (mpasm or mpasmwin v2.0) or MPLAB C18 C compiler.

MPLINK™ Object Linker

MPLINK linker is an object linker for the Microchip MPASM assembler and the Microchip C18 C compiler. MPLINK linker also may be used with the Microchip MPLIB librarian. MPLINK linker is designed to be used with MPLAB IDE, though it does not have to be.

MRU

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

N

Native Data Size

For Native trace, the size of the variable used in a Watch window must be of the same size as the selected device's data memory: bytes for PIC18 devices and words for 16-bit devices.

Nesting Depth

The maximum level to which macros can include other macros.

Node

MPLAB IDE project component.

Non-Extended Mode (PIC18 MCUs)

In Non-Extended mode, the compiler will not utilize the extended instructions nor the indexed with literal offset addressing.

Non Real Time

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

Non-Volatile Storage

A storage device whose contents are preserved when its power is off.

NOP

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

O

Object Code/Object File

Object code is the machine code generated by an assembler or compiler. An object file is a file containing machine code and possibly debug information. It may be immediately executable or it may be relocatable, requiring linking with other object files, e.g., libraries, to produce a complete executable program.

Object File Directives

Directives that are used only when creating an object file.

Octal

The base 8 number system that only uses the digits 0-7. The rightmost digit counts ones, the next digit counts multiples of 8, then $8^2 = 64$, etc.

Off-Chip Memory

Off-chip memory refers to the memory selection option for the PIC18 device where memory may reside on the target board, or where all program memory may be supplied by the emulator. The **Memory** tab accessed from *Options>Development Mode* provides the Off-Chip Memory selection dialog box.

Opcodes

Operational Codes. *See* Mnemonics.

Operators

Symbols, like the plus sign ‘+’ and the minus sign ‘-’, that are used when forming well-defined expressions. Each operator has an assigned precedence that is used to determine order of evaluation.

OTP

One Time Programmable. EPROM devices that are not in windowed packages. Since EPROM needs ultraviolet light to erase its memory, only windowed devices are erasable.

P

Pass Counter

A counter that decrements each time an event (such as the execution of an instruction at a particular address) occurs. When the pass count value reaches zero, the event is satisfied. You can assign the Pass Counter to break and trace logic, and to any sequential event in the complex trigger dialog.

PC

Personal Computer or Program Counter.

PC Host

Any PC running a supported Windows operating system.

Persistent Data

Data that is never cleared or initialized. Its intended use is so that an application can preserve data across a device reset.

Phantom Byte

An unimplemented byte in the dsPIC DSC architecture that is used when treating the 24-bit instruction word as if it were a 32-bit instruction word. Phantom bytes appear in dsPIC DSC hex files.

PIC MCUs

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

PICkit™ 2 and 3 Programmer/Debugger

Microchip’s developmental device programmers with debug capability through Debug Express. See the Readme files for each tool to see which devices are supported.

Plug-ins

The MPLAB IDE has both built-in components and plug-in modules to configure the system for a variety of software and hardware tools. Several plug-in tools may be found under the Tools menu.

Pod

The enclosure for an in-circuit emulator or debugger. Other names are “Puck”, if the enclosure is round, and “Probe”, not be confused with logic probes.

Power-on-Reset Emulation

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

Pragma

A directive that has meaning to a specific compiler. Often a pragma is used to convey implementation-defined information to the compiler. MPLAB C30 uses attributes to convey this information.

Precedence

Rules that define the order of evaluation in expressions.

Production Programmer

A production programmer is a programming tool that has resources designed in to program devices rapidly. It has the capability to program at various voltage levels and completely adheres to the programming specification. Programming a device as fast as possible is of prime importance in a production environment where time is of the essence as the application circuit moves through the assembly line.

Profile

For MPLAB SIM simulator, a summary listing of executed stimulus by register.

Program Counter

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

Program Counter Unit

32-bit assembler – A conceptual representation of the layout of program memory. The program counter increments by 2 for each instruction word. In an executable section, 2 program counter units are equivalent to 3 bytes. In a read-only section, 2 program counter units are equivalent to 2 bytes.

Program Memory

MPLAB IDE – The memory area in a device where instructions are stored. Also, the memory in the emulator or simulator containing the downloaded target application firmware.

32-bit assembler/compiler – The memory area in a device where instructions are stored.

Project

A project contains the files needed to build an application (source code, linker script files, etc.) along with their associations to various build tools and build options.

Prologue

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

Prototype System

A term referring to a user's target application, or target board.

PWM Signals

Pulse Width Modulation Signals. Certain PIC MCU devices have a PWM peripheral.

Q

Qualifier

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

R

Radix

The number base, hex, or decimal, used in specifying an address.

RAM

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

Raw Data

The binary representation of code or data associated with a section.

Read Only Memory

Memory hardware that allows fast access to permanently stored data but prevents addition to or modification of the data.

Real Time

When an in-circuit emulator or debugger is released from the halt state, the processor runs in Real Time mode and behaves exactly as the normal chip would behave. In Real Time mode, the real time trace buffer of an emulator is enabled and constantly captures all selected cycles, and all break logic is enabled. In an in-circuit emulator or debugger, the processor executes in real time until a valid breakpoint causes a halt, or until the user halts the execution.

In the simulator, real time simply means execution of the microcontroller instructions as fast as they can be simulated by the host CPU.

Real-Time Watch

A Watch window where the variables change in real-time as the application is run. See individual tool documentation to determine how to set up a real-time watch. Not all tools support real-time watches.

Recursive Calls

A function that calls itself, either directly or indirectly.

Recursion

The concept that a function or macro, having been defined, can call itself. Great care should be taken when writing recursive macros; it is easy to get caught in an infinite loop where there will be no exit from the recursion.

Reentrant

A function that may have multiple, simultaneously active instances. This may happen due to either direct or indirect recursion or through execution during interrupt processing.

Relaxation

The process of converting an instruction to an identical, but smaller instruction. This is useful for saving on code size. MPLAB ASM30 currently knows how to RELAX a CALL instruction into an RCALL instruction. This is done when the symbol that is being called is within +/- 32k instruction words from the current instruction.

Relocatable

An object whose address has not been assigned to a fixed location in memory.

Relocatable Section

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

Relocation

A process performed by the linker in which absolute addresses are assigned to relocatable sections and all symbols in the relocatable sections are updated to their new addresses.

ROM

Read Only Memory (Program Memory). Memory that cannot be modified.

Run

The command that releases the emulator from halt, allowing it to run the application code and change or respond to I/O in real time.

Run-time Model

Describes the use of target architecture resources.

S

Scenario

For MPLAB SIM simulator, a particular setup for stimulus control.

Section

A portion of an application located at a specific address of memory.

Section Attribute

A characteristic ascribed to a section (e.g., an `access` section).

Sequenced Breakpoints

Breakpoints that occur in a sequence. Sequence execution of breakpoints is bottom-up; the last breakpoint in the sequence occurs first.

Serialized Quick Turn Programming

Serialization allows you to program a serial number into each microcontroller device that the Device Programmer programs. This number can be used as an entry code, password or ID number.

Shell

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

Simulator

A software program that models the operation of devices.

Single Step

This command steps though code, one instruction at a time. After each instruction, MPLAB IDE updates register windows, watch variables, and status displays so you can analyze and debug instruction execution. You can also single step C compiler source code, but instead of executing single instructions, MPLAB IDE will execute all assembly level instructions generated by the line of the high level C statement.

Skew

The information associated with the execution of an instruction appears on the processor bus at different times. For example, the executed opcodes appears on the bus as a fetch during the execution of the previous instruction, the source data address and value and the destination data address appear when the opcodes is actually executed, and the destination data value appears when the next instruction is executed. The trace buffer captures the information that is on the bus at one instance. Therefore, one trace buffer entry will contain execution information for three instructions. The number of captured cycles from one piece of information to another for a single instruction execution is referred to as the skew.

Skid

When a hardware breakpoint is used to halt the processor, one or more additional instructions may be executed before the processor halts. The number of extra instructions executed after the intended breakpoint is referred to as the skid.

Source Code

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

Source File

An ASCII text file containing source code.

Special Function Registers (SFRs)

The portion of data memory (RAM) dedicated to registers that control I/O processor functions, I/O status, timers or other modes or peripherals.

SQTP^{sm}

See Serialized Quick Turn Programming.

Stack, Hardware

Locations in PIC microcontroller where the return address is stored when a function call is made.

Stack, Software

Memory used by an application for storing return addresses, function parameters, and local variables. This memory is typically managed by the compiler when developing code in a high-level language.

MPLAB Starter Kit for *Device*

Microchip's starter kits contains everything needed to begin exploring the specified device. View a working application and then debug and program you own changes.

Static RAM or SRAM

Static Random Access Memory. Program memory you can read/write on the target board that does not need refreshing frequently.

Status Bar

The Status Bar is located on the bottom of the MPLAB IDE window and indicates such current information as cursor position, development mode and device, and active tool bar.

Step Into

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

Step Over

Step Over allows you to debug code without stepping into subroutines. When stepping over a `CALL` instruction, the next breakpoint will be set at the instruction after the `CALL`. If for some reason the subroutine gets into an endless loop or does not return properly, the next breakpoint will never be reached. The Step Over command is the same as Single Step except for its handling of `CALL` instructions.

Step Out

Step Out allows you to step out of a subroutine which you are currently stepping through. This command executes the rest of the code in the subroutine and then stops execution at the return address to the subroutine.

Stimulus

Input to the simulator, i.e., data generated to exercise the response of simulation to external signals. Often the data is put into the form of a list of actions in a text file. Stimulus may be asynchronous, synchronous (pin), clocked and register.

Stopwatch

A counter for measuring execution cycles.

Storage Class

Determines the lifetime of the memory associated with the identified object.

Storage Qualifier

Indicates special properties of the objects being declared (e.g., `const`).

Symbol

A symbol is a general purpose mechanism for describing the various pieces which comprise a program. These pieces include function names, variable names, section names, file names, struct/enum/union tag names, etc. Symbols in MPLAB IDE refer mainly to variable names, function names and assembly labels. The value of a symbol after linking is its value in memory.

Symbol, Absolute

Represents an immediate value such as a definition through the assembly `.equ` directive.

System Window Control

The system window control is located in the upper left corner of windows and some dialogs. Clicking on this control usually pops up a menu that has the items “Minimize,” “Maximize,” and “Close.”

T

Target

Refers to user hardware.

Target Application

Software residing on the target board.

Target Board

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

Target Processor

The microcontroller device on the target application board.

Template

Lines of text that you build for inserting into your files at a later time. The MPLAB Editor stores templates in template files.

Tool Bar

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

Trace

An emulator or simulator function that logs program execution. The emulator logs program execution into its trace buffer which is uploaded to MPLAB IDE’s trace window.

Trace Memory

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

Trace Macro

A macro that will provide trace information from emulator data. Since this is a software trace, the macro must be added to code, the code must be recompiled or reassembled, and the target device must be programmed with this code before trace will work.

Trigger Output

Trigger output refers to an emulator output signal that can be generated at any address or address range, and is independent of the trace and breakpoint settings. Any number of trigger output points can be set.

Trigraphs

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

U

Unassigned Section

A section which has not been assigned to a specific target memory block in the linker command file. The linker must find a target memory block in which to allocate an unassigned section.

Uninitialized Data

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

```
int myVar;
```

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

Upload

The Upload function transfers data from a tool, such as an emulator or programmer, to the host PC or from the target board to the emulator.

USB

Universal Serial Bus. An external peripheral interface standard for communication between a computer and external peripherals over a cable using bi-serial transmission. USB 1.0/1.1 supports data transfer rates of 12 Mbps. Also referred to as high-speed USB, USB 2.0 supports data rates up to 480 Mbps.

V

Vector

The memory locations that an application will jump to when either a reset or interrupt occurs.

W

Warning

MPLAB IDE – An alert that is provided to warn you of a situation that would cause physical damage to a device, software file, or equipment.

32-bit assembler/compiler – Warnings report conditions that may indicate a problem, but do not halt processing. In MPLAB C30, warning messages report the source file name and line number, but include the text ‘`warning`:’ to distinguish them from error messages.

Watch Variable

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

Watch Window

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

Watchdog Timer (WDT)

A timer on a PIC microcontroller that resets the processor after a selectable length of time. The WDT is enabled or disabled and set up using Configuration bits.

Workbook

For MPLAB SIM stimulator, a setup for generation of SCL stimulus.

Index

IAR compatibility 217–222

Symbols

.app_excpt Section 188
.bev_excpt Section 188
.bss 149
.bss Section 194
.config_address 186
.data 149
.data Section 192
.dbg_data Section 192
.dbg_excpt Section 188
.got Section 192
.h files, see header files
.heap 194
.lit4 149
.lit4 Section 193
.lit8 149
.lit8 Section 193
.ramfunc 150
.ramfunc Section 195
.reset Section 187
.rodata Section 191
.sbss 149
.sbss Section 194
.sbss2 Section 192
.sdata 149
.sdata Section 193
.sdata2 Section 191
.stack Section 194
.startup Section 189
.text Section 189
.vector_n Sections 189
"On Bootstrap" Procedure 156
# preprocessor operator 176
## preprocessor operator 176
#define 77
#ident 83
#if 71
#include 78, 79
#line 79
#pragma 68
#pragma config 177
#pragma interrupt 177
#pragma vector 177

Numerics

-msmart-io= 62
PIC32MX Device-Specific Options
    -msmart-io= 62
0b binary radix specifier 102
32-Bit C Compiler Macros 178

A

a0-a3 142
absolute functions 29
absolute variables 29
addressr Attribute 123
alias (symbol) 124
__align qualifier 34
aligned (n) 105, 106
always_inline 124
anonymous unions 99
-ansi 63, 64, 79, 131
ANSI C Standard
    conformance 91
    implementation-defined behavior 92
ANSI C standard 18
ANSI C, Strict 65
ANSI Standard Library Support 14
arrays 111
    as dummy pointer targets 101
    initialization 103
ASCII Character Set 209
ASCII characters
    extended 103
asm 166
asm C statement 42
assembly code
    mixing with C 163
assembly list files 59
Assembly Options 80
    -Wa 80
attribute 105
Attribute, Function
    address 123
    alias (symbol) 124
    always_inline 124
    at_vector 124
    const 124
    deprecated 124
    far 124
    format (type, format_index, first_to_check) 124
    format_arg (index) 125
    interrupt 125
    longcall 125
    malloc 125
    mips16 125
    naked 125
    near 125
    noinline 125
    no_instrument_function 83
    nomips16 125

nonnull (index, ...) 125
noreturn 70, 126
pure 126
ramfunc) 126
section (name) 126
unique_section 127
unused 127
used 127
vector 127
warn_unused_result 127
weak 127
Attribute, Variable
aligned (n) 105, 106
cleanup (function) 106
deprecated 106
packed 106
section ("name") 106
unused 107
weak 107
at_vector Attribute 124
auto variables 110, 112
initilization 149
memory allocation 112–113
Automatic Variable 68, 69
-aux-info 63, 64

B

-B 47, 81
Bad Virtual Address Register 151
BadVAddr. *See* Bad Virtual Address Register
__bank qualifier 33
_BEV_EXCPT_ADDR 185, 188
biased exponent 96
binary constants
C code 102
Bit Fields 64, 65
bit-fields 26, 27, 98–99
bitwise complement operator 118
BMXDKPBA 150
_bmxdkpba_address 150, 157
BMXDUDBA 150
_bmxdudba_address 150, 157
BMXDUPBA 150
_bmxdupba_address 150, 157
Boot Memory Region
kseg0_boot_mem 185
kseg1_boot_mem 185
Bootstrap Exception 142
_bootstrap_exception_handler 158
_bootstrap_exception_handler() 142
Branch Delay 153
_bss_begin 194
_bss_end 194
Bus Matrix Register 150
BMXDKPBA 150
BMXDUDBA 150
BMXDUPBA 150

C

-C 77
-c 62, 80
C Dialect Control Options 63, 64
-ansi 63, 64
-aux-info 63, 64
-ffreestanding 63
-fno-asm 63, 64
-fno-builtin 63, 64
-fno-signed-bitfields 64, 65
-fno-unsigned-bitfields 64, 65
-fsigned-bitfields 64, 65
-fsigned-char 64, 65
-funsigned-bitfields 64, 65
-funsigned-char 64, 65
-fwritable-strings 64, 65
-traditional 131
-ffreestanding 64
C Stack Usage 112
C standard libraries 56
__C32_VERSION__ 179
Call Main 157
calloc 114
Case Ranges 120
Cast 68, 69, 70
casting 117
Cause 153
Cause Register 153
Cause Register 152
CCI 19
char 64, 65, 66, 94, 129
char data types 24
character constants
in C 103
CHAR_BIT 94
CHAR_MAX 94
CHAR_MIN 94
cleanup (function) 106
clearing variables 149
Code Generation Conventions Options 82
-fargument-alias 82
-fargument-noalias 82
-fargument-noalias-global 82
-fcall-saved 82
-fcall-used 82
-ffixed 82
-finstrument-functions 83
-fno-ident 83
-fno-short-double 83
-fno-verbose-asm 83
-fpack-struct 83
-fpcc-struct-return 83
-fshort-enums 83
-fverbose-asm 83
-fvolatile 83
-fvolatile-global 83
-fvolatile-static 83
Code Size, Reduce 72, 73
Command Line Option, Compiler
-fdate-sections 106
-ffunction-sections 126
-fshort-enums 203
-funsigned-bitfields 203
-funsigned-char 94

-iquote 204
-l 80
-mdebugger 188, 192
-mips16 ....56, 125
-mips16 -mno-float 56
-mlong-calls 125
-mno-float 56
-mprocessor 183
-o ex1.out 50
-O3 56
-O3 -mips16 56
-O3 -mips16 -mno-float 56
-O3 -mno-float 56
-Os 56
-Os -mips16 56
-Os -mips16 -mno-float 56
-Os -mno-float 56
-Wall 68
-Wnonnull 125
Command Line Option, Linker
--defsym 183
--defsym_min_stack_size 112
-L 183
Command-Line Simulator 14
Comments 66, 77
common compiler interface 19
Common Subexpression Elimination 73, 74, 75
Common Subexpressions 76
Compare 152
Compare Register 152
Compare Register 152
Compiler
Driver 45, 81
compiler operating mode 13
Conditional Expression 120
Conditionals with Omitted Operands 120
Config 154
Config Register 154
Config1 154
Config1 Register 154
Config2 155
Config2 Register 155
Config3 155
Config3 Register 155
confign 186
Configuration Memory Region
config3, config2, config1, config0 185
Configuration Pragma 86, 87
Configuration Words 86, 87
const 124
const objects
initialization 104
const objects
storage location 113
const qualifier 104
constants
C specifiers 102
character 103
string, see string literals 103
Contact Microchip Technology 226
conversion between types 117
Count 151, 152
Count Register 152
Count Register 152
$\mathtt{Count}_{DM}$ 152
CP0 151
CP0 Access Macros 89
CP0 Registers 151
Customer Support 226

D

-D 77, 78, 79
data memory 110
Data Memory Region
kseg1_data_mem 186
Data Memory Space 114
data types
floating point 97
size of 23, 97
_data_end 193
DBD. *See* Debug Branch Delay
_DBG_CODE_ADDR 185, 188
_DBG_EXCPT_ADDR 185, 188
-dD 77
Debug 155
Debug Branch Delay 156
Debug Exception Program Counter 156
Debug Exception Save Register 156
Debug Executive Memory Region
debug_exec_mem 185
Debug Register 155
Debug Register 152
Debug Sections 196
Debug2 156
Debug2 Register 156
debug_exec_mem 188
_DEBUGGER 188, 192
Debugging Information 72
Debugging Options 72
-g 72
-Q 72
-save-temps 72
Defining Global Register Variables 211
--defsym 183
--defsym _ebase_address=A 154
--defsym _min_heap_size=M 146
--defsym _min_stack_size=N 146
--defsym, _min_heap_size 114
--defsym_min_stack_size 112
DEPC. *See* Debug Exception Program Counter
__deprecate qualifier 38
deprecated Attribute 70
deprecated Attribute 106, 124
DeSave 156
device support 85
diagnostic files 59
Directories 78, 79
Directory Search Options 81
-B 47, 81
-specs= 81

-dM 77
-dN 77
Documentation
  Conventions 9
  Layout 8
double 83, 96, 129
driver
  input files 46
driver option
  CCI 43
  EXT 218
driver options 46, 60–82

E

-E 62, 77, 78, 79, 80
EBase 154
EBase Register 154
_ebase_address 154, 157
__eeprom qualifier 35
EJTAG$_{ver}$ 155
_end 157, 194
endianism 96
ENTRY 182
Environment Variables
  PIC32_C_INCLUDE_PATH 46
  PIC32_C_INCLUDE_PATH 46
  PIC32_COMPILER_PATH 46
  PIC32_EXEC_PREFIX 47
  PIC32_LIBRARY_ PATH 47
  TMPDIR 47
  XC32_C_INCLUDE_PATH 46
  XC32_COMPILER_PATH 46
  XC32_EXEC_PREFIX 47
  XC32_LIBRARY_ PATH 47
EPC 153
EPC Register 142, 153, 156
ERET 146
Error Control Options
  -pedantic-errors 65
  -Werror 70
Error Exception Program Counter 156
ErrorEPC. *See* Error Exception Program Counter
Exception Base Register 154
Exception Memory Region
  exception_mem 185
Exception Program Counter 153
Exception Vector 140
exception_mem 188
EXL Bit 153
exponent 96
extended character set 103
Extensions 79
EXTERN 183
extern 70, 76, 83, 132
External Interrupt Controller 153

F

F constant suffix 103
-falign-functions 73
-falign-labels 73
-falign-loops 73
far 124
__far qualifier 30
-fargument-alias 82
-fargument-noalias 82
-fargument-noalias-global 82
fatal error messages 60
-fcaller-saves 73
-fcall-saved 82
-fcall-used 82
-fcse-follow-jumps 73
-fcse-skip-blocks 74
-fdata-sections 74, 106
-fdefer-pop. *See* -fno-defer
-fexpensive-optimizations 74
-ffixed 82, 212
-fforce-mem 73, 76
-ffreestanding 63, 64
-ffunction-sections 74, 126
-fgcse 74
-fgcse-lm 74
-fgcse-sm 74
File Extensions 48
  file.c 48
  file.h 48
  file.i 48
  file.ii 48
  file.o 48
  file.S 48
  file.s 48
file types
  input 46
file.c 48
file.h 48
file.i 48
file.o 48
file.S 48
file.s 48
-finline-functions 70, 73, 76, 131
-finline-limit=n 76
-finstrument-functions 83
-fkeep-inline-functions 76, 132
-fkeep-static-consts 76
Flags, Positive and Negative 76, 82
float 83, 96, 129
float.h 96
floating-point constant suffixes 103
Floating-Point Format
  double 96
  float 96
  long double 96
floating-point types 97
  biased exponent 96
  exponent 96
  rounding 97
-fmove-all-movables 74
-fno 76, 82
-fno-asm 63, 64
-fno-builtin 63, 64
-fno-defer-pop 74
-fno-function-cse 76

- -fno-ident 83
- -fno-inline 77
- -fno-keep-static-consts 76
- -fno-peephole 74
- -fno-peephole2 74
- -fno-short-double 83
- -fno-show-column 77
- -fno-signed-bitfields 64, 65
- -fno-unsigned-bitfields 64, 65
- -fno-verbose-asm 83
- -fomit-frame-pointer 72, 73, 77
- -foptimize-register-move 74
- -foptimize-sibling-calls 77
- format (type, format_index, first_to_check) 124
- format_arg (index) 125
- fp 142
- -fpack-struct 83
- -fpcc-struct-return 83
- Frame Pointer (W14) 77, 82
- -freduce-all-givs 74
- -fregmove 74
- -frename-registers 74
- -frerun-cse-after-loop 74, 75
- -frerun-loop-opt 74
- -fschedule-insns 75
- -fschedule-insns2 75
- -fshort-enums 83, 203
- -fsigned-bitfields 64, 65
- -fsigned-char 64, 65
- -fstrength-reduce 75
- -fstrict-aliasing 73, 75
- -fsyntax-only 65
- -fthread-jumps 72, 75
- function
  - parameters 112, 128
  - pointers 101
  - specifiers 123
- functions
  - absolute 29
  - static 123
  - written in assembler 163
- -funroll-all-loops 73, 75
- -funroll-loops 73, 75
- -funsigned-bitfields 64, 65, 203
- -funsigned-char 64, 65, 94
- -fverbose-asm 83
- -fvolatile 83
- -fvolatile-global 83
- -fvolatile-static 83
- -fwritable-strings 64, 65

G

- -g 72
- -G num 61
- General Exception 142
- _general_exception_context() 142
- _general_exception_handler 158
- _GEN_EXCPT_ADDR 185, 188
- Global Register Variables 211
- -Gn 149
- _gp 148, 157, 192
- gp 142, 148

H

- -H 77
- Hardware Enable Register 151
- header file
  - search path 22
- Header Files 46, 48, 77, 78, 79
- header files 21, 161
  - device 85, 88
- Heap 146
- _heap 146, 157
- --help 63
- Hex File 50
- hexadecimal constants
  - C code 102
- hi 142
- High-Priority Interrupts 133
- HWREna 151

I

- -I 78, 79
- -I- 78, 79
- identifiers
  - unique length of 23
- -idirafter 78
- -imacros 78, 79
- implementation-defined behavior 92
- -include 78, 79
- Include Files 81
- incremental builds 51
- Inhibit Warnings 65
- initialized variables 149
- Inline 70, 73, 76
- inline 77, 83, 131
- INPUT 183
- input files 46
- int 94, 129
- IntCtl 152
- integer constants 102
- integer suffixes 102
- Integer Values
  - char 94
  - int 94
  - long 94
  - long long 94
  - short 94
  - signed char 94
  - signed int 94
  - signed long 94
  - signed long long 94
  - signed short 94
  - unsigned char 94
  - unsigned int 94
  - unsigned long 94
  - unsigned long long 94
  - unsigned short 94
- integral promotion 117
- Internet Address, Microchip 226
- Interrupt

High Priority 133
Lower Priority 133
interrupt 125
Interrupt Attribute 134
interrupt Attribute 125
Interrupt Control Register 152
interrupt functions
context switching 143
interrupt handler function 134
Interrupt Pragma 135
Interrupt Pragma Clause 140
__interrupt qualifier 35
INT_MAX 95
INT_MIN 95
-iquote 204
__ISR(v, ipl) 136
__ISR_AT_VECTOR(v, ipl) 137
-isystem 81

K

k0 142
k1 142
kseg0_program_mem 189, 191, 192, 193
KSEG1 Data Memory 146
kseg1_boot_mem 187, 188
kseg1_data_mem 192, 193, 194

L

-L 80, 81, 183
-l 80
L constant suffix 102
__LANGUAGE_ASSEMBLY 178
__LANGUAGE_ASSEMBLY__ 178
_LANGUAGE_ASSEMBLY 178
LANGUAGE_ASSEMBLY 178
__LANGUAGE_C 178
__LANGUAGE_C__ 178
_LANGUAGE_C 178
LANGUAGE_C 178
libraries 55
search order 46
user-defined 57
Library 80
ANSI Standard 14
limits.h 94
CHAR_BIT 94
CHAR_MAX 94
CHAR_MIN 94
INT_MAX 95
INT_MIN 95
LLONG_MAX 95
LLONG_MIN 95
LONG_MAX 95
LONG_MIN 95
MB_LEN_MAX 94
SCHAR_MAX 94
SCHAR_MIN 94
SHRT_MAX 94
SHRT_MIN 94
UCHAR_MAX 94
UINT_MAX 95
ULLONG_MAX 95
ULONG_MAX 95
USHRT_MAX 94
link 206
Linker 80
linker scripts 181
Linking Options 80
-L 80, 81
-l 80
-nodefaultlibs 80
-nostdlib 80
-s 81
-u 81
-Wl 81
-Xlinker 81
little endian format 96
little-endian 93
LLONG_MAX 95
LLONG_MIN 95
lo 142
Local Register Variables 211, 212
long 94, 129
Long double 129
long double 83, 96
long long 70, 94, 129
longcall 125
longcall Attribute 127
LONG_MAX 95
LONG_MIN 95
__longramfunc__ 127
Loop Optimizer 74
Loop Unrolling 75
Lower-Priority Interrupts 133

M

-M 78
macro 77, 78, 79, 132
Macros
__C32_VERSION_ 179
__LANGUAGE_ASSEMBLY 178
__LANGUAGE_ASSEMBLY__ 178
_LANGUAGE_ASSEMBLY 178
LANGUAGE_ASSEMBLY 178
__LANGUAGE_C 178
__LANGUAGE_C__ 178
_LANGUAGE_C 178
LANGUAGE_C 178
_mchp_no_float 178
_MCHP_SZINT 178
_MCHP_SZLONG 178
_MCHP_SZPTR 178
__NO_FLOAT 178
__PIC__ 178
__pic__ 178
__PIC32_FEATURE_SET__ 178
__PIC32MX 178
__PIC32MX__ 178
PIC32MX 178
__processor__ 179
__VERSION__ 179
main 145

main function 21, 145
main-line code 133
malloc 114, 125
mantissa 96
map files 59
-mappio-debug 61
MB_LEN_MAX 94
-mcheck-zero-division 61
MCHP 178
_mchp_no_float 178
_MCHP_SZINT 178
_MCHP_SZLONG 178
_MCHP_SZPTR 178
-MD 79
-mdebugger 188, 192
-membedded-data 61
memory allocation 109
    data memory 110
    function code 127
    non-auto variables 110
    program memory 113
    static variables 110
memory models 114
messages
    error, see error messages
    fatal error 60
    types of 60
-MF 79
-MG 79
_min_heap_size 146, 183
_min_stack_size 146, 183
MIPS 180
__mips 180
__mips__ 180
_mips 180
__mips16 180
-mips16 ....56, 61, 125
mips16 125
-mips16 -mno-float 56
_MIPS_ARCH_PIC32MX 180
__MIPSEL 180
__MIPSEL__ 180
_MIPSEL 180
MIPSEL 180
_mips_fpr 180
_MIPS_ISA 180
__mips_isa_rev 180
_mips_no_float 180
__mips_soft_float 180
_MIPS_SZINT 180
_MIPS_SZLONG 180
_MIPS_SZPTR 180
_MIPS_TUNE_PIC32MX 180
-mlong-calls 61, 125
-MM 79
-MMD 79
-mmemcpy 61
-mno-check-zero-division 61
-mno-embedded-data 61
-mno-float 56, 61
-mno-long-calls 61
-mno-memcpy 61
-mno-mips16 ....56, 61
-mno-peripheral-libs 61
-mno-uninit-const-in-rodata 62
-MP 79
-mprocessor 61, 183
-MQ 79
-msoft-float 56
-MT 79
MTC0 Instruction 153
-muninit-const-in-rodata 62
myMicrochip Personalized Notification Service 225

N

naked 125
near 125
__near qualifier 31
_nmi_handler 146
-nodefaultlibs 80
__NO_FLOAT 178
noinline 125
no_instrument_function Attribute 83
NOLOAD 188, 192
nomips16 ....125, 146, 158
nonnull (index, ...) 125
non-volatile RAM 104
NOP 189
noreturn 126
noreturn Attribute 70
-nostdinc 78, 79
-nostdlib 80
NULL macro 28
NULL pointers 101

O

-O 72
-o 50, 62
-o ex1.out 50
-O0 ....56, 72
-O1 72
-O2 ....73, 76
-O3 ....56, 73
-O3 -mips16 56
-O3 -mips16 -mno-float 56
-O3 -mno-float 56
Object File 74, 78, 80
Omitted Operands 120
_on_reset 159
Optimization Control Options 72
    -falign-functions 73
    -falign-labels 73
    -falign-loops 73
    -fcaller-saves 73
    -fcse-follow-jumps 73
    -fcse-skip-blocks 74
    -fdata-sections 74
    -fexpensive-optimizations 74
    -fforce-mem 76
    -ffunction-sections 74
    -fgcse 74

  - -fgcse-lm 74
  - -fgcse-sm 74
  - -finline-functions 76
  - -finline-limit=n 76
  - -fkeep-inline-functions 76
  - -fkeep-static-consts 76
  - -fmove-all-movables 74
  - -fno-defer-pop 74
  - -fno-function-cse 76
  - -fno-inline 77
  - -fno-peephole 74
  - -fno-peephole2 74
  - -fomit-frame-pointer 77
  - -foptimize-register-move 74
  - -foptimize-sibling-calls 77
  - -freduce-all-givs 74
  - -fregmove 74
  - -frename-registers 74
  - -frerun-cse-after-loop 74
  - -frerun-loop-opt 74
  - -fschedule-insns 75
  - -fschedule-insns2 75
  - -fstrength-reduce 75
  - -fstrict-aliasing 75
  - -fthread-jumps 75
  - -funroll-all-loops 75
  - -funroll-loops 75
  - -O 72
  - -O0 72
  - -O1 72
  - -O2 73
  - -O3 73
  - -Os 73
- Optimization, Loop 74
- Optimization, Peephole 74
- Options
  - Assembling 80
  - C Dialect Control 63, 64
  - Code Generation Conventions 82
  - Debugging 72
  - Directory Search 81
  - Linking 80
  - Optimization Control 72
  - Output Control 62
  - Preprocessor Control 77
  - Warnings and Errors Control 65
- -Os 56, 73
- -Os -mips16 56
- -Os -mips16 -mno-float 56
- -Os -mno-float 56
- Output Control Options 62
  - -c 62
  - -E 62
  - --help 63
  - -o 62
  - -S 62
  - -v 62
  - -x 62
- output files
  - names of 58
- OUTPUT_ARCH 182
- OUTPUT_FORMAT 182

P

- -P 79
- __pack qualifier 37
- packed 106
- parameters, see function, parameters
- -pedantic 65, 70
- -pedantic-errors 65
- Peephole Optimization 74
- __persisten qualifier 32
- persistent qualifier 149
- persistent qualifier 159
- __PIC__ 178
- __pic__ 178
- PIC32_C_INCLUDE_PATH 46
- PIC32_C_INCLUDE_PATH 46
- PIC32_COMPILER_PATH 46
- PIC32_EXEC_PREFIX 47
- __PIC32_FEATURE_SET__ 178
- PIC32_LIBRARY_ PATH 47
- __PIC32MX 178
- __PIC32MX__ 178
- PIC32MX 178
- PIC32MX Device-Specific Options
  - -G num 61
  - -mappio-debug 61
  - -mcheck-zero-division 61
  - -membedded-data 61
  - -mips16 61
  - -mlong-calls 61
  - -mmemcpy 61
  - -mno-check-zero-division 61
  - -mno-embedded-data 61
  - -mno-float 61
  - -mno-long-calls 61
  - -mno-memcpy 61
  - -mno-mips16 61
  - -mno-peripheral-libs 61
  - -mno-uninit-const-in-rodata 62
  - -mprocessor 61
  - -muninit-const-in-rodata 62
- PIC32MX Start-up Code 146
- pointer
  - comparisons 101
  - definitions 100
  - qualifiers 100
  - types 100
- Pointers 70
  - Frame 77, 82
  - Stack 82
- pointers 100–101
  - assigning dummy targets 101
  - assigning integers 101
  - function 101
- powerup routine 159
- pragma directives 177
- Pragmas
  - #pragma config 86, 87, 177
  - #pragma interrupt 177

#pragma vector 177
Predefined Macros 178
prefix 81
preprocessing 175
preprocessor
    type conversions 176
Preprocessor Control Options 77
    -C 77
    -D 77
    -dD 77
    -dM 77
    -dN 77
    -fno-show-column 77
    -H 77
    -I 78
    -I- 78
    -idirafter 78
    -imacros 78
    -include 78
    -M 78
    -MD 79
    -MF 79
    -MG 79
    -MM 79
    -MMD 79
    -MQ 79
    -MT 79
    -nostdinc 79
    -P 79
    -trigraphs 79
    -U 79
    -undef 79
preprocessor directives 175–176
preprocessor macros
    predefined 41
PRId 154
__processor__ 179
Processor Identification Register 154
processor.o 183
program memory 113
Program Memory Region
    kseg0_program_mem 185
project name 59
PROVIDE 183
Provisions 58
pure 126

Q

-Q 72
qualifier
    __align 34
    auto 112
    __bank 33
    const 104
    __deprecate 38
    __eeprom 35
    __far 30
    __interrupt 35
    __near 31
    __pack 37
    __persistent 32
    persistent 149, 159
    __section 39
    volatile 104
    __xdata 33
    __ydata 33
qualifiers 104–105
    and auto variables 112
    and structures 98

R

__R3000 180
__R3000__ 180
_R3000 180
R3000 180
ra 142
radix specifiers
    C code 102
__ramfunc__ 127
ramfunc 126
_ramfunc_begin 157
_ramfunc_length 150, 157
RAW Dependency 75
RDHWR 151
Reading, Recommended 10
read-only variables 104
realloc 114
Reduce Code Size 72, 73
register 211, 212
Register Conventions 121
registers
    allocation to 114
Requested Interrupt Priority Level 153
_reset 182
reset
    code executed after 159
_RESET_ADDR 185, 187
Return Type 66
runtime startup code
    variable initialization 149
rx 186

S

-S 62, 80
-s 81
s0-s7 142
-save-temps 72
sbrk 146
_sbss_begin 194
_sbss_end 194
SCHAR_MAX 94
SCHAR_MIN 94
Scheduling 75
_sdata_begin 193
_sdata_end 193
SDE Compatibility Macros 180
    __mips 180
    __mips__ 180
    _mips 180
    __mips16 180
    _MIPS_ARCH_PIC32MX 180
    __MIPSEL 180

- __MIPSEL__ 180
- _MIPSEL 180
- MIPSEL 180
- _mips_fpr 180
- _MIPS_ISA 180
- __mips_isa_rev 180
- _mips_no_float 180
- __mips_soft_float 180
- _MIPS_SZINT 180
- _MIPS_SZLONG 180
- _MIPS_SZPTR 180
- _MIPS_TUNE_PIC32MX 180
- __R3000 180
- __R3000__ 180
- _R3000 180
- R3000 180

Section
- Configuration Words 86

section 74
section ("name") 106
section (name) 126
__section qualifier 39
SECTIONS Command 186, 187
SFR Memory Region
- sfrs 186

SFRs 88
Shadow Register Control Register 153
Shadow Register Map Register 153
short 94, 129
SHRT_MAX 94
SHRT_MIN 94
sign bit 96
signed char 94
signed int 94
signed long 94
signed long long 94
signed short 94
Simulator, Command-Line 14
SI_TimerInt 152
size of types 97
sp 142, 146
Special Function Registers 50
special function registers, see SFRs
Specifying Registers for Local Variables 212
-specs= 81
SR 142
SRSCtl 153
SRSMap 153
Stack
- C Usage 112
- Pointer (W15) 82, 86

_stack 146, 157, 195
Stack Location 195
Stack Pointer 146
Standard I/O Functions 14
standard library files
Start-up and Initialization 58
- For C 58
- For C++ 58

static 83
static functions 123
static variables 110
static variables 149
Status 152
Status Register 152
Status$_{BEV}$ 154, 158
storage duration 110
string literals 103
- concatenation 104
- storage location 103
- type of 103

Strings 64, 65
struct types, see structures
Structure 129
structure bit-fields 98
structure qualifiers 98
structures 98
- bit-fields in 98

switch 67
symbol 81
Syntax Check 65
System Function
- link 206
- unlink 206

System Header Files 67, 79

T

t0-t9 142
temporary variables 112
TMPDIR 47
Trace Control Register 155
TraceBPC 155
TraceBPC Register 155
TraceControl 155
TraceControl2 155
-traditional 64, 65, 131
Traditional C 71
Trigraphs 67, 79
-trigraphs 79
Type Conversion 70
type conversions 117

U

-U 77, 78, 79
-u 81
U constant suffix 102
UCHAR_MAX 94
UINT_MAX 95
ULLONG_MAX 95
ULONG_MAX 95
-undef 79
uninitialized variables 149
unions
- anonymous 99
- qualifiers 98

unique_section 127
unlink 206
unnamed bit-fields 99
unnamed structure members 99
Unroll Loop 75
unsigned char 94

- unsigned int, 94
- unsigned long, 94
- unsigned long long, 94
- unsigned short, 94
- unused Attribute, 68, 107, 127
- Unused Function Parameter, 68
- Unused Variable, 68
- unused variables
  - removing, 104
- USB, 244
- used Attribute, 127
- User Trace Data Register, 155
- UserTraceData, 155
- USHRT_MAX, 94

V

- -v, 62
- v0, 142
- v1, 142
- variables
  - absolute, 29
  - auto, 112
  - in program memory, 113–114
  - in registers, 114
  - initialization, 149
  - sizes, 97
  - static, 110
  - storage duration, 110
- Variables in Specified Registers, 211
- vector, 127
- Vector Pragma, 141
- _vector_spacing, 152, 157, 184
- __VERSION__, 179
- volatile, 83
- volatile qualifier, 104

W

- -W, 65, 68, 69, 71
- -w, 65
- w!x, 186
- -Wa, 80
- -Waggregate-return, 69
- -Wall, 65, 68, 69, 71
- warning messages, 60
- Warnings and Errors Control Options, 65
  - -fsyntax-only, 65
  - -pedantic, 65
  - -pedantic-errors, 65
  - -W, 69
  - -w, 65
  - -Waggregate-return, 69
  - -Wall, 65
  - -Wbad-function-cast, 69
  - -Wcast-align, 69
  - -Wcast-qual, 69
  - -Wchar-subscripts, 66
  - -Wcomment, 66
  - -Wconversion, 70
  - -Wdiv-by-zero, 66
  - -Werror, 70
  - -Wformat, 66
  - -Wimplicit, 66
  - -Wimplicit-function-declaration, 66
  - -Wimplicit-int, 66
  - -Winline, 70
  - -Wlarger-than-, 70
  - -Wlong-long, 70
  - -Wmain, 66
  - -Wmissing-braces, 66
  - -Wmissing-declarations, 70
  - -Wmissing-format-attribute, 70
  - -Wmissing-noreturn, 70
  - -Wmissing-prototypes, 70
  - -Wmultichar, 66
  - -Wnested-externs, 70
  - -Wno-long-long, 70
  - -Wno-sign-compare, 71
  - -Wpadded, 70
  - -Wparentheses, 66
  - -Wpointer-arith, 70
  - -Wredundant-decls, 70
  - -Wreturn-type, 66
  - -Wsequence-point, 67
  - -Wshadow, 70
  - -Wsign-compare, 71
  - -Wstrict-prototypes, 71
  - -Wswitch, 67
  - -Wsystem-headers, 67
  - -Wtraditional, 71
  - -Wtrigraphs, 67
  - -Wundef, 71
  - -Wuninitialized, 68
  - -Wunknown-pragmas, 68
  - -Wunreachable-code, 71
  - -Wunused, 68
  - -Wunused-function, 68
  - -Wunused-label, 68
  - -Wunused-parameter, 68
  - -Wunused-value, 68
  - -Wunused-variable, 68
  - -Wwrite-strings, 71
  - -Wno-multichar, 66
- Warnings, Inhibit, 65
- warn_unused_result, 127
- Watchdog Timer, 244
- -Wbad-function-cast, 69
- -Wcast-align, 69
- -Wcast-qual, 69
- -Wchar-subscripts, 66
- -Wcomment, 66
- -Wconversion, 70
- -Wdiv-by-zero, 66
- weak, 107, 127
- Web Site, Microchip, 226
- -Werror, 70
- -Wformat, 66, 70
- -Wimplicit, 66
- -Wimplicit-function-declaration, 66
- -Wimplicit-int, 66
- -Winline, 70, 132
- -Wl, 81

-Wlarger-than- 70
-Wlong-long 70
-Wmain 66
-Wmissing-braces 66
-Wmissing-declarations 70
-Wmissing-format-attribute 70
-Wmissing-noreturn 70
-Wmissing-prototypes 70
-Wmultichar 66
-Wnested-externs 70
-Wno- 65
-Wno-deprecated-declarations 70
-Wno-div-by-zero 66
-Wno-long-long 70
-Wno-multichar 66
-Wnonnull 125
-Wno-sign-compare 69, 71
-Wpadded 70
-Wparentheses 66
-Wpointer-arith 70
-Wredundant-decls 70
-Wreturn-type 66
-Wsequence-point 67
-Wshadow 70
-Wsign-compare 71
-Wstrict-prototypes 71
-Wswitch 67
-Wsystem-headers 67
-Wtraditional 71
-Wtrigraphs 67
-Wundef 71
-Wuninitialized 68
-Wunknown-pragmas 67, 68
-Wunreachable-code 71
-Wunused 68, 69
-Wunused-function 68
-Wunused-label 68
-Wunused-parameter 68
-Wunused-value 68
-Wunused-variable 68
-Wwrite-strings 71

X

-x 62
xc.h header file 85
XC32_C_INCLUDE_PATH 46
XC32_COMPILER_PATH 46
XC32_EXEC_PREFIX 47
xc32-gcc 45
XC32_LIBRARY_ PATH 47
__xdata qualifier 33
-Xlinker 81

Y

__ydata qualifier 33

Worldwide Sales and Service

AMERICAS

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

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Chongqing
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

China - Hangzhou
Tel: 86-571-2819-3187
Fax: 86-571-2819-3189

China - Hong Kong SAR
Tel: 852-2943-5100
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Osaka
Tel: 81-66-152-7160
Fax: 81-66-152-9310

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-5778-366
Fax: 886-3-5770-955

Taiwan - Kaohsiung
Tel: 886-7-213-7828
Fax: 886-7-330-9305

Taiwan - Taipei
Tel: 886-2-2508-8600
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

11/27/12