



Chipsmall Limited consists of a professional team with an average of over 10 year of expertise in the distribution of electronic components. Based in Hongkong, we have already established firm and mutual-benefit business relationships with customers from,Europe,America and south Asia,supplying obsolete and hard-to-find components to meet their specific needs.

With the principle of “Quality Parts,Customers Priority,Honest Operation,and Considerate Service”,our business mainly focus on the distribution of electronic components. Line cards we deal with include Microchip,ALPS,ROHM,Xilinx,Pulse,ON,Everlight and Freescale. Main products comprise IC,Modules,Potentiometer,IC Socket,Relay,Connector.Our parts cover such applications as commercial,industrial, and automotives areas.

We are looking forward to setting up business relationship with you and hope to provide you with the best service and solution. Let us make a better world for our industry!



Contact us

Tel: +86-755-8981 8866 Fax: +86-755-8427 6832

Email & Skype: info@chipsmall.com Web: www.chipsmall.com

Address: A1208, Overseas Decoration Building, #122 Zhenhua RD., Futian, Shenzhen, China





MICROCHIP

**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, KEELOQ, KEELOQ logo, MPLAB, PIC, PICmicro, PICSTART, PIC³² logo, rfPIC 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, MXDEV, MXLAB, SEEVAL and The Embedded Control Solutions Company are registered trademarks of Microchip Technology Incorporated in the U.S.A.

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

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

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

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

 Printed on recycled paper.

ISBN: 978-1-62076-455-8

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

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	127
10.5 Function Size Limits	128
10.6 Function Parameters	128
10.7 Function Return Values	130
10.8 Calling Functions	130
10.9 Inline Functions	130
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	139
11.5 Exception Handlers	141
11.6 Interrupt Service Routine Context Switching	141
11.7 Latency	142
11.8 Nesting Interrupts	142
11.9 Enabling/Disabling Interrupts	142
11.10 ISR Considerations	142
Chapter 12. Main, Runtime Start-up and Reset	
12.1 Introduction	143
12.2 The Main Function	143
12.3 Runtime Start-up Code	143
12.4 The On Reset Routine	157
Chapter 13. Library Routines	
13.1 Using Library Routines	159
Chapter 14. Mixing C/C++ and Assembly Language	
14.1 Introduction	161
14.2 Using Inline Assembly Language	161
14.3 Predefined Assembly Macros	164
Chapter 15. Optimizations	
15.1 Introduction	167
Chapter 16. Preprocessing	
16.1 Introduction	169
16.2 C/C++ Language Comments	169
16.3 Preprocessor Directives	169
16.4 Pragma Directives	171
16.5 Predefined Macros	172
Chapter 17. Linking Programs	
17.1 Introduction	175
17.2 Replacing Library Symbols	175
17.3 Linker-Defined Symbols	175
17.4 Default Linker Script	176
Appendix 18. Implementation-Defined Behavior	
18.1 Introduction	191
18.2 Highlights	191
18.3 Overview	191
18.4 Translation	192
18.5 Environment	192
18.6 Identifiers	193
18.7 Characters	193
18.8 Integers	194
18.9 Floating-Point	194
18.10 Arrays and Pointers	196

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

18.11 Hints	196
18.12 Structures, Unions, Enumerations, and Bit fields	197
18.13 Qualifiers	197
18.14 Declarators	198
18.15 Statements	198
18.16 Pre-Processing Directives	198
18.17 Library Functions	199
18.18 Architecture	202
Appendix 19. ASCII Character Set	
Appendix 20. Deprecated Features	
20.1 Introduction	205
20.2 Variables in Specified Registers	205
Glossary	207
Index	225
Worldwide Sales and Service	238



MPLAB® XC32 C/C++ COMPILER USER'S GUIDE

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
- myMicrochip Personalized Notification Service
- The Microchip Web Site
- Microchip Forums
- Customer Support

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 18. Implementation-Defined Behavior** – details compiler-specific parameters described as implementation-defined in the ANSI standard.
- **Appendix 19. ASCII Character Set** – contains the ASCII character set.
- **Appendix 20. Deprecated Features** – details features that are considered obsolete.

CONVENTIONS USED

The following conventions may appear in this documentation:

DOCUMENTATION CONVENTIONS

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	<i>MPLAB[®] IDE User's Guide</i>
	Emphasized text	...is the <i>only</i> compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	select Enable Programmer
Quotes	A field name in a window or dialog	"Save project before build"
Underlined, italic text with right angle bracket	A menu path	<u><i>File>Save</i></u>
Bold characters	A dialog button	Click OK
	A tab	Click the Power tab
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		
	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/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/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/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/>

myMICROCHIP PERSONALIZED NOTIFICATION SERVICE

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 <http://www.microchip.com/pcn> 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/C++ compilers, assemblers, linkers and other language tools. These include all MPLAB C/C++ compilers; all MPLAB assemblers (including MPASM™ assembler); all MPLAB linkers (including MPLINK™ object linker); and all MPLAB librarians (including MPLIB™ object librarian).
- **Emulators** – The latest information on Microchip in-circuit emulators. These include the MPLAB REAL ICE™ and MPLAB ICE 2000 in-circuit emulators
- **In-Circuit Debuggers** – The latest information on Microchip in-circuit debuggers. These include the MPLAB ICD 2 and 3 in-circuit debuggers and PICkit™ 2 and 3 debug express.
- **MPLAB IDE/MPLAB X IDE** – The latest information on Microchip MPLAB IDE, the Windows® Integrated Development Environment, or MPLAB X IDE, the open source, cross-platform Integrated Development Environment. These lists focus on the IDE, Project Manager, Editor and 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 MPLAB ICD 2 in-circuit debugger, PICSTART® Plus and PICkit 2 and 3.
- **Starter/Demo Boards** – These include MPLAB Starter Kit boards, PICDEM™ demo boards, and various other evaluation boards.

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

THE MICROCHIP WEB SITE

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

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

MICROCHIP FORUMS

Microchip provides additional online support via our web forums at <http://www.microchip.com/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 is available through the web site at <http://support.microchip.com>.

Documentation errors or comments may be emailed to docerrors@microchip.com.

DOCUMENT REVISION HISTORY

Revision D (January 2012)

- Changed product name from MPLAB C32 C Compiler to MPLAB XC32 C/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

Chapter 1. Compiler Overview

1.1 INTRODUCTION

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

1.3 COMPILER DESCRIPTION AND DOCUMENTATION

The MPLAB XC32 C/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/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/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

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

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/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 conform¹. But, if it is too weak, programmers would see wildly differing results within different compilers and architectures, and the standard would lose its effectiveness.

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

Implementation-defined behavior

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

Unspecified behavior

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

Undefined behavior

This is behavior for which the standard imposes no requirements.

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

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

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 illustrates 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 conform to the CCI.

Observe refinements to ANSI by the CCI

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

Use the CCI extensions to the language

Use the CCI extensions rather than the native language extensions

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

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

2.4 ANSI STANDARD REFINEMENT

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

2.4.1 Source File Encoding

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

2.4.1.1 EXAMPLE

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

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

2.4.1.2 DIFFERENCES

All compilers have used this character set.

2.4.1.3 MIGRATION TO THE CCI

No action required.

2.4.2 The Prototype for `main`

The prototype for the `main()` function is

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

2.4.2.1 EXAMPLE

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

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

2.4.2.2 DIFFERENCES

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

2.4.2.3 MIGRATION TO THE CCI

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

2.4.3 Header File Specification

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

2.4.3.1 EXAMPLE

The following example shows two conforming include directives.

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

2.4.3.2 DIFFERENCES

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

2.4.3.3 MIGRATION TO THE CCI

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

For example, the following code:

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

should be changed to:

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

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

```
-Ilcd
```

2.4.4 Include Search Paths

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

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

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

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

2.4.4.1 EXAMPLE

If including a header file as in the following directive

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

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

2.4.4.2 DIFFERENCES

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

2.4.4.3 MIGRATION TO THE CCI

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

2.4.5 The Number of Significant Initial Characters in an Identifier

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

2.4.5.1 EXAMPLE

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

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

2.4.5.2 DIFFERENCES

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

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

2.4.5.3 MIGRATION TO THE CCI

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

2.4.6 Sizes of Types

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

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

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

2.4.6.1 EXAMPLE

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

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

2.4.6.2 DIFFERENCES

This is consistent with previous types implemented by the compiler.

2.4.6.3 MIGRATION TO THE CCI

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

2.4.7 Plain char Types

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

2.4.7.1 EXAMPLE

The following example

```
char foobar;
```

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

2.4.7.2 DIFFERENCES

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

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

2.4.7.3 MIGRATION TO THE CCI

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

```
signed char foobar;
```

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

2.4.8 Signed Integer Representation

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

2.4.8.1 EXAMPLE

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

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

2.4.8.2 DIFFERENCES

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

2.4.8.3 MIGRATION TO THE CCI

No action required.

2.4.9 Integer conversion

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

2.4.9.1 EXAMPLE

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

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

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

2.4.9.2 DIFFERENCES

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

2.4.9.3 MIGRATION TO THE CCI

No action required.

2.4.10 Bit-wise Operations on Signed Values

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

2.4.10.1 EXAMPLE

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

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

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

2.4.10.2 DIFFERENCES

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

2.4.10.3 MIGRATION TO THE CCI

No action required.

2.4.11 Right-shifting Signed Values

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

2.4.11.1 EXAMPLE

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

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

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