



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<sup>®</sup> XC8 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, PIC<sup>32</sup> 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, PCKit, 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: 978-1-62076-375-9

**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<sup>®</sup> MCUs and dsPIC<sup>®</sup> DSCs, KEELOQ<sup>®</sup> 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**

<b>Preface</b> .....	<b>7</b>
<b>Chapter 1. Compiler Overview</b>	
1.1 Introduction .....	11
1.2 Compiler Description and Documentation .....	11
1.3 Device Description .....	12
<b>Chapter 2. Common C Interface</b>	
2.1 Introduction .....	13
2.2 Background – The Desire for Portable Code .....	13
2.3 Using the CCI .....	16
2.4 ANSI Standard Refinement .....	17
2.5 ANSI Standard Extensions .....	25
2.6 Compiler Features .....	39
<b>Chapter 3. How To's</b>	
3.1 Introduction .....	41
3.2 Installing and Activating the Compiler .....	41
3.3 Invoking the Compiler .....	43
3.4 Writing Source Code .....	46
3.5 Getting My Application to Do What I Want .....	56
3.6 Understanding the Compilation Process .....	60
3.7 Fixing Code That Does Not Work .....	67
<b>Chapter 4. XC8 Command-line Driver</b>	
4.1 Introduction .....	71
4.2 Invoking the Compiler .....	72
4.3 The Compilation Sequence .....	75
4.4 Runtime Files .....	81
4.5 Compiler Output .....	84
4.6 Compiler Messages .....	86
4.7 XC8 Driver Options .....	91
4.8 Option Descriptions .....	92
4.9 MPLAB IDE V8 Universal Toolsuite Equivalents .....	117
4.10 MPLAB X Universal Toolsuite Equivalents .....	124
<b>Chapter 5. C Language Features</b>	
5.1 Introduction .....	131
5.2 ANSI C Standard Issues .....	131
5.3 Device-Related Features .....	133
5.4 Supported Data Types and Variables .....	143
5.5 Memory Allocation and Access .....	165

# MPLAB® XC8 C Compiler User's Guide

---

5.6 Operators and Statements .....	179
5.7 Register Usage .....	181
5.8 Functions .....	182
5.9 Interrupts .....	189
5.10 Main, Runtime Startup and Reset .....	194
5.11 Library Routines .....	198
5.12 Mixing C and Assembly Code .....	200
5.13 Optimizations .....	208
5.14 Preprocessing .....	210
5.15 Linking Programs .....	222
<b>Chapter 6. Macro Assembler</b>	
6.1 Introduction .....	241
6.2 Assembler Usage .....	241
6.3 Options .....	242
6.4 MPLAB XC8 Assembly Language .....	246
6.5 Assembly-Level Optimizations .....	268
6.6 Assembly List Files .....	269
<b>Chapter 7. Linker</b>	
7.1 Introduction .....	277
7.2 Operation .....	277
7.3 Relocation and Psects .....	285
7.4 Map Files .....	286
<b>Chapter 8. Utilities</b>	
8.1 Introduction .....	291
8.2 Librarian .....	291
8.3 OBJTOHEX .....	295
8.4 CREF .....	297
8.5 CROMWELL .....	300
8.6 HEXMATE .....	303
<b>Appendix A. Library Functions</b>	
<b>Appendix B. Error and Warning Messages</b>	
<b>Appendix C. Implementation-Defined Behavior</b>	
C.1 Translation (G.3.1) .....	479
C.2 Environment (G.3.2) .....	479
C.3 Identifiers (G.3.3) .....	480
C.4 Characters (G.3.4) .....	480
C.5 Integers (G.3.5) .....	481
C.6 Floating-Point (G.3.6) .....	482
C.7 Arrays and Pointers (G.3.7) .....	482
C.8 Registers (G.3.8) .....	482
C.9 Structures, Unions, Enumerations, and Bit-Fields (G.3.9) .....	483
C.10 Qualifiers (G.3.10) .....	483
C.11 Declarators (G.3.11) .....	483

---

---

C.12 Statements (G.3.12) .....	483
C.13 Preprocessing Directives (G.3.13) .....	484
C.14 Library Functions (G.3.14) .....	485
<b>Glossary .....</b>	<b>487</b>
<b>Index .....</b>	<b>507</b>
<b>Worldwide Sales and Service .....</b>	<b>518</b>

# MPLAB® XC8 C Compiler User's Guide

---

NOTES:

---

---

## Preface

---

---

### NOTICE TO CUSTOMERS

All documentation becomes dated, and this manual is no exception. Microchip tools and documentation are constantly evolving to meet customer needs, so some actual dialogs and/or tool descriptions may differ from those in this document. Please refer to our web site ([www.microchip.com](http://www.microchip.com)) to obtain the latest documentation available.

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

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

## INTRODUCTION

This chapter contains general information that will be useful to know before using the MPLAB® XC8 C Compiler User's Guide. Items discussed in this chapter include:

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

## DOCUMENT LAYOUT

This document describes how to use the MPLAB XC8 C Compiler. The manual layout is as follows:

- **Chapter 1. Compiler Overview**
- **Chapter 3. How To's**
- **Chapter 4. XC8 Command-line Driver**
- **Chapter 5. C Language Features**
- **Chapter 6. Macro Assembler**
- **Chapter 7. Linker**
- **Chapter 8. Utilities**
- **Appendix A. Library Functions**
- **Appendix B. Error and Warning Messages**
- **Appendix C. Implementation-Defined Behavior**
- **Glossary**
- **Index**



# MPLAB® XC8 C Compiler User's Guide

## CONVENTIONS USED IN THIS GUIDE

This manual uses the following documentation conventions:

### DOCUMENTATION CONVENTIONS

Description	Represents	Examples
<b>Arial font:</b>		
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&gt;Save</i></u>
Bold characters	A dialog button	Click <b>OK</b>
	A tab	Click the <b>Power</b> tab
N'Rnnnn	A number in verilog format, where N is the total number of digits, R is the radix and n is a digit.	4'b0010, 2'hF1
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
<b>Courier New font:</b>		
Plain Courier New	Sample source code	#define START
	Filenames	autoexec.bat
	File paths	c:\mcc18\h
	Keywords	_asm, _endasm, static
	Command-line options	-Opa+, -Opa-
	Bit values	0, 1
	Constants	0xFF, 'A'
Italic Courier New	A variable argument	<i>file.o</i> , where <i>file</i> can be any valid filename
Square brackets [ ]	Optional arguments	mcc18 [options] <i>file</i> [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) { ... }

## WARRANTY REGISTRATION

Please complete the enclosed Warranty Registration Card and mail it promptly. Sending in the Warranty Registration Card entitles users to receive new product updates. Interim software releases are available at the Microchip web site.

## RECOMMENDED READING

This user's guide describes how to use Chapter Name. Other useful documents are listed below. The following Microchip documents are available and recommended as supplemental reference resources.

## Readme for Chapter Name

For the latest information on using Chapter Name, read the “Readme for Chapter Name.txt” file (an ASCII text file) in the Readmes subdirectory of the MPLAB® IDE installation directory. The Readme file contains update information and known issues that may not be included in this user’s guide.

## Readme Files

For the latest information on using other tools, read the tool-specific Readme files in the Readmes subdirectory of the MPLAB IDE installation directory. The Readme files contain update information and known issues that may not be included in this user’s guide.

## THE MICROCHIP WEB SITE

Microchip provides online support via our web site at [www.microchip.com](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

## DEVELOPMENT SYSTEMS CUSTOMER CHANGE NOTIFICATION SERVICE

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

To register, access the Microchip web site at [www.microchip.com](http://www.microchip.com), click on Customer Change Notification and follow the registration instructions.

The Development Systems product group categories are:

- **Compilers** – The latest information on Microchip C compilers and other language tools. These include the MPLAB® C18 and MPLAB® C30 C compilers; MPASM™ and MPLAB® ASM30 assemblers; MPLINK™ and MPLAB LINK30 object linkers; and MPLIB™ and MPLAB® LIB30 object librarians.
- **Emulators** – The latest information on Microchip in-circuit emulators. This includes the MPLAB® ICE 2000 and MPLAB ICE 4000.
- **In-Circuit Debuggers** – The latest information on the Microchip In-Circuit Debugger, MPLAB® ICD 2.
- **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® SIM simulator, MPLAB® IDE Project Manager and general editing and debugging features.
- **Programmers** – The latest information on Microchip programmers. These include the MPLAB® PM3 and PRO MATE® II device programmers and the PICSTART® Plus and PICKit™ 1 development programmers.

## CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

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

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

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

## DOCUMENT REVISION HISTORY

### Revision B (July 2012)

- Added 'how tos' chapter.
- Expanded section relating to PIC18 erratas.
- Updated the section relating to compiler optimization settings.
- Updated MPLAB v8 and MPLAB X IDE project option dialogs.
- Added sections describing PIC18 far qualifier and inline function qualifier.
- Expanded section describing the operation of the main() function
- Expanded information about equivalent assembly symbols for Baseline parts.
- Updated the table of predefined macro symbols.
- Added section on `#pragma addrqual`
- Added sections to do with inline-ing functions
- Updated diagrams and text associated with call graphs in the list file
- Updated library function section to be consistent with packaged libraries
- Added new compiler warnings and errors.
- Added new chapter describing the Common Compiler Interface Standard (CCI)

### Revision A (February 2012)

Initial release of this document.

---

---

## Chapter 1. Compiler Overview

---

---

### 1.1 INTRODUCTION

This chapter is an overview of the MPLAB XC8 C Compiler, including these topics.

- Compiler Description and Documentation
- Device Description

### 1.2 COMPILER DESCRIPTION AND DOCUMENTATION

The MPLAB® XC8 C Compiler is a free-standing, optimizing ANSI C compiler. It supports all 8-bit PIC® microcontrollers: PIC10, PIC12, PIC16 and PIC18 series devices, as well as the PIC14000 device.

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

The compiler is available in three operating modes: Free, Standard or PRO. The Standard and PRO operating modes are licensed modes and require a serial number 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.

#### 1.2.1 Conventions

Throughout this manual, the term “compiler” is used. It can refer to all, or a subset of, the collection of applications that comprise the MPLAB XC8 C Compiler. When it is not important to identify which application performed an action, it will be attributed to the compiler.

Likewise, “compiler” is often used to refer to the command-line driver. Although specifically, the driver for the MPLAB XC8 C Compiler package is called `xc8`. The driver and its options are discussed in **Section 4.7 “XC8 Driver Options”**. Accordingly, “compiler options” commonly relates to command-line driver options.

In a similar fashion, “compilation” refers to all or a selection of steps involved in generating source code into an executable binary image.

## 1.3 DEVICE DESCRIPTION

This compiler supports 8-bit Microchip PIC devices with baseline, Mid-Range, Enhanced Mid-Range, and PIC18 cores. The following descriptions indicate the distinctions within those device cores:

The baseline core uses a 12-bit-wide instruction set and is available in PIC10, PIC12 and PIC16 part numbers.

The Mid-Range core uses a 14-bit-wide instruction set that includes more instructions than the baseline core. It has larger data memory banks and program memory pages, as well. It is available in PIC12, PIC14 and PIC16 part numbers.

The Enhanced Mid-Range core also uses a 14-bit-wide instruction set, but incorporates additional instructions and features. There are both PIC12 and PIC16 part numbers that are based on the Enhanced Mid-Range core.

The PIC18 core instruction set is 16-bits wide and features additional instructions and an expanded register set. PIC18 core devices have part numbers that begin with PIC18.

The compiler takes advantage of the target device's instruction set, addressing modes memory and registers whenever possible.

See **Section 4.8.21 "--CHIPINFO: Display List of Supported Devices"** for information on finding the full list of devices supported by the compiler.

---

---

## 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 PIC24 MCU using the MPLAB XC16 C compiler.

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

The following topics are examined in this chapter of the MPLAB XC8 C Compiler User's Guide:

- ANSI Standard Extensions
- Using the CCI
- ANSI Standard Refinement
- ANSI Standard Extensions

### 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.<sup>1</sup> 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).

## 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 = foobar.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. 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.