Contact us

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

ISBN: 978-1-5224-0679-2

Table of Contents

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.

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

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

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

- Document Layout
- Conventions Used
- Recommended Reading

DOCUMENT LAYOUT

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

- Chapter 1. Compiler Overview – describes the compiler, development tools and feature set.
- Chapter 2. Common C Interface – explains what you need to know about making code portable.
- Chapter 3. How To's – contains help and references for frequently encountered situations when building projects.
- Chapter 4. XC32 Toolchain and MPLAB X IDE – guides you through the toolchain and IDE setup.
- Chapter 5. Compiler Command Line Driver – describes how to use the compiler from the command line.
- Chapter 6. ANSI C Standard Issues – describes the differences between the C/C++ language supported by the compiler syntax and the standard ANSI-89 C.
- Chapter 7. Device-Related Features – describes the compiler header and register definition files, as well as how to use them with the SFRs.
- Chapter 8. Supported Data Types and Variables – describes the compiler integer and pointer data types.
- Chapter 9. Memory Allocation and Access – describes the compiler run-time model, including information on sections, initialization, memory models, the software stack and much more.

- Chapter 10. Fixed-Point Arithmetic Support – describes the fixed-point types and operations supported.
- Chapter 11. Operators and Statements – discusses operators and statements.
- Chapter 12. Register Usage – explains how to access and use SFRs.
- Chapter 13. Functions – details available functions.
- Chapter 14. Interrupts – describes how to use interrupts.
- Chapter 15. Main, Runtime Start-up and Reset – describes important elements of C/C++ code.
- Chapter 16. Library Routines – explains how to use libraries.
- Chapter 17. Mixing C/C++ and Assembly Language – provides guidelines for using the compiler with 32-bit assembly language modules.
- Chapter 18. Optimizations – describes optimization options.
- Chapter 19. Preprocessing – details the preprocessing operation.
- Chapter 20. Linking Programs – explains how linking works.
- Appendix A. Embedded Compiler Compatibility Mode – discusses using the compiler in compatibility mode.
- Appendix B. Implementation-Defined Behavior – details compiler-specific parameters described as implementation-defined in the ANSI standard.
- Appendix C. Deprecated Features – details features that are considered obsolete.
- Appendix D. Built-In Functions – lists the built-in functions of the C compiler.
- Appendix E. ASCII Character Set – contains the ASCII character set.
- Appendix F. Document Revision History – information on previous and current revisions of this document.

CONVENTIONS USED

The following conventions may appear in this documentation:

DOCUMENTATION CONVENTIONS

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	*MPLAB® X IDE User's Guide*
	Emphasized text	...is the *only* compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	select Enable Programmer
Quotes	A field name in a window or dialog	"Save project before build"
Underlined, italic text with right angle bracket	A menu path	<u>*File>Save*</u>
Bold characters	A dialog button	Click **OK**
	A tab	Click the **Power** tab
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
Courier font:		
Plain Courier	Sample source code	`#define START`
	Filenames	`autoexec.bat`
	File paths	`c:\mcc18\h`
	Keywords	`_asm, _endasm, static`
	Command-line options	`-Opa+, -Opa-`
	Bit values	`0, 1`
	Constants	`0xFF, 'A'`
Italic Courier	A variable argument	*`file.o`*, where *`file`* can be any valid filename
Square brackets [ ]	Optional arguments	`mpasmwin [options] file [options]`
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; an OR selection	`errorlevel {0|1}`
Ellipses...	Replaces repeated text	`var_name [, var_name...]`
	Represents code supplied by user	`void main (void)` `{ ...` `}`
Sidebar Text		
DD	Device Dependent. This feature is not supported on all devices. Devices supported will be listed in the title or text.	`xmemory` attribute

RECOMMENDED READING

The MPLAB® XC32 language toolsuite for PIC32 MCUs consists of a C compilation driver (`xc32-gcc`), a C++ compilation driver (`xc32-g++`), an assembler (`xc32-as`), a linker (`xc32-ld`), and an archiver/librarian (`xc32-ar`). This document 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® XC32 Assembler, Linker and Utilities User's Guide (DS50002186)

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

32-Bit Language Tools Libraries (DS50001685)

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.

PIC32 Configuration Settings

Lists the Configuration Bit settings for the Microchip PIC32 devices supported by the `#pragma config` of the MPLAB XC32 C/C++ Compiler.

Device-Specific Documentation

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

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

C Standards Information

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

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

C++ Standards Information

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

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

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

C Reference Manuals

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

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

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

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

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

GCC Documents

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

http://sourceware.org/binutils/

NOTES:

Chapter 1. Compiler Overview

1.1 INTRODUCTION

The MPLAB XC32 C/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 Mac 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 5.9 "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 18. "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, timekeeping and math functions (trigonometric, exponential and hyperbolic). The standard I/O functions for file handling are also included, and, as distributed, they support full access to the host file system using the command-line simulator. The fully functional source code for the low-level file I/O functions is provided in the compiler distribution, and may be used as a starting point for applications that require this capability.

1.3.5 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 compilation drivers automatically select the XC libc and their respective include-file directory for you. The xc32-g++ compilation drivers automatically select the Dinkumware libc and their respective include-file directory for you. The Dinkum C libraries can only be used with the C++ compiler. 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 v8.84 or higher, and the MPLAB X IDE. For C++, MPLAB X IDE v1.50 or higher is required. For simplicity, both IDEs are referred to throughout this book as simply MPLAB IDE.

Note: Building an MPLAB XC32 C++ project and debugging C++ code requires MPLAB X IDE v1.50 (or later). MPLAB IDE v8.xx does not support C++ projects.

Features that are unique to specific devices, and therefore specific compilers, are noted with “DD” in 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® XC32 Assembler, Linker and Utilities User’s Guide”* (DS50002186).
- MPLAB IDE v8.xx and MPLAB X IDE (C++ required MPLAB X IDE v1.30 or higher)
- The MPLAB Simulator
- All Microchip debug tools and programmers
- Demo boards and starter kits that support 32-bit devices

NOTES:

Chapter 2. Common C Interface

2.1 INTRODUCTION

The Common C Interface (CCI) is available with all MPLAB XC C compilers and is designed to enhance code portability between these compilers. For example, CCI-conforming code would make it easier to port from a PIC18 MCU using the MPLAB XC8 C compiler to a PIC32 MCU using the MPLAB XC32 C/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 can 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 can 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 cannot allow the compiler to conform.[1] But, if it is too weak, programmers would see wildly differing results within different compilers and architectures, and the standard would 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 in which 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 was 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 MPLAB 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. For example, 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 are 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 can differ across each compiler, and any arguments to the keywords can be device specific.
Coding guidelines	The CCI can 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 can 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 X IDE widget *Use CCI Syntax* in your project, or use the command-line option that is equivalent.
- **Include <xc.h> in every module**

  Some CCI features are only enabled if this header is seen by the compiler.
- **Ensure ANSI compliance**

  Code that does not conform to the ANSI C Standard does not confirm to the CCI.
- **Observe refinements to ANSI by the CCI**

  Some ANSI implementation-defined behavior is defined explicitly by the CCI.
- **Use the CCI extensions to the language**

  Use the CCI extensions rather than the native language extensions.

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

If any implementation-defined behavior or any non-standard extension is not discussed in this document, then it is not part of the CCI. For example, GCC case ranges, label addresses and 24-bit `short long` types are not part of the CCI. Programs which use these features do not conform to the CCI. The compiler may issue a warning or error to indicate a non-CCI feature has been used 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 can be terminated using a *line feed* (\n) or *carriage return* (\r) that is immediately followed by a *line feed*. Escaped characters can be used in character constants or string literals to represent extended characters that are 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 was compiled under other host operating systems. Under the CCI, no directory separators 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 X 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 X 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 that are detailed below.

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

Header files specified in quote characters `" "` should be discoverable in the current working directory or in the same directories that are searched when the header files are specified in angle bracket delimiters (as above). 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 to the same directories searched when the header file 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. A header file being located elsewhere does not conform to the CCI.

Differences

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

2.4.4.2 MIGRATION TO THE CCI

Remove any option that specifies header file search paths other than the `-I` option (or the equivalent MPLAB X 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 that 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 can 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>`.