MPLAB® XC32 C/C++ Compiler User’s Guide for PIC32C/SAM MCUs

ISBN: 978-1-5224-4735-1

For information regarding Microchip's Quality Management Systems, please visit www.microchip.com/quality.

MPLAB® XC32 C/C++ COMPILER USER'S GUIDE FOR PIC32C/SAM MCUs

Table of Contents

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 for PIC32C/SAM documentation and support information is discussed here:

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. Operators and Statements – discusses operators and statements.
- Chapter 11. Fixed-Point Arithmetic Support – describes the fixed-point types and operations supported.

- 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. Built-In Functions – lists the built-in functions of the C compiler.
- Appendix D. ASCII Character Set – contains the ASCII character set.
- Appendix E. 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) { ... }`

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 Compleat Libraries is available on the My MPLAB X IDE tab, References & Featured Links section of the MPLAB X IDE application.

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/

Arm Reference Document

Arm® C Language Extensions, Release 1.1, Document number IHI 0053B, Date of Issue 12/11/13.

This document specifies the Arm C Language Extensions to enable C/C++ programmers to exploit the Arm architecture with minimal restrictions on source code portability.

NOTES:

Chapter 1. Compiler Overview

1.1 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 toolchain supports the PIC32C and SAM microcontroller families using an Arm® Cortex®-Mx cores. 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 Windows®, Linux® and Mac OS® X.

The compiler can run in Free or PRO operating mode. The PRO operating mode is a licensed mode and requires an activation key and Internet connectivity to enable it. 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.

1.1.1 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 5.8 "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.

1.1.2 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.

1.1.3 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".

1.1.4 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.

1.1.5 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., `/pic32c/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.

1.1.6 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.

1.1.7 Documentation

This version of the C compiler is supported under MPLAB X IDE v5.20 or higher is required.

1.2 DEVICE SUPPORT

The MPLAB XC32 C/C++ Compiler fully supports most Microchip PIC32C, SAM, CEC17, MEC15 and MEC17 devices.

1.3 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 X IDE (v5.20 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

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.

2.1 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.

2.1.1 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, for example, 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.

For freestanding implementations (or for what are typically called 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.

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.

2.1.2 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.

2.2 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 option *Use CCI Syntax* in your project, or use the command-line option that is equivalent (See Section 2.5.1 "Enabling the CCI").
- **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 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.

2.3 ANSI STANDARD REFINEMENT

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

2.3.1 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.

2.3.1.1 EXAMPLE

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

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

2.3.1.2 DIFFERENCES

All compilers have used this character set.

2.3.1.3 MIGRATION TO THE CCI

No action required.

2.3.2 The Prototype for `main`

The prototype for the `main()` function is:

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

2.3.2.1 EXAMPLE

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

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

2.3.2.2 DIFFERENCES

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

2.3.2.3 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.

2.3.3 Header File Specification

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

2.3.3.1 EXAMPLE

The following example shows two conforming include directives.

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

2.3.3.2 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.

2.3.3.3 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
```

2.3.4 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 point 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.

2.3.4.1 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.

2.3.4.2 DIFFERENCES

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

2.3.4.3 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.

2.3.5 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.

2.3.5.1 EXAMPLE

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

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

2.3.5.2 DIFFERENCES

The XC32 compilers did not impose a limit on the number of significant characters.

2.3.5.3 MIGRATION TO THE CCI

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

2.3.6 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>`, for example, `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.

2.3.6.1 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;
```

2.3.6.2 DIFFERENCES

This is consistent with previous types implemented by the compiler.

2.3.6.3 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>`.

2.3.7 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.

2.3.7.1 EXAMPLE

The following example

```
char foobar;
```

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

2.3.7.2 DIFFERENCES

None.

2.3.7.3 MIGRATION TO THE CCI

Any definition of an object defined as a plain `char` 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 can use the `-funsigned-char` option on MPLAB XC32 to change the type of plain `char`, but the code is not strictly conforming.

2.3.8 Signed Integer Representation

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

2.3.8.1 EXAMPLE

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

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

2.3.8.2 DIFFERENCES

The XC32 compilers have represented signed integers in the way described in this section.

2.3.8.3 MIGRATION TO THE CCI

No action required.

2.3.9 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.

2.3.9.1 EXAMPLE

The following shows an assignment of a value that is truncated.

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

Under the CCI, the value of `destination` after the alignment is -2 (that is, the bit pattern 0xFE).

2.3.9.2 DIFFERENCES

The XC32 compiler has performed integer conversion in an identical fashion to that described in this section.

2.3.9.3 MIGRATION TO THE CCI

No action required.

2.3.10 Bitwise Operations on Signed Values

Bitwise operations on signed values act on the two's complement representation, including the sign bit (see also Section 2.3.11 "Right-shifting Signed Values").

2.3.10.1 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 is 0x72.

2.3.10.2 DIFFERENCES

The XC32 compiler has performed bitwise operations in an identical fashion to that described in this section.

2.3.10.3 MIGRATION TO THE CCI

No action required.

2.3.11 Right-shifting Signed Values

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

2.3.11.1 EXAMPLE

The following example shows a negative quantity involved in a right-shift operation.

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

Under the CCI, the value of `output` after the assignment is -2 (that is, the bit pattern 0xFE).

2.3.11.2 DIFFERENCES

The XC32 compiler has performed right-shifting as described in this section.

2.3.11.3 MIGRATION TO THE CCI

No action required.

2.3.12 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.

2.3.12.1 EXAMPLE

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

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

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

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

2.3.12.2 DIFFERENCES

The XC32 compiler has not converted union members accessed via other members.

2.3.12.3 MIGRATION TO THE CCI

No action required.

2.3.13 Default Bit-field int Type

The type of a bit-field specified as a plain `int` is identical to that of one defined using `unsigned int`. This is quite different from 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.

2.3.13.1 EXAMPLE

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

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

2.3.13.2 DIFFERENCES

The XC32 compilers have implemented bit-fields defined using `int` as having a `signed int` type, unless the option `-funsigned-bitfields` was specified.

2.3.13.3 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`. In the following example:

```
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;
};
```

2.3.14 Bit-Fields Straddling a Storage Unit Boundary

The standard indicates that implementations can determine whether bit-fields cross a storage unit boundary. In the CCI, bit-fields do not straddle a storage unit boundary; a new storage unit is allocated to the structure, and padding bits 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 (for example, `int`) from which the bit-field type is derived. For the XC32 compiler, it is 32 bits in size.

2.3.14.1 EXAMPLE

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

```
struct {
        unsigned first  : 30;
        unsigned second :6;
} order;
```

Under the CCI and using MPLAB XC32, the storage allocation unit is (32-bit) word sized. The bit-field, `second`, is 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`, is 2 32-bit words (8 bytes).

2.3.14.2 DIFFERENCES

This allocation is identical with that used by the XC32 compilers.

2.3.14.3 MIGRATION TO THE CCI

No action required.

2.3.15 The Allocation Order of Bit-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 is the least significant bit (LSb) of the storage unit in which it is allocated.

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

2.3.15.2 DIFFERENCES

This is identical with the previous operation of the XC32 compilers.

2.3.15.3 MIGRATION TO THE CCI

No action required.

2.3.16 The NULL Macro

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

2.3.16.1 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 converted to the destination type.

2.3.16.2 DIFFERENCES

The XC32 compilers previously assigned `NULL` the expression `((void *)0)`.

2.3.16.3 MIGRATION TO THE CCI

No action required.

2.3.17 Floating-Point Sizes

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

2.3.17.1 EXAMPLE

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

```
float outY;
```

2.3.17.2 MIGRATION TO THE CCI

No migration is required for the XC32 compiler.

2.4 ANSI STANDARD EXTENSIONS

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

2.4.1 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.

2.4.1.1 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>
```

2.4.1.2 MIGRATION TO THE CCI

No changes required.

2.4.2 Absolute Addressing

Variables and functions can be placed at an absolute address by using the `__at()` construct. Stack-based (`auto` and parameter) variables cannot use the `__at()` specifier.

2.4.2.1 EXAMPLE

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

> **Note:** PIC32C supports only 4-byte aligned absolute addresses.

```
int scanMode __at(0x200);
const char keys[] __at(124) = { 'r', 's', 'u', 'd'};

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

2.4.2.2 DIFFERENCES

The XC32 compilers have used the `address` attribute to specify an object's address.

2.4.2.3 MIGRATION TO THE CCI

Avoid making objects and functions absolute if possible.

I

In MPLAB XC32, change code, for example, from:

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

to:

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

2.4.3 Persistent Objects

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

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

2.4.3.1 EXAMPLE

The following shows a variable qualified using `__persistent`.

```
__persistent int serialNo;
```

2.4.3.2 DIFFERENCES

The XC32 compilers have used the `persistent` attribute with variables to indicate they were not to be cleared.

2.4.3.3 MIGRATION TO THE CCI

Change any occurrence of the `persistent` attribute to `__persistent`, for example, from:

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

to:

```
int __persistent tblIdx;
```

2.4.3.4 CAVEATS

None.

2.4.4 Alignment of Objects

The `__align(`*`alignment`*`)` specifier can be used to indicate that variables must be aligned on a memory address that is a multiple of the alignment specified. The alignment term must be a power of 2. Positive values request that the object's start address be aligned.

Note:	The compiler supports only positive alignment values for PIC32C/SAM MCUs.

2.4.4.1 EXAMPLE

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

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

2.4.4.2 DIFFERENCES

The XC32 compilers used the `aligned` attribute with variables.

2.4.4.3 MIGRATION TO THE CCI

Change any occurrence of the `aligned` attribute to `__align`, for example, from:

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

to:

```
__align(4) char mode;
```

2.4.5 Interrupt Functions

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

For details on the interrupt types supported by this compiler, see Chapter 14. "Interrupts".

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

2.4.5.1 EXAMPLE

The following shows a function qualified using `__interrupt`.

```
static unsigned long tick_counter;

void __interrupt()
SysTick_Handler(void) {
    tick_counter += 1;
}
```

2.4.5.2 DIFFERENCES

The XC32 compilers have used the `interrupt` attribute to define interrupt functions.

For PIC32C compilers, the `__interrupt()` keyword takes an optional parameter, the kind of interrupt to be handled. Change code that uses the `interrupt` attribute, similar to these examples:

```
void __attribute__ ((interrupt ("IRQ")))
irq_handler (void)
{
  /* ... */
}
```

to:

```
void __interrupt ("IRQ")
irq_handler (void)
{
  /* ... */
}
```

2.4.5.3 CAVEATS

None.

2.4.6 Packing Objects

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

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

2.4.6.1 EXAMPLE

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

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

2.4.6.2 DIFFERENCES

The XC32 compilers have used the `packed` attribute to indicate that a structure member was not aligned with a memory gap.

2.4.6.3 MIGRATION TO THE CCI

Change any occurrence of the `packed` attribute, for example, from:

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

to:

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

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

2.4.6.4 CAVEATS

None.

2.4.7 Indicating Antiquated Objects

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

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

2.4.7.1 EXAMPLE

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

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

2.4.7.2 DIFFERENCES

The XC32 compilers have used the `deprecated` attribute (note the different spelling) to indicate that objects should be avoided, if possible.

2.4.7.3 MIGRATION TO THE CCI

Change any occurrence of the `deprecated` attribute to `__deprecate`, for example, from:

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

to:

```
int __deprecate intMask;
```

2.4.7.4 CAVEATS

None.

2.4.8 Assigning Objects to Sections

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

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

2.4.8.1 EXAMPLE

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

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

2.4.8.2 DIFFERENCES

The XC32 compilers have used the `section` attribute to indicate a different destination section name. The `__section()` specifier works in a similar way to the attribute.

2.4.8.3 MIGRATION TO THE CCI

Change any occurrence of the `section` attribute, for example, from:

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

to:

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

2.4.8.4 CAVEATS

None.

2.4.9 Specifying Configuration Bits

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

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

where *`setting`* is a configuration setting descriptor (for example, `WDT`), *`state`* is a descriptive value (for example, `ON`) and *`value`* is a numerical value.

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

2.4.9.1 EXAMPLE

The following shows Configuration bits being specified using this pragma.

```
// ATSAME70Q21B Configuration Bit Settings
#pragma config SECURITY_BIT = CLEAR
#pragma config BOOT_MODE = CLEAR
#pragma config TCM_CONFIGURATION = 0x0 // Enter Hexadecimal value
```

2.4.9.2 DIFFERENCES

The XC32 compilers supported the use of `#pragma config`.

2.4.9.3 MIGRATION TO THE CCI

No migration is required.

2.4.9.4 CAVEATS

None.

2.4.10 Manifest Macros

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

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

TABLE 2-1: MANIFEST MACROS DEFINED BY THE CCI

Name	Meaning if defined	Example
`__XC__`	Compiled with an MPLAB XC compiler	`__XC__`
`__CCI__`	Compiler is CCI compliant and CCI enforcement is enabled	`__CCI__`
`__XC##__`	The specific XC compiler used (`##` can be `8`, `16` or `32`)	`__XC32__`
`__`*`DEVICEFAMILY`*`__`	The family of the selected target device	`__SAME70__`
`__`*`DEVICENAME`*`__`	The selected target device name	`__SAME70J19B__`

2.4.10.1 EXAMPLE

Code conditionally compiled for a particular device family:

```
#ifdef __SAME70__
void E70_specific_func (void);
#else
void general_func (void);
#endif
```

2.4.10.2 DIFFERENCES

Some of these CCI macros are new (for example, `__CCI__`) and others have different names to previous symbols with identical meaning (for example, `__SAME70J19B` is now `__SAME70J19B__`).

2.4.10.3 MIGRATION TO THE CCI

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

2.4.10.4 CAVEATS

None.

2.4.11 In-line Assembly

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

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

2.4.11.1 EXAMPLE

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

```
asm("NOP");
```

2.4.11.2 DIFFERENCES

This is the same syntax used by the XC32 compilers.

2.4.11.3 MIGRATION TO THE CCI

No migration is required.

2.4.11.4 CAVEATS

None.

2.5 COMPILER FEATURES

The following item details the compiler options used to control the CCI.

2.5.1 Enabling the CCI

It is assumed that you are using the MPLAB X IDE to build projects that use the CCI. The option in the MPLAB X IDE Project Properties to enable CCI conformance is *Use CCI Syntax*, which can be found under *XC32 > xc132-gcc > Preprocessing and messages*, if you are using the MPLAB XC32 compiler.

If you are not using this IDE, then the command-line options are `-mcci` for MPLAB XC32.

2.5.1.1 DIFFERENCES

This option has never been implemented previously.

2.5.1.2 MIGRATION TO THE CCI

Enable the option.

2.5.1.3 CAVEATS

None.

Chapter 3. How To's

This section contains help and reference for situations that are frequently encountered when building projects for Microchip 32-bit devices. Click the links at the beginning of each section to assist in finding the topic relevant to your question. Some topics are indexed in multiple sections.

Start here:

- Installing and Activating the Compiler
- Invoking the Compiler
- Writing Source Code
- Getting My Application to Do What I Want
- Understanding the Compilation Process
- Fixing Code That Does Not Work

3.1 INSTALLING AND ACTIVATING THE COMPILER

This section details questions that might arise when installing or activating the compiler.

- How Do I Install and Activate My Compiler?
- How Can I Tell if the Compiler has Activated Successfully?
- Can I Install More Than One Version of the Same Compiler?

3.1.1 How Do I Install and Activate My Compiler?

Installation and activation of the license are performed simultaneously by the XC compiler installer. The guide *Installing and Licensing MPLAB XC C Compilers* (DS52059) is available on www.microchip.com. It provides details on single-user and network licenses, as well as how to activate a compiler for evaluation purposes.

3.1.2 How Can I Tell if the Compiler has Activated Successfully?

If you think the compiler may not have installed correctly or is not working, it is best to verify its operation outside of MPLAB X IDE to isolate possible problems. Try running the compiler from the command line to check for correct operation. You do not actually have to compile code.

From your terminal or command-line prompt, run the license manager xclm with the option -status. This option instructs the license manager to print all MPLAB XC licenses installed on your system and exit. So, under 32-bit Windows, for example, type the following line, replacing the path information with a path that is relevant to your installation.

```
"C:\Program Files\Microchip\xc32\v1.00\bin\xclm" -status
```

The license manager should run, print all of the MPLAB XC compiler license available on your machine, and quit. Confirm that the license is listed as activated (e.g., Product:swxc32-pro). Note: if it is not activated properly, the compiler will continue to operate, but only in the Free mode. If an error is displayed, or the compiler indicates Free mode, then activation was not successful.

3.1.3 Can I Install More Than One Version of the Same Compiler?

Yes, the compilers and installation process has been designed to allow you to have more than one version of the same compiler installed. For MPLAB X IDE, you can easily swap between version by changing options in the IDE (see Section 3.2.4 "How Can I Select Which Compiler I Want to Build With?").

Compilers should be installed into a directory whose name is related to the compiler version. This is reflected in the default directory specified by the installer. For example, the MPLAB XC32 compilers v1.00 and v1.10 would typically be placed in separate directories.

```
C:\Program Files\Microchip\xc32\v1.00\
C:\Program Files\Microchip\xc32\v1.10\
```

3.2 INVOKING THE COMPILER

This section discusses how the compiler is run, both on the command-line and from the IDE. It includes information about how to get the compiler to do what you want in terms of options and the build process itself.

- How Do I Compile from Within MPLAB X IDE?
- How Do I Compile on the Command-line?
- How Do I Compile Using a Make Utility?
- How Can I Select Which Compiler I Want to Build With?
- How Can I Change the Compiler's Operating Mode?
- How Do I Build Libraries?
- How Do I Know What Compiler Options Are Available and What They Do?
- How Do I Know What the Build Options in MPLAB X IDE do?
- What is Different About an MPLAB X IDE Debug Build?
- See also How Do I Stop the Compiler Using Certain Memory Locations?
- See also What Do I Need to Do When Compiling to Use a Debugger?
- See also How Do I Use Library Files In My Project?
- See also What Optimizations Are Employed By The Compiler?

3.2.1 How Do I Compile from Within MPLAB X IDE?

See the following documentation for information on how to set up a project:

Section 4.4 "Project Setup" - MPLAB X IDE

3.2.2 How Do I Compile on the Command-line?

The compiler driver is called `xc32-gcc` for all 32-bit devices; e.g., in Windows, it is named `xc32-gcc.exe`. This application should be invoked for all aspects of compilation. It is located in the `bin` directory of the compiler distribution. Avoid running the individual compiler applications (such as the assembler or linker) explicitly. You can compile and link in the one command, even if your project is spread among multiple source files.

The driver is introduced in Section 5.1 "Invoking the Compiler". See Section 3.2.4 "How Can I Select Which Compiler I Want to Build With?" to ensure you are running the correct driver if you have more than one installed. The command-line options to the driver are detailed in Section 5.8 "Driver Option Descriptions". The files that can be passed to the driver are listed and described in Section 5.1.3 "Input File Types".

3.2.3 How Do I Compile Using a Make Utility?

When compiling using a make utility (such as `make`), the compilation is usually performed as a two-step process: first generating the intermediate files and then the final compilation and link step to produce one binary output. This is described in Section 5.2.2 "Multi-step C Compilation".

3.2.4 How Can I Select Which Compiler I Want to Build With?

The compilation and installation process has been designed to allow you to have more than one compiler installed at the same time For MPLAB X IDE, you can create a project and then build this project with different compilers by simply changing a setting in the project properties.

In MPLAB X IDE, you select which compiler to use when building a project by opening the Project Properties window (*File>Project Properties*) and selecting the Configuration category (`Conf: [default]`). A list of MPLAB XC32 compiler versions is shown in the Compiler Toolchain, on the far right. Select the MPLAB XC32 compiler you require.

Once selected, the controls for that compiler are then shown by selecting the XC32 global options, XC32 Compiler and XC32 Linker categories. These reveal a pane of options on the right; each category has several panes which can be selected from a pull-down menu that is near the top of the pane.

3.2.5 How Can I Change the Compiler's Operating Mode?

The compiler's operating mode (Free, Evaluation or PRO) is based on its level of optimizations (see Chapter 18. "Optimizations") which can be specified as a command line option (see Section 5.8.7 "Options for Controlling Optimization"). If you are building under MPLAB X IDE, go to the Project Properties window, click on the compiler name (xc32-gcc for C language projects or xc32-g++ for C++ language projects), and select the Optimization option category to set optimization levels - see Section 4.4.3 "xc32-gcc (32-bit C Compiler)".

When building your project, the compiler will emit a warning message if you have selected an option that is not available for your licensed operating mode. The compiler will continue compilation with the option disabled.

3.2.6 How Do I Build Libraries?

When you have functions and data that are commonly used in applications, you can make all the C source and header files available so other developers can copy these into their projects. Alternatively, you can build these modules into object files and package them into library archives, which along with the accompanying header files, can then be built into an application.

Libraries can be more convenient because there are fewer files to manage. However, libraries do need to be maintained. MPLAB XC32 uses *.a library archives. Be sure to rebuild your library objects when you move your project to a new release of the compiler toolchain.

Using the compiler driver, libraries can begin to be built by listing all the files that are to be included into the library on the command line. None of these files should contain a `main()` function, nor settings for configuration bits or any other such data.

For information on how to create your own libraries, see Section 5.4.1.2 "User-defined Libraries".

3.2.7 How Do I Know What Compiler Options Are Available and What They Do?

A list of all compiler options can be obtained by using the `--help` option on the command line. Alternatively, all options are listed in Section 5.8 "Driver Option Descriptions" in this user's guide. If you are compiling in MPLAB X IDE, see Section 4.4 "Project Setup".

3.2.8 How Do I Know What the Build Options in MPLAB X IDE do?

Most of the widgets and controls in the MPLAB X IDE Project Properties window, XC32 options, map directly to one command-line driver option or suboption. See Section 4.4 "Project Setup" for a list of options and any corresponding command-line options.

3.2.9 What is Different About an MPLAB X IDE Debug Build?

The main difference between a command-line debug build and an MPLAB X IDE debug build is the setting of a preprocessor macro called `__DEBUG` to be 1 when a debug is selected. This macro is not defined if it is not a debug build.

You may make code in your source conditional on this macro using `#ifdef` directives, (see Section 5.8.8 "Options for Controlling the Preprocessor") so that you can have your program behave differently when you are still in a development cycle. Some compiler errors are easier to track down after performing a debug build.

In MPLAB X IDE, memory will be reserved for your debugger only when you perform a debug build. See Section 3.4.3 "What Do I Need to Do When Compiling to Use a Debugger?".

3.3 WRITING SOURCE CODE

This section presents issues pertaining to the source code you write. It has been subdivided into the following sections.

- C Language Specifics
- Device-Specific Features
- Memory Allocation
- Variables
- Functions
- Interrupts
- Assembly Code

3.3.1 C Language Specifics

This section discusses source code issues that directly relate to the C language itself but which are commonly asked.

- When Should I Cast Expressions?
- Can Implicit Type Conversions Change the Expected Results of My Expressions?
- How Do I Enter Non-English Characters Into My Program?
- How Can I Use a Variable Defined in Another Source File?
- How Do I Port My Code to Different Device Architectures?

3.3.1.1 WHEN SHOULD I CAST EXPRESSIONS?

Expressions can be explicitly cast using the cast operator -- a type in round brackets, e.g., `(int)`. In all cases, conversion of one type to another must be done with caution and only when absolutely necessary.

Consider the example:

```
unsigned long l;
unsigned short s;

s = l;
```

Here, a `long` type is being assigned to a `int` type and the assignment will truncate the value in `l`. The compiler will automatically perform a type conversion from the type of the expression on the right of the assignment operator (`long`) to the type of the lvalue on the left of the operator (`short`).This is called an implicit type conversion. The compiler will typically produce a warning concerning the potential loss of data by the truncation.

A cast to type `short` is not required and should not be used in the above example if a `long` to `short` conversion was intended. The compiler knows the types of both operands and will perform the conversion accordingly. If you did use a cast, there is the potential for mistakes if the code is later changed. For example, if you had:

```
s = (short)l;
```

the code will work the in the same way; but if in the future, the type of `s` is changed to a `long` (for example), then you must remember to either adjust the cast or remove it, otherwise the contents of `l` will continue to be truncated by the assignment, which may not be correct. Most importantly, the warning issued by the compiler will not be produced if the cast is in place.

Use a cast only in situations where the types used by the compiler are not the types that you require. For example consider the result of a division assigned to a floating-point variable:

```
int i, j;
float fl;

fl = i/j;
```

In this case integer division is performed, then the rounded integer result is converted to a `float` format. So if `i` contained 7 and `j` contained 2, the division will yield 3 and this will be implicitly converted to a `float` type (3.0) and then assigned to `fl`. If you wanted the division to be performed in a `float` format, then a cast is necessary:

```
fl = (float)i/j;
```

(Casting either `i` or `j` will force the compiler to encode a floating-point division). The result assigned to `fl` now be 3.5.

An explicit cast may suppress warnings that might otherwise have been produced. This can also be the source of many problems. The more warnings the compiler produces, the better chance you have of finding potential bugs in your code.

3.3.1.2 CAN IMPLICIT TYPE CONVERSIONS CHANGE THE EXPECTED RESULTS OF MY EXPRESSIONS?

Yes! The compiler will always use integral promotion and there is no way to disable this (see Section 10.1 “Integral Promotion”). In addition, the types of operands to binary operators are usually changed so that they have a common type as specified by the C Standard. Changing the type of an operand can change the value of the final expression so it is very important that you understand the type C Standard conversion rules that apply when dealing with binary operators. You can manually change the type of an operand by casting (see Section 3.3.1.1 “When Should I Cast Expressions?”).

3.3.1.3 HOW DO I ENTER NON-ENGLISH CHARACTERS INTO MY PROGRAM?

The ANSI standard and MPLAB XC C do not support extended characters set in character and string literals in the source character set. See Section 8.8 “Constant Types and Formats” to see how these characters can be entered using escape sequences.

3.3.1.4 HOW CAN I USE A VARIABLE DEFINED IN ANOTHER SOURCE FILE?

Provided the variable defined in the other source file is not `static` (see Section 9.2.2 “Static Variables”) or `auto` (see Section 9.3 “Auto Variable Allocation and Access”), adding a declaration for that variable in the current file will allow you to access it. A declaration consists of the keyword `extern` in addition to the type and name of the variable as specified in its definition, for example:

```
extern int systemStatus;
```

This is part of the C language and your favorite C text will give you more information.

The position of the declaration in the current file determines the scope of the variable, i.e., if you place the declaration inside a function, it will limit the scope of the variable to that function and if placed outside of a function, it allows access to the variable in all functions for the remainder of the current file.

Often, declarations are placed in header files and these are then `#include`d into the C source code (see Section 19.3 “Pragma Directives”).

3.3.1.5 HOW DO I PORT MY CODE TO DIFFERENT DEVICE ARCHITECTURES?

Microchip devices have three basic architectures: 8-bit, which is a Harvard architecture with a separate program and data memory bus; 16-bit, which is a modified Harvard architecture also with a separate program and data memory bus; and 32-bit, which is a MIPS and Arm architecture. Porting code to different devices within an architectural family requires a minimum update to application code. However, porting between architectural families can require significant rewrite.

In an attempt to reduce the work to port between architectures, a Common C Interface, or CCI, has been developed. If you use these coding styles, your code will more easily migrate upward. For more on CCI, see Chapter 2. “Common C Interface”.

3.3.2 Device-Specific Features

This section discusses the code that needs to be written to set up or control a feature that is specific to Microchip PIC devices.

- How Do I Set the Configuration Bits?
- How Do I Determine the Cause of Reset?
- How Do I Access SFRs?
- How Do I Stop the Compiler Using Certain Memory Locations?

Also, see the following linked information in other sections.

What Do I Need to Do When Compiling to Use a Debugger?

3.3.2.1 HOW DO I SET THE CONFIGURATION BITS?

These should be set in your code using either a macro or pragma. Earlier versions of MPLAB X IDE allowed you to set these bits in a dialog, but MPLAB X IDE requires that they be specified in your source code. Config bits are set in source code using the config pragma. See Section 7.4 "Configuration Bit Access" for more information on the config pragma.

3.3.2.2 HOW DO I DETERMINE THE CAUSE OF RESET?

The bits in the Reset Control (RCON) Register allow you to determine the cause of a Reset. Most MCUs have a peripheral register that you can use to determine the cause of a reset. The exact register name and functionality varies by device but is often called RCON, RCAUSE, or RSTC_SR. Check the data sheet for your target device for more information.

3.3.2.3 HOW DO I ACCESS SFRS?

The compiler ships with header files that define variables which are mapped over the top of memory-mapped SFRs. Since these are C variables, they can be used like any other C variable and no new syntax is required to access these registers.

The name assigned to the variable is usually the same as the name specified in the device data sheet. See Section 3.3.2.4 "How Do I Find The Names Used to Represent SFRs and Bits?" if these names are not recognized.

3.3.2.4 HOW DO I FIND THE NAMES USED TO REPRESENT SFRS AND BITS?

Special function registers and the bits within those are accessed via special variables that map over the register, Section 3.3.2.3 "How Do I Access SFRs?"; however, the names of these variables sometimes differ from those indicated in the data sheet for the device you are using.

View the device-specific header file which allows access to these special variables. Begin by searching for the data sheet SFR name. If that is not found, search on what the SFR represents, as comments in the header often spell out what the macros under the comment do.

3.3.3 Memory Allocation

Here are questions relating to how your source code affects memory allocation.

- How Do I Position Variables at an Address I Nominate?
- How Do I Position Functions at an Address I Nominate?
- How Do I Place Variables in Program Memory?
- How Do I Stop the Compiler Using Certain Memory Locations?
- Why are Some Objects Positioned into Memory that I Reserved?

3.3.3.1 HOW DO I POSITION VARIABLES AT AN ADDRESS I NOMINATE?

The easiest way to do this is to make the variable absolute by using the `address` attribute (see Section 8.11 "Variable Attributes") or the `__at()` CCI construct (see Section 2.4.2 "Absolute Addressing"). This means that the address you specify is used in preference to the variable's symbol in generated code. Since you nominate the address, you have full control over where objects are positioned, but you must also ensure that absolute variables do not overlap.

See also Section 9.3 "Auto Variable Allocation and Access" for information on moving auto variables, Section 9.2.1 "Non-auto Variable Allocation" for moving non-auto variables and Section "The source code for this is found in the pic32c-libs.zip file located at:" for moving program-space variables.

3.3.3.2 HOW DO I POSITION FUNCTIONS AT AN ADDRESS I NOMINATE?

The easiest way to do this is to make the functions absolute, by using the `address` attribute (see Section 13.2.1 "Function Attributes"). This means that the address you specify is used in preference to the function's symbol in generated code. Since you nominate the address, you have full control over where functions are positioned, but must also ensure that absolute functions do not overlap.

3.3.3.3 HOW DO I PLACE VARIABLES IN PROGRAM MEMORY?

The `const` qualifier implies that the qualified variable is read only. As a consequence of this, any variables (except auto variables or function parameters) qualified `const` are placed in program memory, thus freeing valuable data RAM (see Section "The source code for this is found in the pic32c-libs.zip file located at:"). Variables qualified `const` can also be made absolute, so that they can be positioned at an address you nominate.

3.3.3.4 HOW DO I STOP THE COMPILER USING CERTAIN MEMORY LOCATIONS?

Concatenating an address attribute with the `noload` attribute can be used to block out sections of memory. For more on variable attributes and options, see the following sections in this user's guide:

Section 8.11 "Variable Attributes"

Section 5.8.1 "Options Specific to PIC32C/SAM Devices"

See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide* (DS50002186) for details on linker scripts.

3.3.4 Variables

This examines questions that relate to the definition and usage of variables and types within a program.

- Why Are My Floating-point Results Not Quite What I Am Expecting?
- How Can I Access Individual Bits of a Variable?
- How Long Can I Make My Variable and Macro Names?
- How Do I Share Data Between Interrupt and Main-line Code?
- How Do I Position Variables at an Address I Nominate?
- How Do I Place Variables in Program Memory?
- How Can I Rotate a Variable?
- How Do I Find Out Where Variables and Functions Have Been Positioned?

3.3.4.1 WHY ARE MY FLOATING-POINT RESULTS NOT QUITE WHAT I AM EXPECTING?

First, make sure that if you are watching floating-point variables in MPLAB X IDE that the type and size of these match how they are defined. In MPLAB XC32 for PIC32C/SAM targets, the float type is a 32-bit floating-point type. The double and long double types are 64-bit floating-point types.

Since floating-point variables only have a finite number of bits to represent the values they are assigned, they will hold an approximation of their assigned value. A floating-point variable can only hold one of a set of discrete real number values. If you attempt to assign a value that is not in this set, it is rounded to the nearest value. The more bits used by the mantissa in the floating-point variable, the more values can be exactly represented in the set and the average error due to the rounding is reduced.

Whenever floating-point arithmetic is performed, rounding also occurs. This can also lead to results that do not appear to be correct.

3.3.4.2 HOW CAN I ACCESS INDIVIDUAL BITS OF A VARIABLE?

There are several ways of doing this. The simplest and most portable way is to define an integer variable and use macros to read, set, or clear the bits within the variable using a mask value and logical operations, such as the following.

```
#define  testbit(var, bit)   ((var) & (1 <<(bit)))
#define  setbit(var, bit)    ((var) |= (1 << (bit)))
#define  clrbit(var, bit)    ((var) &= ~(1 << (bit)))
```

These test to see if the bit number (`bit`) in the integer (`var`) is set; as well as set the corresponding `bit` in `var`; and clear the corresponding `bit` in `var`. Alternatively, a union of an integer variable and a structure with bit-fields (see Section 8.5.2 "Bit Fields in Structures") can be defined, for example:

```
union both {
   unsigned char byte;
   struct {
      unsigned b0:1, b1:1, b2:1, b3:1, b4:1, b5:1, b6:1, b7:1;
   } bitv;
} var;
```

This allows you to access `byte` as a whole (using var.byte), or any bit within that variable independently (using `var.bitv.b0` through `var.bitv.b7`).

3.3.4.3 HOW LONG CAN I MAKE MY VARIABLE AND MACRO NAMES?

The C Standard indicates that only a number of initial characters in an identifier are significant, but it does not actually state what the number is and how it varies from compiler to compiler. For MPLAB XC32, no limit is imposed, but for CCI there is a limit (see Section 2.3.5 "The Number of Significant Initial Characters in an Identifier"). CCI Compliant names are more portable across Microchip architectures.

If two identifiers only differ in the non-significant part of the name, they are considered to represent the same object, which will almost certainly lead to code failure.

3.3.5 Functions

This section examines questions that relate to functions.

- What is the Optimum Size For Functions?
- How Can I Tell How Big a Function Is?
- How Do I Know What Resources Are Being Used by Each Function?
- How Do I Find Out Where Variables and Functions Have Been Positioned?
- How Do I Use Interrupts in C?
- How Do I Stop An Unused Function Being Removed?
- How Do I Make a Function Inline?

3.3.5.1 WHAT IS THE OPTIMUM SIZE FOR FUNCTIONS?

Generally speaking, the source code for functions should be kept small as this aids in readability and debugging. It is much easier to describe and debug the operation of a function which performs a small number of tasks. Also smaller-sized functions typically have less side effects, which can be the source of coding errors. On the other hand, in the embedded programming world, a large number of small functions and the calls necessary to execute them, may result in excessive memory and stack usage. Therefore a compromise is often necessary.

Function size can cause issues with memory paging, as addressed in Section 13.5 "Function Size Limits". The smaller the functions, the easier it is for the linker to allocate them to memory without errors.

3.3.5.2 HOW DO I STOP AN UNUSED FUNCTION BEING REMOVED?

The `__attribute__((keep))` may be applied to a function. The `keep` attribute will prevent the linker from removing the function with `--gc-sections`, even if it is unused. See the *"MPLAB® XC32 Assembler, Linker and Utilities User's Guide"* (DS50002186) for more information on section garbage collection using the `--gc-sections` option.

3.3.5.3 HOW DO I MAKE A FUNCTION INLINE?

The XC32 compiler does not inline any functions when not optimizing.

By declaring a function inline, you can direct the XC32 compiler to make calls to that function faster. One way XC32 can achieve this is to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included. The effect on code size is less predictable; as the object code may be larger or smaller with function inlining, depending on the particular case.

To declare a function inline, use the inline keyword in its declaration, like this:

```
static inline int
inc (int *a)
{
  return (*a)++;
}
```

When a function is both inline and static, if all calls to the function are integrated into the caller and the function's address is never used, then the function's own assembler code is never referenced. In this case, XC32 does not actually output assembler code for the function. Some calls cannot be integrated for various reasons (in particular, calls that precede the function's definition, nor recursive calls within the definition). If there is a non-integrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that can't be inlined. Enable optimization level -O1 or greater to enable function inlining.

3.3.6 Interrupts

Interrupt and interrupt service routine questions are discussed in this section.

- How Do I Use Interrupts in C?
- How Can I Make My Interrupt Routine Faster?
- How Do I Share Data Between Interrupt and Main-line Code?

3.3.6.1 HOW DO I USE INTERRUPTS IN C?

First, be aware of what interrupt hardware is available on your target device. 32-bit devices implement several separate interrupt vector locations and use a priority scheme. For more information, see Section 14.1 "Interrupt Operation".

Prior to any interrupt occurring, your program must ensure that peripherals are correctly configured and that interrupts are enabled. For details, see Section 14.7 "Enabling/Disabling Interrupts".

For all other interrupt related tasks, including specifying the interrupt vector, context saving, nesting and other considerations, consult Chapter 14. "Interrupts".

3.3.7 Assembly Code

This section examines questions that arise when writing assembly code as part of a C project.

- How Should I Combine Assembly and C Code?
- What do I need Other than Instructions in an Assembly Source File?
- How Do I Access C Objects from Assembly Code?
- How Can I Access SFRs From Within Assembly Code?
- What Things Must I Manage When Writing Assembly Code?

3.3.7.1 HOW SHOULD I COMBINE ASSEMBLY AND C CODE?

Ideally, any hand-written assembly should be written as separate routines that can be called. This offers some degree of protection from interaction between compiler-generated and hand-written assembly code. Such code can be placed into a separate assembly module that can be added to your project, as specified in Section 17.1 "Mixing Assembly Language and C Variables and Functions".

If necessary, assembly code can be added in-line with C code by using either of two forms of the `asm` instruction; simple or extended. An explanation of these forms, and some examples, are shown in Section 17.2 "Using Inline Assembly Language".

Macros are provided which in-line several simple instructions, as discussed in Section 17.3 "Predefined Macro". More complex inline assembly that changes register contents and the device state should be used with caution.

See Chapter 12. "Register Usage" for those registers used by the compiler.

3.3.7.2 WHAT DO I NEED OTHER THAN INSTRUCTIONS IN AN ASSEMBLY SOURCE FILE?

Assembly code typically needs assembler directives as well as the instructions themselves. The operation of all the directives is described in the *"MPLAB® XC32 Assembler, Linker and Utilities User's Guide"* (DS50002186). Two common directives are discussed below.

All assembly code must be placed in a section, using the `.section` directive, so it can be manipulated as a whole by the linker and placed in memory. See the "Linker Processing" chapter of the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide* (DS50002186) for more information.

Another commonly used directive is `.global` which is used to make symbols accessible across multiple source files. Find more on this directive in the afore-mentioned user's guide.

3.3.7.3 HOW DO I ACCESS C OBJECTS FROM ASSEMBLY CODE?

Most C objects are accessible from assembly code. There is a mapping between the symbols used in the C source and those used in the assembly code generated from this source. Your assembly should access the assembly-equivalent symbols which are detailed in Section 17.1 "Mixing Assembly Language and C Variables and Functions".

Instruct the assembler that the symbol is defined elsewhere by using the `.global` assembler directive. This is the assembly equivalent of a C declaration, although no type information is present. This directive is not needed and should not be used if the symbol is defined in the same module as your assembly code.

Any C variable accessed from assembly code will be treated as if it were qualified `volatile` (see Section 8.9.2 "Volatile Type Qualifier"). Specifying the `volatile` qualifier in C code is preferred as it makes it clear that external code may access the object.

3.3.7.4 HOW CAN I ACCESS SFRS FROM WITHIN ASSEMBLY CODE?

The safest way to gain access to SFRs in assembly code is to have symbols defined in your assembly code that equate to the corresponding SFR address. For the XC32 compiler, the xc.h include file can be used from either preprocessed assembly code or C/C++ code.

There is no guarantee that you will be able to access symbols generated by the compilation of C code, even code that accesses the SFR you require.

3.3.7.5 WHAT THINGS MUST I MANAGE WHEN WRITING ASSEMBLY CODE?

There are several things that you must manage if you are hand-writing assembly code.

- You must place any assembly code you write into a section. See the "Linker Processing" chapter of the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide"* (DS50002186) for more information.
  Assembly code that is placed inline with C code will be placed in the same section as the compiler-generated assembly and you should not place this into a separate section.
- You must ensure that any registers you write to in assembly code are not already in use by compiler-generated code. If you write assembly in a separate module, then this is less of an issue as the compiler will, by default, assume that all registers are used by these routines (see Chapter 12. "Register Usage", registers). No assumptions are made for inline assembly (see Section 17.1 "Mixing Assembly Language and C Variables and Functions") and you must be careful to save and restore any resources that you use (write) and which are already in use by the surrounding compiler-generated code.

3.4 GETTING MY APPLICATION TO DO WHAT I WANT

This section provides programming techniques, applications and examples. It also examines questions that relate to making an application perform a specific task.

- What Can Cause Glitches on Output Ports?
- How Do I Link Bootloaders and Downloadable Applications?
- What Do I Need to Do When Compiling to Use a Debugger?
- How Do I Share Data Between Interrupt and Main-line Code?
- How Can I Prevent Misuse of My Code?
- How Do I Use Printf to Send Text to a Peripheral?
- How Can I Implement a Delay in My Code?
- How Can I Rotate a Variable?

3.4.1 What Can Cause Glitches on Output Ports?

In most cases, this is caused by using ordinary variables to access port bits or the entire port itself. These variables should be qualified `volatile` (see Section 8.9.2 "Volatile Type Qualifier").

The value stored in a variable mapped over a port (hence the actual value written to the port) directly translates to an electrical signal. It is vital that the values held by these variables only change when the code intends them to, and that they change from their current state to their new value in a single transition. The compiler attempts to write to volatile variables in one operation.

3.4.2 How Do I Link Bootloaders and Downloadable Applications?

Exactly how this is done depends on the device you are using and your project requirements, but the general approach when compiling applications that use a bootloader is to allocate discrete program memory space to the bootloader and application so they have their own dedicated memory. In this way, the operation of one cannot affect the other. This will require that either the bootloader or the application is offset in memory. That is, the Reset and interrupt location are offset from address 0 and all program code is offset by the same amount.

Typically the application code is offset, and the bootloader is linked with no offset so that it populates the Reset and interrupt code locations. The bootloader Reset and interrupt code merely contains code which redirects control to the real Reset and interrupt code defined by the application and which is offset.

The contents of the Hex file for the bootloader can be merged with the code of the application by using loadable projects in MPLAB X IDE (see MPLAB X IDE documentation for details). This results in a single Hex file that contains the bootloader and application code in the one image. Check for warnings from this application about overlap, which may indicate that memory is in use by both bootloader and the downloadable application.

3.4.3 What Do I Need to Do When Compiling to Use a Debugger?

You can use debuggers, such as the PICkit™ 4 in-circuit debugger or the MPLAB ICD 4 in-circuit debugger, to debug code built with the MPLAB XC32 compiler. These debuggers use some of the data and program memory of the device for its own use, so it is important that your code does not also use these resources.

The command-line option `-g` (see Section 5.8.6 "Options for Debugging") is used to tell the compiler to generate debugging information. The compiler can then reserve the memory used by the debugger so that your code will not be placed in these locations.

In the MPLAB X IDE, the appropriate debugger option is specified if you perform a Debug Run. It will not be specified if you perform a regular Run, Build Project, or Clean and Build.

Since some device memory is being reserved for use by the debugger, there is less available for your program and it is possible that your code or data may no longer fit in the device when a debugger is selected. For 32-bit devices, some boot flash memory is required for debugging. In addition, some data memory (RAM) is used by the debug tool and may impact the variable allocation in your application.

The specific memory locations used by the debuggers are attributes of MPLAB X, the debug tool in use, and the target device. If you move a project to a new version of the IDE, the resources required may change. For this reason, you should not manually reserve memory for the debugger, or make any assumptions in your code as to what memory is used. A summary of the debugger requirements is available in the MPLAB X IDE help files.

To verify that the resources reserved by the compiler match those required by the debugger, you may view the boot-flash, application-flash, and data-memory usage in the map file or memory-usage report.

To create a map file in MPLAB X IDE, open the Project Properties window (*File>Project Properties*) and click on the linker category (xc32-ld). Under "Option Categories," select "Diagnostics." Next to "Generate map file," enter a path and name for the map file. The logical place to put the map file is in the project folder.

Debug Run your code to generate the map file. View in your favorite text viewer.

See also Section 3.5.14 "Why are Some Objects Positioned into Memory that I Reserved?".

3.4.4 How Do I Share Data Between Interrupt and Main-line Code?

Variables accessed from both interrupt and main-line code can easily become corrupted or misread by the program. The `volatile` qualifier (see Section 8.9.2 "Volatile Type Qualifier") tells the compiler to avoid performing optimizations on such variables. This will fix some of the issues associated with this problem.

The other issues relates to whether the compiler/device can access the data atomically. With 32-bit PIC devices, this is rarely the case. An atomic access is one where the entire variable is accessed in only one instruction. Such access is uninterruptible. You can determine if a variable is being accessed atomically by looking at the assembler list file (see the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide,* DS50002186, for more information). If the variable is accessed in one instruction, it is atomic. Since the way variables are accessed can vary from statement to statement it is usually best to avoid these issues entirely by disabling interrupts prior to the variable being accessed in main-line code, then re-enable the interrupts afterwards (see Section 14.7 "Enabling/Disabling Interrupts" for more information).

3.4.5 How Can I Prevent Misuse of My Code?

First, many devices with flash program memory allow all or part of this memory to be write protected. The device configuration bits need to be set correctly for this to take place (see Section 7.4 "Configuration Bit Access", Section 2.4.9 "Specifying Configuration Bits" for CCI and your device data sheet).

Second, you can prevent third-party code being programmed at unused locations in the program memory, by filling these locations with a value rather than leaving them in their default unprogrammed state. You can chose a fill value that corresponds to an instruction or set all the bits so as the values cannot be further modified.

Consider what will happen if your program somehow reaches and starts executing from these filled values. What instruction will be executed?

The fill-memory feature is not yet available for the PIC32C/SAM compiler.

3.4.6 How Do I Use Printf to Send Text to a Peripheral?

The `printf` function does two things: it formats text based on the format string (along with placeholders) you specify and sends (prints) this formatted text to a destination (or stream). You may choose the `printf` output go to an LCD, SPI module or USART, for example.

For more on the ANSI C function `printf`, see the *32-bit Language Tool Libraries* manual (DS51685).

To check what is passed to the `printf` function, you may attempt to statically analyze format strings passed to the function by using the `-msmart-io` option (Section 5.8.1 "Options Specific to PIC32C/SAM Devices"). You may also use the `-Wformat` option to specify a warning when the arguments supplied to the function do not have types appropriate to the format string specified (see Section 5.8.5 "Options for Controlling Warnings and Errors").

If you wish to create your own `printf`-type function, you will need to use the attributes `format` and `format_arg` (as discussed in Section 13.2.1 "Function Attributes").

3.4.7 How Can I Implement a Delay in My Code?

If an accurate delay is required, or if there are other tasks that can be performed during the delay, then using a timer to generate an interrupt is the best way to proceed.

Microchip does not recommend using a software delay on PIC32/SAM devices as there are many variables that can affect timing, such as the configuration of the L1 cache, prefetch cache, & Flash wait states. On these devices, you may choose to use a hardware timer for timing purposes.

3.4.8 How Can I Rotate a Variable?

The C language does not have a rotate operator, but rotations can be performed using the shift and bitwise OR operators. Since the 32-bit devices have a rotate instruction, the compiler will look for code expressions that implement rotates (using shifts and ORs) and use the rotate instruction in the generated output wherever possible.

If you are using CCI, you should consult Section 2.3.10 "Bitwise Operations on Signed Values" and Section 2.3.11 "Right-shifting Signed Values" if you will be using signed variables.

For the following example C code:

```
int rotate_left (unsigned a, unsigned s)
{
  return (a << s) | (a >> (32 - s));
}
```

the compiler may generate assembly instructions similar to the following:

```
rotate_left:
   subu      $2,$0,$5
   jr        $31
   ror       $2,$4,$2
```

3.5 UNDERSTANDING THE COMPILATION PROCESS

This section tells you how to find out what the compiler did during the build process, how it encoded output code, where it placed objects, etc. It also discusses the features that are supported by the compiler.

- What's the Difference Between the Free and PRO Modes?
- How Can I Make My Code Smaller?
- How Can I Reduce RAM Usage?
- How Can I Make My Code Faster?
- How Does the Compiler Place Everything in Memory?
- How Can I Make My Interrupt Routine Faster?
- How Big Can C Variables Be?
- What Optimizations Will Be Applied to My Code?
- What Devices are Supported by the Compiler?
- How Do I Know What Code the Compiler Is Producing?
- How Can I Tell How Big a Function Is?
- How Do I Know What Resources Are Being Used by Each Function?
- How Do I Find Out Where Variables and Functions Have Been Positioned?
- Why are Some Objects Positioned into Memory that I Reserved?
- How Do I Know How Much Memory Is Still Available?
- How Do I Use Library Files In My Project?

- How Do I Customize the C Runtime Startup Code?
- What Optimizations Are Employed By The Compiler?
- How Do I Set Up Warning/Error Messages?
- How Do I Find the Code that Caused Compiler Errors Or Warnings in My Program?
- How Can I Stop Spurious Warnings from Being Produced?
- Why Can't I Even Blink an LED?
- What Can Cause Corrupted Variables and Code Failure When Using Interrupts?
- How Do I Build Libraries?
- What is Different About an MPLAB X IDE Debug Build?
- How Do I Stop An Unused Function Being Removed?
- How Do I Use Library Files In My Project?

3.5.1 What's the Difference Between the Free and PRO Modes?

These modes, or editions, mainly differ in the optimizations that are performed when compiling (see Chapter 18. "Optimizations"). Compilers operating in Free mode can compile for all the same devices as supported by the Pro mode. The code compiled in Free or PRO modes can use all the available memory for the selected device. What will be different is the size and speed of the generated compiler output. Free mode output will be less efficient when compared to that produced in Pro mode.

3.5.2 How Can I Make My Code Smaller?

There are a number of ways that this can be done, but results vary from one project to the next. Use the assembly list file to observe the assembly code produced by the compiler to verify that the following tips are relevant to your code.For information on the list file, see the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide* (DS50002186).

Use the smallest data types possible as less code is needed to access these. This also reduces RAM usage. For example, a `short` integer type exists for this compiler (see Chapter 8. "Supported Data Types and Variables" for all data types and sizes).

There are two sizes of floating-point type as well, and these are discussed in the same section. Replace floating-point variables with integer variables wherever possible. For many applications, scaling a variable's value makes eliminating floating-point operations possible.

Use unsigned types, if possible, instead of signed types, particularly if they are used in expressions with a mix of types and sizes. Try to avoid an operator acting on operands with mixed sizes whenever possible.

Whenever you have a loop or condition code, use a "strong" stop condition, for example:

```
for(i=0; i!=10; i++)
```

is preferable to:

```
for(i=0; i<10; i++)
```

A check for equality (`==` or `!=`) is usually more efficient to implement than the weaker `<` comparison.

In some situations, using a loop counter that decrements to zero is more efficient than one that starts at zero and counts up by the same number of iterations. So you might be able to rewrite the above as:

```
for(i=10; i!=0; i--)
```

Ensure that you enable all the optimizations allowed for the edition of your compiler (see Chapter 18. "Optimizations"). If you have a Pro edition, you can use the `-Os` option (see Section 5.8.7 "Options for Controlling Optimization") to optimize for size. Otherwise, pick the highest optimization available.

Be aware of what optimizations the compiler performs so you can take advantage of them and don't waste your time manually performing optimizations in C code that the compiler already handles, e.g., don't turn a multiply-by-4 operation into a shift-by-2 operation as this sort of optimization is already detected.

3.5.3 How Can I Reduce RAM Usage?

Consider using `auto` variables rather than `global` or `static` variables as there is the potential that these may share memory allocated to other `auto` variables that are not active at the same time. Memory allocation of auto variables is made on a stack, described in Section 9.3 "Auto Variable Allocation and Access".

Rather than pass large objects to, or from, functions, pass pointers which reference these objects. This is particularly true when larger structures are being passed.

Objects that do not need to change throughout the program can be located in program memory using the `const` qualifier (see Section "The source code for this is found in the pic32c-libs.zip file located at:"). This frees up precious RAM, but slows execution.

3.5.4 How Can I Make My Code Faster?

To a large degree, smaller code is faster code, so efforts to reduce code size often decrease execution time (to accomplish this, see Section 3.5.2 "How Can I Make My Code Smaller?" and Section 3.5.6 "How Can I Make My Interrupt Routine Faster?"). However, there are ways some sequences can be sped up at the expense of increased code size.

Depending on your compiler edition (see Chapter 18. "Optimizations"), you may be able to use the `-O3` option (see Section 5.8.7 "Options for Controlling Optimization") to optimize for speed. This will use alternate output in some instances that is faster, but larger.

Generally, the biggest gains to be made in terms of speed of execution come from the algorithm used in a project. Identify which sections of your program need to be fast. Look for loops that might be linearly searching arrays and choose an alternate search method such as a hash table and function. Where results are being recalculated, consider if they can be cached.

3.5.5 How Does the Compiler Place Everything in Memory?

In most situations, assembly instructions and directives associated with both code and data are grouped into sections, and these are then positioned into containers which represent the device memory. To see what sections objects are placed in, use the option `-ai` to view this information in the assembler listing file.

The exception is for absolute variables, which are placed at a specific address when they are defined and which are not placed in a section. For setting absolute variables, use the `address()` attribute (specified in Section 8.11 "Variable Attributes").

3.5.6 How Can I Make My Interrupt Routine Faster?

Consider suggestions made in Section 3.5.2 "How Can I Make My Code Smaller?" (code size) for any interrupt code. Smaller code is often faster code.

In addition to the code you write in the ISR, there is the code the compiler produces to switch context. Because this is executed immediately after an interrupt occurs and immediately before the interrupt returns, it must be included in the time taken to process an interrupt. This code is optimal, in that only registers used in the ISR will be saved by this code. Thus, the fewer registers used in your ISR will mean potentially less context switch code to be executed.

Generally simpler code will require fewer resources than more complicated expressions. Use the assembly list file to see which registers are being used by the compiler in the interrupt code. For information on the list file, see the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide* (DS50002186).

Avoid calling other functions from the ISR. In addition to the extra overhead of the function call, the compiler also saves all general purpose registers that may or may not be used by the called function. Consider having the ISR simply set a flag and return. The flag can then be checked in main-line code to handle the interrupt. This has the advantage of moving the complicated interrupt-processing code out of the ISR so that it no longer contributes to its register usage. Always use the `volatile` qualifier (see Section 8.9.2 "Volatile Type Qualifier") for variables shared by the interrupt and main-line code (see Section 3.4.4 "How Do I Share Data Between Interrupt and Main-line Code?").

3.5.7 How Big Can C Variables Be?

This question specifically relates to the size of individual C objects, such as arrays or structures. The total size of all variables is another matter.

To answer this question you need to know in which memory space the variable will be located. Objects qualified `const` will be located in program memory; other objects will be placed in data memory. Program memory object sizes are discussed in Section 9.4.1 "Size Limitations of `const` Variables". Objects in data memory are broadly grouped into autos and non-autos, with the size limitations of these objects, respectively, discussed in Section 9.2.1 "Non-auto Variable Allocation" and Section 9.2.3 "Non-auto Variable Size Limits".

3.5.8 What Optimizations Will Be Applied to My Code?

Code optimizations available depend on the edition of your compiler (see Chapter 18. "Optimizations"). A description of optimization options can be found under Section 5.8.7 "Options for Controlling Optimization".

3.5.9 What Devices are Supported by the Compiler?

New devices are usually added with each compiler release. Check the readme document for a full list of devices supported by a compiler release.

3.5.10 How Do I Know What Code the Compiler Is Producing?

The assembly list file may be set up using assembler listing file options, to contain a great deal of information about the code, such as the assembly output for almost the entire program, including library routines linked in to your program; section information; symbol listings; and more.

The list file may be produced as follows:

- On the command line, create a basic list file using the option: `-Wa, -a=`*`projectname`*`.lst`.
- For MPLAB X IDE, right click on your project and select "Properties." In the Project Properties window, click on "xc32-as" under "Categories." From "Option categories," select "Listing file options" and ensure "List to file" is checked.

By default, the assembly list file will have a `.lst` extension.

For information on the list file, see the *"MPLAB® XC32 Assembler, Linker and Utilities User's Guide"* (DS50002186).

3.5.11 How Can I Tell How Big a Function Is?

This size of a function (the amount of assembly code generated for that function) can be determined from the assembly list file. See Section 3.5.10 "How Do I Know What Code the Compiler Is Producing?" for more on creating an assembly listing file.

3.5.12 How Do I Know What Resources Are Being Used by Each Function?

In the assembly list file there is information printed for every C function, including library functions. See Section 3.5.10 "How Do I Know What Code the Compiler Is Producing?" for more on creating an assembly listing file.

To see information on functions calls, you can view the Call Graph in MPLAB X IDE (*Window>Output>Call Graph*). You must be in debug mode to see this graph. Right click on a function and select "Show Call Graph" to see what calls this function and what it calls.

Auto, parameter, and temporary variables used by a function may overlap with those from other functions as these are placed in a compiled stack by the compiler (see Section 9.3 "Auto Variable Allocation and Access").

3.5.13 How Do I Find Out Where Variables and Functions Have Been Positioned?

You can determine where variables and functions have been positioned from either the assembly list file (generated by the assembler) or the map file (generated by the linker). Only global symbols are shown in the map file; all symbols (including locals) are listed in the assembly list file.

There is a mapping between C identifiers and the symbols used in assembly code, which are the symbols shown in both of these files. The symbol associated with a variable is assigned the address of the lowest byte of the variable; for functions it is the address of the first instruction generated for that function.

For more on assembly list files and linker map files, see the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide* (DS50002186).

3.5.14 Why are Some Objects Positioned into Memory that I Reserved?

Most variables and functions are placed into sections that are defined in the linker script. See the *"MPLAB® XC32 Assembler, Linker and Utilities User's Guide"* (DS50002186) for details on linker scripts. However, some variables and function are explicitly placed at an address rather than being linked anywhere in an address range, as described in 3.3.3.1 "How Do I Position Variables at an Address I Nominate?" and 3.3.3.2 "How Do I Position Functions at an Address I Nominate?".

Check the assembly list file to determine the names of sections that hold objects and code. Check the linker options in the map file to see if sections have been linked explicitly or if they are linked anywhere in a class. See the *"MPLAB® XC32 Assembler, Linker and Utilities User's Guide"* (DS50002186) for information on each of these files.

3.5.15 How Do I Know How Much Memory Is Still Available?

A memory usage summary is available from the compiler after compilation (`--report-mem` option), from MPLAB X IDE in the Dashboard window. All of these summaries indicate the amount of memory used and the amount still available, but none indicate whether this memory is one contiguous block or broken into many small chunks. Small blocks of free memory cannot be used for larger objects and so out-of-memory errors may be produced even though the total amount of memory free is apparently sufficient for the objects to be positioned.

Consult the linker map file to determine exactly what memory is still available in each linker class. This file also indicates the largest contiguous block in that class if there are memory page divisions. See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide* (DS50002186) for information on the map file.

3.5.16 How Do I Use Library Files In My Project?

See Section 3.2.6 "How Do I Build Libraries?" for information on how you build your own library files. The compiler will automatically include any applicable standard library into the build process when you compile, so you never need to control these files.

To use one or more library files that were built by yourself or a colleague, include them in the list of files being compiled on the command line. The library files can be specified in any position in the file list relative to the source files, but if there is more than one library file, they will be searched in the order specified in the command line.

For example:

```
xc32-gcc -mprocessor=ATSAME70J19B main.c int.c mylib.a
```

If you are using MPLAB X IDE to build a project, add the library file(s) to the Libraries folder that will shown in your project, in the order in which they should be searched. The IDE will ensure that they are passed to the compiler at the appropriate point in the build sequence.

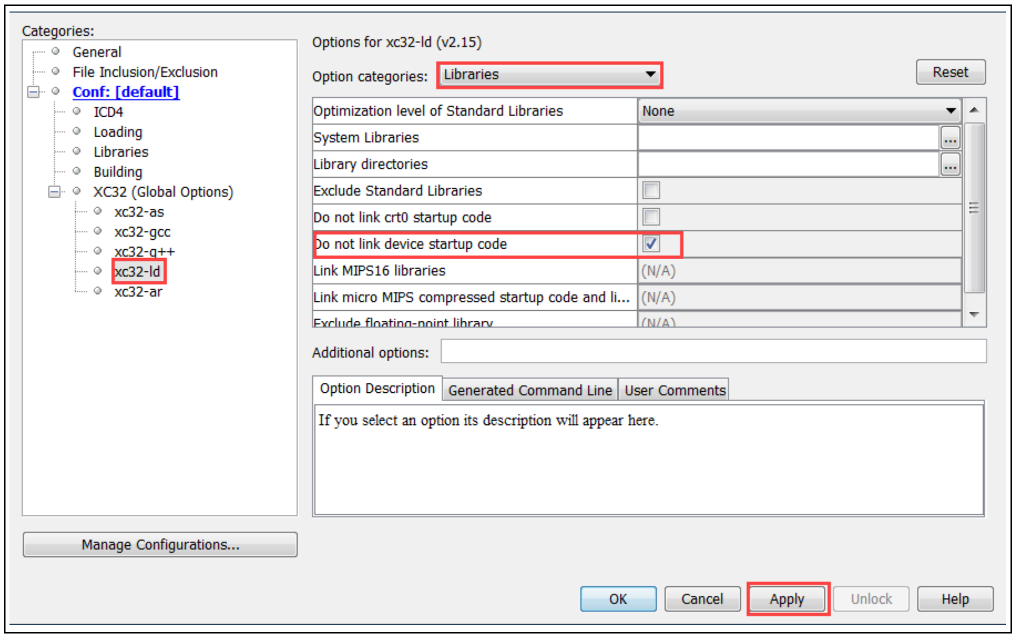
3.5.17 How Do I Customize the C Runtime Startup Code?

Some applications may require an application-specific version of the C runtime startup code. For instance, you may want to modify the startup code for an application loaded by a bootloader.

To customize the startup code for your application:

1. Start with the default startup code, a copy of which is located in `<install-directory>/pic32c/lib/proc/<device>/start-up_<device>.c`
   You may also choose to get this file from the Device Family Pack (DFP). Make a copy of this `.c` file, rename it, and add it to your project.
2. Change your MPLAB X project to exclude the default startup code by passing the `-mno-device-startup-code` option to the xc32-gcc driver at link time. This option is available as "Do not link device startup code" in the MPLAB X project properties under Options for xc32-ld in the Libraries category. When you build your project, the MPLAB X will build your new application-specific copy of the startup code rather than linking in the default code.

FIGURE 3-1: STARTUP CODE PROPERTIES SETTING



Categories:
- General
- File Inclusion/Exclusion
- Conf: [default]
  - ICD4
  - Loading
  - Libraries
  - Building
  - XC32 (Global Options)
    - xc32-as
    - xc32-gcc
    - xc32-g++
    - xc32-ld
    - xc32-ar

Options for xc32-ld (v2.15)

Option categories: Libraries

Optimization level of Standard Libraries	None
System Libraries	
Library directories	
Exclude Standard Libraries	☐
Do not link crt0 startup code	☐
Do not link device startup code	✓
Link MIPS16 libraries	(N/A)
Link micro MIPS compressed startup code and li...	(N/A)
Exclude floating-point library	(N/A)

Additional options:

Option Description | Generated Command Line | User Comments

If you select an option its description will appear here.

3.5.18 What Optimizations Are Employed By The Compiler?

Code optimizations available depend on the edition of your compiler (see Chapter 18. "Optimizations"). A description of optimization options can be found under Section 5.8.7 "Options for Controlling Optimization".

3.6 FIXING CODE THAT DOES NOT WORK

This section examines issues relating to projects that do not build due to compiler errors, or which build but do not work as expected.

- How Do I Set Up Warning/Error Messages?
- How Do I Find the Code that Caused Compiler Errors Or Warnings in My Program?
- How Can I Stop Spurious Warnings from Being Produced?
- Why Can't I Even Blink an LED?
- What Can Cause Corrupted Variables and Code Failure When Using Interrupts?
- Invoking the Compiler
- What Can Cause Corrupted Variables and Code Failure When Using Interrupts?
- Why are Some Objects Positioned into Memory that I Reserved?

3.6.1 How Do I Set Up Warning/Error Messages?

To control message output, see Section 5.8.5 "Options for Controlling Warnings and Errors".

3.6.2 How Do I Find the Code that Caused Compiler Errors Or Warnings in My Program?

In most instances, where the error is a syntax error relating to the source code, the message produced by the compiler indicates the offending line of code. If you are compiling in MPLAB X IDE, then you can double-click the message and have the editor take you to the offending line. But identifying the offending code is not always so easy.

In some instances, the error is reported on the line of code following the line that needs attention. This is because a C statement is allowed to extend over multiple lines of the source file. It is possible that the compiler may not be able to determine that there is an error until it has started to scan the next statement. Consider the following code:

```
input = PORTB   // oops - forgot the semicolon
if(input>6)
  // ...
```

The missing semicolon on the assignment statement will be flagged on the following line that contains the `if()` statement.

In other cases, the error might come from the assembler, not the code generator. If the assembly code was derived from a C source file, then the compiler will try to indicate the line in the C source file that corresponds to the assembly that is at fault. If the source being compiled is an assembly module, the error directly indicates the line of assembly that triggered the error. In either case, remember that the information in the error relates to some problem is the assembly code, not the C code.

Finally, there are errors that do not relate to any particular line of code at all. An error in a compiler option or a linker error are examples of these. If the program defines too many variables, there is no one particular line of code that is at fault; the program as a whole uses too much data. Note that the name and line number of the last processed file and source may be printed even though code is not the direct source of the error.

At the top of each message description, on the right in brackets, is the name of the application that produced this message. Knowing the application that produced the error makes it easier to track down the problem. The compiler application names are indicated in Chapter 4. "XC32 Toolchain and MPLAB X IDE".

If you need to see the assembly code generated by the compiler, look in the assembly list file. For information on where the linker attempted to position objects, see the map file. See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide* (DS50002186) for information about the list and map files.

3.6.3 How Can I Stop Spurious Warnings from Being Produced?

Warnings indicate situations that could possibly lead to code failure. Always check your code to confirm that it is not a possible source of error. In many situations the code is valid and the warning is superfluous. In this case, you may:

- Inhibit specific warnings by using the `-Wno-` version of the option.
- Inhibit all warnings with the `-w` option.
- In MPLAB X IDE, inhibit warnings in the Project Properties window under each tool category. Also look in the Tool Options window, Embedded button, Suppressible Messages tab.

See Section 5.8.5 "Options for Controlling Warnings and Errors" for details.

3.6.4 Why Can't I Even Blink an LED?

Even if you have set up the port register and written a value to the port, there are several things that can prevent such a seemingly simple program from working.

- Make sure that the device's configuration registers are set up correctly, as discussed in Section 7.4 "Configuration Bit Access". Make sure that you explicitly specify every bit in these registers and don't just leave them in their default state. All the configuration features are described in your device data sheet. If the configuration bits that specify the oscillator source are wrong, for example, the device clock may not even be running.
- If the internal oscillator is being used, in addition to configuration bits there may be SFRs you need to initialize to set the oscillator frequency and modes. See Section 7.4 "Configuration Bit Access" and your device data sheet.
- To ensure that the device is not resetting because of the watchdog time, either turn off the timer in the configuration bits or clear the timer in your code. There are library functions you can use to handle the watchdog timer, described in the *32-bit Language Tool Libraries* manual (DS51685). If the device is resetting, it may never reach the lines of code in your program that blink the LED. Turn off any other features that may cause device Reset until your test program is working.
- The device pins used by the port bits are often multiplexed with other peripherals. A pin might be connected to a bit in a port, or it might be an analog input, or it might be the output of a comparator, for example. If the pin connected to your LED is not internally connected to the port you are using, then your LED will never operate as expected. The port function tables in your device data sheets will show other uses for each pin which will help you identify peripherals to investigate.

3.6.5 What Can Cause Corrupted Variables and Code Failure When Using Interrupts?

This is usually caused by having variables used by both interrupt and main-line code. If the compiler optimizes access to a variable or access is interrupted by an interrupt routine, then corruption can occur. See Section 3.4.4 "How Do I Share Data Between Interrupt and Main-line Code?" for more information.

NOTES:

Chapter 4. XC32 Toolchain and MPLAB X IDE

The 32-bit language tools may be used together under MPLAB X IDE to provide GUI development of application code for the PIC32 MCU families of devices. The tools are:

- MPLAB XC32 C/C++ Compiler
- MPLAB XC32 Assembler
- MPLAB XC32 Object Linker
- MPLAB XC32 Object Archiver/Librarian and other 32-bit utilities

4.1 MPLAB X IDE AND TOOLS INSTALLATION

In order to use the 32-bit language tools with MPLAB X IDE, you must install:

- MPLAB X IDE, which is available for free on the Microchip website.
- MPLAB XC32 C/C++ Compiler, which includes all of the 32-bit language tools. The compiler is available for free (Free and Evaluation editions) or for purchase (Pro edition) on the Microchip website.

The 32-bit language tools will be installed, by default, in the directory:

- Windows OS - `C:\Program Files\Microchip\xc32\`*`x.xx`*
- Mac OS - `Applications/microchip/xc32/`*`x.xx`*
- Linux OS - `/opt/microchip/xc32/`*`x.xx`*

where *`x.xx`* is the version number.

The executables for each tool will be in the `bin` subdirectory:

- C Compiler - `xc32-gcc.exe`
- Assembler - `xc32-as.exe`
- Object Linker - `xc32-ld.exe`
- Object Archiver/Librarian - `xc32-ar.exe`
- Other Utilities

All device include (header) files are located in the `/pic32c/include/proc` subdirectory. These files are automatically incorporated when you `#include` the xc.h header file.

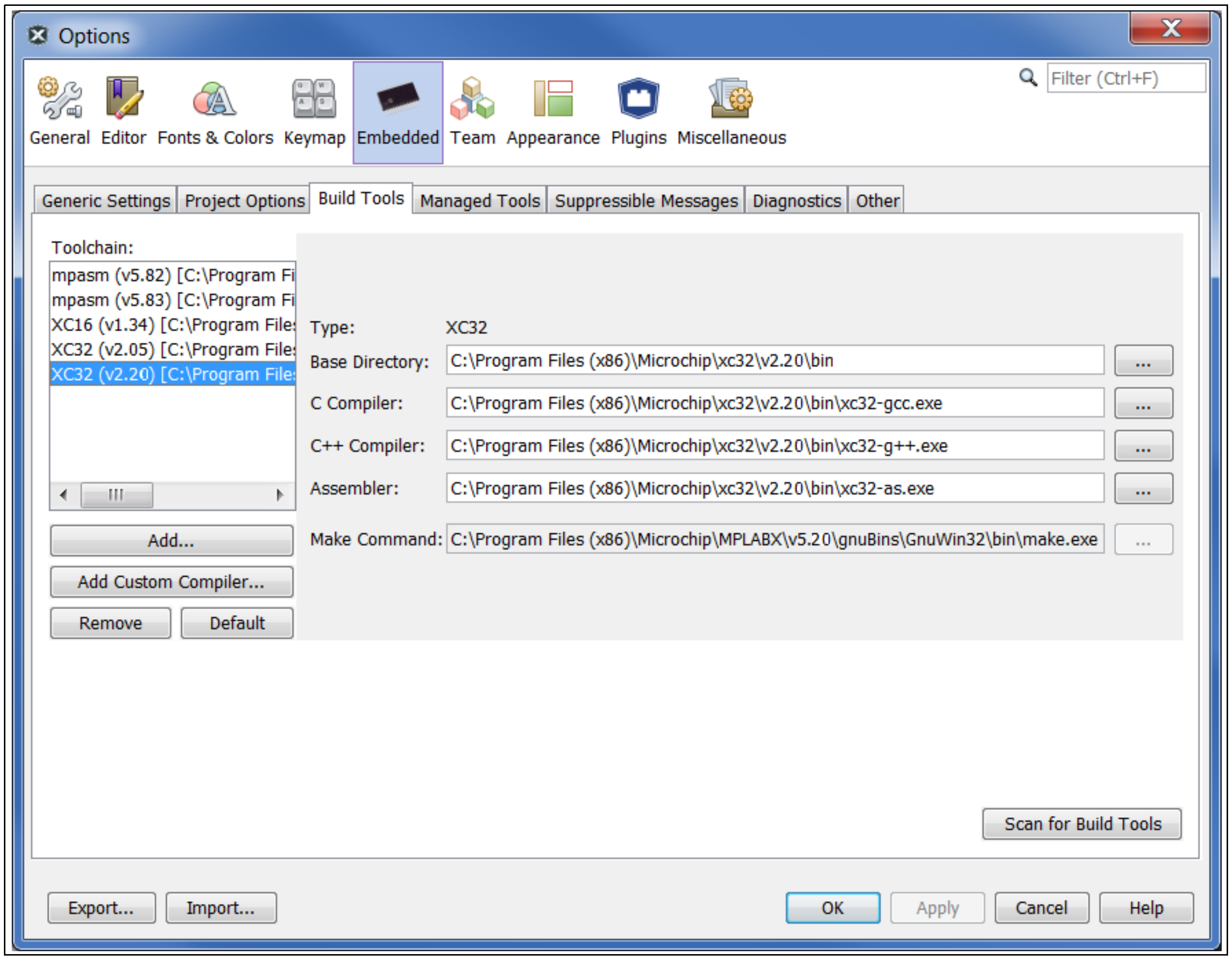
Code examples are located in the examples directory.

4.2 MPLAB X IDE SETUP

Once MPLAB X IDE is installed on your PC, launch the application and check the settings below to ensure that the 32-bit language tools are properly recognized.

1. From the MPLAB X IDE menu bar, select *Tools>Options* to open the Options dialog. Click on the **Embedded** button and select the "Build Tools" tab.
2. Click on "XC32" under "Toolchain." Ensure that the paths are correct for your installation.
3. Click **OK**.

FIGURE 4-1: XC32 TOOLSUITE LOCATIONS IN WINDOWS® OS



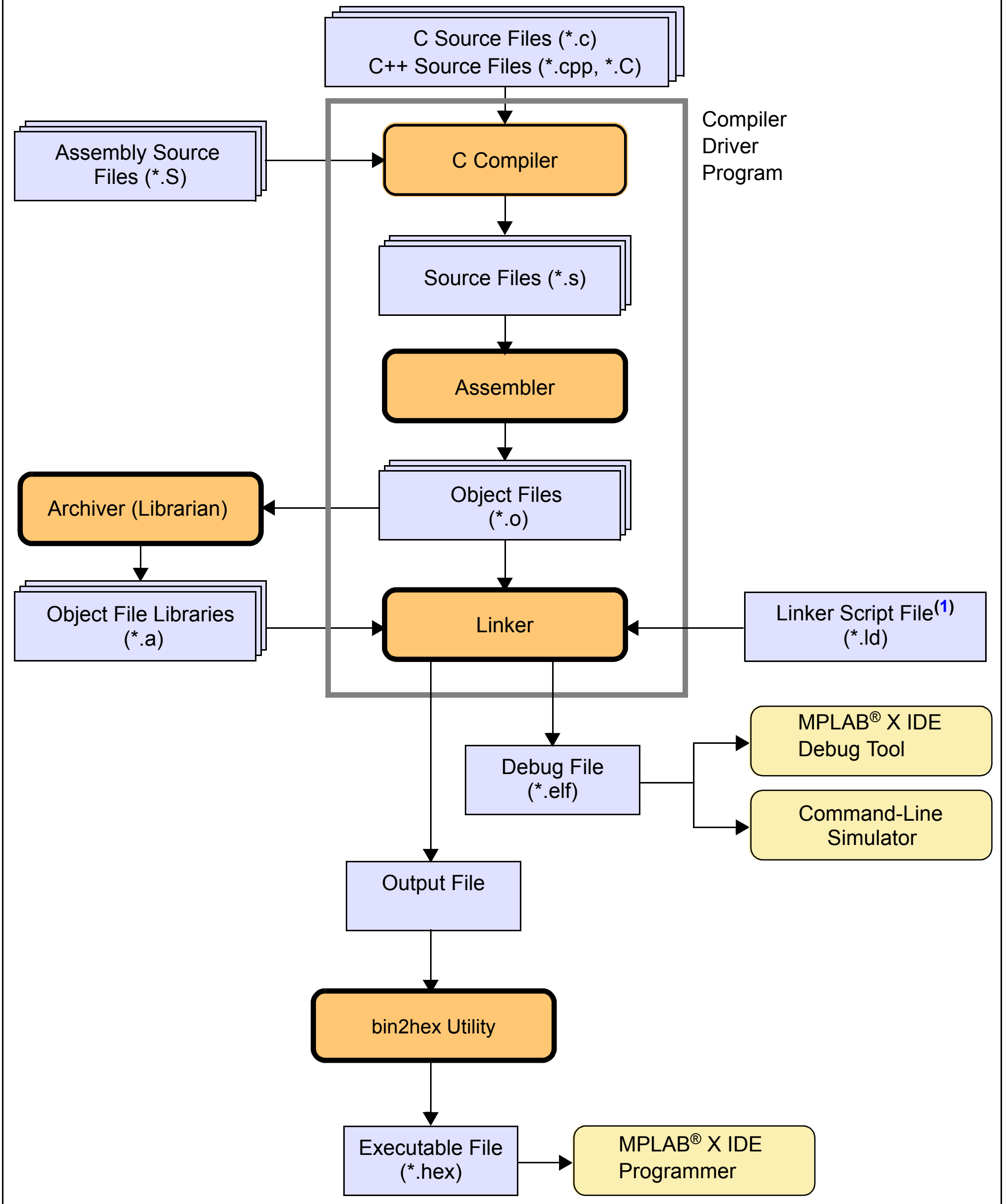
Toolchain:
mpasm (v5.82) [C:\Program Fi
mpasm (v5.83) [C:\Program Fi
XC16 (v1.34) [C:\Program Files
XC32 (v2.05) [C:\Program Files
XC32 (v2.20) [C:\Program Files

Type:	XC32
Base Directory:	C:\Program Files (x86)\Microchip\xc32\v2.20\bin
C Compiler:	C:\Program Files (x86)\Microchip\xc32\v2.20\bin\xc32-gcc.exe
C++ Compiler:	C:\Program Files (x86)\Microchip\xc32\v2.20\bin\xc32-g++.exe
Assembler:	C:\Program Files (x86)\Microchip\xc32\v2.20\bin\xc32-as.exe
Make Command:	C:\Program Files (x86)\Microchip\MPLABX\v5.20\gnuBins\GnuWin32\bin\make.exe

4.3 MPLAB X IDE PROJECTS

A project in MPLAB X IDE is a group of files needed to build an application, along with their associations to various build tools. Below is a generic MPLAB X IDE project.

FIGURE 4-2: COMPILER PROJECT RELATIONSHIPS



Note 1: The linker can choose the correct linker script file for your project.

In this MPLAB X IDE project, C source files are shown as input to the compiler. The compiler will generate source files for input into the assembler. For more information on the compiler, see the compiler documentation.

Assembly source files are shown as input to the C preprocessor. The resulting source files are input to the assembler. The assembler will generate object files for input into the linker or archiver. For more information on the assembler, see the assembler documentation.

Object files can be archived into a library using the archiver/librarian. For more information on the archiver, see the archiver/librarian documentation.

The object files and any library files, as well as a linker script file (generic linker scripts are added automatically), are used to generate the project output files via the linker. The output file that may be generated by the linker is a debug file (`.elf`) used by the simulator and debug tools which may be input into the bin2hex utility to produce an executable file (`.hex`). For more information on linker script files and using the object linker, see the linker documentation.

For more on projects, and related workspaces, see MPLAB X IDE documentation.

4.4 PROJECT SETUP

To set up an MPLAB X IDE project for the first time, use the built-in Project Wizard (*File>New Project*.) In this wizard, you will be able to select a language toolsuite that uses the 32-bit language tools. For more on the wizard, and MPLAB X IDE projects, see MPLAB X IDE documentation.

Once you have a project set up, you may then set up properties of the tools in MPLAB X IDE.

1. From the MPLAB X IDE menu bar, select *File>Project Properties* to open a window to set/check project build options.
2. Under “Conf:[*default*]”, select a tool from the tool collection to set up.

4.4.1 XC32 (Global Options)

Set up global options for all 32-bit language tools (see also Section 4.4.6 “Options Page Features”).

TABLE 4-1: XC32 (GLOBAL OPTIONS) ALL OPTIONS CATEGORY

Option	Description	Command Line
Don’t delete intermediate files	Don’t delete intermediate Files. Place them in the object directory and name them based on the source file.	`-save-temps=obj`
Use Whole-Program and Link-Time Optimizations	When this feature is enabled, the build will be constrained in the following ways: - The per-file build settings will be ignored - The build will no longer be an incremental one (full build only)	`-fwhole-program` `-flto`
Common include dirs	Directory paths entered here will be appended to the already existing include paths of the compiler. Relative paths are from the MPLAB X IDE project directory.	`-I"`*`dir`*`"`

4.4.2 xc32-as (32-bit Assembler)

A subset of command-line options may be specified in MPLAB X IDE. Select a category and then set up assembler options. For additional options, see MPLAB XC32 Assembler documentation (and Section 4.4.6 “Options Page Features”).

TABLE 4-2: XC32-AS GENERAL OPTIONS CATEGORY

Option	Description	Command Line
Have symbols in production build	Generate debugging information for source-level debugging in MPLAB X.	`--gdwarf-2`
Keep local symbols	Check to keep local symbols, i.e., labels beginning with `.L` (upper case only). Uncheck to discard local symbols.	`--keep-locals`
Preprocessor macro definitions	Project-specific preprocessor macro defines passed via the compiler's `-D` option.	
Assembler symbols	Define symbol 'sym' to a given 'value'.	`--defsym` *`sym=value`*
Preprocessor Include directories	Relative paths are from MPLAB X project directory.	
Assembler Include directories	Relative paths are from MPLAB X project directory. Add a directory to the list of directories the assembler searches for files specified in `.include` directives. You may add as many directories as necessary to include a variety of paths. The current working directory is always searched first and then `-I` directories in the order in which they were specified (left to right) here.	`-I`

TABLE 4-3: XC32-AS OTHER OPTIONS CATEGORY

Option	Description	Command Line

Diagnostics level	Select warnings to display in the Output window.
	- Generate warnings
	- Suppress warnings
	- Fatal warnings
		
		`--warn`
		`--no-warn`
		`--fatal-warnings`

Include source code	Check for a high-level language listing. High-level listings require that the assembly source code is generated by a compiler, a debugging option like `-g` is given to the compiler, and assembly listings (`-al`) are requested. Uncheck for a regular listing.	`-ah`
Expand macros	Check to expand macros in a listing. Uncheck to collapse macros.	`-am`
Include false conditionals	Check to include false conditionals (`.if`, `.ifdef`) in a listing. Uncheck to omit false conditionals.	`-ac`
Omit forms processing	Check to turn off all forms processing that would be performed by the listing directives `.psize, .eject, .title and .sbttl.` Uncheck to process by listing directives.	`-an`
Include assembly	Check for an assembly listing. This `-a` suboption may be used with other suboptions. Uncheck to exclude an assembly listing.	`-al`
List symbols	Check for a symbol table listing. Uncheck to exclude the symbol table from the listing.	`-as`

TABLE 4-3: XC32-AS OTHER OPTIONS CATEGORY (CONTINUED)

Option	Description	Command Line
Omit debugging directives	Check to omit debugging directives from a listing. This can make the listing cleaner. Uncheck to included debugging directives.	`-ad`
List to file	Use this option if you want the listing for a file. The list file will have the same name as the asm file plus `.lst`.	`-a=${CURRENT_QUOTED_IF_SPACED_OBJECT_FILE_MINUS_EXTENSION}.lst`

4.4.3 xc32-gcc (32-bit C Compiler)

Although the MPLAB XC32 C/C++ Compiler works with MPLAB X IDE, it must be acquired separately. The full version may be purchased, or a student (limited-feature) version may be downloaded for free. See the Microchip website (www.microchip.com) for details.

A subset of command-line options may be specified in MPLAB X IDE. Select a category, and then set up compiler options. For additional options, see the *MPLAB X IDE User’s Guide* (DS50002027), also available on the Microchip website.

See also Section 4.4.6 “Options Page Features”.

TABLE 4-4: XC32-GCC GENERAL CATEGORY

Option	Description	Command Line
Have symbols in production build	Build for debugging in a production build image.	`-g`
Isolate each function in a section	This option is often used with the linker's `--gc-sections` option to remove unreferenced functions. Check to place each function into its own section in the output file. The name of the function determines the section’s name in the output file. Note: When you specify this option, the assembler and linker may create larger object and executable files and will also be slower. Uncheck to place multiple functions in a section.	`-ffunction-sections`
Place data into its own section	This option is often used with the linker's `--gc-sections` option to remove unreferenced statically-allocated variables. Place each data item into its own section in the output file. The name of the data item determines the name of the section. When you specify this option, the assembler and linker may create larger object and executable files and will also be slower.	`-fdata-sections`
Use indirect calls	Enable full-range calls.	`-mlong-calls`

TABLE 4-5: XC32-GCC OPTIMIZATION CATEGORY

Option	Description	Command Line
Optimization Level	Select an optimization level. Your compiler edition may support only some optimizations. Equivalent to `-On` option, where *n* is an option below: • 0 - Do not optimize.The compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. • 1 - Optimize. Optimizing compilation takes somewhat longer, and a lot more host memory for a large function. The compiler tries to reduce code size and execution time. • 2 - Optimize even more. The compiler performs nearly all supported optimizations that do not involve a space-speed trade-off. • 3 - Optimize yet more favoring speed (superset of O2). • s - Optimize yet more favoring size (superset of O2).	`-On`
Unroll loops	This option often increases execution speed at the expense of larger code size. Check to perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or run time. Uncheck to not unroll loops.	`-funroll-loops`
Omit frame pointer	Check to not keep the Frame Pointer in a register for functions that don't need one. Uncheck to keep the Frame Pointer.	`-fomit-frame-pointer`

Pre-optimization instruction scheduling	Default for optimization level:
	- Disable
	- Enable
		
		`-fno-schedule-insns`
		`-fschedule-insns`

Post-optimization instruction scheduling	Default for optimization level:
	- Disable
	- Enable
		
		`-fno-schedule-insns2`
		`-fschedule-insns2`

TABLE 4-6: XC32-GCC PREPROCESSING AND MESSAGES CATEGORY

Option	Description	Command Line
Preprocessor macros	Project-specific preprocessor macro defines passed via the compiler's `-D` option.	
Include directories	Search these directories for project-specific include files.	
Make warnings into errors	Check to halt compilation based on warnings as well as errors. Uncheck to halt compilation based on errors only.	`-Werror`
Additional warnings	Check to enable all warnings. Uncheck to disable warnings.	`-Wall`
support-ansi	Check to issue all warnings demanded by strict ANSI C. Uncheck to issue all warnings.	`-ansi`
strict-ansi	Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++.	`-pedantic`
Use CCI syntax	Enable support for the CCI syntax (see Chapter 2. "Common C Interface").	`-mcci`

4.4.4 xc32-g++(32-bit C++ Compiler)

A subset of command-line options may be specified in MPLAB X IDE. Select a category, and then set up linker options. For additional options, see MPLAB Object Linker for 32-bit Devices documentation (and Section 4.4.6 "Options Page Features").

TABLE 4-7: XC32-G++ C++ SPECIFIC CATEGORY

Option	Description	Command Line
Generate run time type descriptor information	Enable generation of information about every class with virtual functions for use by the C++ runtime type identification features ('dynamic_cast' and 'typeid'). If you don't use those parts of the language, you can save some space by disabling this option. Note that exception handling uses the same information, but it will generate it as needed. The 'dynamic_cast' operator can still be used for casts that do not require runtime type information, i.e., casts to void * or to unambiguous base classes.	`-frtti`
Enable C++ exception handling	Enable exception handling. Generates extra code needed to propagate exceptions.	`-fexceptions`
Check that the pointer returned by operator 'new' is non-null	Check that the pointer returned by operator new is non-null before attempting to modify the storage allocated.	`-fcheck-new`
Generate code to check for violation of exception specification	Generate code to check for violation of exception specifications at runtime. This option violates the C++ standard, but may be useful for reducing code size in production builds.	`-fenforce-eh-specs`

TABLE 4-8: XC32-G++ GENERAL CATEGORY

Option	Description	Command Line
Have symbols in production build	Build for debugging in a production build image.	`-g`
Isolate each function in a section	Place each function into its own section in the output file if the target supports arbitrary sections. The name of the function or the name of the data item determines the section's name in the output file. This option is useful when combined with the linker's `--gc-sections` option to remove unreferenced functions.	`-ffunction-sections`
Place data into its own section	Place each data item into its own section in the output file if the target supports arbitrary sections. The name of the function or the name of the data item determines the section's name in the output file. This option is useful when combined with the linker's `--gc-sections` option to remove unreferenced variables.	`-fdata-sections`
Use indirect calls	Enable full-range calls.	`-mlong-calls`

TABLE 4-9: XC32-G++ OPTIMIZATION CATEGORY

Option	Description	Command Line
Optimization Level	Select an optimization level. Your compiler edition may support only some optimizations. Equivalent to `-On` option, where *n* is an option below: • 0 - Do not optimize.The compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. • 1 - Optimize. Optimizing compilation takes somewhat longer, and a lot more host memory for a large function. The compiler tries to reduce code size and execution time. • 2 - Optimize even more. The compiler performs nearly all supported optimizations that do not involve a space-speed trade-off. • 3 - Optimize yet more favoring speed (superset of O2). • s - Optimize yet more favoring size (superset of O2).	`-On`
Unroll loops	Check to perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or run time. Uncheck to not unroll loops.	`-funroll-loops`
Omit frame pointer	Check to not keep the Frame Pointer in a register for functions that don't need one. Uncheck to keep the Frame Pointer.	`-fomit-frame-pointer`
Pre-optimization instruction scheduling	Default for optimization level:	
	- Disable	`-fno-schedule-insns`
	- Enable	`-fschedule-insns`
Post-optimization instruction scheduling	Default for optimization level:	
	- Disable	`-fno-schedule-insns2`
	- Enable	`-fschedule-insns2`

TABLE 4-10: XC32-G++ OPTIMIZATION CATEGORY

Option	Description	Command Line
Preprocessor macros	Project-specific preprocessor macro defines passed via the compiler's `-D` option.	
Include directories	Search these directories for project-specific include files.	
Make warnings into errors	Check to halt compilation based on warnings as well as errors. Uncheck to halt compilation based on errors only.	`-Werror`
Additional warnings	Check to enable all warnings. Uncheck to disable warnings.	`-Wall`
support-ansi	Check to issue all warnings demanded by strict ANSI C. Uncheck to issue all warnings.	`-ansi`
strict-ansi	Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++.	`-pedantic`
Use CCI syntax	Enable support for the CCI syntax (Chapter 2. "Common C Interface").	`-mcci`

4.4.5 xc32-ld (32-Bit Linker)

A subset of command-line options may be specified in MPLAB X IDE. Select a category, and then set up linker options. For additional options, see MPLAB Object Linker for 32-bit Devices documentation (and Section 4.4.6 “Options Page Features”).

TABLE 4-11: XC32-LD GENERAL CATEGORY

Option	Description	Command Line
Heap Size (bytes)	Specify the size of the heap in bytes. Allocate a run-time heap of size bytes for use by C programs. The heap is allocated from unused data memory. If not enough memory is available, an error is reported.	`--defsym=_min_heap_size=<size>`
Minimum stack size (bytes)	Specify the minimum size of the stack in bytes. By default, the linker allocates all unused data memory for the run-time stack. Alternatively, the programmer may allocate the stack by declaring two global symbols: `__SP_init` and `__SPLIM_init`. Use this option to ensure that at least a minimum sized stack is available. The actual stack size is reported in the link map output file. If the minimum size is not available, an error is reported.	`--defsym=_min_stack_size=<size>`
Allow overlapped sections	Check to not check section addresses for overlaps. Uncheck to check for overlaps.	`--check-sections` `--no-check-sections`
Remove unused sections	Check to not enable garbage collection of unused input sections (on some targets). Uncheck to enable garbage collection.	`--no-gc-sections` `--gc-sections`
Use response file to link	Pass linker options in a file rather than on the command line. On Windows systems, this option allows you to properly link projects with a large number of object files that would normally overrun the command-line length limitation of the Windows OS.	`True`
Additional driver options	Enter any additional driver options not existing in this GUI otherwise. The string you introduce here will be emitted as is in the driver invocation command.	

TABLE 4-12: XC32-LD LIBRARIES CATEGORY

Option	Description	Command Line
Optimization level of Standard Libraries	Select an optimization level. Your compiler edition may support only some optimizations. Equivalent to -On option, where n is an option below: • 0 - Do not optimize.The compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. • 1 - Optimize. Optimizing compilation takes somewhat longer, and a lot more host memory for a large function. The compiler tries to reduce code size and execution time. • 2 - Optimize even more. The compiler performs nearly all supported optimizations that do not involve a space-speed trade-off. • 3 - Optimize yet more favoring speed (superset of O2). • s - Optimize yet more favoring size (superset of O2).	`-O`*`n`*
System Libraries	Add libraries to be linked with the project files. You may add more than one.	`--library=name`
Library directories	Add a library directory to the library search path. You may add more than one.	`--library-path="name"`
Exclude Standard Libraries	Check to not use the standard system startup files or libraries when linking. Only use library directories specified on the command line. Uncheck to use the standard system startup files and libraries.	`-nostdlib`
Do not link startup code	Exclude the default startup code because the project provides application-specific startup code.	`-nostartfiles`
Do not link device startup code	Do not link the default device-specific startup code (for example, `startup_,device>.c`)	`-mno-device-startup-code`

TABLE 4-13: XC32-LD DIAGNOSTICS CATEGORY

Option	Description	Command Line
Generate map file	Create a map file.	`-Map="`*`file`*`"`
Display memory usage	Check to print memory usage report. Uncheck to not print a report.	`--report-mem`
Generate cross-reference file	Check to create a cross-reference table. Uncheck to not create this table.	`--cref`
Warn on section realignment	Check to warn if start of section changes due to alignment. Uncheck to not warn.	`--warn-section-align`
Trace Symbols	Add/remove trace symbols.	`--trace-symbol=symbol`

TABLE 4-14: XC32-LD SYMBOLS AND MACROS CATEGORY

Option	Description	Command Line
Linker symbols	Create a global symbol in the output file containing the absolute address (*expr*). You may use this option as many times as necessary to define multiple symbols in the command line. A limited form of arithmetic is supported for the *expr* in this context: you may give a hexadecimal constant or the name of an existing symbol, or use + and - to add or subtract hexadecimal constants or symbols.	`--defsym=sym`
Preprocessor macro definitions	Add linker macros.	`-Dmacro`

Symbols	Specify symbol information in the output.
	- Keep all
	- Strip debugging info
	- Strip all symbol info
		
		—
		`--strip-debug (-S)`
		`--strip-all (-s)`

4.4.6 Options Page Features

The Options section of the Properties page has the following features for all tools:

TABLE 4-15: PAGE FEATURES OPTIONS

Reset	Reset the page to default values.
Additional options	Enter options in a command-line (non-GUI) format.
Option Description	Click on an option name to see information on the option in this window. Not all options have information in this window.
Generated Command Line	Click on an option name to see the command-line equivalent of the option in this window.

4.5 PROJECT EXAMPLE

In this example, you will create an MPLAB X IDE project with two C code files.

4.5.1 Run the Project Wizard

In MPLAB X IDE, select <u>*File>New Project*</u> to launch the wizard.

1. **Choose Project:** Select "Microchip Embedded" for the category and "Standalone Project" for the project. Click **Next>** to continue.
2. **Select Device:** Select the dsPIC30F6014. Click **Next>** to continue.
3. **Select Header:** There is no header for this device so this is skipped.
4. **Select Tool:** Choose a development tool from the list. Tool support for the selected device is shown as a colored circle next to the tool. Mouse over the circle to see the support as text. Click **Next>** to continue.
5. **Select Compiler:** Choose a version of the XC32 toolchain. Click **Next>** to continue.
6. **Select Project Name and Folder:** Enter a project name, such as `MyXC32Project`. Then select a location for the project folder. Click **Finish** to complete the project creation and setup.

Once the Project Wizard has completed, the Project window should contain the project tree. For more on projects, see the MPLAB X IDE documentation.

4.5.2 Set Build Options

Select *File>Project Properties* or right click on the project name and select "Properties" to open the Project Properties dialog.

1. Under "Conf:[default]>XC32 (Global Options)", select "xc32-gcc."
2. Under "Conf:[default]>XC32 (Global Options)", select "xc32-ld."
3. Select "Diagnostics" from the "Option Categories". Then enter a file name to "Generate map file," i.e., `example.map`.
4. Click **OK** on the bottom of the dialog to accept the build options and close the dialog.

4.5.3 Build the Project

Right click on the project name, "MyXC32Project," in the project tree and select "Build" from the pop-up menu. The Output window displays the build results.

If the build did not complete successfully, check these items:

1. Review the previous steps in this example. Make sure you have set up the language tools correctly and have all the correct project files and build options.
2. If you modified the sample source code, examine the Build tab of the Output window for syntax errors in the source code. If you find any, click on the error to go to the source code line that contains that error. Correct the error, and then try to build again.

4.5.4 Output Files

View the project output files by opening the files in MPLAB X IDE.

1. Select *File>Open File*. In the Open dialog, find the project directory.
2. Under "Files of type" select "All Files" to see all project files.
3. Select *File>Open File*. In the Open dialog, select "example.map." Click **Open** to view the linker map file in an MPLAB X IDE editor window. For more on this file, see the linker documentation.
4. Select *File>Open File*. In the Open dialog, return to the project directory and then go to the *dist>default>production* directory. Notice that there is only one hex file, "MyXC32Project.X.production.hex." This is the primary output file. Click **Open** to view the hex file in an MPLAB X IDE editor window. For more on this file, see the Utilities documentation.
   There is also another file, "MyXC32Project.X.production.elf." This file contains debug information and is used by debug tools to debug your code. For information on selecting the type of debug file, see Section 4.4.1 "XC32 (Global Options)".

4.5.5 Further Development

Usually, your application code will contain errors and not work the first time. Therefore, you will need a debug tool to help you develop your code. Using the output files previously discussed, several debug tools exist that work with MPLAB X IDE to help you do this. You may choose from simulators, in-circuit emulators or in-circuit debuggers, either manufactured by Microchip Technology or third-party developers. Please see the documentation for these tools to learn how they can help you. When debugging, you will use *Debug>Debug Project* to run and debug your code. Please see MPLAB X IDE documentation for more information.

Once you have developed your code, you will want to program it into a device. Again, there are several programmers that work with MPLAB X IDE to help you do this. Please see the documentation for these tools to see how they can help you. When programming, you will use the **Make and Program Device Project** button on the debug toolbar. Please see MPLAB X IDE documentation concerning this control.

Chapter 5. Compiler Command Line Driver

The command line driver (`xc32-gcc` or `xc32-g++`) is the application that can be invoked to perform all aspects of compilation, including C/C++ code generation, assembly and link steps. Even if you use an IDE to assist with compilation, the IDE will ultimately call `xc32-gcc` for C projects or `xc32-g++` for C++ projects. MPLAB X IDE uses various heuristics to determine the project language. In some cases, it will add the -x flag to select the correct language, C or C++, and use any of the two drivers.

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

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

5.1 INVOKING THE COMPILER

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

5.1.1 Driver Command Line Format

The compilation driver program (`xc32-gcc`) compiles, assembles and links C and assembly language modules and library archives. The `xc32-g++` driver must be used when the module source is written in C++. Most of the compiler command line options are common to all implementations of the GCC toolset (MPLAB XC32 uses the GCC toolset; XC8 does not). A few are specific to the compiler.

The basic form of the compiler command line is:

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

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

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

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

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

The available options are described in Section 5.8 "Driver Option Descriptions". It is conventional to supply *`options`* (identified by a leading *dash* "-" before the filenames), although this is not mandatory.

The *files* may be any mixture of C/C++ and assembler source files, relocatable object files (`.o`) or archive files. The order of the files is important. It may affect the order in which code or data appears in memory or the search order for symbols. Typically archive files are specified after source files. The file types are described in Section 5.1.3 "Input File Types".

Note: Command line options and file names are case sensitive.

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

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

5.1.2 Environment Variables

The variables in this section are optional, but if defined, they will be used by the compiler. The compiler driver, or other subprogram, may choose to determine an appropriate value for some of the following environment variables if they are not set. The driver, or other subprogram, takes advantage of internal knowledge about the installation of the compiler. As long as the installation structure remains intact, with all subdirectories and executables remaining in the same relative position, the driver or subprogram will be able to determine a usable value. The "XC32" variables should be used for new projects; however, the "PIC32" variables may be used for legacy projects.

TABLE 5-1: COMPILER-RELATED ENVIRONMENT VARIABLES

Option	Definition
`XC32_C_INCLUDE_PATH` `PIC32_C_INCLUDE_PATH`	This variable's value is a semicolon-separated list of directories, much like `PATH`. When the compiler searches for header files, it tries the directories listed in the variable, after the directories specified with `-I` but before the standard header file directories. If the environment variable is undefined, the preprocessor chooses an appropriate value based on the standard installation. By default, the following directories are searched for include files: `<install-path>\xc32\include`
`XC32_COMPILER_PATH` `PIC32_COMPILER_PATH`	The value of `XC32_COMPILER_PATH` is a semicolon-separated list of directories, much like `PATH`. The compiler tries the directories thus specified when searching for subprograms, if it can't find the subprograms using `XC32_EXEC_PREFIX`.
`XC32_EXEC_PREFIX` `PIC32_EXEC_PREFIX`	If `XC32_EXEC_PREFIX` is set, it specifies a prefix to use in the names of subprograms executed by the compiler. No directory delimiter is added when this prefix is combined with the name of a subprogram, but you can specify a prefix that ends with a slash if you wish. If the compiler cannot find the subprogram using the specified prefix, it tries looking in your `PATH` environment variable. If the `XC32_EXEC_PREFIX` environment variable is unset or set to an empty value, the compiler driver chooses an appropriate value based on the standard installation. If the installation has not been modified, this will result in the driver being able to locate the required subprograms. Other prefixes specified with the `-B` command line option take precedence over the user- or driver-defined value of `XC32_EXEC_PREFIX`. Under normal circumstances it is best to leave this value undefined and let the driver locate subprograms itself.
`XC32_LIBRARY_PATH` `PIC32_LIBRARY_PATH`	This variable's value is a semicolon-separated list of directories, much like `PATH`. This variable specifies a list of directories to be passed to the linker. The driver's default evaluation of this variable is: `<install-path>\lib; <install-path>\xc32\lib.`

TABLE 5-1: COMPILER-RELATED ENVIRONMENT VARIABLES (CONTINUED)

Option	Definition
`TMPDIR`	If `TMPDIR` is set, it specifies the directory to use for temporary files. The compiler uses temporary files to hold the output of one stage of compilation that is to be used as input to the next stage: for example, the output of the preprocessor, which is the input to the compiler proper.

5.1.3 Input File Types

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

TABLE 5-2: FILE NAMES

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

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

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

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

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

5.2 THE C COMPILATION SEQUENCE

5.2.1 Single-step C Compilation

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

5.2.1.1 COMPILING A SINGLE C FILE

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

- `<install-dir>/pic32c/include` - the directory for standard C header files.
- `<install-dir>/pic32c/include/proc` - the directory for PIC32 device-specific header files.
- `<install-dir>/pic32c/include/peripheral` - the directory for PIC32 peripheral library include files.
- `<install-dir>/pic32c/lib` - the directory structure for standard libraries and start-up files.
- `<install-dir>/pic32c/lib/proc` - the directory for device-specific linker script fragments, register definition files and configuration data may be found.

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

```
/* ex1.c*/
// ATSAME70N20B Configuration Bit Settings

// Config Source code for XC32 compiler.
// GPNVMBITS
#pragma config SECURITY_BIT = CLEAR
#pragma config BOOT_MODE = CLEAR
#pragma config TCM_CONFIGURATION = 0x0 // Enter Hexadecimal value

// ONLY Set LOCKBITS are generated.
// LOCKBIT_WORD0

// LOCKBIT_WORD1

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.

#include <xc.h>

unsigned int x, y, z;

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

int
main(void) {
    x = 2;
    y = 5;
    z = add(x, y);
    return 0;
}
```

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

Compile the program by typing the following at the prompt:

```
xc32-gcc -mprocessor=ATSAME70N20B -o ex1.elf ex1.c
```

The command line option `-o ex1.elf` names the output executable file (if the `-o` option is not specified, then the output file is named `a.out`). The executable file may be imported into MPLAB X IDE with `File-Import-Hex/Elf (prebuilt) File`.

If a hex file is required to load in a device programmer, use the following command:

```
xc32-bin2hex ex1.elf
```

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

5.2.1.2 COMPILING MULTIPLE C FILES

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

File 1

```
/* ex1.c*/
// ATSAME70N20B Configuration Bit Settings

// Config Source code for XC32 compiler.
// GPNVMBITS
#pragma config SECURITY_BIT = CLEAR
#pragma config BOOT_MODE = CLEAR
#pragma config TCM_CONFIGURATION = 0x0 // Enter Hexadecimal value
// ONLY Set LOCKBITS are generated.
// LOCKBIT_WORD0
// LOCKBIT_WORD1
// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.

#include <xc.h>

int main(void);
unsigned int add(unsigned int a, unsigned int b);
unsigned int x, y, z;

int main(void) {
    x = 2;
    y = 5;
    z = add(x, y);
    return 0;
}
```

File 2

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

Compile both files by typing the following at the prompt:

```
xc32-gcc -mprocessor=ATSAME70N20B -o ex1.elf ex1.c add.c
```

This command compiles the modules `ex1.c` and `add.c`. The compiled modules are linked with the compiler libraries and the executable file `ex1.elf` is created.

5.2.2 Multi-step C Compilation

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

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

If the compiler is being invoked using a make utility, the make file will need to be configured to use the intermediate files (`.o` files) and the options used to generate the intermediate files (`-c`, see Section 5.8.2 "Options for Controlling the Kind of Output"). Make utilities typically call the compiler multiple times: once for each source file to generate an intermediate file, and once to perform the second stage compilation.

For example, the files `ex1.c` and `add.c` listed in Section 5.2.1.2 "Compiling Multiple C Files" can be compiled separately with:

```
xc32-gcc -mprocessor=ATSAME70N20B -c ex1.c
xc32-gcc -mprocessor=ATSAME70N20B -c add.c
xc32-gcc -mprocessor=ATSAME70N20B -o ex1.elf ex1.o add.o
```

5.3 THE C++ COMPILATION SEQUENCE

5.3.1 Single-step C++ Compilation

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

5.3.1.1 COMPILING A SINGLE C++ FILE

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

- `<install-dir>/pic32c/include/c++` - the directory for standard C++ header files.
- `<install-dir>/pic32c/include/proc` - the directory for PIC32 device-specific header files.
- `<install-dir>/pic32c/include/peripheral` - the directory for PIC32 peripheral library include files.
- `<install-dir>/pic32c/lib` - the directory structure for standard libraries and start-up files.
- `<install-dir>/pic32c/lib/proc` - the directory for device-specific linker script fragments, register definition files and configuration data.

The following is a simple C++ program. Create the following program with any plain-text editor and save it as `ex1.cpp`.

File 1

```
/* ex1.cpp */

// ATSAME70Q21B Configuration Bit Settings
#pragma config SECURITY_BIT = CLEAR
#pragma config BOOT_MODE = CLEAR
#pragma config TCM_CONFIGURATION = 0x0 // Enter Hexadecimal value

#include <xc.h>
#include <iostream>
using namespace std;

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

int main(void)
{
    int myvalue = 6;

    std::cout << "original value: " << myvalue << endl;
    myvalue = add(myvalue, 3);
    std::cout << "new value: " << myvalue << endl;
    while (1);
}
```

The program includes the header file `xc.h`, which provides definitions for all Special Function Registers (SFRs) on the target device and iostream header file, which provides the necessary prototypes for the peripheral library. For completion, the user must provide actual implementations of functions `_mon_getc` and `_mon_putc` for file IO. By default the XC32 compiler links do-nothing stubs for these functions.

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

```
xc32-g++ -mprocessor=ATSAME70Q21B -Wl,--defsym=_min_heap_size=0xF000
-o ex1.elf ex1.cpp
```

The option `-o ex1.elf` names the output executable file. This elf file may be imported into MPLAB X IDE.

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

```
xc32-bin2hex ex1.elf
```

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

5.3.2 Compiling Multiple C++ files

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

File 1

```
/* ex1.cpp */

// ATSAME70Q21B Configuration Bit Settings
#pragma config SECURITY_BIT = CLEAR
#pragma config BOOT_MODE = CLEAR
#pragma config TCM_CONFIGURATION = 0x0 // Enter Hexadecimal value
```

```
#include <xc.h>
#include <iostream>
using namespace std;

extern unsigned int add(unsigned int a, unsigned int b);

int main(void)
{
    int myvalue = 6;

    std::cout << "original value: " << myvalue << endl;
    myvalue = add(myvalue, 3);
    std::cout << "new value: " << myvalue << endl;
    while (1);
}
```

File 2

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

Compile both files by typing the following at the prompt:

```
xc32-g++ -mprocessor=ATSAME70Q21B -Wl,--defsym=_min_heap_size=0xF000
-o ex1.elf ex1.cpp add.cpp
```

The command compiles the modules `ex1.cpp` and `add.cpp`. The compiled modules are linked with the compiler libraries for C++ and the executable file `ex1.elf` is created.

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

5.4 RUNTIME FILES

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

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

5.4.1 Library Files

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

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

By default, the 32-bit language tools use the directory `<install-directory>/lib/gcc/pic32c/<gcc-version>` to store the startup libraries and the directory `<install-directory>/pic32c/lib` to store the target-specific libraries. Both of these paths contain subdirectories for each of the multilib combinations specified above.

The target libraries that are distributed with the compiler are built for the corresponding command-line options:

- Alignment (`-mno-unaligned-access`)
- Floating-point application binary interface (`-mfloat-abi softfp, hard`)

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

1.
   ```
   xc32-gcc foo.c
   xc32-g++ foo.cpp
   ```
   For this example, no command line options have been specified (i.e., the default command line options are being used). In this case, the default directories mentioned above are used.

2.
   ```
   xc32-gcc -mfloat-abi=softfp foo.c
   xc32-g++ -mfloat-abi=softfp foo.cpp
   ```
   For this example, `soft` multilib subdirectories are used.

3.
   ```
   xc32-gcc -mfloat-abi=hard foo.c
   xc32-g++ -mfloat-abi=hard foo.cpp
   ```
   For this example, the `hard` multilib subdirectories are used.

5.4.1.1 STANDARD LIBRARIES

The C/C++ standard libraries contain a standardized collection of functions, such as string, math and input/output routines. How to used these functions is described in Chapter 16. “Library Routines”.

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

5.4.1.2 USER-DEFINED LIBRARIES

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

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

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

5.4.2 Peripheral Library Functions

For All PIC32 devices, see 16.1 “Using Library Routines”.

5.5 START-UP AND INITIALIZATION

The C/C++ runtime startup code is device specific. The xc32-gcc and xc32-g++ compilation drivers select the appropriate startup code when linking using the `-mprocessor=`*`device`* option.

- The startup code initializes the L1 cache when available.
- It enables the DSPr2 engine when available.
- It also initializes the Translation Lookaside Buffer (TLB) of the Memory Management Unit (MMU) for the External Bus Interface (EBI) or Serial Quad Interface (SQI) when available. The device-specific linker script creates a table of TLB initialization values that the startup code uses to initialize the TLB at startup.

For C, C++:

The source code for the startup module, which initializes the runtime environment is platform specific and can be found in the `pic32c-libs.zip` file located at:

```
<install-directory>/pic32-libs/
```

Inside the zip archive, the source code can be found at:

```
pic32c-libs/proc/<DEVICENAME>/startup_<devicename>.c
```

The same files are found in: `<install-directory>/pic32c/lib/proc/<DEVICENAME>/startup_<devicename>.c`

Prebuilt startup object files are found in architecture specific directories in:

```
<install_directory>/lib/gcc/pic32c/<gcc-version>/
```

The object files for startup and runtime are named: `crti.o`, `crtn.o`, `crtbegin.o` and `crtend.o`. Other related libraries can be found there as well. Multilib versions of these modules exist in order to support architectural differences between device families.

For more information about what the code in these start-up modules actual does, see Section 15.2 "Runtime Start-up Code".

5.6 COMPILER OUTPUT

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

5.6.1 Output Files

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

TABLE 5-3: FILE NAMES

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

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

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

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

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

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

The default behavior of `xc32-gcc` and `xc32-g++` is to produce an ELF output. To make changes to the file's output or the file names, see Section 5.8 "Driver Option Descriptions".

5.6.2 Diagnostic Files

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

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

The option to create a listing file in the assembler is `-a` (or `-Wa,-a` if passed to the driver). There are many variants to this option, which may be found in the *"MPLAB® XC32 Assembler, Linker and Utilities User's Guide"* (DS50002186). To pass the option from the compiler, see Section 5.8.9 "Options for Assembling".

There is one list file produced for each build. There is one assembler listing file for each translation unit. This is a pre-link assembler listing so it will not show final addresses. Thus, if you require a list file for each source file, these files must be compiled separately, see Section 5.2.2 "Multi-step C Compilation". This is the case if you build using MPLAB X IDE. Each list file will be assigned the module name and extension `.lst`.

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

The linker option to create a map file is `-Map` *`file`* (or `-Wl,-Map=file`, if passed to the driver), which can be found in the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide* (DS50002186). To pass the option from the compiler driver, see Section 5.8.10 "Options for Linking".

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

5.7 COMPILER MESSAGES

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

- **Warning Messages** indicate source code or some other situation that can be compiled, but is unusual and may lead to a runtime failure of the code. The code or situation that triggered the warning should be investigated; however, compilation of the current module will continue, as will compilation of any remaining modules.
- **Error Messages** indicate source code that is illegal or that compilation of this code cannot take place. Compilation will be attempted for the remaining source code in the current module, but no additional modules will be compiled and the compilation process will then conclude.
- **Fatal Error Messages** indicate a situation that cannot allow compilation to proceed and which requires the compilation process to stop immediately.

For information on options that control compiler output of errors, warnings or comments, see Section 5.8.5 "Options for Controlling Warnings and Errors".

5.8 DRIVER OPTION DESCRIPTIONS

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

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

- Options Specific to PIC32C/SAM Devices
- Options for Controlling the Kind of Output
- Options for Controlling the C Dialect
- Options for Controlling the C++ Dialect
- Options for Controlling Warnings and Errors
- Options for Debugging
- Options for Controlling Optimization
- Options for Controlling the Preprocessor
- Options for Assembling
- Options for Linking
- Options for Directory Search
- Options for Code Generation Conventions

5.8.1 Options Specific to PIC32C/SAM Devices

These driver options are specific to the PIC32C/SAM devices, not the compiler.

TABLE 5-4: PIC32C/SAM DEVICE-SPECIFIC OPTIONS

Option	Definition
`-mprocessor=`*`device`*	Selects the device for which to compile. (e.g., `-mprocessor=ATSAME70N20B`)
`-mprint-builtins`	Print a list of enabled builtin functions.
`-msmart-io=[0|1|2]`	This option attempts to statically analyze format strings passed to `printf`, `scanf`, and the '`f`' and '`v`' variations of these functions. Uses of nonfloating-point format arguments will be converted to use an integer-only variation of the library function. For many applications, this feature can reduce program-memory usage. `-msmart-io=0` disables this option, while `-msmart-io=2` causes the compiler to be optimistic and convert function calls with variable or unknown format arguments. `-msmart-io=1` is the default and will convert only when the compiler can prove that floating-point support is not required.
`-mfloat-abi=`***`name`***	Specifies which floating-point ABI to use. Permissible values are: '`soft`', '`softfp`' and '`hard`'. Specifying '`soft`' causes XC32 to generate output containing library calls for floating-point operations. '`softfp`' allows the generation of code using hardware floating-point instructions, but still uses the soft-float calling conventions. '`hard`' allows generation of floating-point instructions and uses FPU-specific calling conventions. The default depends on the specific target configuration. Note that the hard-float and soft-float ABIs are not link-compatible; you must compile your entire program with the same ABI, and link with a compatible set of libraries.
`-mlong-calls` `-mno-long-calls`	Tells the compiler to perform function calls by first loading the address of the function into a register and then performing a subroutine call on this register. This switch is needed if the target function lies outside of the 64-megabyte addressing range of the offset-based version of subroutine call instruction. Even if this switch is enabled, not all function calls are turned into long calls. The heuristic is that static functions, functions that have the `short_call` attribute, functions that are inside the scope of a #`pragma no_long_calls` directive, and functions whose definitions have already been compiled within the current compilation unit are not turned into long calls. The exceptions to this rule are that weak function definitions, functions with the `long_call` attribute or the section attribute, and functions that are within the scope of a `#pragma long_calls` directive are always turned into long calls. This feature is not enabled by default. Specifying `-mno-long-calls` restores the default behavior, as does placing the function calls within the scope of a `#pragma long_calls_off` directive. Note these switches have no effect on how the compiler generates code to handle function calls via function pointers.
`-mthumb`	Generate code that executes in Thumb state.
`-munaligned-access` `-mno-unaligned-access`	Enables (or disables) reading and writing of 16- and 32-bit values from addresses that are not 16- or 32-bit aligned. By default, unaligned access is disabled for all pre-Arm v6 and all Arm v6-M architectures, and enabled for all other architectures. If unaligned access is not enabled then words in packed data structures are accessed a byte at a time.
`--nofallback`	Require an MPLAB XC32 Pro license and do not fall back to a lesser license.

5.8.2 Options for Controlling the Kind of Output

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

TABLE 5-5: KIND-OF-OUTPUT CONTROL OPTIONS

Option	Definition
`-c`	Compile or assemble the source files, but do not link. The default file extension is `.o`.
`-E`	Stop after the preprocessing stage (i.e., before running the compiler proper). The default output file is `stdout`.
`-fexceptions`	Enable exception handling. You may need to enable this option when compiling C code that needs to interoperate properly with exception handlers written in C++.
`-o file`	Place the output in *`file`*.
`-S`	Stop after compilation proper (i.e., before invoking the assembler). The default output file extension is `.s`.
`-v`	Print the commands executed during each stage of compilation.

`-x`	You can specify the input language explicitly with the `-x` option:
	`-x` *`language`*
	Specify explicitly the language for the following input files (rather than letting the compiler choose a default based on the file name suffix). This option applies to all following input files until the next `-x` option. The following values are supported by the compiler:
	`c`
	`c++`
	`c-header`
	`cpp-output`
	`assembler`
	`assembler-with-cpp`
	
	`-x none`
	Turn off any specification of a language, so that subsequent files are handled according to their file name suffixes. This is the default behavior but is needed if another `-x` option has been used. For example:
	`xc32-gcc -x assembler foo.asm bar.asm -x none main.c mabonga.s`
	
	Without the `-x none`, the compiler assumes all the input files are for the assembler.

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

5.8.3 Options for Controlling the C Dialect

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

TABLE 5-6: C DIALECT CONTROL OPTIONS

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

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

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

`-ffreestanding`	Assert that compilation takes place in a freestanding environment. This implies `-fno-builtin`. A freestanding environment is one in which the standard library may not exist, and program start-up may not necessarily be at main. The most obvious example is an OS kernel. This is equivalent to `-fno-hosted`.
`-fno-asm`	Do not recognize `asm`, `inline` or `typeof` as a keyword, so that code can use these words as identifiers. You can use the keywords `__asm__`, `__inline__` and `__typeof__` instead. `-ansi` implies `-fno-asm`.
`-fno-builtin` `-fno-builtin-function`	Don't recognize built-in functions that do not begin with `__builtin_` as prefix.
`-fno-exceptions`	Disable C++ exception handling. This option disables the generation of extra code needed to propagate exceptions.
`-fno-rtti`	Enable/Disable runtime type-identification features. The `-fno-rtti` option disables generation of information about every class with virtual functions for use by the C++ runtime type identification features ('`dynamic_cast`' and '`typeid`'). If you don't use those parts of the language, you can save some space by using this flag. Note that exception handling uses the same information, but it will generate it as needed. The '`dynamic_cast`' operator can still be used for casts that do not require runtime type information, i.e., casts to `void *` or to unambiguous base classes.
`-fsigned-char`	Let the type `char` be signed, like `signed char`.

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

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

5.8.4 Options for Controlling the C++ Dialect

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

TABLE 5-7: C++ DIALECT CONTROL OPTIONS

Option	Definition
`-ansi`	Support all (and only) ANSI-standard C++ programs.
`-aux-info filename`	Output to the given filename prototyped declarations for all functions declared and/or defined in a translation unit, including those in header files. This option is silently ignored in any language other than C++. Besides declarations, the file indicates, in comments, the origin of each declaration (source file and line), whether the declaration was implicit, prototyped or unprototyped (`I`, `N` for new or `O` for old, respectively, in the first character after the line number and the colon), and whether it came from a declaration or a definition (`C` or `F`, respectively, in the following character). In the case of function definitions, a K&R-style list of arguments followed by their declarations is also provided, inside comments, after the declaration.
`-ffreestanding`	Assert that compilation takes place in a freestanding environment. This implies `-fno-builtin`. A freestanding environment is one in which the standard library may not exist, and program start-up may not necessarily be at main. The most obvious example is an OS kernel. This is equivalent to `-fno-hosted`.
`-fno-asm`	Do not recognize `asm`, `inline` or `typeof` as a keyword, so that code can use these words as identifiers. You can use the keywords `__asm__`, `__inline__` and `__typeof__` instead. `-ansi` implies `-fno-asm`.
`-fno-builtin` `-fno-builtin-function`	Don’t recognize built-in functions that do not begin with `__builtin_` as prefix.
`-fsigned-char`	Let the type `char` be signed, like `signed char`.

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

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

5.8.5 Options for Controlling Warnings and Errors

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

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

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

TABLE 5-8: WARNING AND ERROR OPTIONS IMPLIED BY ALL WARNINGS

Option	Definition
`-fsyntax-only`	Check the code for syntax, but don’t do anything beyond that.
`-pedantic`	Issue all the warnings demanded by strict ANSI C. Reject all programs that use extensions.
`-pedantic-errors`	Like `-pedantic`, except that errors are produced rather than warnings.
`-w`	Inhibit all warning messages.

TABLE 5-8: WARNING AND ERROR OPTIONS IMPLIED BY ALL WARNINGS (CONTINUED)

Option	Definition
`-Wall`	This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros. Note that some warning flags are not implied by `-Wall`. Some of them warn about constructions that users generally do not consider questionable, but which occasionally you might wish to check for; others warn about constructions that are necessary or hard to avoid in some cases, and there is no simple way to modify the code to suppress the warning. Some of them are enabled by `-Wextra` but many of them must be enabled individually.
`-Waddress`	Warn about suspicious uses of memory addresses. These include using the address of a function in a conditional expression, such as `void func(void);` if `(func)`, and comparisons against the memory address of a string literal, such as if `(x == "abc")`. Such uses typically indicate a programmer error: the address of a function always evaluates to true, so their use in a conditional usually indicates that the programmer forgot the parentheses in a function call; and comparisons against string literals result in unspecified behavior and are not portable in C, so they usually indicate that the programmer intended to use `strcmp`.
`-Wchar-subscripts`	Warn if an array subscript has type `char`.
`-Wcomment`	Warn whenever a comment-start sequence `/*` appears in a `/*` comment, or whenever a Backslash-Newline appears in a `//` comment.
`-Wdiv-by-zero`	Warn about compile-time integer division by zero. To inhibit the warning messages, use `-Wno-div-by-zero`. Floating-point division by zero is not warned about, as it can be a legitimate way of obtaining infinities and NaNs. **(This is the default.)**
`-Wformat`	Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified.
`-Wimplicit`	Equivalent to specifying both `-Wimplicit-int` and `-Wimplicit-function-declaration`.
`-Wimplicit-function-declaration`	Give a warning whenever a function is used before being declared.
`-Wimplicit-int`	Warn when a declaration does not specify a type.
`-Wmain`	Warn if the type of `main` is suspicious. `main` should be a function with external linkage, returning `int`, taking either zero, two or three arguments of appropriate types.

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

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

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

TABLE 5-8: WARNING AND ERROR OPTIONS IMPLIED BY ALL WARNINGS (CONTINUED)

Option	Definition
`-Wsequence-point`	Warn about code that may have undefined semantics because of violations of sequence point rules in the C standard. The C standard defines the order in which expressions in a C program are evaluated in terms of sequence points, which represent a partial ordering between the execution of parts of the program: those executed before the sequence point and those executed after it. These occur after the evaluation of a full expression (one which is not part of a larger expression), after the evaluation of the first operand of a `&&`, `||`, `? :` or `,` (comma) operator, before a function is called (but after the evaluation of its arguments and the expression denoting the called function), and in certain other places. Other than as expressed by the sequence point rules, the order of evaluation of subexpressions of an expression is not specified. All these rules describe only a partial order rather than a total order, since, for example, if two functions are called within one expression with no sequence point between them, the order in which the functions are called is not specified. However, the standards committee has ruled that function calls do not overlap. It is not specified when between sequence points modifications to the values of objects take effect. Programs whose behavior depends on this have undefined behavior. The C standard specifies that "Between the previous and next sequence point, an object shall have its stored value modified, at most once, by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored." If a program breaks these rules, the results on any particular implementation are entirely unpredictable. Examples of code with undefined behavior are `a = a++;`, `a[n] = b[n++]` and `a[i++] = i;`. Some more complicated cases are not diagnosed by this option, and it may give an occasional false positive result, but in general it has been found fairly effective at detecting this sort of problem in programs.
`-Wswitch`	Warn whenever a `switch` statement has an index of enumeral type and lacks a case for one or more of the named codes of that enumeration. (The presence of a default label prevents this warning.) `case` labels outside the enumeration range also provoke warnings when this option is used.
`-Wsystem-headers`	Print warning messages for constructs found in system header files. Warnings from system headers are normally suppressed on the assumption that they usually do not indicate real problems and would only make the compiler output harder to read. Using this command line option tells the compiler to emit warnings from system headers as if they occurred in user code. However, note that using `-Wall` in conjunction with this option does not warn about unknown pragmas in system headers. For that, `-Wunknown-pragmas` must also be used.
`-Wtrigraphs`	Warn if any trigraphs are encountered (assuming they are enabled).
`-Wuninitialized`	Warn if an automatic variable is used without first being initialized. These warnings are possible only when optimization is enabled, because they require data flow information that is computed only when optimizing. These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared `volatile`, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers. Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.
`-Wunknown-pragmas`	Warn when a `#pragma` directive is encountered which is not understood by the compiler. If this command line option is used, warnings will even be issued for unknown pragmas in system header files. This is not the case if the warnings were only enabled by the `-Wall` command line option.

TABLE 5-8: WARNING AND ERROR OPTIONS IMPLIED BY ALL WARNINGS (CONTINUED)

Option	Definition
`-Wunused`	Warn whenever a variable is unused aside from its declaration, whenever a function is declared static but never defined, whenever a label is declared but not used, and whenever a statement computes a result that is explicitly not used. In order to get a warning about an unused function parameter, both `-W` and `-Wunused` must be specified. Casting an expression to void suppresses this warning for an expression. Similarly, the `unused` attribute suppresses this warning for unused variables, parameters and labels.
`-Wunused-function`	Warn whenever a static function is declared but not defined or a non-inline static function is unused.
`-Wunused-label`	Warn whenever a label is declared but not used. To suppress this warning, use the `unused` attribute.
`-Wunused-parameter`	Warn whenever a function parameter is unused aside from its declaration. To suppress this warning, use the `unused` attribute.
`-Wunused-variable`	Warn whenever a local variable or non-constant static variable is unused aside from its declaration. To suppress this warning, use the `unused` attribute.
`-Wunused-value`	Warn whenever a statement computes a result that is explicitly not used. To suppress this warning, cast the expression to void.

The following `-W` options are not implied by `-Wall`. Some of them warn about constructions that users generally do not consider questionable, but which you might occasionally wish to check for. Others warn about constructions that are necessary or hard to avoid in some cases, and there is no simple way to modify the code to suppress the warning.

TABLE 5-9: WARNING AND ERROR OPTIONS NOT IMPLIED BY ALL WARNINGS

Option	Definition

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

`-Waggregate-return`	Warn if any functions that return structures or unions are defined or called.
`-Wbad-function-cast`	Warn whenever a function call is cast to a non-matching type. For example, warn if `int foof()` is cast to anything `*`.
`-Wcast-align`	Warn whenever a pointer is cast, such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *`.
`-Wcast-qual`	Warn whenever a pointer is cast, so as to remove a type qualifier from the target type. For example, warn if a `const char *` is cast to an ordinary `char *`.
`-Wconversion`	Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, as well as conversions that change the width or signedness of a fixed point argument, except when the same as the default promotion. Also, warn if a negative integer constant expression is implicitly converted to an unsigned type. For example, warn about the assignment `x = -1` if `x` is unsigned. But do not warn about explicit casts like `(unsigned) -1`.
`-Werror`	Make all warnings into errors.

TABLE 5-9: WARNING AND ERROR OPTIONS NOT IMPLIED BY ALL WARNINGS (CONTINUED)

Option	Definition
`-Winline`	Warn if a function can not be inlined and it was either declared as inline, or else the `-finline-functions` option was given.
`-Wlarger-than-len`	Warn whenever an object of larger than `len` bytes is defined.
`-Wlong-long` `-Wno-long-long`	Warn if `long long` type is used. This is default. To inhibit the warning messages, use `-Wno-long-long`. Flags `-Wlong-long` and `-Wno-long-long` are taken into account only when `-pedantic` flag is used.
`-Wmissing-declarations`	Warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a prototype.
`-Wmissing-format-attribute`	If `-Wformat` is enabled, also warn about functions that might be candidates for format attributes. Note these are only possible candidates, not absolute ones. This option has no effect unless `-Wformat` is enabled.
`-Wmissing-noreturn`	Warn about functions that might be candidates for attribute `noreturn`. These are only possible candidates, not absolute ones. Care should be taken to manually verify functions. In fact, do not ever return before adding the `noreturn` attribute, otherwise subtle code generation bugs could be introduced.
`-Wmissing-prototypes`	Warn if a global function is defined without a previous prototype declaration. This warning is issued even if the definition itself provides a prototype. (This option can be used to detect global functions that are not declared in header files.)
`-Wnested-externs`	Warn if an `extern` declaration is encountered within a function.
`-Wno-deprecated-declarations`	Do not warn about uses of functions, variables and types marked as deprecated by using the `deprecated` attribute.
`-Wpadded`	Warn if padding is included in a structure, either to align an element of the structure or to align the whole structure.
`-Wpointer-arith`	Warn about anything that depends on the size of a function type or of `void`. The compiler assigns these types a size of 1, for convenience in calculations with `void *` pointers and pointers to functions.
`-Wredundant-decls`	Warn if anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.
`-Wshadow`	Warn whenever a local variable shadows another local variable.
`-Wsign-compare` `-Wno-sign-compare`	Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. This warning is also enabled by `-W`. To get the other warnings of `-W` without this warning, use `-W -Wno-sign-compare`.
`-Wstrict-prototypes`	Warn if a function is declared or defined without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration which specifies the argument types.)
`-Wtraditional`	Warn about certain constructs that behave differently in traditional and ANSI C. • Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C. • A function declared external in one block and then used after the end of the block. • A switch statement has an operand of type `long`. • A nonstatic function declaration follows a static one. This construct is not accepted by some traditional C compilers.
`-Wundef`	Warn if an undefined identifier is evaluated in an `#if` directive.

TABLE 5-9: WARNING AND ERROR OPTIONS NOT IMPLIED BY ALL WARNINGS (CONTINUED)

Option	Definition
`-Wunreachable-code`	Warn if the compiler detects that code will never be executed. It is possible for this option to produce a warning even though there are circumstances under which part of the affected line can be executed, so care should be taken when removing apparently unreachable code. For instance, when a function is inlined, a warning may mean that the line is unreachable in only one inlined copy of the function.
`-Wwrite-strings`	Give string constants the type `const char[length]` so that copying the address of one into a non-const `char *` pointer gets a warning. At compile time, these warnings help you find code that you can try to write into a string constant, but only if you have been very careful about using `const` in declarations and prototypes. Otherwise, it's just a nuisance, which is why `-Wall` does not request these warnings.

5.8.6 Options for Debugging

The following options are used for debugging.

TABLE 5-10: DEBUGGING OPTIONS

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

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

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

5.8.7 Options for Controlling Optimization

The following options control compiler optimizations.

TABLE 5-11: GENERAL OPTIMIZATION OPTIONS

Option	Edition	Definition
`-O0`	All	Do not optimize. **(This is the default.)** Without `-O`, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code. The compiler only allocates variables declared `register` in registers.
`-O` `-O1`	All	Optimization level 1. Optimizing compilation takes somewhat longer and a lot more host memory for a large function. With `-O`, the compiler tries to reduce code size and execution time. When `-O` is specified, the compiler turns on `-fthread-jumps` and `-fdefer-pop`. The compiler turns on `-fomit-frame-pointer`.
`-O2`	PRO	Optimization level 2. The compiler performs nearly all supported optimizations that do not involve a space-speed trade-off. `-O2` turns on all optional optimizations except for loop unrolling (`-funroll-loops`), function inlining (`-finline-functions`), and strict aliasing optimizations (`-fstrict-aliasing`). It also turns on force copy of memory operands (`-fforce-mem`) and Frame Pointer elimination (`-fomit-frame-pointer`). As compared to `-O`, this option increases both compilation time and the performance of the generated code.
`-O3`	PRO	Optimization level 3. `-O3` turns on all optimizations specified by `-O2` and also turns on the `inline-functions` option.
`-Os`	PRO	Optimize for size. `-Os` enables all `-O2` optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

`-flto`	PRO	This option runs the standard link-time optimizer. When invoked with source code, the compiler adds an internal bytecode representation of the code to special sections in the object file. When the object files are linked together, all the function bodies are read from these sections and instantiated as if they had been part of the same translation unit.
		To use the link-timer optimizer, specify `-flto` both at compile time and during the final link. For example
		`xc32-gcc -c -O1 -flto -mprocessor=ATSAME70Q21B foo.c`
		`xc32-gcc -c -O1 -flto -mprocessor=ATSAME70Q21B bar.c`
		`xc32-gcc -o myprog.elf -flto -O3 -mprocessor=ATSAME70Q21B foo.o bar.o`
		
		Another (simpler) way to enable link-time optimization is,
		`xc32-gcc -o myprog.elf -flto -O3 -mprocessor=ATSAME70Q21B foo.c bar.c`
		
		Link time optimizations do not require the presence of the whole program to operate. If the program does not require any symbols to be exported, it is possible to combine `-flto` with `-fwhole-program` to allow the interprocedural optimizers to use more aggressive assumptions which may lead to improved optimization opportunities.
		Regarding portability: The bytecode files are versioned and there is a strict version check, so bytecode files generated in one version of XC32 may not work with an older/newer version of the XC32 compiler.

TABLE 5-11: GENERAL OPTIMIZATION OPTIONS (CONTINUED)

Option	Edition	Definition
`-ftoplevel-reorder`	All	Allow reordering top-level functions, variables, and asm statements. They may not be output in the same order that they appear in the input file. This option also allows removal of unreferenced static variables. Use this options with optimization level `-O1` or greater.
`-fwhole-program`	PRO	Assume that the current compilation unit represents the whole program being compiled. All public functions and variables, with the exception of main and those merged by attribute `externally_visible`, become static functions and in effect are optimized more aggressively by interprocedural optimizers. While this option is equivalent to proper use of the static keyword for programs consisting of a single file, in combination with option `-flto`, this flag can be used to compile many smaller scale programs since the functions and variables become local for the whole combined compilation unit, not for the single source file itself.

The following options control specific optimizations. The `-O2` option turns on all of these optimizations except `-funroll-loops`, `-funroll-all-loops` and `-fstrict-aliasing`.

You can use the following flags in the rare cases when "fine-tuning" of optimizations to be performed is desired.

TABLE 5-12: SPECIFIC OPTIMIZATION OPTIONS

Option	Definition

`-falign-functions`
`-falign-functions=n`
	Align the start of functions to the next power-of-two greater than `n`, skipping up to `n` bytes. For instance, `-falign-functions=32` aligns functions to the next 32-byte boundary, but `-falign-functions=24` would align to the next 32-byte boundary only if this can be done by skipping 23 bytes or less.
	`-fno-align-functions` and `-falign-functions=1` are equivalent and mean that functions are not aligned.
	The assembler only supports this flag when `n` is a power of two, so `n` is rounded up. If `n` is not specified, use a machine-dependent default.

`-falign-labels`
`-falign-labels=n`
	Align all branch targets to a power-of-two boundary, skipping up to `n` bytes like `-falign-functions`. This option can easily make code slower, because it must insert dummy operations for when the branch target is reached in the usual flow of the code.
	If `-falign-loops` or `-falign-jumps` are applicable and are greater than this value, then their values are used instead.
	If `n` is not specified, use a machine-dependent default which is very likely to be 1, meaning no alignment.

`-falign-loops`
`-falign-loops=n`
	Align loops to a power-of-two boundary, skipping up to `n` bytes like `-falign-functions`. The hope is that the loop is executed many times, which makes up for any execution of the dummy operations.
	If `n` is not specified, use a machine-dependent default.

`-fcaller-saves`	Enable values to be allocated in registers that are clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.
`-fcse-follow-jumps`	In common subexpression elimination, scan through jump instructions when the target of the jump is not reached by any other path. For example, when CSE encounters an `if` statement with an `else` clause, CSE follows the jump when the condition tested is false.
`-fcse-skip-blocks`	This is similar to `-fcse-follow-jumps`, but causes CSE to follow jumps which conditionally skip over blocks. When CSE encounters a simple `if` statement with no `else` clause, `-fcse-skip-blocks` causes CSE to follow the jump around the body of the `if`.
`-fexpensive-optimizations`	Perform a number of minor optimizations that are relatively expensive.

TABLE 5-12: SPECIFIC OPTIMIZATION OPTIONS (CONTINUED)

Option	Definition
`-ffunction-sections` `-fdata-sections`	Place each function or data item into its own section in the output file. The name of the function or the name of the data item determines the section's name in the output file. Use these options when there are significant benefits for doing so. When you specify these options, the assembler and linker may create larger object and executable files and is also slower.
`-fgcse`	Perform a global common subexpression elimination pass. This pass also performs global constant and copy propagation.
`-fgcse-lm`	When `-fgcse-lm` is enabled, global common subexpression elimination attempts to move loads which are only killed by stores into themselves. This allows a loop containing a load/store sequence to change to a load outside the loop, and a copy/store within the loop.
`-fgcse-sm`	When `-fgcse-sm` is enabled, a store motion pass is run after global common subexpression elimination. This pass attempts to move stores out of loops. When used in conjunction with `-fgcse-lm`, loops containing a load/store sequence can change to a load before the loop and a store after the loop.
`-fmove-all-movables`	Forces all invariant computations in loops to be moved outside the loop.
`-fno-defer-pop`	Always pop the arguments to each function call as soon as that function returns. The compiler normally lets arguments accumulate on the stack for several function calls and pops them all at once.
`-fno-peephole` `-fno-peephole2`	Disable machine specific peephole optimizations. Peephole optimizations occur at various points during the compilation. `-fno-peephole` disables peephole optimization on machine instructions, while `-fno-peephole2` disables high level peephole optimizations. To disable peephole entirely, use both options.
`-foptimize-register-move` `-fregmove`	Attempt to reassign register numbers in move instructions and as operands of other simple instructions in order to maximize the amount of register tying. `-fregmove` and `-foptimize-register-moves` are the same optimization.
`-freduce-all-givs`	Forces all general-induction variables in loops to be strength reduced. These options may generate better or worse code. Results are highly dependent on the structure of loops within the source code.
`-frename-registers`	Attempt to avoid false dependencies in scheduled code by making use of registers left over after register allocation. This optimization most benefits processors with lots of registers. It can, however, make debugging impossible, since variables no longer stay in a “home register”.
`-frerun-cse-after-loop`	Rerun common subexpression elimination after loop optimizations has been performed.
`-frerun-loop-opt`	Run the loop optimizer twice.
`-fschedule-insns`	Attempt to reorder instructions to eliminate instruction stalls due to required data being unavailable.
`-fschedule-insns2`	Similar to `-fschedule-insns`, but requests an additional pass of instruction scheduling after register allocation has been done.
`-fstrength-reduce`	Perform the optimizations of loop strength reduction and elimination of iteration variables.

TABLE 5-12: SPECIFIC OPTIMIZATION OPTIONS (CONTINUED)

Option	Definition

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

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

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

TABLE 5-13: MACHINE-INDEPENDENT OPTIMIZATION OPTIONS

Option	Definition
`-fforce-mem`	Force memory operands to be copied into registers before doing arithmetic on them. This produces better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register load. The `-O2` option turns on this option.
`-finline-functions`	Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated, and the function is declared `static`, then the function is normally not output as assembler code in its own right.

TABLE 5-13: MACHINE-INDEPENDENT OPTIMIZATION OPTIONS (CONTINUED)

Option	Definition
`-finline-limit=n`	By default, the compiler limits the size of functions that can be inlined. This flag allows the control of this limit for functions that are explicitly marked as inline (i.e., marked with the `inline` keyword). *n* is the size of functions that can be inlined in number of pseudo instructions (not counting parameter handling). The default value of `n` is 10000. Increasing this value can result in more inlined code at the cost of compilation time and memory consumption. Decreasing usually makes the compilation faster and less code is inlined (which presumably means slower programs). This option is particularly useful for programs that use inlining. **Note:** Pseudo instruction represents, in this particular context, an abstract measurement of function's size. In no way does it represent a count of assembly instructions and as such, its exact meaning might change from one release of the compiler to an another.
`-fkeep-inline-functions`	Even if all calls to a given function are integrated, and the function is declared `static`, output a separate run time callable version of the function. This switch does not affect `extern` inline functions.
`-fkeep-static-consts`	Emit variables are declared static const when optimization isn't turned on, even if the variables are not referenced. The compiler enables this option by default. If you want to force the compiler to check if the variable was referenced, regardless of whether or not optimization is turned on, use the `-fno-keep-static-consts` option.
`-fno-function-cse`	Do not put function addresses in registers. Make each instruction that calls a constant function contain the function's address explicitly. This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.
`-fno-inline`	Do not pay attention to the `inline` keyword. Normally this option is used to keep the compiler from expanding any functions inline. If optimization is not enabled, no functions can be expanded inline.
`-fomit-frame-pointer`	Do not keep the Frame Pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore Frame Pointers. It also makes an extra register available in many functions.
`-foptimize-sibling-calls`	Optimize sibling and tail recursive calls.

5.8.8 Options for Controlling the Preprocessor

The following options control the compiler preprocessor.

TABLE 5-14: PREPROCESSOR OPTIONS

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

TABLE 5-14: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
`-I-`	Any directories you specify with `-I` options before the `-I-` options are searched only for the case of `#include "file"`. They are not searched for `#include <file>`. If additional directories are specified with `-I` options after the `-I-`, these directories are searched for all `#include` directives. (Ordinarily all `-I` directories are used this way.) In addition, the `-I-` option inhibits the use of the current directory (where the current input file came from) as the first search directory for `#include "file`." There is no way to override this effect of `-I-`. With `-I.` you can specify searching the directory that was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory. `-I-` does not inhibit the use of the standard system directories for header files. Thus, `-I-` and `-nostdinc` are independent. **NOTE:** Do not specify an MPLAB XC32 system include directory (e.g., /pic32c/include/) in your project properties. The xc32-gcc and xc32-g++ compilation drivers automatically select the default C libc or the C++ libc and their respective include-file directory for you. Manually adding a system include file path may disrupt this mechanism and cause the incorrect libc include files to be compiled into your project, causing a conflict between the include files and the library. Note that adding a system include path to your project properties has never been a recommended practice.
`-Idir`	Add the directory *dir* to the head of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one `-I` option, the directories are scanned in left-to-right order. The standard system directories come after.
`-idirafter dir`	Add the directory `dir` to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that `-I` adds to).
`-imacros file`	Process file as input, discarding the resulting output, before processing the regular input file. Because the output generated from the file is discarded, the only effect of `-imacros` *file* is to make the macros defined in file available for use in the main input. Any `-D` and `-U` options on the command line are always processed before `-imacros` *file*, regardless of the order in which they are written. All the `-include` and `-imacros` options are processed in the order in which they are written.
`-include file`	Process file as input before processing the regular input file. In effect, the contents of file are compiled first. Any `-D` and `-U` options on the command line are always processed before `-include` *file*, regardless of the order in which they are written. All the `-include` and `-imacros` options are processed in the order in which they are written.
`-M`	Tell the preprocessor to output a rule suitable for `make` describing the dependencies of each object file. For each source file, the preprocessor outputs one make-rule whose target is the object file name for that source file and whose dependencies are all the `#include` header files it uses. This rule may be a single line or may be continued with `\-newline` if it is long. The list of rules is printed on standard output instead of the preprocessed C program. `-M` implies `-E` (see Section 5.8.2 "Options for Controlling the Kind of Output").
`-MD`	Like `-M` but the dependency information is written to a file and compilation continues. The file containing the dependency information is given the same name as the source file with a `.d` extension.
`-MF file`	When used with `-M` or `-MM`, specifies a file in which to write the dependencies. If no `-MF` switch is given, the preprocessor sends the rules to the same place it would have sent preprocessed output. When used with the driver options, `-MD` or `-MMD`, `-MF,` overrides the default dependency output file.
`-MG`	Treat missing header files as generated files and assume they live in the same directory as the source file. If `-MG` is specified, then either `-M` or `-MM` must also be specified. `-MG` is not supported with `-MD` or `-MMD`.
`-MM`	Like `-M` but the output mentions only the user header files included with `#include "file"`. System header files included with `#include <file>` are omitted.

TABLE 5-14: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
`-MMD`	Like `-MD` except mention only user header files, not system header files.

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

`-MQ`	Same as `-MT`, but it quotes any characters which are special to `make`. `-MQ '$(objpfx)foo.o'` gives `$$(objpfx)foo.o: foo.c` The default target is automatically quoted, as if it were given with `-MQ`.
`-MT target`	Change the target of the rule emitted by dependency generation. By default, CPP takes the name of the main input file, including any path, deletes any file suffix such as `.c`, and appends the platform's usual object suffix. The result is the target. An `-MT` option sets the target to be exactly the string you specify. If you want multiple targets, you can specify them as a single argument to `-MT`, or use multiple `-MT` options. For example: `-MT '$(objpfx)foo.o'` might give `$(objpfx)foo.o: foo.c`
`-nostdinc`	Do not search the standard system directories for header files. Only the directories you have specified with `-I` options (and the current directory, if appropriate) are searched. See Section 5.8.11 "Options for Directory Search" for information on `-I`. By using both `-nostdinc` and `-I-`, the include-file search path can be limited to only those directories explicitly specified.
`-P`	Tell the preprocessor not to generate `#line` directives. Used with the `-E` option (see Section 5.8.2 "Options for Controlling the Kind of Output").
`-trigraphs`	Support ANSI C trigraphs. The `-ansi` option also has this effect.
`-Umacro`	Undefine macro *`macro`*. `-U` options are evaluated after all `-D` options, but before any `-include` and `-imacros` options.
`-undef`	Do not predefine any nonstandard macros (including architecture flags).

5.8.9 Options for Assembling

The following options control assembler operations.

TABLE 5-15: ASSEMBLY OPTIONS

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

5.8.10 Options for Linking

If any of the options `-c`, `-S` or `-E` are used, the linker is not run and object file names should not be used as arguments.

TABLE 5-16: LINKING OPTIONS

Option	Definition
`-Ldir`	Add directory *`dir`* to the list of directories to be searched for libraries specified by the command line option `-l`.

TABLE 5-16: LINKING OPTIONS (CONTINUED)

Option	Definition

`-llibrary`	Search the library named *`library`* when linking.
	The linker searches a standard list of directories for the library, which is actually a file named `lib`*`library`*`.a`. The linker then uses this file as if it had been specified precisely by name.
	It makes a difference where in the command you write this option. The linker processes libraries and object files in the order they are specified. Thus, `foo.o -lz bar.o` searches library `z` after file `foo.o` but before `bar.o`. If `bar.o` refers to functions in `libz.a`, those functions may not be loaded.
	The directories searched include several standard system directories, plus any that you specify with `-L`.
	Normally the files found this way are library files (archive files whose members are object files). The linker handles an archive file by scanning through it for members which define symbols that have been referenced but not defined yet. But if the file found is an ordinary object file, it is linked in the usual fashion. The only difference between using an `-l` option (e.g., `-lmylib`) and specifying a file name (e.g., `libmylib.a`) is that `-l` searches several directories, as specified.
	By default the linker is directed to search:
	`<install-path>\lib`
	for libraries specified with the `-l` option. On Microsoft Windows, for a compiler installed into the default location, this would be:
	`Program Files(x86)\Microchip\xc32\<version>\lib`
	This behavior can be overridden using the environment variables.
	See also the `INPUT` and `OPTIONAL` linker script directives.

`-nodefaultlibs`	Do not use the standard system libraries when linking. Only the libraries you specify are passed to the linker. The compiler may generate calls to `memcmp`, `memset` and `memcpy`. These entries are usually resolved by entries in the standard compiler libraries. These entry points should be supplied through some other mechanism when this option is specified.
`-nostdlib`	Do not use the standard system start-up files or libraries when linking. No start-up files and only the libraries you specify are passed to the linker. The compiler may generate calls to `memcmp`, `memset` and `memcpy`. These entries are usually resolved by entries in standard compiler libraries. These entry points should be supplied through some other mechanism when this option is specified.
`-s`	Remove all symbol table and relocation information from the executable.
`-u symbol`	Pretend *`symbol`* is undefined to force linking of library modules to define the symbol. It is legitimate to use `-u` multiple times with different symbols to force loading of additional library modules.
`-Wl,option`	Pass *`option`* as an option to the linker. If *`option`* contains commas, it is split into multiple options at the commas.
`-Xlinker option`	Pass *`option`* as an option to the linker. You can use this to supply system-specific linker options that the compiler does not know how to recognize.

5.8.11 Options for Directory Search

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

TABLE 5-17: DIRECTORY SEARCH OPTIONS

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

5.8.12 Options for Code Generation Conventions

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

TABLE 5-18: CODE GENERATION CONVENTION OPTIONS

Option	Definition

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

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

TABLE 5-18: CODE GENERATION CONVENTION OPTIONS (CONTINUED)

Option	Definition
`-fno-ident`	Ignore the `#ident` directive.
`-fpack-struct`	Pack all structure members together without holes. Usually you would not want to use this option, since it makes the code sub-optimal, and the offsets of structure members won’t agree with system libraries.
`-fpcc-struct-return`	Return short `struct` and `union` values in memory like longer ones, rather than in registers. This convention is less efficient, but it has the advantage of allowing capability between 32-bit compiled files and files compiled with other compilers. Short structures and unions are those whose size and alignment match that of an integer type.
`-fshort-enums`	Allocate to an `enum` type only as many bytes as it needs for the declared range of possible values. Specifically, the `enum` type is equivalent to the smallest integer type that has enough room.
`-fverbose-asm` `-fno-verbose-asm`	Put extra commentary information in the generated assembly code to make it more readable. `-fno-verbose-asm`, the default, causes the extra information to be omitted and is useful when comparing two assembler files.
`-fvolatile`	Consider all memory references through pointers to be volatile.
`-fvolatile-global`	Consider all memory references to external and global data items to be volatile. The use of this switch has no effect on static data.
`-fvolatile-static`	Consider all memory references to static data to be volatile.

Chapter 6. ANSI C Standard Issues

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

6.1 DIVERGENCE FROM THE ANSI C STANDARD

There are no divergences from the ANSI C standard.

6.2 EXTENSIONS TO THE ANSI C STANDARD

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

6.2.1 Keyword Differences

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

- Specifying Attributes of Variables – Section 8.11 "Variable Attributes"
- Specifying Attributes of Functions – Section 13.2 "Function Attributes and Specifiers"
- Inline Functions – Section 13.9 "Inline Functions"
- Variables in Specified Registers – Section 8.11 "Variable Attributes"
- Complex Numbers – Section 8.7 "Complex Data Types"
- Double-Word Integers – Section 8.3 "Integer Data Types"
- Referring to a Type with `typeof` – Section 8.9 "Standard Type Qualifiers"

6.2.2 Statement Differences

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

- Labels as Values – Section 10.3 "Labels as Values"
- Conditionals with Omitted Operands – Section 10.4 "Conditional Operator Operands"
- Case Ranges – Section 10.5 "Case Ranges"

6.2.3 Expression Differences

Expression differences are:

Binary constants – Section 8.8 "Constant Types and Formats".

6.3 IMPLEMENTATION-DEFINED BEHAVIOR

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

Chapter 7. Device-Related Features

The MPLAB XC32 C/C++ Compiler supports a number of special features and extensions to the C/C++ language which are designed to ease the task of producing PIC32C/SAM devices. This chapter documents the special language features which are specific to these devices.

7.1 DEVICE SUPPORT

MPLAB XC32 C/C++ Compiler aims to support all PIC32 devices. However, new devices in these families are frequently released.

7.2 DEVICE HEADER FILES

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

Inclusion of this file provides access to SFRs for the core and peripheral modules. See Section 7.5 "Using SFRs From C Code" for further information.

7.3 STACK

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

The PIC32 devices use a dedicated stack pointer register `sp` (register r14) for use as a software Stack Pointer. All processor stack operations, including function calls, interrupts and exceptions, use the software stack. It points to the next free location on the stack. The stack grows downward, towards lower memory addresses.

By default, the size of the stack is 1024 bytes. The size of the stack can be specified by defining the `_min_stack_size` symbol to the desired size in bytes using the `--defsym` linker option. An example of allocating a stack of 2048 bytes using the command line is:

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

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

- Register r14 (`sp`) – This is the Stack Pointer. It points to the next free location on the stack.
- Register r11 (`fp`) – This is the Frame Pointer. It points to the current function's frame.

No stack overflow detection is supplied.

The C/C++ run-time start-up module initializes the stack pointer during the start-up and initialization sequence (see Section 15.2.1 "Initialize Stack Pointer and Heap").

7.4 CONFIGURATION BIT ACCESS

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

The `#pragma config` directive specifies the processor-specific configuration settings (i.e., Configuration bits) to be used by the application. Refer to the "PIC32 Configuration Settings" online help (found under *MPLAB X IDE>Help>Help Contents>XC32 Toolchain*) for more information. (If using the compiler from the command line, this help file is located at the default location at: `Program Files/Microchip/<install-dir>/<version>/docs/PIC32ConfigSet.html.`)

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

- Syntax
- Example

7.4.1 Syntax

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

pragma-config-directive:

# pragma config *setting-list*

setting-list:
setting
| *setting-list, setting*
setting:
setting-name = *value-name*

The setting-name and value-name are device specific and can be determined by using the PIC32ConfigSet.html document located in the installation directory, docs folder.

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

Integer values for config pragmas can be set using the `config` pragma. (Examples: `#pragma config USERID = 0x1234u`)

7.4.2 Example

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

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

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

7.5 USING SFRS FROM C CODE

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

The SFRs may be read or modified using a C language interface. The SFR interface definitions are accessible by including the `<xc.h>` header file in your source code (see Section 7.2 "Device Header Files").

The names used in the C interface for SFRs and SFR fields/bits are based on the names specified in the device data sheet. Each peripheral component's registers are accessed through a fixed-base address defined as, for example, `WDT_BASE_ADDRESS`, which is the address of a structure containing all memory-mapped registers for the component, for example `WDT_CR`.

Multiple ways are provided to work with fields or bits of each SFR. The register field in the component structure can reference the complete register value, or individual bits or fields of the register by name. Macro definitions are also provided to allow accessing individual fields using bit operations. Section 7.5.1 "SFR Register Definitions" provides more information on the SFR interface.

The `<xc.h>` header will include a device-specific header file for the device you are using. These device-specific headers are located in the `pic32c/include/proc` directory of the compiler under a directory with a name that represents the device, e.g., `pic32c/include/proc/32CZ2038CA70144`.

To check the interface(s) for SFRs for the device you are using, inspect the headers in the `component` subdirectory of the device-specific header directory. These headers are automatically included by the device-specific header file and named based on the component names in the device data sheet, e.g., `wdt.h`. Remember that no device-specific headers need to be included directly into your source code – including the `<xc.h>` header will ensure all required headers are included.

In additional to peripheral modules, SFRs controlling aspects of the MCU are also accessible through the same interface. Core SFRs are defined in header files automatically included by all device-specific header files, so including `<xc.h>` is sufficient to allow access to all core SFRs. Core-specific header files are located in the `pic32c/include` directory.

7.5.1 SFR Register Definitions

In this section we describe the conventions of the SFR interface and use the WDT (watchdog timer) component as an example. Always consult the device data sheet and device-specific header files to confirm the specific capabilities and names for your device.

The SFRs within each component are accessed through a base pointer, defined in the header as a macro, for example, `_WDT_REGS`. This is used as a pointer to a structure of type `_wdt_registers_t` containing all SFRs of that component by name, for example, `_WDT_REGS->WDT_CR`. The fields of this structure are typed to reflect whether each register is read-only, write-only, or read-write; however, the device data sheet should always be consulted for details on the read/write properties of SFRs.

Each SFR field is, itself, a structure of the type allowing access to the SFR data as a whole, e.g., `WDT_CR.w`, or access to individual bits or fields, e.g., `WDR_CR.KEY`. Any reserved bits in an SFR are not individually accessible in this way. For example,

```
/* get entire WDT CR contents */
uint32_t _wdt_cr = _WDT_REGS->WDT_CR.w;

/* Set KEY field of CR (8 bits) */
_WDT_REGS->_WDT_CR.KEY = 0x1F;

/* Read LOCKMR bit of CR */
uint32_t lock = _WDT_REGS->WDT_CR.LOCKMR;

/* Set entire WDT_CR - including reserved bits */
_WDT_REGS->WDT_CR.w = 0xDEADBEEFu;
```

Note that this and subsequent examples are intended to demonstrate the language interface only and do not demonstrate any usage of the particular SFRs.

For each field of an SFR, macros are also provided to facilitate access to that field using bit operations. For example, `WDR_CR_KEY_Pos` is defined to the least-significant bit position of the `KEY` field, and `WDT_CR_KEY_Msk` is defined to an integer mask of the same width as the SFR, with all bits of the field set, and all other bits 0. Thus, one can, for example, extract the `KEY` field as:

```
/* Read KEY field by reading entire register and extracting
   bits by mask/shift operations */
uint32_t wdt = _WDT_REGS->WDT_CR.w;
uint32_t key = (wdt & WDT_CR_KEY_Msk) >> WDT_CR_KEY_Pos;

/* Update KEY field by masking/inserting bits */
wdt = (wdt & ~WDT_CR_KEY_Msk) | (0x1F << WDT_CR_KEY_Pos);

/* Set WDRSTT bit by bit operations */
wdt = wdt | (1u << WDT_CR_WDRSTT_Pos);

/* Write back updated register contents */
_WDT_REGS->WDR_CR.w = wdt;
```

In addition, the `WDR_CR_KEY_Value` function-like macro is defined to place a value in the proper bit position for the field, so that `WDR_CR_KEY_Value(0x3)` produces the same value as:

```
WDT_CR_KEY_Msk & ((0x3) << WDT_CR_KEY_Pos)
```

Both explicit bit operations using the mask and position macros and bitfield operations may be freely mixed; however, the `WDT_BASE_ADDRESS` pointer should only be accessed using the defined structures, rather than casting to an integral type. Always ensure that you confirm the operation of peripheral modules from the device data sheet.

7.6 TIGHTLY-COUPLED MEMORIES

Some PIC32C/SAM device families feature a SRAM interface providing a fixed latency called Tightly-Coupled Memory (TCM). These memories may contain code (called instruction TCM or ITCM) or data (called data TCM or DTCM). To support this interface, XC32 provides a new `tcm` attribute. You can apply this attribute to a function or variable and it will be placed into instruction or data TCM as appropriate (e.g., `uint32_t __attribute__((tcm)) var;`). Programmers may choose to use TCM in order to achieve better speed and also more consistent timing.

The internal organization of TCM varies greatly from family to family and consequently so do the compiler options used to control it. From the data sheet for the device you are using, you can learn which category to consult below.

In families where code TCM (ITCM) and data TCM (DTCM) are separate, two separate options are provided to specify the respective sizes of these. To enable TCM, pass the `-mitcm=<size_in_bytes>` and the `-mdtcm=<size_in_bytes>` options to the xc32-gcc/g++ driver. See the device data sheet for the size values supported by your target device. The device-specific startup code and the device-specific linker script then work together to set up, initialize, and enable TCM at startup, before your `main()` function is called.

In families where code TCM (ITCM) and data TCM (DTCM) are combined, pass the single option `-mtcm=<size_in_bytes>`. The device start-up code and linker script will work together similarly.

Users can also optionally allocate the vector table and (separately) the stack to TCM.

There are two options to control this:

1. For the vector table, the default behavior is to place the vectors in TCM if possible. Users may pass the option `-mno-vectors-in-tcm` to change this. This might be desirable on some smaller devices with TCM.
2. You may also choose to move your stack to DTCM by passing the `-mstack-in-tcm` option to the xc32-gcc/g++ driver at compile and link time. The linker will allocate a stack to DTCM and the startup code will transfer the stack from System SRAM to DTCM before calling your `main()` function.

NOTES:

Chapter 8. Supported Data Types and Variables

The MPLAB XC32 C/C++ Compiler supports a variety of data types and attributes. These data types and variables are discussed here. For information on where variables are stored in memory, see Chapter 9. "Memory Allocation and Access" or Chapter 9. "Memory Allocation and Access".

8.1 IDENTIFIERS

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

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

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

8.2 DATA REPRESENTATION

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

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

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

8.3 INTEGER DATA TYPES

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

Type	Bits	Min	Max
`char`, `signed char`	8	-128	127
`unsigned char`	8	0	255
`short`, `signed short`	16	-32768	32767
`unsigned short`	16	0	65535
`int`, `signed int`, `long`, `signed long`	32	-2^{31}	$2^{31}-1$
`unsigned int`, `unsigned long`	32	0	$2^{32}-1$
`long long`, `signed long long`	64	-2^{63}	$2^{63}-1$
`unsigned long long`	64	0	$2^{64}-1$

8.3.1 Signed and Unsigned Character Types

Each implementation must define whether a plain `char` is signed or unsigned. For PIC32C and all other ARM platforms, a plain `char` defaults to be unsigned. Note that this behavior differs from PIC32M. The options `-funsigned-char` and `-fsigned-char` can always be used to alter the default type for a given translation unit.

8.3.2 `limits.h`

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

TABLE 8-1: LIMITS.H HEADER FILE

Macro Name	Value	Description
`CHAR_BIT`	8	The size, in bits, of the smallest non-bit field object.
`SCHAR_MIN`	-128	The minimum value possible for an object of type `signed char`.
`SCHAR_MAX`	127	The maximum value possible for an object of type `signed char`.
`UCHAR_MAX`	255	The maximum value possible for an object of type `unsigned char`.
`CHAR_MIN`	-128 (or 0, see **Section 8.3.1 "Signed and Unsigned Character Types"**)	The minimum value possible for an object of type `char`.
`CHAR_MAX`	127 (or 255, see **Section 8.3.1 "Signed and Unsigned Character Types"**)	The maximum value possible for an object of type `char`.
`MB_LEN_MAX`	16	The maximum length of multibyte character in any locale.
`SHRT_MIN`	-32768	The minimum value possible for an object of type `short int`.
`SHRT_MAX`	32767	The maximum value possible for an object of type `short int`.
`USHRT_MAX`	65535	The maximum value possible for an object of type `unsigned short int`.
`INT_MIN`	-2^{31}	The minimum value possible for an object of type `int`.
`INT_MAX`	$2^{31}-1$	The maximum value possible for an object of type `int`.
`UINT_MAX`	$2^{32}-1$	The maximum value possible for an object of type `unsigned int`.
`LONG_MIN`	-2^{31}	The minimum value possible for an object of type `long`.
`LONG_MAX`	$2^{31}-1$	The maximum value possible for an object of type `long`.
`ULONG_MAX`	$2^{32}-1$	The maximum value possible for an object of type `unsigned long`.
`LLONG_MIN`	-2^{63}	The minimum value possible for an object of type `long long`.
`LLONG_MAX`	$2^{63}-1$	The maximum value possible for an object of type `long long`.

TABLE 8-1: LIMITS.H HEADER FILE (CONTINUED)

Macro Name	Value	Description
`ULLONG_MAX`	2^{64}-1	The maximum value possible for an object of type `unsigned long long`.

8.4 FLOATING-POINT DATA TYPES

For the Cortex-M based devices, such as the MEC17, CEC17, and SAM families, XC32 defaults to using the hardware Floating-Point Unit (FPU) where available. For cases where want a specific FPU calling convention, you can specify the following command-line options to the xc32-gcc compilation driver at both compile and link time:

- -mfloat-abi=soft -- Specifying 'soft' causes XC32 to generate output containing library calls for floating-point operations. This setting is the default for devices that do not feature a hardware FPU.
- -mfloat-abi=softfp -- Specifying 'softfp' allows the generation of code using hardware floating-point instructions, but still uses the soft-float calling conventions
- -mfloat-abi=hard -- Specifying 'hard' allows generation of floating-point instructions and uses FPU-specific calling conventions. This setting is the default for devices that feature a hardware FPU.

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

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

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

This format is described in Table 8-2, where:

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

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

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

and for 64-bit values

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

Here is an example of the IEEE 754 32-bit format shown in Table 8-2. Note that the Most Significant bit of the mantissa column (i.e., the bit to the left of the radix point) is the implied bit, which is assumed to be 1 unless the exponent is zero (in which case the float is zero).

TABLE 8-2: FLOATING-POINT FORMAT EXAMPLE IEEE 754

Format	Number	Biased Exponent	1.mantissa	Decimal
32-bit	7DA6B69Bh	11111011b	1.01001101011011010011011b	2.77000e+37
		(251)	(1.302447676659)	—

The example in Table 8-2 can be calculated manually as follows.

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

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

which becomes:

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

which is approximately equal to:

2.77000*e*+37

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

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

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

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

The characteristics of the floating-point formats are summarized in Table 8-3. The symbols in this table are preprocessor macros which are available after including `<float.h>` in your source code. Two sets of macros are available for `float` and `double` types, where *`XXX`* represents `FLT` and `DBL`, respectively. For example, `FLT_MAX` represents the maximum floating-point value of the `float` type. `DBL_MAX` represents the same values for the `double` type. As the size and format of floating-point data types are not fully specified by the ANSI Standard, these macros allow for more portable code which can check the limits of the range of values held by the type on this implementation.

TABLE 8-3: RANGES OF FLOATING-POINT TYPE VALUES

Symbol	Meaning	32-bit Value	64-bit Value
*XXX*_RADIX	Radix of exponent representation	2	2
*XXX*_ROUNDS	Rounding mode for addition	1	
*XXX*_MIN_EXP	Min. *n* such that FLT_RADIX^{n-1} is a normalized float value	-125	-1021
*XXX*_MIN_10_-EXP	Min. *n* such that 10^{n} is a normalized float value	-37	-307
*XXX*_MAX_EXP	Max. *n* such that FLT_RADIX^{n-1} is a normalized float value	128	1024
*XXX*_MAX_10_-EXP	Max. *n* such that 10^{n} is a normalized float value	38	308
*XXX*_MANT_DIG	Number of FLT_RADIX mantissa digits	24	53
*XXX*_EPSILON	The smallest number which added to 1.0 does not yield 1.0	1.1920929e-07	2.22044604925 03131e-16

8.5 STRUCTURES AND UNIONS

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

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

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

8.5.1 Structure and Union Qualifiers

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

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

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

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

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

8.5.2 Bit Fields in Structures

MPLAB XC32 C/C++ Compiler fully supports bit fields in structures.

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

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

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

will produce a structure occupying 1 byte.

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

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

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

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

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

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

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

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

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

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

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

8.6 POINTER TYPES

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

8.6.1 Combining Type Qualifiers and Pointers

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

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

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

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

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

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

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

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

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

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

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

8.6.2 Data Pointers

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

8.6.3 Function Pointers

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

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

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

8.6.4 Special Pointer Targets

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.7 COMPLEX DATA TYPES

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

8.8 CONSTANT TYPES AND FORMATS

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

The formats used to specify the radices are given in Table 8-4. The letters used to specify binary or hexadecimal radices are case insensitive, as are the letters used to specify the hexadecimal digits.

TABLE 8-4: RADIX FORMATS

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

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

The default types of constants may be changed by the addition of a suffix after the digits, e.g., `23U`, where `U` is the suffix. Table 8-5 shows the possible combination of suffixes and the types that are considered when assigning a type. For example, if the suffix `l` is specified and the value is a decimal constant, the compiler will assign the type `long int`, if that type will hold the constant; otherwise, it will be assigned `long long int`. If the constant was specified as an octal or hexadecimal constant, then unsigned types are also considered.

TABLE 8-5: SUFFIXES AND ASSIGNED TYPES

Suffix	Decimal	Octal or Hexadecimal

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.9 STANDARD TYPE QUALIFIERS

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

8.9.1 Const Type Qualifier

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

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

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

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

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

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

8.9.2 Volatile Type Qualifier

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

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

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

The `volatile` qualifier does not guarantee that any access will be atomic, but the compiler will try to implement this.

The code produced by the compiler to access `volatile` objects may be different than that to access ordinary variables, and typically the code will be longer and slower for `volatile` objects, so only use this qualifier if it is necessary. However failure to use this qualifier when it is required may lead to code failure.

Another use of the `volatile` keyword is to prevent variables from being removed if they are not used in the C/C++ source. If a non-`volatile` variable is never used, or used in a way that has no effect on the program's function, then it may be removed before code is generated by the compiler.

A C/C++ statement that consists only of a `volatile` variable's name will produce code that reads the variable's memory location and discards the result. For example the entire statement:

```
PORTB;
```

will produce assembly code that reads `PORTB`, but does nothing with this value. This is useful for some peripheral registers that require reading to reset the state of interrupt flags. Normally such a statement is not encoded as it has no effect.

8.10 COMPILER-SPECIFIC QUALIFIERS

There are currently no non-standard qualifiers implemented in MPLAB XC32 C/C++ Compiler. Attributes are used to control variables and functions.

8.11 VARIABLE ATTRIBUTES

The compiler keyword `__attribute__` allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses.

You may also specify attributes with __ (double underscore) preceding and following each keyword (e.g., `__aligned__` instead of `aligned`). This allows you to use them in header files without being concerned about a possible macro of the same name.

To specify multiple attributes, separate them by commas within the double parentheses, for example:

```
__attribute__ ((aligned (16), packed)).
```

Note: It is important to use variable attributes consistently throughout a project. For example, if a variable is defined in file A with the `aligned` attribute, and declared `extern` in file B without `aligned`, then a link error may result.

`address (addr)`

Specify an absolute virtual address for the variable. This attribute can be used in conjunction with a section attribute.

This attribute can be used to start a group of variables at a specific address:

```
int foo __attribute__((section("mysection"),address(0xA0001000)));
int bar __attribute__((section("mysection")));
int baz __attribute__((section("mysection")));
```

Keep in mind that the compiler performs no error checking on the specified address. The section will be located at the specified address regardless of the memory-region ranges listed in the linker script or the actual ranges on the target device. This application code is responsible for ensuring that the address is valid for the target device and application.

In addition, to make effective use of absolute sections and the new best-fit allocator, standard program-memory and data-memory sections should not be mapped in the linker script. The built-in linker script does not map most standard sections such as the `.text`, `.data`, `.bss`, or `.ramfunc` section. By not mapping these sections in the linker script, we allow these sections to be allocated using the best-fit allocator rather than the sequential allocator. Sections that are unmapped in the linker script can flow around absolute sections whereas sections that are linker-script mapped are grouped together and allocated sequentially, potentially causing conflicts with absolute sections.

Note: In almost all cases, you will want to combine the address attribute with the space attribute to indicate code or data with `space(prog)` or `space(data)`, respectively. See the description for the attribute **`space(memory-space)`**.

`aligned (n)`

The attributed variable will be aligned on the next `n` byte boundary.

The `aligned` attribute can also be used on a structure member. Such a member will be aligned to the indicated boundary within the structure.

If the alignment value `n` is omitted, the alignment of the variable is set 8 (the largest alignment value for a basic data type).

Note that the `aligned` attribute is used to increase the alignment of a variable, not reduce it. To decrease the alignment value of a variable, use the `packed` attribute.

`cleanup (function)`

Indicate a function to call when the attributed automatic function scope variable goes out of scope.

The indicated function should take a single parameter, a pointer to a type compatible with the attributed variable, and have `void` return type.

`packed`

The attributed variable or structure member will have the smallest possible alignment. That is, no alignment padding storage will be allocated for the declaration. Used in combination with the `aligned` attribute, `packed` can be used to set an arbitrary alignment restriction greater or lesser than the default alignment for the type of the variable or structure member.

`section ("section-name")`

Place the variable into the named section.

For example,

```
unsigned int dan __attribute__ ((section (".quixote")))
```

Variable `dan` will be placed in section `.quixote`.

The `-fdata-sections` command line option has no effect on variables defined with a `section` attribute unless `unique_section` is also specified.

space(memory-space)

The space attribute can be used to direct the compiler to allocate a variable in a specific memory space. Valid memory spaces are `prog` for program memory, `data` for data memory, and `serial_mem` for serial memory such as SPI Flash. The `data` space is the default space for non-const variables.

The `prog`, `data`, and `serial_mem` spaces normally correspond to the `kseg0_prog_mem`, `ksegN_data_mem`, and `serial_mem` memory regions, respectively, as specified in the default device-specific linker scripts.

This attribute also controls how initialized data is handled. The linker generates an entry in the data-initialization template for the default `space(data)`. But, it does not generate an entry for `space(prog)` or `space(serial_mem)`, since the variable is located in non-volatile memory. Typically, this means that `space(data)` applies to variables that will be initialized at runtime startup; while `space(prog)` and `space(serial_mem)` apply to variables that will be programmed by an in-circuit programmer or a bootloader.

For example,

```
const unsigned int __attribute__((space(prog))) jack = 10;
const unsigned int __attribute__((space(serial_mem))) zori = 1;
signed int __attribute__((space(data))) oz = 5;
```

unique_section

Place the variable in a uniquely named section, just as if `-fdata-sections` had been specified. If the variable also has a `section` attribute, use that section name as the prefix for generating the unique section name.

For example,

```
int tin __attribute__ ((section (".ofcatfood"), unique_section)
```

Variable `tin` will be placed in section `.ofcatfood`.

unused

Indicate to the compiler that the variable may not be used. The compiler will not issue a warning for this variable if it is not used.

weak

The `weak` attribute causes the declaration to be emitted as a weak symbol. A weak symbol indicates that if a global version of the same symbol is available, that version should be used instead.

When `weak` is applied to a reference to an external symbol, the symbol is not required for linking. For example:

```
extern int __attribute__((weak)) s;
int foo() {
  if (&s) return s;
  return 0; /* possibly some other value */
}
```

In the above program, if `s` is not defined by some other module, the program will still link but `s` will not be given an address. The conditional verifies that `s` has been defined (and returns its value if it has). Otherwise '`0`' is returned. There are many uses for this feature, mostly to provide generic code that can link with an optional library.

Chapter 9. Memory Allocation and Access

There are two broad groups of RAM-based variables: auto/parameter variables, which are allocated to some form of stack and global/static variables, which are positioned freely throughout the data memory space. The memory allocation of these two groups is discussed separately in the following sections.

9.1 ADDRESS SPACES

Unlike the 8- and 16-bit PIC devices, the PIC32 has a unified programming model. PIC32 devices provide a single 32-bit wide address space for all code, data, peripherals and Configuration bits.

Memory regions within this single address space are designated for different purposes; for example, as memory for instruction code or memory for data. Internally the device uses separate buses[1] to access the instructions and data in these regions, thus allowing for parallel access. The terms program memory and data memory, which are used on the 8- and 16-bit PIC devices, are still relevant on PIC32 devices, but the smaller parts implement these in different address spaces.

All addresses used by the CPU within the device are virtual addresses. These are mapped to physical addresses by the system control processor.

9.2 VARIABLES IN DATA MEMORY

Most variables are ultimately positioned into the data memory. The exceptions are non-`auto` variables which are qualified as `const`, which are placed in the program memory space, see Section 8.9.1 "Const Type Qualifier".

Due to the fundamentally different way in which `auto` variables and non-`auto` variables are allocated memory, they are discussed separately. To use the C/C++ language terminology, these two groups of variables are those with automatic storage duration and those with permanent storage duration, respectively.

Note: The terms "local" and "global" are commonly used to describe variables, but are not ones defined by the language standard. The term "local variable" is often taken to mean a variable which has scope inside a function, and "global variable" is one which has scope throughout the entire program. However, the C/C++ language has three common scopes: block, file (i.e., internal linkage) and program (i.e., external linkage), so using only two terms to describe these can be confusing. For example, a `static` variable defined outside a function has scope only in that file, so it is not globally accessible, but it can be accessed by more than one function inside that file, so it is not local to any one function either. In terms of memory allocation, variables are allocated space based on whether it is an `auto` or not, hence the grouping in the following sections.

1.The device can be considered a Harvard architecture in terms of its internal bus arrangement.

9.2.1 Non-auto Variable Allocation

Non-`auto` variables (those with permanent storage duration) are located by the compiler into any of the available data banks. This is done in a two-stage process: placing each variable into an appropriate section and later linking that section into data memory.

The compiler considers three categories of non-`auto` variable which all relate to the value the variable should contain by the time the program begins. The following sections are used for the categories described.

- `.bss` These sections (also .sbss) contain any uninitialized variables, which are not assigned a value when they are defined, or variables which should be cleared by the runtime start-up code.
- `.data` These sections (also .sdata) contain the RAM image of any initialized variables, which are assigned a non-zero initial value when they are defined and which must have a value copied to them by the runtime start-up code.

Note that the data section used to hold initialized variables is the section that holds the RAM variables themselves. There is a corresponding section (called `.dinit`) that is placed into program memory (so it is non-volatile) and which is used to hold the initial values that are copied to the RAM variables by the runtime start-up code.

9.2.2 Static Variables

All `static` variables have permanent storage duration, even those defined inside a function which are "local static" variables. Local `static` variables only have scope in the function or block in which they are defined, but unlike `auto` variables, their memory is reserved for the entire duration of the program. Thus, they are allocated memory like other non-`auto` variables. Static variables may be accessed by other functions via pointers, since they have permanent duration.

Variables which are `static` are guaranteed to retain their value between calls to a function, unless explicitly modified via a pointer.

Variables which are `static` and initialized have their initial value assigned only once during the program's execution. Thus, they may be preferable over initialized `auto` objects which are assigned a value every time the block they are defined in begins execution. Any initialized static variables are initialized in the same way as other non-`auto` initialized objects by the runtime start-up code (see Section 5.4.2 "Peripheral Library Functions"). Static variables are located in the same sections as their non-`static` counterparts.

9.2.3 Non-auto Variable Size Limits

Arrays of any type (including arrays of aggregate types) are fully supported by the compiler. So too are the structure and union aggregate types (see Section 8.5 "Structures and Unions"). There are no theoretical limits on how large these objects can be made.

9.2.4 Changing the Default Non-auto Variable Allocation

There are several ways in which non-`auto` variables can be located in locations other than the default.

Variables can be placed in other device memory spaces by the use of qualifiers. For example if you wish to place variables in the program memory space, then the `const` specifier should be used (see Section 8.9.1 "Const Type Qualifier").

If you wish to prevent all variables from using one or more data memory locations so that these locations can be used for some other purpose, it is best to define a variable (or array) using the `address` attribute so that it consumes the memory space (see Section 8.11 "Variable Attributes").

If only a few non-`auto` variables are to be located at specific addresses in data space memory, then the variables can be located using the `address` attribute. This attribute is described in Section 8.11 "Variable Attributes".

9.3 AUTO VARIABLE ALLOCATION AND ACCESS

This section discusses allocation of `auto` variables (those with automatic storage duration). This also includes function parameter variables, which behave like `auto` (short for *automatic*) variables, as well as temporary variables defined by the compiler.

The `auto` variables are the default type of local variable. Unless explicitly declared to be `static`, a local variable will be made `auto`. The `auto` keyword may be used if desired.

The `auto` variables, as their name suggests, automatically come into existence when a function is executed, then disappear once the function returns. Since they are not in existence for the entire duration of the program, there is the possibility to reclaim memory they use when the variables are not in existence and allocate it to other variables in the program.

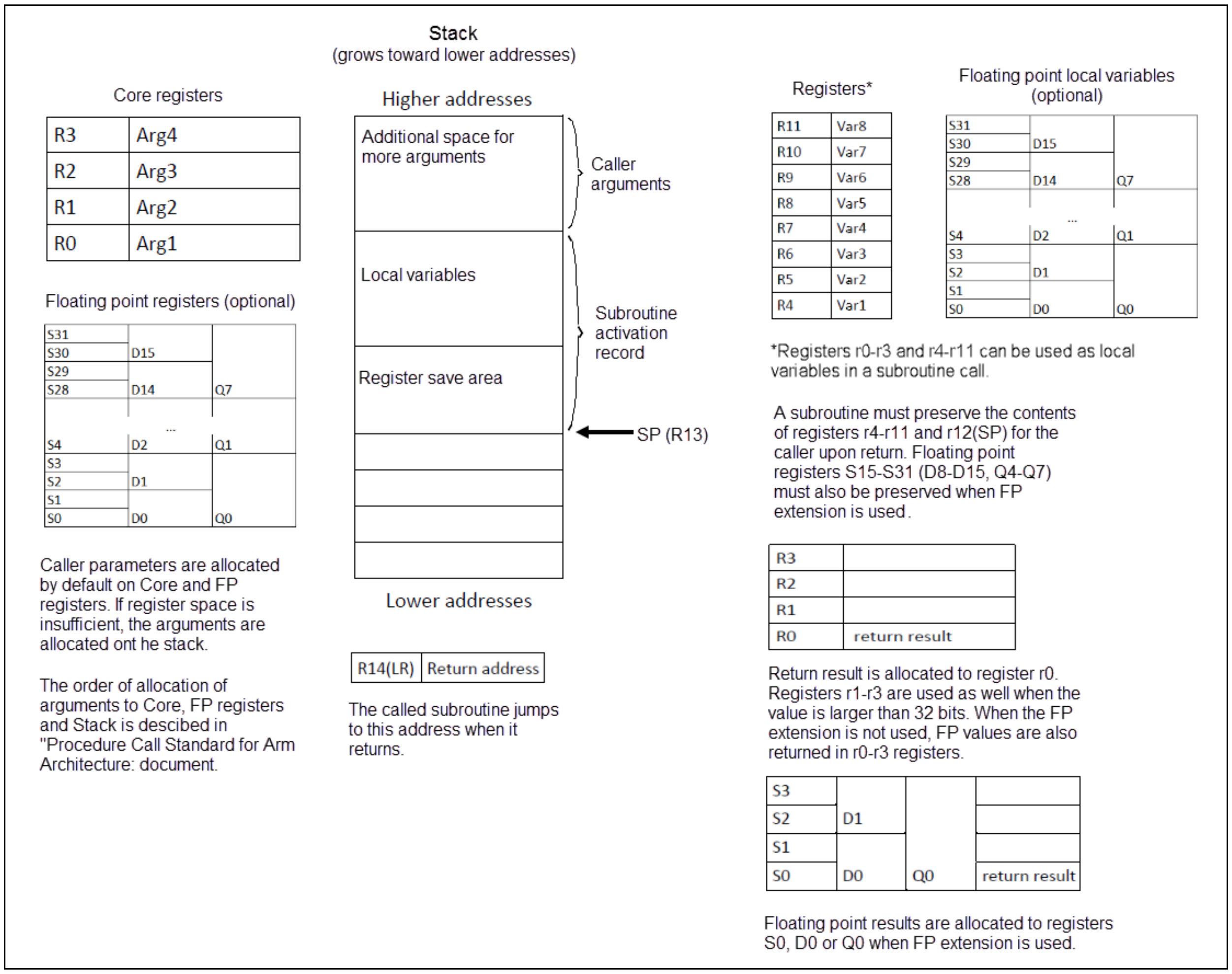
On PIC32C devices, the registers r4-r8, r10, and r11 are used to hold the values of a function's automatic variables. A function must preserve the contents of the registers r4-r8, r10, r11, and r13/SP.

Often times, the software stack of the PIC32C is used to store `auto` variables. The values from registers are pushed onto the stack. Functions are reentrant and each instance of the function has its own area of memory on the stack for its auto and parameter variables.

On PIC32C devices, the Stack Pointer (SP) is register r13. The core uses a full descending stack. A full descending stack means that the stack pointer holds the address of the last stacked item in memory and the stack grows to lower addresses. When the core pushes a new item onto the stack, it decrements the stack pointer and then writes to the item to the new memory location.

The standard qualifiers `const` and `volatile` may both be used with `auto` variables and these do not affect how they are positioned in memory. This implies that a local `const`-qualified object is still an `auto` object and, as such, will be allocated memory on the stack in the data memory, not in the program memory like with non-`auto const` objects.

FIGURE 9-1: STACK


Stack
(grows toward lower addresses)
Core registers
R3 Arg4
R2 Arg3
R1 Arg2
R0 Arg1
Floating point registers (optional)
Caller parameters are allocated by default on Core and FP registers. If register space is insufficient, the arguments are allocated ont he stack.
The order of allocation of arguments to Core, FP registers and Stack is descibed in "Procedure Call Standard for Arm Architecture: document.
Higher addresses
Additional space for more arguments
Caller arguments
Local variables
Subroutine activation record
Register save area
SP (R13)
Lower addresses
R14(LR) Return address
The called subroutine jumps to this address when it returns.
Registers*
Floating point local variables (optional)
*Registers r0-r3 and r4-r11 can be used as local variables in a subroutine call.
A subroutine must preserve the contents of registers r4-r11 and r12(SP) for the caller upon return. Floating point registers S15-S31 (D8-D15, Q4-Q7) must also be preserved when FP extension is used.
return result
Return result is allocated to register r0. Registers r1-r3 are used as well when the value is larger than 32 bits. When the FP extension is not used, FP values are also returned in r0-r3 registers.
Floating point results are allocated to registers S0, D0 or Q0 when FP extension is used.

9.3.1 Local Variable Size Limits

There is no theoretical maximum size for auto variables.

9.4 VARIABLES IN PROGRAM MEMORY

The only variables that are placed into program memory are those that are not `auto` and which have been qualified `const`. Any `auto` variables qualified `const` are placed on the stack along with other `auto` variables.

Any `const`-qualified (`auto` or non-`auto`) variable will always be read-only and any attempt to write to these in your source code will result in an error being issued by the compiler.

A `const` object is usually defined with initial values, as the program cannot write to these objects at runtime. However, this is not a requirement. An uninitialized `const` object is allocated space in the `bss` section, along with other uninitialized RAM variables, but is still treated as read-only by the compiler.

```
const char IOtype = 'A'; // initialized const object
const char buffer[10]; // I just reserve memory in RAM
```

9.4.1 Size Limitations of `const` Variables

There is no theoretical maximum size for `const` variables.

9.4.2 Changing the Default Allocation

If you intend to prevent all variables from using one or more program memory locations so that you can use those locations for some other purpose, you may choose to adjust the memory regions in a custom linker script.

If only a few non-`auto const` variables are to be located at specific addresses in program space memory, then the variables should use the address attribute to locate them at the desired location. This attribute is described in Section 8.11 "Variable Attributes".

9.5 VARIABLES IN REGISTERS

Allocating variables to registers, rather than to a memory location, can make code more efficient. With MPLAB XC32 C/C++ Compiler, variables may be allocated to registers as part of code optimizations. For optimization levels 1 and higher, a value assigned to a variable may be stored in a register. During this time, the memory location associated with the variable may not hold the live value.

The register keyword may be used to indicate your preference for the variable to be allocated a register, but this is just a recommendation and may not be honored. The specific register may be indicated as well, but this is not recommended as your register choice may conflict with the needs of the compiler. Using a specific register in your code may cause the compiler to generate less efficient code.

EXAMPLE 9-1: VARIABLES IN REGISTERS

```
volatile unsigned int special;
unsigned int example (void)
{
register unsigned int my_reg __asm__("v1");
my_reg += special;
return my_reg;
}
```

As indicated in Section 12.2 "Register Conventions", parameters may be passed to a function via a register.

The source code for this is found in the pic32c-libs.zip file located at:

`<install-directory>/pic32-libs/`

Once the file is unzipped, the source code can be found at:

`pic32m-libs/libpic32/stubs/pic32_init_tlb_ebi.S.`

9.6 DYNAMIC MEMORY ALLOCATION

The run-time heap is an uninitialized area of data memory that is used for dynamic memory allocation using the standard C library dynamic memory management functions, `calloc`, `malloc` and `realloc` along with the C++ new operator. Most C++ applications will require a heap.

If you do not use any of these functions, then you do not need to allocate a heap. By default, a heap is not created.

In MPLAB X, you can specify a heap size in the project properties for the xc32-ld linker. MPLAB X will automatically pass the option to the linker when building your project.

If you do want to use dynamic memory allocation, either directly, by calling one of the memory allocation functions, or indirectly, by using a standard C library function that uses one of these functions, then a heap must be created. A heap is created by specifying its size on the linker command line using the `--defsym,min_heap_size` linker command line option. An example of allocating a heap of 512 bytes using the command line is:

```
xc32-gcc foo.c -Wl,--defsym,min_heap_size=512
```

An example of allocating a heap of `0xF000` bytes using the xc32-g++ driver is:

```
xc32-g++ vector.cpp -Wl,--defsym,min_heap_size=0xF000
```

The linker allocates the heap immediately before the stack.

Chapter 10. Operators and Statements

The MPLAB XC32 C/C++ Compiler supports all ANSI operators. The exact results of some of these are implementation-defined. Implementation-defined behavior is fully documented in **Appendix B. "Implementation-Defined Behavior"**. The following sections illustrate code operations that are often misunderstood, as well as additional operations that the compiler is capable of performing.

10.1 INTEGRAL PROMOTION

When there is more than one operand to an operator, they typically must be of exactly the same type. The compiler will automatically convert the operands, if necessary, so they do have the same type. The conversion is to a "larger" type so there is no loss of information; however, the change in type can cause different code behavior to what is sometimes expected. These form the standard type conversions.

Prior to these type conversions, some operands are unconditionally converted to a larger type, even if both operands to an operator have the same type. This conversion is called *integral promotion* and is part of Standard C behavior. The MPLAB XC32 C/C++ Compiler performs these integral promotions where required, and there are no options that can control or disable this operation. If you are not aware that the type has changed, the results of some expressions are not what would normally be expected.

Integral promotion is the implicit conversion of enumerated types, `signed` or `unsigned` varieties of `char`, `short int` or bit field types to either `signed int` or `unsigned int`. If the result of the conversion can be represented by an `signed int`, then that is the destination type, otherwise the conversion is to `unsigned int`.

Consider the following example:

```
unsigned char count=0, a=0, b=50;
if (a - b < 10)
  count++;
```

The `unsigned char` result of `a - b` is 206 (which is not less than 10), but both `a` and `b` are converted to `signed int` via integral promotion before the subtraction takes place. The result of the subtraction with these data types is -50 (which *is* less than 10) and hence the body of the `if()` statement is executed.

If the result of the subtraction is to be an `unsigned` quantity, then apply a cast. For example:

```
if((unsigned int)(a - b) < 10)
  count++;
```

The comparison is then done using `unsigned int`, in this case, and the body of the `if()` would not be executed.

Another problem that frequently occurs is with the bitwise compliment operator, `~`. This operator toggles each bit within a value. Consider the following code:

```
unsigned char count=0, c;
c = 0x55;
if (~c == 0xAA)
  count++;
```

If c contains the value 0x55, it is often assumed that ~c will produce 0xAA, however the result is 0xFFFFFFAA and so the comparison in the above example would fail. The compiler may be able to issue a mismatched comparison error to this effect in some circumstances. Again, a cast could be used to change this behavior.

The consequence of integral promotion as illustrated above is that operations are not performed with char -type operands, but with int -type operands. However there are circumstances when the result of an operation is identical regardless of whether the operands are of type char or int. In these cases, the MPLAB XC32 C/C++ Compiler will not perform the integral promotion so as to increase the code efficiency. Consider the following example:

```
unsigned char a, b, c;
a = b + c;
```

Strictly speaking, this statement requires that the values of b and c should be promoted to unsigned int, the addition performed, the result of the addition cast to the type of a, and then the assignment can take place. Even if the result of the unsigned int addition of the promoted values of b and c was different to the result of the unsigned char addition of these values without promotion, after the unsigned int result was converted back to unsigned char, the final result would be the same. If an 8-bit addition is more efficient than a 32-bit addition, the compiler will encode the former.

If, in the above example, the type of a was unsigned int, then integral promotion would have to be performed to comply with the ANSI C standard.

10.2 TYPE REFERENCES

Another way to refer to the type of an expression is with the typeof keyword. This is a non-standard extension to the language. Using this feature reduces your code portability.

The syntax for using this keyword looks like sizeof, but the construct acts semantically like a type name defined with typedef.

There are two ways of writing the argument to typeof: with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This assumes that x is an array of functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is a pointer to int.

If you are writing a header file that must work when included in ANSI C programs, write __typeof__ instead of typeof.

A typeof construct can be used anywhere a typedef name could be used. For example, you can use it in a declaration, in a cast, or inside of sizeof or typeof.

- This declares y with the type of what x points to:
  ```
  typeof (*x) y;
  ```
- This declares y as an array of such values:
  ```
  typeof (*x) y[4];
  ```
- This declares y as an array of pointers to characters:
  ```
  typeof (typeof (char *)[4]) y;
  ```
  It is equivalent to the following traditional C declaration:
  ```
  char *y[4];
  ```

To see the meaning of the declaration using `typeof` and why it might be a useful way to write, let's rewrite it with these macros:

```
#define pointer(T) typeof(T *)
#define array(T, N) typeof(T [N])
```

Now the declaration can be rewritten this way:

```
array (pointer (char), 4) y;
```

Thus, `array (pointer (char), 4)` is the type of arrays of four pointers to `char`.

10.3 LABELS AS VALUES

You can get the address of a label defined in the current function (or a containing function) with the unary operator '`&&`'. This is a non-standard extension to the language. Using this feature reduces your code portability.

The value returned has type `void *`. This value is a constant and can be used wherever a constant of that type is valid. For example:

```
void *ptr;
...
ptr = &&foo;
```

To use these values, you need to be able to jump to one. This is done with the computed goto statement, `goto *exp`;. For example:

```
goto *ptr;
```

Any expression of type `void *` is allowed.

One way of using these constants is in initializing a static array that will serve as a jump table:

```
static void *array[] = { &&foo, &&bar, &&hack };
```

Then you can select a label with indexing, like this:

```
goto *array[i];
```

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

Such an array of label values serves a purpose much like that of the `switch` statement. The `switch` statement is cleaner and therefore preferable to an array.

Another use of label values is in an interpreter for threaded code. The labels within the interpreter function can be stored in the threaded code for fast dispatching.

This mechanism can be misused to jump to code in a different function. The compiler cannot prevent this from happening, so care must be taken to ensure that target addresses are valid for the current function.

10.4 CONDITIONAL OPERATOR OPERANDS

The middle operand in a conditional expression may be omitted. Then if the first operand is nonzero, its value is the value of the conditional expression. This is a non-standard extension to the language. Using this feature reduces your code portability.

Therefore, the expression:

```
x ? : y
```

has the value of `x` if that is nonzero; otherwise, the value of `y`.

This example is equivalent to:

```
x ? x : y
```

In this simple case, the ability to omit the middle operand is not especially useful. When it becomes useful is when the first operand does, or may (if it is a macro argument), contain a side effect. Then repeating the operand in the middle would perform the side effect twice. Omitting the middle operand uses the value already computed without the undesirable effects of recomputing it.

10.5 CASE RANGES

You can specify a range of consecutive values in a single case label, like this:

```
case low ... high:
```

This has the same effect as the proper number of individual case labels, one for each integer value from *`low`* to *`high`*, inclusive. This is a non-standard extension to the language. Using this feature reduces your code portability.

This feature is especially useful for ranges of ASCII character codes:

```
case 'A' ... 'Z':
```

Be careful: Write spaces around the..., otherwise it may be parsed incorrectly when you use it with integer values. For example, write this:

```
case 1 ... 5:
```

rather than this:

```
case 1...5:
```

Chapter 11. Fixed-Point Arithmetic Support

The MPLAB XC32 C/C++ Compiler supports fixed-point arithmetic according to the N1169 draft of ISO/IEC TR 18037, the ISO C99 Technical Report on Embedded C. It is available at: http://www.open-std.org/JTC1/SC22/WG14/www/projects#18037.

This chapter describes the implementation-specific details of the types and operations supported by the compiler under the N1169 draft standard.

11.1 ENABLING FIXED-POINT ARITHMETIC SUPPORT

Fixed-point arithmetic support is enabled by default by the MPLAB XC32 C/C++ compiler, allowing use of built-in fixed-point types, literals and operators as described in **Section 11.2 "Data Types"**. Additional headers may be included to provide convenient definitions as described in **Section 11.3 "External Definitions"**.

11.2 DATA TYPES

All of the 12 primary fixed-point types and their aliases, as described in Section 4.1 "Overview and principles of the fixed-point data types" of the N1169 draft of ISO/IEC TR 18037, are supported by the compiler. Fixed-point data values consist of fractional and optional integral parts. The format of fixed-point data types supported by the compiler are specified in Table 11-1 below.

In the formats shown, "s" indicates the sign bit for signed types (absent for unsigned types). Numeric values indicate the width of the integer and fractional parts, with a period separating the optional integer part from the fractional part.

TABLE 11-1: FIXED POINT FORMATS

Type	Format	Description
`short _Fract`	s.7	1 bit sign, 7 bits fraction
`unsigned short _Fract`	0.8	8 bits fraction
`_Fract`	s.15	1 bit sign, 15 bits fraction
`unsigned _Fract`	0.16	16 bits fraction
`long _Fract`	s.31	1 bit sign, 31 bits fraction
`unsigned long _Fract`	0.32	32 bits fraction
`long long _Fract`	s.63	1 bit sign, 63 bits fraction
`unsigned long long _Fract`	0.64	64 bits fraction
`short _Accum`	s8.7	1 bit sign, 8 bits integer, 7 bits fraction
`unsigned short _Accum`	8.8	8 bits integer, 8 bits fraction
`_Accum`	s16.15	1 bit sign, 16 bits integer, 15 bits fraction
`unsigned _Accum`	16.16	16 bits integer, 16 bits fraction
`long _Accum`	s32.31	1 bit sign, 32 bits integer, 31 bits fraction
`unsigned long _Accum`	32.32	32 bits integer, 32 bits fraction
`long long _Accum`	s32.31	1 bit sign, 32 bits integer, 31 bits fraction
`unsigned long long _Accum`	32.32	32 bits integer, 32 bits fraction

The _Sat type modifier may be used with any type in Table 11-1 to indicate that values are saturated, as described in ISO/IEC TR 18037 draft N1169. For example, _Sat short _Fract is the saturating form of short _Fract. Signed types saturate at the largest-magnitude negative and position numbers representable by the type. Unsigned types saturate at 0 and the largest-magnitude (positive) value representable by the type.

The default behavior of overflow on signed or unsigned types is saturation. The pragmas described in Section 4.1.3 "Rounding and Overflow" of the N1169 draft of ISO/IEC TR 18037 to control the rounding and overflow behavior are not supported.

Table 11-2 describes the fixed-point literal suffixes supported to form fixed-point literals of each type.

TABLE 11-2: FIXED-POINT LITERAL SUFFIXES

Type	Suffixes
`short _Fract`	`hr, HR`
`unsigned short _Fract`	`uhr, UHR`
`_Fract`	`r, R`
`unsigned _Fract`	`ur, UR`
`long _Fract`	`lr, LR`
`unsigned long _Fract`	`ulr, ULR`
`long long _Fract`	`llr, LLR`
`unsigned long long _Fract`	`ullr, ULLR`
`short _Accum`	`hk, HK`
`unsigned short _Accum`	`uhk, UHK`
`_Accum`	`k, K`
`unsigned _Accum`	`uk, UK`
`long _Accum`	`lk, LK`
`unsigned long _Accum`	`ulk, ULK`
`long long _Accum`	`llk, LLK`
`unsigned long long _Accum`	`ullk, ULLK`

11.3 EXTERNAL DEFINITIONS

The stdfix.h file included with the compiler provides various pre-processor macros related to fixed-point support. This header defines aliases for the ISO/IEC TR 18037 draft N1169 type specifiers/modifiers, for example fract as an alias for _Fract, as well as macros related to the precision and limits of each type. An example of usage follows:

```
#include <stdfix.h>
int main(void)
{
  int i;
  fract a[5] = {0.5,0.4,0.2,0.0,-0.1};
  fract b[5] = {0.1,0.8,0.6,0.5,-0.1};
  accum dp = 0;
  /* compute dp = dot product of a[] and b[] */
  for (i = 0; i < 5; i++) {
    dp += a[i] * b[i];
  }
  return 0;
}
```

11.4 C OPERATORS

The following C language operators are supported for fixed-point types:

- prefix and postfix increment and decrement operators (`++`, `--`)
- unary arithmetic operators (`+`, `-`, `!`)
- binary arithmetic operators (`+`, `-`, `*`, `/`)
- binary shift operators (`<<`, `>>`)
- relational operators (`<`, `<=`, `>=`, `>`)
- assignment operators (`+=`, `-=`, `*=`, `/=`, `<<=`, `>>=`)
- conversions to and from integer, floating-point, or fixed-point types

11.5 UNSUPPORTED FEATURES

The fixed-point conversion specifiers for formatted I/O, as described in Section 4.1.9 “Formatted I/O functions for fixed-point arguments” of ISO/IEC TR 18037 draft N1169, are not supported by the current MPLAB XC32 standard C libraries. Fixed-point arguments must be used in formatted I/O routines by conversion to or from an appropriate floating-point representation. For example:

```
#include <stdio.h>
#include <stdfix.h>

int main(void)
{
  fract a = 0.5;
  accum b;
  double d;

  scanf ("%lf", &d); /* read into floating-point type */
  b = (accum) d;    /* convert to fixed-point type */
  printf ("%1.4f", (float) a); /* cast to floating-point type for
output */
  return 0;
}
```

The fixed-point functions described in Section 4.1.7 of ISO/IEC TR 18037 draft N1169 are not provided by the current MPLAB XC32 standard C libraries.

NOTES:

Chapter 12. Register Usage

This chapter examines registers used by the compiler to generate assembly from C/C++ source code.

12.1 REGISTER USAGE

This chapter describes the usage of registers in compiler-generated assembly and hardware register conventions.

When generating assembly from C/C++ source code, the compiler assumes that register contents will not be modified by external functions according to the calling conventions, or by inline assembly statements. The extended inline assembly syntax may be used to indicate the hardware registers used and/or modified by inline assembly so that the compiler may generate correct code in the presence of these statements.

12.2 REGISTER CONVENTIONS

The 16 general-purpose registers in the Arm® Cortex®-Mx core of PIC32C devices are shown in Table 12-1. Certain registers are assigned to a dedicated purpose by the compiler, or have synonyms indicating their usage in the procedure call standard. The special names for use in assembly code and for dedicated usage, when applicable, are indicated.

TABLE 12-1: REGISTER CONVENTIONS

Register Number	Special Name	Use
R0-R6		General purpose registers.
R7		Syscall number register.
R8-R10		General purpose registers.
R11	`FP`	Frame pointer.
R12	`IP`	Intra-procedure-call scratch register.
R13	`SP`	Stack pointer.
R14	`LR`	Link register.
R15	`PC`	Program counter.

Note: The special registers R7 and R11 may be available for general-purpose use if not required for their dedicated use. Note also that all register names, including synonyms and special names, are case-insensitive in assembly language.

NOTES:

Chapter 13. Functions

The following sections describe how function definitions are written and specifically how they can be customized to suit your application. The conventions used for parameters and return values, as well as the assembly call sequences are also discussed.

13.1 WRITING FUNCTIONS

Functions may be written in the usual way in accordance with the C/C++ language.

The only specifier that has any effect on function is `static`. Interrupt functions are defined with the use of the `interrupt` attribute (see Section 13.2 "Function Attributes and Specifiers").

A function defined using the `static` specifier only affects the scope of the function, i.e., limits the places in the source code where the function may be called. Functions that are `static` may only be directly called from code in the file in which the function is defined. The equivalent symbol used in assembly code to represent the function may change if the function is `static` (see Section 9.2.2 "Static Variables"). This specifier does not change the way the function is encoded.

13.2 FUNCTION ATTRIBUTES AND SPECIFIERS

13.2.1 Function Attributes

`address(addr)`

The address attribute specifies an absolute physical address at which the function will be placed in memory.

The compiler performs no error checking on the address value, so the application must ensure the value is valid for the target device. The section containing the function will be located at the specified address regardless of the memory-regions specified in the linker script or the actual memory ranges on the target device. The application code must ensure that the address is valid for the target device.

To make effective use of absolute sections and the new best-fit allocator, standard program-memory and data-memory sections should not be mapped in the linker script. The built-in linker script does not map most standard sections, such as the `.text`, `.data`, or `.bss` sections. By not mapping these sections in the linker script, we allow these sections to be allocated using the best-fit allocator rather than the sequential allocator. Sections that are unmapped in the linker script can flow around absolute sections, whereas sections that are linker-script mapped are grouped together and allocated sequentially, potentially causing conflicts with absolute sections.

`alias ("symbol")`

Indicates that the function is an alias for another symbol. For example:

```
void foo (void) { /* stuff */ }
__attribute__ ((alias("foo"))) void bar (void);
```

In the above example, symbol `bar` is considered to be an alias for the symbol `foo`.

always_inline

Instructs the compiler to always inline a function declared as `inline`, even if no optimization level was specified.

const

If the result of a pure function is determined exclusively from its parameters (i.e., does not depend on the values of any global variables), it may be declared with the `const` attribute, allowing for even more aggressive optimization. Note that a function which de-references a pointer argument cannot be declared `const` if it depends on the referenced value, as the referenced storage is not considered a parameter of the function.

deprecated
deprecated (msg)

When a function specified as `deprecated` is used, the compiler will generate a warning. The optional `msg` argument, which must be a string, will be printed in the warning if present. The `deprecated` attribute may also be used for variables and types.

format (type, format_index, first_to_check)

The `format` attribute indicates that the function takes a `printf`, `scanf`, `strftime`, or `strfmon` style format string at position `index` in the argument list, and instructs the compiler to type-check the arguments starting at `first_to_check` against the conversion specifiers in the format string, just as it does for the standard library functions.

The `type` parameter is one of `printf`, `scanf`, `strftime` or `strfmon` (optionally with surrounding double underscores, e.g., `__printf__`) and determines how the format string will be interpreted.

The `format_index` parameter specifies the position of the format string in the function's parameters. Function parameters are numbered from the left, starting from index 1.

The `first_to_check` parameter specifies the position of the first parameter to check against the format string. All parameters following the parameter indicated by `first_to_check` will be checked. If `first_to_check` is zero, type checking is not performed, and the compiler only checks the format string for consistency.

format_arg (index)

The `format_arg` attribute specifies that a function manipulates a `printf` style format string and that the compiler should check the format string for consistency. The `index` parameter gives the position of the format string in the parameter list of the function, numbered from the left beginning at index 1.

interrupt
interrupt(type)

Functionally equivalent to the `isr` attribute.

isr
isr(type)

Instructs the compiler to generate prologue and epilogue code for the function as an interrupt handler function. See Chapter 14. "Interrupts". The optional `type` argument specifies the type of the interrupt, which may be one of the identifiers `irq`, `fiq`, `abort`, `undef` or `swi`, in lowercase or uppercase.

`keep`

The `keep` attribute prevents the linker from removing an unused function when `--gc-sections` is in effect.

`long_call`

Always call the function with an indirect call instruction.

`malloc`

The `malloc` attribute asserts that any non-null pointer return value from the function will not be aliased to any other pointer which is live at the point of return from the function. This allows the compiler to perform more aggressive optimizations.

`naked`

Indicates that the compiler should generate no prologue or epilogue code for the function.

`noinline`

A function declared with the `noinline` attribute will never be considered for inlining, regardless of optimization level.

`nonnull (index, ...)`

Indicate to the compiler that one or more pointer arguments to the function must be non-null. When the `-Wnonnull` option is in effect, the compiler will issue a warning diagnostic if it can determine that the function is called with a null pointer supplied for any `nonnull` argument. The `index` argument(s) indicate the position of the pointer arguments required to be non-null in the parameter list of the function, numbered from the left starting at index 1. If no arguments are provided, all pointer arguments of the function will be marked as non-null.

`noreturn`

Indicate to the compiler that the function will never return control to its caller. In some situations, this can allow the compiler to generate more efficient code in the calling function, since optimizations can be performed without regard to behavior if the function ever did return. Functions declared with `noreturn` should always have a void return type.

`optimize(opt_level)`

The optimize attribute may be used to specify different optimization options for individual functions within a source file. The `opt_level` argument may be an integer, which will be assumed to be an optimization level (i.e., `-Oopt_level`, or a string specifying an optimization option. Strings that begin with `O` are assumed to be an optimization option. An example usage of this attribute is to have frequently-executed functions compiled with more aggressive optimization options to produce faster and larger code, while using lower optimization levels to avoid code size increases for less frequently used functions.

`pure`

Indicates to the compiler that the function is pure, allowing for more aggressive optimizations in the presence of calls to the function. A `pure` function has no side effects other than its return value, and the return value is dependent only on parameters and/or (non-volatile) global variables.

section("name")

Place the function into the given named section. For example:

```
void __attribute__ ((section (".wilma"))) baz () {return;}
```

In the above example, function `baz` will be placed in section `.wilma`. The `-ffunction-sections` command line option has no effect on functions declared with the `section` attribute.

space(id)

Place the function in the memory space identified by the `id` argument. The `id` argument may be `prog`, which places the function in the program space (i.e., ROM), or `data`, which places the function in a data section (i.e., RAM). Unlike the `section` attribute, the actual section is not explicitly specified.

short_call

Always call the function using an absolute call instruction, even when the `-mlong-calls` command line option is specified.

tcm

Attempt to place the function in tightly-coupled memory (TCM), providing highly consistent access times. The actual section will be determined by the compiler, possibly based on other attributes such as `space`. For example:

```
void __attribute__((tcm)) foo (void) {return;}
```

In the above example, the compiler will attempt to place `foo` in tightly-coupled program memory. Note that the amount of TCM available in program or data memory on the target device may vary, so the compiler cannot ensure all functions with the `tcm` attribute can be placed in TCM.

Also see Section 7.6 "Tightly-Coupled Memories".

unique_section

Place the function in a uniquely named section, as if `-ffunction-sections` were in effect. If the function also has a `section` attribute, the given section name will be used as a prefix for the generated unique section name. For example:

```
void __attribute__ ((section (".fred"), unique_section) foo (void)
{return;}
```

In the above example, function `foo` will be placed in section `.fred.foo`.

unsupported

Indicate to the compiler that the function is not supported, similar to the `deprecated` attribute. A warning will be issued if the function is called.

unused

Indicate to the compiler that the function may not be used. The compiler will not issue a warning for this function if it is not used.

used

Indicate to the compiler that the function is always used and code must be generated for the function even if the compiler cannot determine that the function is called, e.g., if a status function is only called from an inline assembly statement.

`warn_unused_result`

A warning will be issued if the return value of the indicated function is unused by a caller.

`weak`

A weak symbol indicates that if another version of the same symbol is available, that version should be used instead. For example, this is useful when a library function is implemented such that it can be overridden by a user written function.

13.3 ALLOCATION OF FUNCTION CODE

Code associated with C/C++ functions is normally always placed in the program Flash memory of the target device.

An alternative to Flash memory is to place functions in TMC on those devices if available.

13.4 CHANGING THE DEFAULT FUNCTION ALLOCATION

The assembly code associated with a C/C++ function can be placed at an absolute address. This can be accomplished by using the `address` attribute and specifying the virtual address of the function (see Section 8.11 “Variable Attributes”).

Functions can also be placed at specific positions by placing them in a user-defined section and then linking this section at an appropriate address (see Section 8.11 “Variable Attributes”).

13.5 FUNCTION SIZE LIMITS

There are no theoretical limits as to how large functions can be made.

13.6 FUNCTION PARAMETERS

MPLAB XC uses a fixed convention to pass arguments to a function. The method used to pass the arguments depends on the size and number of arguments involved.

> **Note:** The names “argument” and “parameter” are often used interchangeably, but typically an argument is the actual value that is passed to the function and a parameter is the variable defined by the function to store the argument.

The Stack Pointer is always aligned on an 8-byte boundary.

- All integer types smaller than a 32-bit integer are first converted to a 32-bit value. The first four 32 bits of arguments are passed via registers `a0`-`a3` (see Table 13-1 for how many registers are required for each data type).
- Although some arguments may be passed in registers, space is still allocated on the stack for all arguments to be passed to a function (see Figure 13-1). Application code should not assume that the current argument value is on the stack, even when space is allocated.
- When calling a function:
  - Registers `a0`-`a3` are used for passing arguments to functions. Values in these registers are not preserved across function calls.
  - Registers `t0`-`t7` and `t8`-`t9` are caller saved registers. The calling function must push these values onto the stack for the registers’ values to be saved.

- Registers `s0`-`s7` are called saved registers. The function being called must save any of these registers it modifies.
- Register `s8` is a saved register if the optimizer eliminates its use as the Frame Pointer. `s8` is a reserved register otherwise.
- Register `ra` contains the return address of a function call.

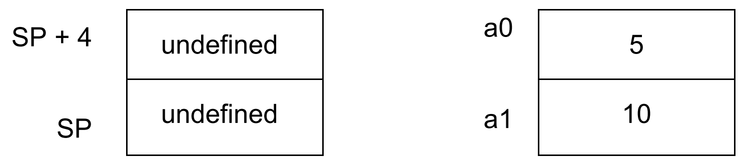
TABLE 13-1: REGISTERS REQUIRED

Data Type	Number of Registers Required
`char`	1
`short`	1
`int`	1
`long`	1
`long long`	2
`float`	1
`double`	1
`long double`	2
`structure`	Up to 4, depending on the size of the struct.

FIGURE 13-1: PASSING ARGUMENTS

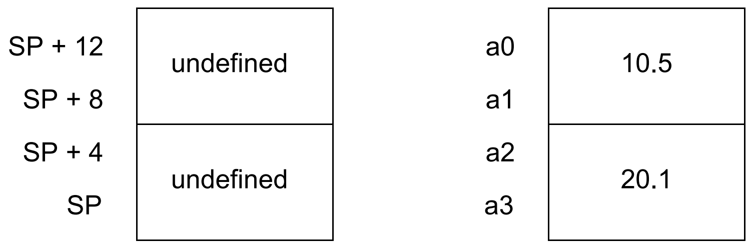
Example 1:

```
int add (int, int)
a= add (5, 10);
```



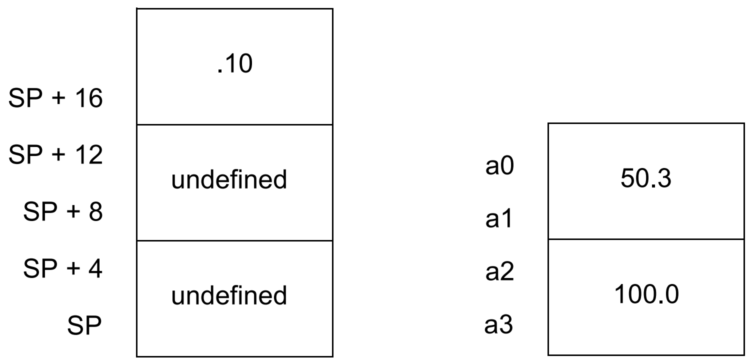
Example 2:

```
void foo (long double, long double)
call= foo (10.5, 20.1);
```



Example 3:

```
void calculate (long double, long double, int)
calculate (50.3, 100.0, .10);
```



13.7 FUNCTION RETURN VALUES

Function return values are returned in registers.

Integral or pointer value are placed in register `v0`. All floating-point values, regardless of precision, are returned in floating-point register `$f0`.

If a function needs to return an actual structure or union – not a pointer to such an object – the called function copies this object to an area of memory that is reserved by the caller. The caller passes the address of this memory area in register $4 when the function is called. The function also returns a pointer to the returned object in register `v0`. Having the caller supply the return object's space allows re-entrance.

13.8 CALLING FUNCTIONS

By default, functions are called using the direct form of the call (`jal`) instruction. This allows calls to destinations within a 256 MB segment. This operation can be changed through the use of attributes applied to functions or command-line options so that a longer, but unrestricted, call is made.

The `-mlong-calls` option, (see Section 5.8.1 "Options Specific to PIC32C/SAM Devices"), forces a register form of the call to be employed by default. Generated code is longer, but calls are not limited in terms of the destination address.

The attributes `longcall` or `far` can be used with a function definition to always enforce the longer call sequence for that function. The `near` attribute can be used with a function so that calls to it use the shorter direct call, even if the `-mlong-calls` option is in force.

13.9 INLINE FUNCTIONS

By declaring a function `inline`, you can direct the compiler to integrate that function's code into the code for its callers. This usually makes execution faster by eliminating the function-call overhead. In addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time, so that not all of the inline function's code needs to be included. The effect on code size is less predictable. Machine code may be larger or smaller with inline functions, depending on the particular case.

Note: Function inlining will only take place when the function's definition is visible (not just the prototype). In order to have a function inlined into more than one source file, the function definition may be placed into a header file that is included by each of the source files.

To declare a function inline, use the `inline` keyword in its declaration, like this:

```
inline int
inc (int *a)
{
  (*a)++;
}
```

(If you are using the `-traditional` option or the `-ansi` option, write `__inline__` instead of `inline`.) You can also make all “simple enough” functions inline with the command-line option `-finline-functions`. The compiler heuristically decides which functions are simple enough to be worth integrating in this way, based on an estimate of the function’s size.

Note: The `inline` keyword will only be recognized with `-finline` or optimizations enabled.

Certain usages in a function definition can make it unsuitable for inline substitution. Among these usages are: use of `varargs`, use of `alloca`, use of variable-sized data, use of computed `goto` and use of nonlocal `goto`. Using the command-line option `-Winline` will warn when a function marked `inline` could not be substituted, and will give the reason for the failure.

In compiler syntax, the `inline` keyword does not affect the linkage of the function.

When a function is both `inline` and `static`, if all calls to the function are integrated into the caller and the function’s address is never used, then the function’s own assembler code is never referenced. In this case, the compiler does not actually output assembler code for the function, unless you specify the command-line option `-fkeep-inline-functions`. Some calls cannot be integrated for various reasons (in particular, calls that precede the function’s definition cannot be integrated and neither can recursive calls within the definition). If there is a non-integrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that can’t be inlined. The compiler will only eliminate `inline` functions if they are declared to be `static` and if the function definition precedes all uses of the function.

When an `inline` function is not `static`, then the compiler must assume that there may be calls from other source files. Since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-`static` inline function is always compiled on its own in the usual fashion.

If you specify both `inline` and `extern` in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function and had not defined it.

This combination of `inline` and `extern` has a similar effect to a macro. Put a function definition in a header file with these keywords and put another copy of the definition (lacking `inline` and `extern`) in a library file. The definition in the header file will cause most calls to the function to be inlined. If any uses of the function remain, they will refer to the single copy in the library.

NOTES:

Chapter 14. Interrupts

Interrupt processing is an important aspect of most microcontroller applications. Interrupts may be used to synchronize software operations with events that occur in real time. When an interrupt occurs, the normal flow of software execution is suspended and special functions are invoked to process the event. These special functions and are often called interrupt handlers or Interrupt Service Routines (ISRs). At the completion of interrupt processing, previous context information is restored and normal execution resumes.

PIC32C/SAM devices support multiple interrupts, from both internal and external sources. The devices allow high-priority interrupts to override any lower priority interrupts that may be in progress.

The compiler provides full support for interrupt processing in C/C++ or inline assembly code. This section presents an overview of interrupt processing.

14.1 INTERRUPT OPERATION

Each interrupt typically has a control bit in a special function register (SFR) that can disable that interrupt source. Most also have a configurable priority level. Check your device data sheet and the appropriate technical reference manual for full information on how your device handles interrupts.

The compiler incorporates features allowing interrupts to be fully handled from C/C++ code. *Interrupt code* is the name given to any code that executes as a result of an interrupt occurring. Interrupt code completes at the point where the corresponding return from interrupt instruction is executed. This contrasts with *main-line code*, which for a freestanding application, is the main part of the program that executes after Reset.

14.2 WRITING AN INTERRUPT SERVICE ROUTINE

An interrupt service routine takes no arguments and returns no results, that is, its argument list is void and it returns type void. This pattern is not enforced but executing an ISR, that does not follow it, will result in unpredictable behavior.

On the Cortex-M cores found in PIC32C/SAM devices, the hardware takes care of context save and restore. When an ISR is executed, in addition to other things, the hardware saves registers r0, r1, r2, r3, r12, and r14 on the stack and restores them when the function exits. Consequently, any function with no return value and no arguments that conforms to the procedure call standard of the platform can be used as a handler function. See the appropriate architecture reference manual for the full details on exception entry and exit behavior.

Given the hardware support, it is not mandatory to tell the compiler that a function is an interrupt handler since no special entry or exit code needs to be generated. That said, a function should still be marked as an interrupt handler function using the `interrupt` attribute so that the compiler knows that it is used. With the `interrupt` attribute, the compiler generates code to guarantee the stack is 8-byte (double-word) aligned when the function is entered and restores the stack pointer to its original value when the function exits. Such an action is a safeguard, but the extra instructions are not necessary if the device is configured to guarantee an 8-byte aligned stack upon exception entry. See the architecture reference manual for details.

14.2.1 Interrupt Attribute

`__attribute__((interrupt))`

Use of the `interrupt` attribute tells the compiler that the function is an interrupt handler and generates code to ensure the stack pointer is aligned on 8 bytes upon function entry.

Note: The `interrupt` attribute can take arguments as described in the GCC manual but are ignored when targeting PIC32C/SAM devices.

14.3 ASSOCIATING A HANDLER FUNCTION WITH AN EXCEPTION

Each exception handler, be it internal or external, is associated with a function. The exception handled by that function depends on its name. The name of a handler function corresponds to the name of the exception it handles, suffixed with `_Handler`. For example, `SysTick_Handler` is the exception handler for the `SysTick` interrupt. To define a custom handler, write a function with the name matching that of the default handler for that exception and link it into your application. Aside from the standard internal handlers (see next section), names of handler functions are specific to the device. To see the full list of handler function names, check the appropriate device header file in `pic32c/include_mcc/proc`.

The following example shows how to create a custom `SysTick_Handler`.

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

const static uint32_t LOWEST_IRQ_PRIORITY =
  (1UL << __NVIC_PRIO_BITS) - 1UL;

static uint32_t tick_counter;

__attribute__((interrupt)) void SysTick_Handler(void) {
    tick_counter += 1;
}

int main(void) {
    // Get the reload value for 10ms.
    uint32_t ticks = SysTick->CALIB & SysTick_CALIB_TENMS_Msk;

    // Set the IRQ priority, the SysTick reload value, the counter
    // value, then enable the interrupt.  The same can be achieved
    // using the function SysTick_Config from the CMSIS-Core(M) API.
    NVIC_SetPriority(SysTick_IRQn, LOWEST_IRQ_PRIORITY);
    SysTick->LOAD = (uint32_t) ticks - 1UL;
    SysTick->VAL = 0UL;
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk
      | SysTick_CTRL_TICKINT_Msk
      | SysTick_CTRL_ENABLE_Msk;

    // Ensure the changes are written before continuing.
    __DSB();
    while (1) { __builtin_nop(); }
    return 0;
}
```

Some PIC32C/SAM devices have faults that must enabled in software. The following example associates a handler with the `UsageFault` and enables divide-by-zero errors.

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

__attribute__((interrupt)) void UsageFault_Handler(void);

void UsageFault_Handler(void) {
    __builtin_software_breakpoint();
}

uint32_t DemoFunction(uint32_t dividend, uint32_t divisor) {
    return dividend / divisor;
}

int main(void) {
    // Enable UsageFault and divide by zero errors
    SCB->SHCSR |= SCB_SHCSR_USGFAULTENA_Msk;
    SCB->CCR |= SCB_CCR_DIV_0_TRP_Msk;
    // Ensure the changes are written before continuing.
    __DSB();

    DemoFunction(2, 0);

    return 0;
}
```

Below is a longer example that uses two interrupts to safely access shared data without having to disable interrupts. The space for `tick_counter` is only accessed by the `SysTick` and `PendSV` handlers. Both run at the same priority meaning that they cannot interrupt each other. Hence, access to `tick_counter` is properly serialized.

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

const static uint32_t LIMIT = 50;
const static uint32_t LOWEST_IRQ_PRIORITY =
    (1UL << __NVIC_PRIO_BITS) - 1UL;

static uint32_t tick_counter = 0;

__attribute__((interrupt)) void SysTick_Handler(void) {
    tick_counter += 1;
    __conditional_software_breakpoint(tick_counter <= LIMIT);
}

static uint32_t tick_limit = 0;
static uint32_t result = 0;

__attribute__((interrupt)) void PendSV_Handler(void) {
    if (tick_counter == tick_limit) {
        tick_counter = 0;
        result = 1;
    } else {
        result = 0;
    }
}

static inline void TriggerPendSV(void) {
    SCB->ICSR = SCB_ICSR_PENDSVSET_Msk;
    __DSB();
    __ISB();
}

uint32_t TickCounterReached(uint32_t limit) {
    tick_limit = limit;
    TriggerPendSV();
    return result;
}

int main(void) {
    const uint32_t ticks =
        (SysTick->CALIB & SysTick_CALIB_TENMS_Msk);

    SysTick_Config(ticks);
    NVIC_SetPriority(SysTick_IRQn, LOWEST_IRQ_PRIORITY);
    NVIC_SetPriority(PendSV_IRQn, LOWEST_IRQ_PRIORITY);
    __DSB();

    while (1) {
        if (TickCounterReached(LIMIT)) {
            __builtin_software_breakpoint();
        }
    }

    return 0;
}
```

In order to set interrupt handlers at runtime, it is best to copy the vector table into RAM. The vector table is located via the *Vector Table Offset Register (VTOR)* in the *System Control Block (SCB)*. It is described by the `DeviceVectors` structure and is found in the variable `exception_table`. These definitions are available in the device specific header file included via `xc.h`. To set interrupt handlers dynamically, create a `DeviceVectors` structure, copy `exception_table` into it, point the `VTOR` at it, then change the handlers as desired. The actual structure definition is found in the default device startup code. It is defined as weak and is put in the `.vectors.default` section, which is mapped to the flash memory region by the default linker script. In C code, the definition is the following, with the actual function pointers elided.

```
__attribute__((section(".vectors.default"), weak, externally_visible))
const DeviceVectors exception_table = { ... }
```

Setting a handler is best done using the Interrupts and Exceptions portion of the CMSIS-Core(M) API, provided via `xc.h`. Each interrupt has an IRQ number found in the `IRQn_Type` enumeration. The name for the IRQ number is similar to the name of the handler function, except with the `_IRQn` suffix. To set the handler, use the function `NVIC_SetHandler()` giving it the IRQ number and the function address.

Alignment of the vector table is important but is specific to the device (see the appropriate architecture manual for details). For Cortex-M7 devices (such as the SAME70) the table must be aligned on a power of two greater than or equal to four times the number of exceptions, with a minimum of 128-byte alignment. At present, this must be determined manually if you create your own vector table.

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

// For a SAME70 device. Supports 90 exceptions (16 + 74).
// 90 * 4 = 360. Next highest power of 2 is 512.
#define TBL_ALIGN 512
```

```
DeviceVectors exn_table __attribute__((aligned(TBL_ALIGN)));
extern DeviceVectors exception_table;

static uint32_t counter;
__attribute__((interrupt)) void CustomSysTick_Handler(void) {
    counter += 1;
}

int main(void)
{
    memcpy(&exn_table, &exception_table, sizeof(DeviceVectors));

    // Disable interrupts, set the VTOR, ensure all data
    // operations are complete, then enable interrupts.
    __disable_irq();
    SCB->VTOR = (uint32_t) exn_table;
    __DSB();
    __enable_irq();

    NVIC_SetVector(SysTick_IRQn, (uint32_t) CustomSysTick_Handler);

    // Code continues here.
}
```

14.4 EXCEPTION HANDLERS

Cortex-M cores support exception handlers for internal events, although the specific set of internal events depends on the architecture. All internal and external exception handlers-except for the Reset handler-have a weak default implementation. This default implementation executes `__builtin_software_breakpoint()` in a debug build and goes into an infinite loop in a production build. The default implementation is a weak function called `Dummy_Handler`; you may override it if you wish to define your own default handler.

For convenience, the common set of exception handlers for internal events are briefly described below. See the appropriate architecture reference manual for the list of all internal events and how they are triggered.

14.4.1 Reset

The Reset exception is handled by the function `Reset_Handler` and has IRQ number `Reset_IRQn`. It is always enabled and has the highest priority of all interrupts. The handler is part of the C runtime start-up code and should not be altered. You can augment the reset handler by defining certain functions. See the **Chapter 15. "Main, Runtime Start-up and Reset"** for details.

14.4.2 NMI (Non-Maskable Interrupt)

The NMI exception is handled by the function `NonMaskableInt_Handler` with IRQ number `NonMaskableInt_IRQn`. It is always enabled and has a higher priority than any other interrupt except Reset. Hardware typically triggers an NMI, although software can trigger one using the following code.

```
SCB->ICSR = SCB_ICSR_NMIPENDSET_Msk;
```

14.4.3 HardFault

The HardFault exception is the generic fault mechanism, used when no other exception mechanism applies for a fault. It is handled by the function `HardFault_Handler` with IRQ number `HardFault_IRQn`. Like NMI, it is always enabled and has a higher priority than any other interrupt, except Reset and NMI.

14.4.4 SVCall

The SVCall exception is triggered by the Supervisor Call (`SVC`) instruction. Its handler is `SVCall_Handler`, its IRQ number is `SVCall_IRQn`, it is always enabled and has configurable priority.

14.4.5 PendSV

PendSV is an internal interrupt, typically used to force a context switch in software. Its handler is `PendSV_Handler` with IRQ number `PendSV_IRQn`. It is always enabled and has configurable priority, normally configured to be at the lowest priority.

You can trigger a PendSV interrupt with the following code.

```
SCB->ICSR = SCB_ICSR_PENDSVSET_Msk;
```

14.4.6 SysTick

SysTick is an internal interrupt used to handle interrupts raised by the system timer. Its handler is `SysTick_Handler` with IRQ number `SysTick_IRQn`, it is always enabled and has configurable priority. This handler can also be triggered via software using the following code.

```
SCB->ICSR = SCB_ICSR_PENDSTSET_Msk;
```

14.5 INTERRUPT SERVICE ROUTINE CONTEXT SWITCHING

As mentioned earlier, the hardware takes care of context switching on PIC32C/SAM devices. In particular, the hardware saves and restores the argument registers (r0 to r3), the IP register (r12) the link register (r14), the return address, and the program status registers. Any other registers must be preserved by the interrupt handler function; however, this is standard for any function that follows the procedure call standard.

It is possible for other exceptions to occur during the context switch. The work for this is handled in hardware, the details of which can be found in the appropriate architecture reference manual. In short, the context switching does not happen twice. Instead, the higher priority interrupt is run with the other interrupt set to pending.

14.6 LATENCY

The time between interrupt generation and the execution of the first instruction of your ISR is known as *interrupt latency*. There are two elements that affect it.

- **Processor Servicing of Interrupt** - This is the amount of time it takes the processor to recognize the interrupt and branch to the associated ISR.
- **Saving ISR Code Context** - The amount of time it takes to save registers on the stack before entering the ISR.

For the most part, these are determined solely by the hardware on PIC32C/SAM devices. In particular, the hardware saves the context on the stack, eliminating the need for the compiler to generate such code. As a result, if your ISR is written such that it does not need to save any registers beyond what is saved by the hardware, the interrupt latency for your ISR is entirely hardware dependent.

To determine the value of the interrupt latency, see the data sheet for the device and the appropriate Cortex-M technical reference manual.

14.7 ENABLING/DISABLING INTERRUPTS

The following functions from the CMSIS-Core(M) API are used to manipulate the interrupt state of the CPU:

```
__enable_irq()
__disable_irq()
```

The Nested Vector Interrupt Controller (NVIC), which controls the aspects of nearly all internal and external interrupts, can be manipulated using the CMSIS-Core(M) API. See the *Interrupts and Exceptions (NVIC)* reference section of the API for more information. The NVIC API is made available when you include the `xc.h` header.

14.8 ISR CONSIDERATIONS

There are a few things to consider when writing an interrupt service routine.

As with all compilers, limiting the number of registers used by the interrupt function, or any functions called by the interrupt function, may result in less context switch code being generated and executed by the compiler. Keeping interrupt functions small and simple will help you achieve this.

When interrupt execution speed is a concern, avoid calling other functions from your ISR. You may be able to replace a function call with a volatile flag that is handled by your application's main control loop.

NOTES:

Chapter 15. Main, Runtime Start-up and Reset

When creating C/C++ applications targeting PIC32C/SAM/CEC devices, there are specific steps required to initialize the device, core registers, and the C/C++ runtime environment after reset and before the application `main()` function is called. The XC32 compiler provides start-up code for each supported device to execute user-defined functions at various points in the start-up process. These features, as well as the general steps taken after a reset and before the `main()` function is called, are described in this section.

15.1 THE MAIN FUNCTION

The identifier `main()` is reserved as the entry point for application code. The start-up code for the device will call `main()` after performing all other initialization steps. Upon returning from the `main()` function, control returns to the start-up function and will enter an infinite loop. Calling the standard `exit` or `abort` functions will also cause execution to enter an infinite loop. The return value of `main()` is not used by the start-up function.

15.2 RUNTIME START-UP CODE

A C/C++ application requires certain initialization steps and the processor to be in a particular state before the execution of `main()` can proceed. Certain special functions may be defined to execute at specific points during these initialization steps. The start-up function supplied by the compiler to perform this initialization is called `Reset_Handler()`. This section will describe the general sequence of operations performed by the `Reset_Handler()` function.

The operations performed by `Reset_Handler()` are as follows:

1. Ensure the stack pointer (`sp`) register is initialized to point to the top of the system stack.
2. Call the function `void _on_reset()` if it is defined by the application.
3. Enable the Floating Point Unit (FPU) device if present and enabled by the compiler options.
4. Enable the instruction and data caches, if present.
5. Configure the instruction and/or data Tightly-Coupled Memory (TCM), if present and enabled (also see **Section 7.6 "Tightly-Coupled Memories"**).
6. Initialize the memory sections which must be filled with zeros, or initialized to other known values and copy sections which should be placed into TCM as needed.
7. If requested, relocate the stack to data TCM.
8. Initialize the `VTOR` (Vector Table on Reset) register to the address of the interrupt vector table.
9. Perform the initialization for the standard C library.
10. Call the function `void _on_bootstrap()` if it is defined by the application.
11. Call the application `main()` function.
12. On return from `main()`, enter an infinite loop.

Each step will be described in further detail in the following sections.

15.2.1 Initialize Stack Pointer and Heap

This step is only explicitly performed for some devices. The initial stack pointer value is defined using the symbol `_stack` which is defined by the linker to be located in data (RAM) memory. A minimum amount of stack space is reserved by defining the symbol `_min_stack_size` to a positive value, e.g. by using the linker `--defsym` option. Note that while some implementations may locate the stack base at the highest RAM address, the XC32 linker may place it elsewhere in RAM. On devices which support placing the stack in TCM with the `-mstack-in-dtcm` option, the stack pointer will be updated to the new location in TCM following TCM initialization steps (see **Section 15.2.7 "Relocate Stack to TCM"** and **Section 7.6 "Tightly-Coupled Memories"**).

15.2.2 Call the `_on_reset()` Function

The `_on_reset()` should be defined in an application if special initialization steps are required upon device reset. When implementing `_on_reset()`, one must take care, particular if writing in C, to account for the state of the device. In particular, the stack pointer will be initialized, but no data or library initialization will be performed, nor will any static constructors be called for C++ applications. References to non-automatic variables in C/C++ applications may yield unexpected or unpredictable results.

The `_on_reset()` function is useful for cases where hardware must be initialized before data is initialized. For instance, you may need to initialize a memory controller before initializing data in that memory.

15.2.3 Enable the FPU Device

On devices with a FPU and for applications where code may be generated using the Floating Point Unit, the unit will be enabled. This step is only performed if the `-mfloat-abi=hard|softfp` option is in effect at the linker step, which is the default behavior for devices which have an FPU present.

15.2.4 Configure Tightly-Coupled Memories

On devices supporting one or more TCMs, when enabled, device-specific code will be called to perform any configuration or initialization required to satisfy the requested TCM configuration (also see **Section 7.6 "Tightly-Coupled Memories"**).

15.2.5 Enable Caches

On devices supporting instruction or data cacheable memory, caches will be initialized based on definitions in device-specific files controlled by the `-mprocessor` option.

15.2.6 Data Initialization

The internal library function `__pic32c_data_initialization()` is called to perform any required data initialization based on the contents of the linker-generated `.dinit` section, as well as clearing any uninitialized memory (e.g. the `.bss` and `.sbss` sections) as needed.

15.2.7 Relocate Stack to TCM

With the `-mstack-in-itcm` or similar options, the runtime stack may be placed into TCM at this point. Following this step, the runtime stack will be located in the requested memory region.

15.2.8 Set VTOR Register

The VTOR, or Vector Table Offset Register, on Arm Cortex-M MCUs, is set to reflect the starting address of the Interrupt Vector Table (IVT). This value is determined by the special symbol __svectors defined by the XC32 linker.

15.2.9 C Library Initialization

The function __libc_init_array() is called to perform all initialization required by the standard C library. Before this step, standard C library routines may produce unexpected results.

15.2.10 Call the _on_bootstrap() Function

The _on_bootstrap() should be defined in an application if special initialization steps are required after memory, CPU and library initialization is done but before main() is called. Unlike _on_reset(), this function may be implemented in C with no caveats.

15.2.11 Call the Main Function

The main() function is called, with any return value unused. Following the return from main(), control will return to Reset_Handler() and execution will enter an infinite loop. On devices which support the Thumb-2 instruction set, the preprocessor macro __DEBUG may be defined to insert a software breakpoint instruction (BKPT) immediately after the return from main().

15.2.12 Exception Handlers

For devices based on Arm Cortex-M cores, an exception table is defined (placed in the section .vectors), containing the initial stack pointer and start address (e.g. the Reset_Handler() function address) as well as the interrupt service routine (ISR) vector. The device-specific start-up code defines a default vector table exception_table, as well as a default ISR named _Dummy_Handler(). Apart from the Reset_Handler(), all pointers in exception_table are initialized to point to _Dummy_Handler(), which simply enters an infinite loop. For devices supporting the Thumb-2 instruction set, a software breakpoint instruction will be inserted before the infinite loop when __DEBUG is defined.

The symbols exeception_table, Reset_Handler() and _Dummy_Handler() may be redefined by user code to provide custom implementations.

NOTES:

Chapter 16. Library Routines

16.1 USING LIBRARY ROUTINES

Library functions or routines (and any associated variables) will be automatically linked into a program once they have been referenced in your source code. The use of a function from one library file will not include any other functions from that library. Only used library functions will be linked into the program output and consume memory.

Your program will require declarations for any functions or symbols used from libraries. These are contained in the standard C header (`.h`) files. Header files are not library files and the two files types should not be confused. Library files contain precompiled code, typically functions and variable definitions; the header files provide declarations (as opposed to definitions) for functions, variables and types in the library files, as well as other preprocessor macros.

```
#include <math.h>    // declare function prototype for sqrt

int main(void)
{
  double i;

  // sqrt referenced; sqrt will be linked in from library file
  i = sqrt(23.5);
}
```

MPLAB® Harmony includes a set of peripheral libraries, drivers, and system services that are readily accessible for application development. For access to the `plib.h` (peripheral header files), go to the Microchip web site (www.microchip.com), click on the **Design** tab, then click on **Software** and download MPLAB Harmony and MPLAB Code Configurator. The path to the installed peripheral libraries is:

For Windows: `C:\microchip\harmony\<version>\framework\peripheral`

For Mac/Linux: `~\microchip\harmony\<version>\framework\peripheral`

NOTES:

Chapter 17. Mixing C/C++ and Assembly Language

Assembly language code can be mixed with C/C++ code using two different techniques: writing assembly code and placing it into a separate assembler module, or including it as in-line assembly in a C/C++ module.This section describes how to use assembly language and C/C++ modules together. It gives examples of using C/C++ variables and functions in assembly code and examples of using assembly language variables and functions in C/C++.

The more assembly code a project contains, the more difficult and time consuming its maintenance will be. As the project is developed, the compiler may work in different ways as some optimizations look at the entire program. The assembly code is more likely to fail if the compiler is updated due to differences in the way the updated compiler may work. These factors do not affect code written in C/C++

Note: If assembly must be added, it is preferable to write this as self-contained routine in a separate assembly module rather than in-lining it in C code.

17.1 MIXING ASSEMBLY LANGUAGE AND C VARIABLES AND FUNCTIONS

The following guidelines indicate how to interface separate assembly language modules with C modules.

- Follow the register conventions described in Section 12.2 "Register Conventions". In particular, registers r0-r3 are used for parameter passing. An assembly language function will receive parameters and should pass arguments to called functions, in these registers.
- Table 12-1 describes which registers must be saved across non-interrupt function calls
- Interrupt functions must preserve all registers. Unlike a normal function call, an interrupt may occur at any point during the execution of a program. When returning to the normal program, all registers must be as they were before the interrupt occurred.
- Variables or functions declared within a separate assembly file that will be referenced by any C source file should be declared as global using the assembler directive `.global`. Undeclared symbols used in assembly files will be treated as externally defined.

The following example shows how to use variables and functions in both assembly language and C regardless of where they were originally defined.

The file `ex1.c` defines `cFunction` and `cVariable` to be used in the assembly language file. The C file also shows how to call an assembly function, `asmFunction`, and how to access the assembly defined variable, `asmVariable`.

EXAMPLE 17-1: MIXING C AND ASSEMBLY

```
        .syntax unified
        .cpu cortex-m7
        .thumb

        .global asmVariable
        .type   asmVariable,%object

        .data
        .align  2
asmVariable:
        .space  4

        @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        @ char *asmFunction (char *s)
        @ {
        @   asmVariable = 0;
        @   if (s) {
        @     char *d = s, c;
        @     while ((c = *d)) {
        @       if (cFunction (c)) {
        @         *d = c & cVariable;
        @         ++asmVariable;
        @       }
        @       ++d;
        @     }
        @   }
        @   return s;
        @ }
        @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        .global asmFunction
        .type   asmFunction,%function

        .text
        .align  1

        .thumb_func
asmFunction:
        @ if the input string is not NULL
        cbnz     r0, .L_not_NULL

        @ set 'asmVariable' to zero and return
        ldr      r1, =asmVariable
        str      r0, [r1]
        bx       lr

.L_not_NULL:
        @ r4-r7 are callee-saved registers
        @ LR contains the return address
        @ r0 is the first argument and also the return value of the
function
        push    {r0, r4-r7, lr}

        @ d = s;
        mov     r4, r0

        @ r6 - the value of the C variable (8-bit AND mask)
        @ r7 - counter of changed chars
        ldr     r6, =cVariable
        movs    r7, #0
        ldrb    r6, [r6]
```

```
.L_while:
        @ while ((c = *d))
        ldrb    r5, [r4]
        cbz     r5, .L_end_while

        @ if (cFunction (c))
        mov     r0, r5
        bl      cFunction
        cbz     r0, .L_next_char

        @ *d = c & cVariable;
        ands    r5, r5, r6
        strb    r5, [r4]

        @ ++ the number of changed chars
        adds    r7, r7, #1

.L_next_char:
        @ ++d;
        adds    r4, r4, #1
        b       .L_while

.L_end_while:
        @ write the no. of changes to 'asmVariable'
        ldr     r0, =asmVariable
        str     r7, [r0]

        @ return s;
        pop     {r0, r4-r7, pc}

        .pool
        .size asmFunction, .-asmFunction
```

The file `ex1.S` defines `asmFunction` and `asmVariable` as required for use in a linked application. The assembly file also shows how to call a C function, `cFunction`, and how to access a C defined variable, `cVariable`.

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

extern int asmVariable;
extern char *asmFunction (char *s);

char cVariable = 0xDF;

char cFunction (char c)
{
    return c >= 'a' && c <= 'z';
}

int main()
{
    char s[] = "heLLo, wOrlD!";
    printf ("%s\n", s);

    char *d = asmFunction (s);
    printf ("%s\nchanges: %d", d, asmVariable);

    return 0;
}
```

In the C file, `ex2.c`, although for the function declaration this isn't required, note that `asmFunction` is a `char *` function and is declared accordingly.

In the assembly file, `ex1.S`, the symbols `asmFunction` and `asmVariable` are made globally visible through the use of the `.global` assembler directive and can be accessed by any other source file.

17.2 USING INLINE ASSEMBLY LANGUAGE

Within a C/C++ function, the `asm` statement may be used to insert a line of assembly language code into the assembly language that the compiler generates. Inline assembly has two forms: simple and extended.

In the **simple** form, the assembler instruction is written using the syntax:

```
asm ("instruction");
```

where `instruction` is a valid assembly-language construct. If you are writing inline assembly in ANSI C programs, write `__asm__` instead of `asm`.

> **Note:** Only a single string can be passed to the simple form of inline assembly.

In an **extended** assembler instruction using `asm`, the operands of the instruction are specified using C/C++ expressions. The extended syntax is:

```
asm("template" [ : [ "constraint"(output-operand) [ , ... ] ]
                  [ : [ "constraint"(input-operand) [ , ... ] ]
                       [ "clobber" [ , ... ] ]
                ]
              ]);
```

You must specify an assembler instruction `template`, plus an operand `constraint` string for each operand. The `template` specifies the instruction mnemonic, and optionally placeholders for the operands. The `constraint` strings specify operand constraints, for example, that an operand must be in a register (the usual case), or that an operand must be an immediate value.

Constraint letters and modifiers supported by the compiler are listed in Table 17-1 through Table 17-3.

TABLE 17-1: REGISTER CONSTRAINT LETTERS SUPPORTED BY THE COMPILER

Letter	Constraint
`l`	In Thumb State, the core registers r0-r7. In Arm state, this is an alias for the 'r' constraint.
`h`	In Thumb state, the core registers r8-r15.
`t`	In Arm/Thumb-2 state, the VFP floating-point registers s0-s31.
`w`	In Arm/Thumb-2 state, the VFP floating-point registers d0-d15, or d0-d31 for VFPv3.
`x`	In Arm/Thumb-2 state, the VFP floating-point registers d0-d7.
`Ts`	If `-mrestrict-it` is specified (for Arm-v8), the core registers r0-r7. Otherwise, `GENERAL_REGS` (r0-r12 and r14).

TABLE 17-2: INTEGER CONSTRAINT LETTERS SUPPORTED BY THE COMPILER

Letter	Constraint
`G`	In Arm/Thumb-2 state, the floating-point constant 0.
`I`	In Arm /Thumb-2 state, a constant that can be used as an immediate value in a Data Processing instruction (that is, an integer in the range 0 to 255 rotated by a multiple of 2). In Thumb-1 state, a constant in the range 0..255.
`j`	In Arm /Thumb-2 state, a constant suitable for a `MOVW` instruction.
`J`	In Arm /Thumb-2 state, a constant in the range -4095..4095. In Thumb-1 state, a constant in the range -255..-1.
`K`	In Arm /Thumb-2 state, a constant that satisfies the 'I' constraint if inverted (one's complement). In Thumb-1 state, a constant that satisfies the 'I' constraint multiplied by any power of 2.
`L`	In Arm /Thumb-2 state, a constant that satisfies 'I' constraint if negated (two's complement). In Thumb-1 state, a constant in the range -7..7.
`M`	In Thumb-1 state, a constant that is a multiple of 4 in the range 0..1020.
`N`	In Thumb-1 state, a constant in the range 0-31.
`O`	In Thumb-1 state, a constant that is a multiple of 4 in the range -508..508.
`Pf`	Memory models except relaxed, consume or release ones.

TABLE 17-3: CONSTRAINT MODIFIERS SUPPORTED BY THE COMPILER

Letter	Constraint
`=`	Means that this operand is write-only for this instruction: the previous value is discarded and replaced by output data.
`+`	Means that this operand is both read and written by the instruction
`&`	Means that this operand is an `earlyclobber` operand, which is modified before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is used as an input operand or as part of any memory address

17.2.1 Examples:

- Insert Bit Field
- Multiple Assembler Instructions

17.2.1.1 INSERT BIT FIELD

This example demonstrates how to use the `BFI` instruction to insert a bit field into a 32-bit wide variable. This function-like macro uses inline assembly to emit the `BFI` instruction, which is not commonly generated from C/C++ code.

```
/* Thumb2 insert bits */
#define _ins(tgt,val,pos,sz) __extension__({                \
    unsigned int __t = (tgt), __v = (val);                  \
    __asm__ ("bfi\t%0,%1,%2,%3"                /* template */ \
             : "+r" (__t)                       /* output   */ \
             : "r" (__v), "M" (pos), "M" (sz)); /* input    */ \
    __t;                                                      \
})
```

Here `__v`, `pos`, and `sz` are input operands. The `__v` operand is constrained to be of type 'r' (a register). The `pos` and `sz` operands are constrained to be of type 'M' (a constant in the range 0-32 or any power of 2).

The `__t` output operand is constrained to be of type 'r' (a register). The '+' modifier means that this operand is both read and written by the instruction and so the operand is both an input and an output.

The following example shows this macro in use.

```
unsigned int result;
void example (void)
{
    unsigned int insertval = 0x12;
    result = 0xAAAAAAAAu;
    result = _ins(result, insertval, 4, 8);
    /* result is now 0xAAAAA12A */
}
```

For this example, the compiler may generate assembly code similar to the following.

```
        movs    r2, #18             @ 0x12
        mov     r3, #-1431655766    @ 0xaaaaaaaa

        bfi     r3,r2,#4,#8         @ inline assembly

        ldr     r2, .L2             @ load result address
        str     r3, [r2]            @ assign the result
        bx      lr                  @ return
        ...
  .L2:
        .word   result
```

17.2.1.2 MULTIPLE ASSEMBLER INSTRUCTIONS

This example demonstrates how to use a couple of `REV` instructions to perform a 64-bit byte swap. The `REV` instruction is swapping (reversing the order of) the bytes in a 32-bit word. This function-like macro uses inline assembly to create a “byte-swap double word” using instructions that are not commonly generated from C/C++ code. However, the same functionality can be gained by using one of the GCC built-in functions, `__builtin_bswap64()`. As a general rule, built-ins should be preferred over inline assembly, whenever possible.

The following shows the definition of the function-like macro, `_bswapdw`.

```
/* Thumb2 byte-swap double word */
#define _bswapdw(val) __extension__({             \
  union { uint32_t i[2]; uint64_t l; } __i, __o; \
  __i.l = (val);                                  \
  __asm__ ("rev\t%0, %3\n\t"                      \
           "rev\t%1, %2"      /* template */      \
           : "=&r" (__o.i[0]), "=r" (__o.i[1])    \
           : "r"  (__i.i[0]), "r"  (__i.i[1]));   \
  __o.l; \
})
```

A union is used to reference the two 32-bit halves of a 64-bit integer. For example, the C expressions for the input operands are `'__i.i[0]'` and `'__i.i[1]'` and the ones for the output operands are `'__o.i[0]'` and `'__o.i[1]'`, respectively.

All operands use the constraint 'r' (32-bit register). To be noted the '&' modifier for operand 0, indicating that it is an "early-clobber" (written before all the input operands are consumed, with the implication that the compiler will allocate a register different that the input ones). This is needed because the 32-bit halves themselves need to be swapped.

The function-like macro is shown in the following example assigning to `result` the content of `value`, swapped.

```
uint64_t result;
int example (void)
{
   uint64_t value = 0x0123456789ABCDEFull;
   result = _bswapdw (value);
   /* result == 0xEFCDAB8967452301 */
}
```

The compiler may generate assembly code similar to the following for this example:

```
 ldr r2, .L6           @ r2 = 0x01234567
 ldr r3, .L6+4         @ r3 = 0x89ABCDEF

 rev r1, r2            @ from inline asm
 rev r3, r3            @ from inline asm

 ldr r2, .L6+8         @ r2 = address of 'result'
 stm r2, {r1, r3}      @ store value to 'result'
 bx lr               @ return
 ...
 .align 2
.L6:
 .word 19088743      @ 0x01234567
 .word -1985229329   @ 0x89ABCDEF
 .word result
```

17.2.2 Equivalent Assembly Symbols

C/C++ symbols can be accessed directly with no modification in extended assembly code.

17.3 PREDEFINED MACRO

There is one predefined macro available once you include <xc.h>. It is `_nop()`. This macro inserts a `nop` instruction.

NOTES:

Chapter 18. Optimizations

Different MPLAB XC32 C/C++ Compiler editions support different levels of optimization. Some editions are free to download and others must be purchased. Visit http://www.microchip.com/MPLABXC compilers for more information on C and C++ licenses.

The compiler editions are:

Edition	Cost	Description
Professional (PRO)	Yes	Implemented with the highest optimizations and performance levels.
Free	No	Implemented with the most code optimizations restrictions.
Evaluation (EVAL)	No	PRO edition enabled for 60 days and then reverts to Free edition.

Setting Optimization Levels

Different optimizations may be set ranging from no optimization to full optimization, depending on your compiler edition. When debugging code, you may wish to not optimize your code to ensure expected program flow.

For details on compiler options used to set optimizations, see Section 5.8.7 "Options for Controlling Optimization".

NOTES:

Chapter 19. Preprocessing

All C/C++ source files are preprocessed before compilation. Assembly source files that use the .S extension (upper case) are also preprocessed. A large number of options control the operation of the preprocessor and preprocessed code (see **Section 5.8.8 "Options for Controlling the Preprocessor"**).

19.1 PREPROCESSOR DIRECTIVES

MPLAB XC32 C/C++ Compiler accepts all the standard preprocessor directives, which are listed in Table 19-1.

TABLE 19-1: PREPROCESSOR DIRECTIVES

Directive	Meaning	Example
`#`	Preprocessor null directive, do nothing	`#`
`#assert`	Generate error if condition false	`#assert SIZE > 10`

`#define`	Define preprocessor macro	`#define SIZE 5`
		`#define FLAG`
		`#define add(a,b) ((a)+(b))`

`#elif`	Short for `#else #if`	`see #ifdef`
`#else`	Conditionally include source lines	`see #if`
`#endif`	Terminate conditional source inclusion	`see #if`
`#error`	Generate an error message	`#error Size too big`

`#if`	Include source lines if constant expression true	`#if SIZE < 10`
		`  c = process(10)`
		`#else`
		`  skip();`
		`#endif`

`#ifdef`	Include source lines if preprocessor symbol defined	`#ifdef FLAG`
		`  do_loop();`
		`#elif SIZE == 5`
		`  skip_loop();`
		`#endif`

`#ifndef`	Include source lines if preprocessor symbol not defined	`#ifndef FLAG`
		`  jump();`
		`#endif`

`#include`	Include text file into source	`#include <stdio.h>`
		`#include "project.h"`

`#line`	Specify line number and file name for listing	`#line 3 final`
`#`*nn*	(Where *nn* is a number) short for `#line` *nn*	`#20`
`#pragma`	Compiler specific options	Refer to **Section 19.3 "Pragma Directives"**
`#undef`	Undefines preprocessor symbol	`#undef FLAG`
`#warning`	Generate a warning message	`#warning Length not set`

Macro expansion using arguments can use the `#` character to convert an argument to a string, and the `##` sequence to concatenate arguments. If two expressions are being concatenated, consider using two macros in case either expression requires substitution itself, so for example,

```
#define   paste1(a,b)   a##b
#define   paste(a,b)    paste1(a,b)
```

lets you use the `paste` macro to concatenate two expressions that themselves may require further expansion. The replacement token is rescanned for more macro identifiers, but remember that once a particular macro identifier has been expanded, it will not be expanded again if it appears after concatenation.

The type and conversion of numeric values in the preprocessor domain is the same as in the C domain. Preprocessor values do not have a type, but acquire one as soon as they are converted by the preprocessor. Expressions may overflow their allocated type in the same way that C expressions may overflow.

Overflow may be avoided by using a constant suffix. For example, an `L` after the number indicates it should be interpreted as a long once converted.

So, for example:

```
#define MAX 100000*100000
```

and

```
#define MAX 100000*100000L
```

(note the `L` suffix) will define the values `0x540be400` and `0x2540be400`, respectively.

19.2 C/C++ LANGUAGE COMMENTS

A C/C++ comment is ignored by the compiler and can be used to provide information to someone reading the source code. They should be used freely.

Comments may be added by enclosing the desired characters within `/*` and `*/`. The comment can run over multiple lines, but comments cannot be nested. Comments can be placed anywhere in C/C++ code, even in the middle of expressions, but cannot be placed in character constants or string literals.

Since comments cannot be nested, it may be desirable to use the `#if` preprocessor directive to comment out code that already contains comments, for example:

```
#if 0
   result = read();  /* TODO: Jim, check this function is right */
#endif
```

Single-line, C++ style comments may also be specified. Any characters following `//` to the end of the line are taken to be a comment and will be ignored by the compiler, as shown below:

```
   result = read();  // TODO: Jim, check this function is right
```

19.3 PRAGMA DIRECTIVES

The `#pragma` directive may be used to modify the behavior of the compiler. The general format of a pragma directive is:

```
#pragma [GCC] keyword options
```

where `keyword` is one of a set of supported keywords, some of which may be followed by a number of `options`.

Certain keywords must be preceded by `GCC` indicating that the keyword is a GCC extension. Any keyword not understood by the compiler will be ignored. The keywords supported for PIC32C/SAM devices are given below.

19.3.1 Pragmas to Control Function Attributes

`#pragma long_calls`

Set all functions following the pragma to have the `long_call` function attribute.

`#pragma no_long_calls`

Set all functions following the pragma to have the `short_call` attribute.

`#pragma long_calls_off`

Disable the effect of any preceding `long_calls` or `long_calls_off` pragma, so that following functions will not have any `long_call` or `short_call` attribute implicitly set.

19.3.2 Pragmas to Control Options/Optimization

`#pragma GCC target ("string" ...)`

This pragma may be used to set target-specific options for all subsequent function definitions. The arguments allowed are any options prefixed with `-m`, such that `-m` will be prepended to each string given to form the target options, i.e., `#pragma` GCC target `("arch=armv7e-m")`. All function definitions following this pragma will behave as if the attribute `((target("string"))` were applied to the definition. The parentheses are optional.

`#pragma GCC optimize ("string" ...)`

This pragma may be used to set optimization options for all subsequent function definitions. The arguments allowed may be:

- A number `n`, to be interpreted as an optimization level, i.e., the `-On` option
- A string beginning with `O`, which is interpreted as an optimization option, i.e., `-Ostring`
- Otherwise, `string` should be an option with the prefix `-f`.

All function definitions following the pragma behave as if the attribute `((optimize("string"))` were specified for the definition. The parentheses are optional.

`#pragma GCC push_options`

`#pragma GCC pop_options`

These pragmas allow for maintaining a stack of `target` and `optimize` options. The `push_options` pragma will push the current options onto the stack, which will be the command-line options if no `target` or `optimize` pragma are in effect. The `pop_options` will restore the options in effect to those last pushed onto the stack.

`#pragma GCC reset_options`

Clears any current options set via the `target` or `optimize` pragmas for all subsequent function definitions.

19.3.3 MPLAB XC32 Pragmas

The following pragma directives are specific to the MPLAB XC32 compiler.

`#pragma config identifier = value`

The `config` pragma allows for the setting of device-specific configuration bits for an application. See **Section 7.4 "Configuration Bit Access"** for a description of the syntax for the `config` options.

19.4 PREDEFINED MACROS

These are the predefined 32-bit C/C++ compiler macros available for use with the compiler.

The compiler provides a number of macro definitions which characterize the various target-specific options and other aspects of the compiler and host environment.

TABLE 19-2: MACRO DEFINITIONS

Macro	Meaning
`__PIC__` `__pic__`	The translation unit is being compiled for position independent code.
`__PIC32C` `__PIC32C__`	Defined when a PIC32CX device is specified with the `-mprocessor` option. Always defined when targeting a PIC32C/SAM device.
`__PIC32CZ`	Defined when a PIC32CZ device is specified with the `-mprocessor` option.

`__LANGUAGE_ASSEMBLY`
`__LANGUAGE_ASSEMBLY__`
`_LANGUAGE_ASSEMBLY`
	Defined if compiling a pre-processed assembly file (.S files).

`LANGUAGE_ASSEMBLY`	Defined if compiling a pre-processed assembly file (.S files) and `-ansi` is not specified.

`__LANGUAGE_C`
`__LANGUAGE_C__`
`_LANGUAGE_C`
	Defined if compiling a C file.

`LANGUAGE_C`	Defined if compiling a C file and `-ansi` is not specified.

`__LANGUAGE_C_PLUS_PLUS`
`__cplusplus`
`_LANGUAGE_C_PLUS_PLUS__`
	Defined if compiling a C++ file.

`__EXCEPTIONS`	Defined if C++ exceptions are enabled.
`__GXX_RTTI`	Defined if runtime type information is enabled.
`__processor__`	Where "processor" is the capitalized argument to the `-mprocessor` option. For example, `-mprocessor=32CX0525SG12144` will define `__32CX05255SG12144__`
`__XC`	Always defined to indicate that this is a Microchip XC compiler.
`__XC32`	Always defined to indicate this the XC32 compiler.

TABLE 19-2: MACRO DEFINITIONS (CONTINUED)

Macro	Meaning
`__VERSION__`	The `__VERSION__` macro expands to a string constant describing the compiler in use. Do not rely on its contents having any particular form, but it should contain at least the release number. Use the `__XC32_VERSION` macro for a numeric version number.
`__XC32_VERSION` or `__C32_VERSION__`	The C compiler defines the constant `__XC32_VERSION`, giving a numeric value to the version identifier. This macro can be used to construct applications that take advantage of new compiler features while still remaining backward compatible with older versions. The value is based upon the major and minor version numbers of the current release. For example, release version 1.03 will have a `__XC32_VERSION` definition of 1030. This macro can be used, in conjunction with standard preprocessor comparison statements, to conditionally include/exclude various code constructs.
`__arm__`	Defined when compiling for ARM architectures, regardless of whether generating Thumb or ARM code.
`__thumb__`	Defined to indicate the compiler is generating Thumb code. This definition is subject to the `-mthumb` and `-marm` options.
`__thumb2__`	Defined when generating Thumb code for a target processor supporting the Thumb-2 instruction set.
`__SOFTFP__`	Defined when compiling for software floating-point, i.e. when `-mfloat-abi=soft` is in effect.
`__ARM_FP`	Defined to an integer mask describing the floating-point capability of the current target processor. This is 0 when software floating point is in effect. Otherwise, bits 1, 2 and 3 of the mask are set to indicate support for 16, 32 and 64-bit hardware floating point, respectively.

See also the device-specific include files (pic32c/include/proc/p32*.h) for other macros that can be used to determine the features available on the selected device. You will find these macros near the end of the header file.

NOTES:

Chapter 20. Linking Programs

See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide* (DS50002186) for more detailed information on the linker.

The compiler will automatically invoke the linker unless the compiler has been requested to stop after producing an intermediate file.

Linker scripts are used to specify the available memory regions and where sections should be positioned in those regions.

The linker creates a map file which details the memory assigned to sections. The map file is the best place to look for memory information.

20.1 REPLACING LIBRARY SYMBOLS

Unlike with the Microchip MPLAB XC8 compiler, not all library functions can be replaced with user-defined routines using MPLAB XC32 C/C++ Compiler. Only weak library functions (see Section 8.11 "Variable Attributes") can be replaced in this way. For those that are weak, any function you write in your code will replace an identically named function in the library files.

20.2 LINKER-DEFINED SYMBOLS

The 32-bit linker defines several symbols that can be used in your C code development. Please see the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide* (DS50002186) for more information.

The linker defines the symbols `_ramfunc_begin` and `_bmxdkpba_address`, which represent the starting address in RAM where ram functions will be accessed, and the corresponding address in the program memory from which the functions will be copied. They are used by the default runtime start-up code to initialize the bus matrix if ram functions exist in the project (see Section 13.3 "Allocation of Function Code").

The linker also defines the symbol `_stack`, which is used by the runtime start-up code to initialize the stack pointer. This symbol represents the starting address for the software stack.

All the above symbols are rarely required for most programs, but may assist you if you are writing your own runtime start-up code.

NOTES:

Appendix A. Embedded Compiler Compatibility Mode

Since very different device architectures may be targeted by other compilers, the semantics of the non-standard extensions may be different to that in the MPLAB XC compilers. This document indicates when the original C code may need to be reviewed.

A.1 COMPILING IN COMPATIBILITY MODE

An option is used to enable vendor-specific syntax compatibility. When using MPLAB XC8, this option is `--ext=vendor`; when using MPLAB XC16 or MPLAB XC32, the option is `-mext=vendor`. The argument *`vendor`* is a key that is used to represent the syntax. See Table A-1 for a list of all keys usable with the MPLAB XC compilers.

TABLE A-1: VENDOR KEYS

Vendor key	Syntax	XC8 Support	XC16 Support	XC32 Support
`cci`	Common Compiler Interface	Yes	Yes	Yes
`iar`	IAR C/C++ Compiler™ for Arm	Yes	Yes	Yes

The Common Compiler Interface is a language standard that is common to all Microchip MPLAB XC compilers. The non-standard extensions associated with this syntax are already described in **Chapter 2. "Common C Interface"** and are not repeated here.

A.2 SYNTAX COMPATIBILITY

The goal of this syntax compatibility feature is to ease the migration process when porting source code from other C compilers to the native MPLAB XC compiler syntax.

Many non-standard extensions are not required when compiling for Microchip devices and, for these, there are no equivalent extensions offered by MPLAB XC compilers. These extensions are then simply ignored by the MPLAB XC compilers, although a warning message is usually produced to ensure that you are aware of the different compiler behavior. You should confirm that your project will still operate correctly with these features disabled.

Other non-standard extensions are not compatible with Microchip devices. Errors will be generated by the MPLAB XC compiler if these extensions are not removed from the source code. You should review the ramifications of removing the extension and decide whether changes are required to other source code in your project.

Table A-2 indicates the various levels of compatibility used in the tables that are presented throughout this guide.

TABLE A-2: LEVEL OF SUPPORT INDICATORS

Level	Explanation
support	The syntax is accepted in the specified compatibility mode, and its meaning will mimic its meaning when it is used with the original compiler.
support (no args)	In the case of pragmas, the base pragma is supported in the specified compatibility mode, but the arguments are ignored.
native support	The syntax is equivalent to that which is already accepted by the MPLAB XC compiler, and the semantics are compatible. You can use this feature without a vendor compatibility mode having been enabled.
ignore	The syntax is accepted in the specified compatibility mode, but the implied action is not required or performed. The extension is ignored and a warning will be issued by the compiler.
error	The syntax is not accepted in the specified compatibility mode. An error will be issued and compilation will be terminated.

Note that even if a C feature is supported by an MPLAB XC compiler, addresses, register names, assembly instructions, or any other device-specific argument is unlikely to be valid when compiling for a Microchip device. Always review code which uses these items in conjunction with the data sheet of your target Microchip device.

A.3 DATA TYPE

Some compilers allow use of the boolean type, `bool`, as well as associated values `true` and `false`, as specified by the C99 ANSI Standard. This type and these values may be used by all MPLAB XC compilers when in compatibility mode[1], as shown in Table A-3.

As indicated by the ANSI Standard, the `<stdbool.h>` header must be included for this feature to work as expected when it is used with MPLAB XC compilers.

TABLE A-3: SUPPORT FOR C99 BOOL TYPE

IAR Compatibility Mode			
Type	**XC8**	**XC16**	**XC32**
`bool`	support	support	support

Do not confuse the boolean type, `bool`, and the integer type, `bit`, implemented by MPLAB XC8.

A.4 OPERATOR

The `@` operator may be used with other compilers to indicate the desired memory location of an object. As Table A-4 indicates, support for this syntax in MPLAB C is limited to MPLAB XC8 only.

Any address specified with another device is unlikely to be correct on a new architecture. Review the address in conjunction with the data sheet for your target Microchip device.

Using `@` in a compatibility mode with MPLAB XC8 will work correctly, but will generate a warning. To prevent this warning from appearing again, use the reviewed address with the MPLAB C `__at()` specifier instead.

1. Not all C99 features have been adopted by all Microchip MPLAB XC compilers.

For MPLAB XC16/32, consider using the `address` attribute.

TABLE A-4: SUPPORT FOR NON-STANDARD OPERATOR

IAR Compatibility Mode			
Operator	**XC8**	**XC16**	**XC32**
`@`	native support	error	error

A.5 EXTENDED KEYWORDS

Non-standard extensions often specify how objects are defined or accessed. Keywords are usually used to indicate the feature. The non-standard C keywords corresponding to other compilers are listed in Table A-5, as well as the level of compatibility offered by MPLAB XC compilers. The table notes offer more information about extensions.

TABLE A-5: SUPPORT FOR NON-STANDARD KEYWORDS

IAR Compatibility Mode			
Keyword	**XC8**	**XC16**	**XC32**
`__section_begin`	ignore	support	support
`__section_end`	ignore	support	support
`__section_size`	ignore	support	support
`__segment_begin`	ignore	support	support
`__segment_end`	ignore	support	support
`__segment_size`	ignore	support	support
`__sfb`	ignore	support	support
`__sfe`	ignore	support	support
`__sfs`	ignore	support	support
`__asm or asm(`[1]	support[2]	native support	native support
`__arm`	ignore	ignore	ignore
`__big_endian`	error	error	error
`__fiq`	support	error	error
`__intrinsic`	ignore	ignore	ignore
`__interwork`	ignore	ignore	ignore
`__irq`	support	error	error
`__little_endian`[3]	ignore	ignore	ignore
`__nested`	ignore	ignore	ignore
`__no_init`	support	support	support
`__noreturn`	ignore	support	support
`__ramfunc`	ignore	ignore	support[4]
`__packed`	ignore[5]	support	support
`__root`	ignore	support	support
`__swi`	ignore	ignore	ignore
`__task`	ignore	support	support
`__weak`	ignore	support	support
`__thumb`	ignore	ignore	ignore
`__farfunc`	ignore	ignore	ignore
`__huge`	ignore	ignore	ignore
`__nearfunc`	ignore	ignore	ignore
`__inline`	support	native support	native support

Note 1: All assembly code specified by this construct is device-specific and will need review when porting to any Microchip device.

2: The keyword, asm, is supported natively by MPLAB XC8, but this compiler only supports the `__asm` keyword in IAR compatibility mode.

3: This is the default (and only) endianism used by all MPLAB XC compilers.

4: When used with MPLAB XC32, this must be used with the `__longcall__` macro for full compatibility.

5: Although this keyword is ignored, by default, all structures are packed when using MPLAB XC8, so there is no loss of functionality.

A.6 INTRINSIC FUNCTIONS

Intrinsic functions can be used to perform common tasks in the source code. The MPLAB XC compilers' support for the intrinsic functions offered by other compilers is shown in Table A-6.

TABLE A-6: SUPPORT FOR NON-STANDARD INTRINSIC FUNCTIONS

IAR Compatibility Mode			
Function	**XC8**	**XC16**	**XC32**
__disable_fiq[(1)]	support	ignore	ignore
__disable_interrupt	support	support	support
__disable_irq[(1)]	support	ignore	ignore
__enable_fiq[(1)]	support	ignore	ignore
__enable_interrupt	support	support	support
__enable_irq[(1)]	support	ignore	ignore
__get_interrupt_state	ignore	support	support
__set_interrupt_state	ignore	support	support

Note 1: These intrinsic functions map to macros which disable or enable the global interrupt enable bit on 8-bit PIC® devices.

The header file `<xc.h>` must be included for supported functions to operate correctly.

A.7 PRAGMAS

Pragmas may be used by a compiler to control code generation. Any compiler will ignore an unknown pragma, but many pragmas implemented by another compiler have also been implemented by the MPLAB XC compilers in compatibility mode. Table A-7 shows the pragmas and the level of support when using each of the MPLAB XC compilers.

Many of these pragmas take arguments. Even if a pragma is supported by an MPLAB XC compiler, this support may not apply to all of the pragma's arguments. This is indicated in the table.

TABLE A-7: SUPPORT FOR NON-STANDARD PRAGMAS

IAR Compatibility Mode			
Pragma	**XC8**	**XC16**	**XC32**
`bitfields`	ignore	ignore	ignore
`data_alignment`	ignore	support	support
`diag_default`	ignore	ignore	ignore
`diag_error`	ignore	ignore	ignore
`diag_remark`	ignore	ignore	ignore
`diag_suppress`	ignore	ignore	ignore

TABLE A-7: SUPPORT FOR NON-STANDARD PRAGMAS (CONTINUED)

IAR Compatibility Mode			
Pragma	**XC8**	**XC16**	**XC32**
`diag_warning`	ignore	ignore	ignore
`include_alias`	ignore	ignore	ignore
`inline`	support (no args)	support (no args)	support (no args)
`language`	ignore	ignore	ignore
`location`	ignore	support	support
`message`	support	native support	native support
`object_attribute`	ignore	ignore	ignore
`optimize`	ignore	native support	native support
`pack`	ignore	native support	native support
`__printf_args`	support	support	support
`required`	ignore	support	support
`rtmodel`	ignore	ignore	ignore
`__scanf__args`	ignore	support	support
`section`	ignore	support	support
`segment`	ignore	support	support
`swi_number`	ignore	ignore	ignore
`type_attribute`	ignore	ignore	ignore
`weak`	ignore	native support	native support

NOTES:

Appendix B. Implementation-Defined Behavior

B.1 OVERVIEW

ISO C requires a conforming implementation to document the choices for behaviors defined in the standard as "implementation-defined." The following sections list all such areas, the choices made for the compiler and the corresponding section number from the ISO/IEC 9899:1999 standard.

B.2 TRANSLATION

ISO Standard:	"How a diagnostic is identified (3.10, 5.1.1.3)."
Implementation:	All output to `stderr` is a diagnostic.
ISO Standard:	"Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2)."
Implementation:	Each sequence of whitespace is replaced by a single character.

B.3 ENVIRONMENT

ISO Standard:	"The name and type of the function called at program start-up in a freestanding environment (5.1.2.1)."
Implementation:	`int main (void);`
ISO Standard:	"The effect of program termination in a freestanding environment (5.1.2.1)."
Implementation:	An infinite loop (branch to self) instruction will be executed.
ISO Standard:	"An alternative manner in which the `main` function may be defined (5.1.2.2.1)."
Implementation:	`int main (void);`
ISO Standard:	"The values given to the strings pointed to by the `argv` argument to `main` (5.1.2.2.1)."
Implementation:	No arguments are passed to `main`. Reference to `argc` or `argv` is undefined.
ISO Standard:	"What constitutes an interactive device (5.1.2.3)."
Implementation:	Application defined.
ISO Standard:	"Signals for which the equivalent of `signal(`*`sig, SIG_IGN`*`);` is executed at program start-up (7.14.1.1)."
Implementation:	Signals are application defined.
ISO Standard:	"The form of the status returned to the host environment to indicate unsuccessful termination when the `SIGABRT` signal is raised and not caught (7.20.4.1)."
Implementation:	The host environment is application defined.
ISO Standard:	"The forms of the status returned to the host environment by the `exit` function to report successful and unsuccessful termination (7.20.4.3)."
Implementation:	The host environment is application defined.

ISO Standard:	“The status returned to the host environment by the `exit` function if the value of its argument is other than zero, `EXIT_SUCCESS`, or `EXIT_FAILURE` (7.20.4.3).”
Implementation:	The host environment is application defined.
ISO Standard:	“The set of environment names and the method for altering the environment list used by the `getenv` function (7.20.4.4).”
Implementation:	The host environment is application defined.
ISO Standard:	“The manner of execution of the string by the system function (7.20.4.5).”
Implementation:	The host environment is application defined.

B.4 IDENTIFIERS

ISO Standard:	“Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).”
Implementation:	No.
ISO Standard:	“The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).”
Implementation:	All characters are significant.

B.5 CHARACTERS

ISO Standard:	“The number of bits in a byte (C90 3.4, C99 3.6).”
Implementation:	8.
ISO Standard:	“The values of the members of the execution character set (C90 and C99 5.2.1).”
ISO Standard:	“The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (C90 and C99 5.2.2).”
Implementation:	The execution character set is ASCII.
ISO Standard:	“The value of a char object into which has been stored any character other than a member of the basic execution character set (C90 6.1.2.5, C99 6.2.5).”
Implementation:	The value of the char object is the 8-bit binary representation of the character in the source character set. That is, no translation is done.
ISO Standard:	“Which of signed char or unsigned char has the same range, representation, and behavior as “plain” char (C90 6.1.2.5, C90 6.2.1.1, C99 6.2.5, C99 6.3.1.1).”
Implementation:	By default on PIC32C, unsigned char is functionally equivalent to plain char.
ISO Standard:	“The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (C90 6.1.3.4, C99 6.4.4.4, C90 and C99 5.1.1.2).”
Implementation:	The binary representation of the source character set is preserved to the execution character set.
ISO Standard:	“The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (C90 6.1.3.4, C99 6.4.4.4).”
Implementation:	The compiler determines the value for a multi-character character constant one character at a time. The previous value is shifted left by eight, and the bit pattern of the next character is masked in. The final result is of type `int`. If the result is larger than can be represented by an `int`, a warning diagnostic is issued and the value truncated to `int` size.

ISO Standard:	"The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (C90 6.1.3.4, C99 6.4.4.4)."
Implementation:	See previous.
ISO Standard:	"The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (C90 6.1.3.4, C99 6.4.4.4)."
Implementation:	LC_ALL
ISO Standard:	"The current locale used to convert a wide string literal into corresponding wide character codes (C90 6.1.4, C99 6.4.5)."
Implementation:	LC_ALL
ISO Standard:	"The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (C90 6.1.4, C99 6.4.5)."
Implementation:	The binary representation of the characters is preserved from the source character set.

B.6 INTEGERS

ISO Standard:	"Any extended integer types that exist in the implementation (C99 6.2.5)."
Implementation:	There are no extended integer types.
ISO Standard:	"Whether signed integer types are represented using sign and magnitude, two's complement, or one's complement, and whether the extraordinary value is a trap representation or an ordinary value (C99 6.2.6.2)."
Implementation:	All integer types are represented as two's complement, and all bit patterns are ordinary values.
ISO Standard:	"The rank of any extended integer type relative to another extended integer type with the same precision (C99 6.3.1.1)."
Implementation:	No extended integer types are supported.
ISO Standard:	"The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (C90 6.2.1.2, C99 6.3.1.3)."
Implementation:	When converting value X to a type of width N, the value of the result is the Least Significant N bits of the 2's complement representation of X. That is, X is truncated to N bits. No signal is raised.
ISO Standard:	"The results of some bitwise operations on signed integers (C90 6.3, C99 6.5)."
Implementation:	Bitwise operations on signed values act on the 2's complement representation, including the sign bit. The result of a signed right shift expression is sign extended. C99 allows some aspects of signed '<<' to be undefined. The compiler does not do so.

B.7 FLOATING-POINT

ISO Standard:	"The accuracy of the floating-point operations and of the library functions in <math.h> and <complex.h> that return floating-point results (C90 and C99 5.2.4.2.2)."
Implementation:	The accuracy is unknown.
ISO Standard:	"The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in <stdio.h>, <stdlib.h>, and <wchar.h> (C90 and C99 5.2.4.2.2)."
Implementation:	The accuracy is unknown.
ISO Standard:	"The rounding behaviors characterized by non-standard values of FLT_ROUNDS (C90 and C99 5.2.4.2.2)."
Implementation:	No such values are used.
ISO Standard:	"The evaluation methods characterized by non-standard negative values of FLT_EVAL_METHOD (C90 and C99 5.2.4.2.2)."
Implementation:	No such values are used.
ISO Standard:	"The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (C90 6.2.1.3, C99 6.3.1.4)."
Implementation:	C99 Annex F is followed.
ISO Standard:	"The direction of rounding when a floating-point number is converted to a narrower floating-point number (C90 6.2.1.4, 6.3.1.5)."
Implementation:	C99 Annex F is followed.
ISO Standard:	"How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (C90 6.1.3.1, C99 6.4.4.2)."
Implementation:	C99 Annex F is followed.
ISO Standard:	"Whether and how floating expressions are contracted when not disallowed by the FP_CONTRACT pragma (C99 6.5)."
Implementation:	The pragma is not implemented.
ISO Standard:	"The default state for the FENV_ACCESS pragma (C99 7.6.1)."
Implementation:	This pragma is not implemented.
ISO Standard:	"Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (C99 7.6, 7.12)."
Implementation:	None supported.
ISO Standard:	"The default state for the FP_CONTRACT pragma (C99 7.12.2)."
Implementation:	This pragma is not implemented.
ISO Standard:	"Whether the "inexact" floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (C99 F.9)."
Implementation:	Unknown.
ISO Standard:	"Whether the "underflow" (and "inexact") floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (C99 F.9)."
Implementation:	Unknown.

B.8 ARRAYS AND POINTERS

ISO Standard:	"The result of converting a pointer to an integer or vice versa (C90 6.3.4, C99 6.3.2.3)."
Implementation:	A cast from an integer to a pointer or vice versa results uses the binary representation of the source type, reinterpreted as appropriate for the destination type. If the source type is larger than the destination type, the Most Significant bits are discarded. When casting from a pointer to an integer, if the source type is smaller than the destination type, the result is sign extended. When casting from an integer to a pointer, if the source type is smaller than the destination type, the result is extended based on the signedness of the source type.
ISO Standard:	"The size of the result of subtracting two pointers to elements of the same array (C90 6.3.6, C99 6.5.6)."
Implementation:	32-bit signed integer.

B.9 HINTS

ISO Standard:	"The extent to which suggestions made by using the register storage-class specifier are effective (C90 6.5.1, C99 6.7.1)."
Implementation:	The register storage class specifier generally has no effect.
ISO Standard:	"The extent to which suggestions made by using the inline function specifier are effective (C99 6.7.4)."
Implementation:	If `-fno-inline` or `-O0` are specified, no functions will be inlined, even if specified with the `inline` specifier. Otherwise, the function may or may not be inlined dependent on the optimization heuristics of the compiler.

B.10 STRUCTURES, UNIONS, ENUMERATIONS, AND BIT FIELDS

ISO Standard:	"A member of a union object is accessed using a member of a different type (C90 6.3.2.3)."
Implementation:	The corresponding bytes of the union object are interpreted as an object of the type of the member being accessed without regard for alignment or other possible invalid conditions.
ISO Standard:	"Whether a "plain" `int` bit field is treated as a `signed int` bit field or as an `unsigned int` bit field (C90 6.5.2, C90 6.5.2.1, C99 6.7.2, C99 6.7.2.1)."
Implementation:	By default on PIC32C, a plain `int` bit field is treated as an unsigned integer. Note that this is different from the PIC32M. The default behavior can be set explicitly by the compiler flags `-funsigned-bitfields` and `-fsigned-bitfields`.
ISO Standard:	"Allowable bit field types other than `_Bool`, `signed int`, and `unsigned int` (C99 6.7.2.1)."
Implementation:	No other types are supported.
ISO Standard:	"Whether a bit field can straddle a storage unit boundary (C90 6.5.2.1, C99 6.7.2.1)."
Implementation:	No.
ISO Standard:	"The order of allocation of bit fields within a unit (C90 6.5.2.1, C99 6.7.2.1)."
Implementation:	Bit fields are allocated left to right.
ISO Standard:	"The alignment of non-bit field members of structures (C90 6.5.2.1, C99 6.7.2.1)."
Implementation:	Each member is located to the lowest available offset allowable according to the alignment restrictions of the member type.

ISO Standard:	"The integer type compatible with each enumerated type (C90 6.5.2.2, C99 6.7.2.2)."
Implementation:	If the enumeration values are all non-negative, the type is `unsigned int`, else it is `int`. The `-fshort-enums` command line option can change this.

B.11 QUALIFIERS

ISO Standard: "What constitutes an access to an object that has volatile-qualified type (C90 6.5.3, C99 6.7.3)."

Implementation: Any expression which uses the value of or stores a value to a volatile object is considered an access to that object. There is no guarantee that such an access is atomic.

If an expression contains a reference to a volatile object but neither uses the value nor stores to the object, the expression is considered an access to the volatile object or not depending on the type of the object. If the object is of scalar type, an aggregate type with a single member of scalar type, or a union with members of (only) scalar type, the expression is considered an access to the volatile object. Otherwise, the expression is evaluated for its side effects but is not considered an access to the volatile object.

For example:

```
volatile int a;
a; /* access to 'a' since 'a' is scalar */
```

B.12 DECLARATORS

ISO Standard:	"The maximum number of declarators that may modify an arithmetic, structure or union type (C90 6.5.4)."
Implementation:	No limit.

B.13 STATEMENTS

ISO Standard:	"The maximum number of case values in a switch statement (C90 6.6.4.2)."
Implementation:	No limit.

B.14 PRE-PROCESSING DIRECTIVES

ISO Standard:	“How sequences in both forms of header names are mapped to headers or external source file names (C90 6.1.7, C99 6.4.7).”
Implementation:	The character sequence between the delimiters is considered to be a string which is a file name for the host environment.
ISO Standard:	“Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (C90 6.8.1, C99 6.10.1).”
Implementation:	Yes.
ISO Standard:	“Whether the value of a single-character `character` constant in a constant expression that controls conditional inclusion may have a negative value (C90 6.8.1, C99 6.10.1).”
Implementation:	Yes.
ISO Standard:	“The places that are searched for an included `< >` delimited header and how the places are specified or the header is identified (C90 6.8.2, C99 6.10.2).”

Implementation:	`<install directory>/lib/gcc/pic32c/6.2.1/include`
	`<install directory>/pic32c/include`

ISO Standard:	“How the named source file is searched for in an included `""` delimited header (C90 6.8.2, C99 6.10.2).”
Implementation:	The compiler first searches for the named file in the directory containing the including file, the directories specified by the `-iquote` command line option (if any), then the directories which are searched for a `< >` delimited header.
ISO Standard:	“The method by which preprocessing tokens are combined into a header name (C90 6.8.2, C99 6.10.2).”
Implementation:	All tokens, including whitespace, are considered part of the header file name. Macro expansion is not performed on tokens inside the delimiters.
ISO Standard:	“The nesting limit for `#include` processing (C90 6.8.2, C99 6.10.2).”
Implementation:	No limit.
ISO Standard:	“The behavior on each recognized non-`STDC #pragma` directive (C90 6.8.6, C99 6.10.6).”
Implementation:	See **Section 8.11 “Variable Attributes”**.
ISO Standard:	“The definitions for `__DATE__` and `__TIME__` when respectively, the date and time of translation are not available (C90 6.8.8, C99 6.10.8).”
Implementation:	The date and time of translation are always available.

B.15 LIBRARY FUNCTIONS

ISO Standard:	“The Null Pointer constant to which the macro NULL expands (C90 7.1.6, C99 7.17).”
Implementation:	`(void *)0`
ISO Standard:	“Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).”
Implementation:	See the *32-Bit Language Tools Libraries* (DS51685).
ISO Standard:	“The format of the diagnostic printed by the `assert` macro (7.2.1.1).”
Implementation:	“Failed assertion ‘*message*’ at line *line* of ‘*filename*’.\n”
ISO Standard:	“The default state for the `FENV_ACCESS` pragma (7.6.1).”
Implementation:	Unimplemented.
ISO Standard:	“The representation of floating-point exception flags stored by the `fegetexceptflag` function (7.6.2.2).”
Implementation:	Unimplemented.
ISO Standard:	“Whether the `feraiseexcept` function raises the inexact exception in addition to the overflow or underflow exception (7.6.2.3).”
Implementation:	Unimplemented.
ISO Standard:	“Floating environment macros other than `FE_DFL_ENV` that can be used as the argument to the `fesetenv` or `feupdateenv` function (7.6.4.3, 7.6.4.4).”
Implementation:	Unimplemented.
ISO Standard:	“Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function (7.11.1.1).”
Implementation:	None.
ISO Standard:	“The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 or greater than 2 (7.12).”
Implementation:	Unimplemented.
ISO Standard:	“The infinity to which the `INFINITY` macro expands, if any (7.12).”
Implementation:	Unimplemented.
ISO Standard:	“The quiet NaN to which the `NAN` macro expands, when it is defined (7.12).”
Implementation:	Unimplemented.
ISO Standard:	“Domain errors for the mathematics functions, other than those required by this International Standard (7.12.1).”
Implementation:	None.
ISO Standard:	“The values returned by the mathematics functions and whether `errno` is set to the value of the macro `EDOM`, on domain errors (7.12.1).”
Implementation:	`errno` is set to `EDOM` on domain errors.
ISO Standard:	“Whether the mathematics functions set `errno` to the value of the macro `ERANGE` on overflow and/or underflow range errors (7.12.1).”
Implementation:	Yes.
ISO Standard:	“The default state for the `FP_CONTRACT` pragma (7.12.2)
Implementation:	Unimplemented.
ISO Standard:	“Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero (7.12.10.1).”
Implementation:	NaN is returned.
ISO Standard:	“The base-2 logarithm of the modulus used by the `remquo` function in reducing the quotient (7.12.10.3).”
Implementation:	Unimplemented.
ISO Standard:	“The set of signals, their semantics, and their default handling (7.14).”

Implementation:	The default handling of signals is to always return failure. Actual signal handling is application defined.
ISO Standard:	“If the equivalent of `signal(sig, SIG_DFL);` is not executed prior to the call of a signal handler, the blocking of the signal that is performed (7.14.1.1).”
Implementation:	Application defined.
ISO Standard:	“Whether the equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler for the signal `SIGILL` (7.14.1.1).”
Implementation:	Application defined.
ISO Standard:	“Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).”
Implementation:	Application defined.
ISO Standard:	“Whether the last line of a text stream requires a terminating new-line character (7.19.2).”
Implementation:	Yes.
ISO Standard:	“Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.19.2).”
Implementation:	Yes.
ISO Standard:	“The number of null characters that may be appended to data written to a binary stream (7.19.2).”
Implementation:	No null characters are appended to a binary stream.
ISO Standard:	“Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.19.3).”
Implementation:	Application defined. The system level function `open` is called with the `O_APPEND` flag.
ISO Standard:	“Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3).”
Implementation:	Application defined.
ISO Standard:	“The characteristics of file buffering (7.19.3).”
ISO Standard:	“Whether a zero-length file actually exists (7.19.3).”
Implementation:	Application defined.
ISO Standard:	“The rules for composing valid file names (7.19.3).”
Implementation:	Application defined.
ISO Standard:	“Whether the same file can be open multiple times (7.19.3).”
Implementation:	Application defined.
ISO Standard:	“The nature and choice of encodings used for multibyte characters in files (7.19.3).”
Implementation:	Encodings are the same for each file.
ISO Standard:	“The effect of the `remove` function on an open file (7.19.4.1).”
Implementation:	Application defined. The system function `unlink` is called.
ISO Standard:	“The effect if a file with the new name exists prior to a call to the `rename` function (7.19.4.2).”
Implementation:	Application defined. The system function `link` is called to create the new file name, then `unlink` is called to remove the old file name. Typically, `link` will fail if the new file name already exists.
ISO Standard:	“Whether an open temporary file is removed upon abnormal program termination (7.19.4.3).”
Implementation:	No.
ISO Standard:	“What happens when the `tmpnam` function is called more than `TMP_MAX` times (7.19.4.4).”
Implementation:	Temporary names will wrap around and be reused.

ISO Standard:	"Which changes of mode are permitted (if any), and under what circumstances (7.19.5.4)."
Implementation:	The file is closed via the system level `close` function and re-opened with the `open` function with the new mode. No additional restriction beyond those of the application defined `open` and `close` functions are imposed.
ISO Standard:	"The style used to print an infinity or NaN, and the meaning of the *n-char-sequence* if that style is printed for a NaN (7.19.6.1, 7.24.2.1)."
Implementation:	No char sequence is printed. NaN is printed as "NaN." Infinity is printed as "[-/+]Inf."
ISO Standard:	"The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.19.6.1, 7.24.2.1)."
Implementation:	Functionally equivalent to `%x`.
ISO Standard:	"The interpretation of a `-` character that is neither the first nor the last character, nor the second where a `^` character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.1)."
Implementation:	Unknown
ISO Standard:	"The set of sequences matched by the `%p` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.2)."
Implementation:	The same set of sequences matched by %x.
ISO Standard:	"The interpretation of the input item corresponding to a `%p` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.2)."
Implementation:	If the result is not a valid pointer, the behavior is undefined.
ISO Standard:	"The value to which the macro `errno` is set by the `fgetpos`, `fsetpos`, or `ftell` functions on failure (7.19.9.1, 7.19.9.3, 7.19.9.4)."
Implementation:	If the result exceeds `LONG_MAX`, `errno` is set to `ERANGE`. Other errors are application defined according to the application definition of the `lseek` function.
ISO Standard:	"The meaning of the *n-char-sequence* in a string converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function (7.20.1.3, 7.24.4.1.1)."
Implementation:	No meaning is attached to the sequence.
ISO Standard:	"Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.20.1.3, 7.24.4.1.1)."
Implementation:	Yes.
ISO Standard:	"Whether the `calloc`, `malloc`, and `realloc` functions return a Null Pointer or a pointer to an allocated object when the size requested is zero (7.20.3)."
Implementation:	A pointer to a statically allocated object is returned.
ISO Standard:	"Whether open output streams are flushed, open streams are closed, or temporary files are removed when the `abort` function is called (7.20.4.1)."
Implementation:	No.
ISO Standard:	"The termination status returned to the host environment by the `abort` function (7.20.4.1)."
Implementation:	By default, there is no host environment.
ISO Standard:	"The value returned by the `system` function when its argument is not a Null Pointer (7.20.4.5)."
Implementation:	Application defined.
ISO Standard:	"The local time zone and Daylight Saving Time (7.23.1)."

Implementation:	Application defined.
ISO Standard:	“The era for the `clock` function (7.23.2.1).”
Implementation:	Application defined.
ISO Standard:	“The positive value for `tm_isdst` in a normalized `tmx` structure (7.23.2.6).”
Implementation:	1.
ISO Standard:	“The replacement string for the `%Z` specifier to the `strftime`, `strfx-time`, `wcsftime`, and `wcsfxtime` functions in the “`C`” locale (7.23.3.5, 7.23.3.6, 7.24.5.1, 7.24.5.2).”
Implementation:	Unimplemented.
ISO Standard:	“Whether or when the trigonometric, hyperbolic, base-*e* exponential, base-*e* logarithmic, error, and log gamma functions raise the inexact exception in an IEC 60559 conformant implementation (F.9).”
Implementation:	No.
ISO Standard:	“Whether the inexact exception may be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (F.9).”
Implementation:	No.
ISO Standard:	“Whether the underflow (and inexact) exception may be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (F.9).”
Implementation:	No.
ISO Standard:	“Whether the functions honor the Rounding Direction mode (F.9).”
Implementation:	The Rounding mode is not forced.

B.16 ARCHITECTURE

ISO Standard:	“The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (C90 and C99 5.2.4.2, C99 7.18.2, 7.18.3).”
Implementation:	See **Section 8.3.2 “`limits.h`”**.
ISO Standard:	“The number, order, and encoding of bytes in any object (when not explicitly specified in the standard) (C99 6.2.6.1).”
Implementation:	Little endian, populated from Least Significant Byte first. See **Section 8.2 “Data Representation”**.
ISO Standard:	“The value of the result of the `size of` operator (C90 6.3.3.4, C99 6.5.3.4).”
Implementation:	See **Section 8.2 “Data Representation”**.

NOTES:

Appendix C. Built-In Functions

This appendix lists the built-in functions that are specific to MPLAB XC32 C/C++ Compiler.

Built-in functions give the C programmer access to assembler operators or machine instructions that are currently only accessible using inline assembly, but are sufficiently useful that they are applicable to a broad range of applications. Built-in functions are coded in C source files syntactically like function calls, but they are compiled to assembly code that directly implements the function, and do not involve function calls or library routines.

There are a number of reasons why providing built-in functions is preferable to requiring programmers to use inline assembly. They include the following:

1. Providing built-in functions for specific purposes simplifies coding.
2. Certain optimizations are disabled when inline assembly is used. This is not the case for built-in functions.
3. For machine instructions that use dedicated registers, coding inline assembly while avoiding register allocation errors can require considerable care. The built-in functions make this process simpler as you do not need to be concerned with the particular register requirements for each individual machine instruction.

C.1 BUILT-IN FUNCTION DESCRIPTIONS (PIC32C)

This section describes the programmer interface to the compiler built-in functions. Since the functions are "built in," there are no header files associated with them. Similarly, there are no command-line switches associated with the built-in functions – they are always available. The built-in function names are chosen such that they belong to the compiler's namespace (they all have the prefix `__builtin_`), so they will not conflict with function or variable names in the programmer's namespace.

Built-In Function List

- void __builtin_nop(void)
- void __builtin_software_breakpoint(void)

void __builtin_nop(void)

Description:	Emit a no-op instruction.
Prototype:	`void __builtin_nop(void);`
Argument:	None.
Return Value:	None.
Assembler Operator/ Machine Instruction:	`nop`
Error Messages	None.

void __builtin_software_breakpoint(void)

Description:	Emit a software breakpoint instruction.
Prototype:	`void __builtin_software_breakpoint(void);`
Argument:	None.
Return Value:	None.
Assembler Operator/ Machine Instruction:	`bkpt`
Error Messages	None.

Appendix D. ASCII Character Set

TABLE D-1: ASCII CHARACTER SET

Most Significant Character

Least Significant Character

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

NOTES:

Appendix E. Document Revision History

DOCUMENT REVISION HISTORY

Revision A (June 2019)

Initial revision of the document.

NOTES:

Support

myMICROCHIP PERSONALIZED NOTIFICATION SERVICE

myMicrochip: http://www.microchip.com/pcn

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

Please visit myMicrochip to begin the registration process and select your preferences to receive personalized notifications. A FAQ and registration details are available on the page, which can be opened by selecting the link above.

When you are selecting your preferences, choosing "Development Systems" will populate the list with available development tools. The main categories of tools are listed below:

- **Compilers** – The latest information on Microchip C compilers, assemblers, linkers and other language tools. These include all MPLAB C compilers; all MPLAB assemblers (including MPASM™ assembler); all MPLAB linkers (including MPLINK™ object linker); and all MPLAB librarians (including MPLIB™ object librarian).
- **Emulators** – The latest information on Microchip in-circuit emulators. This includes the MPLAB REAL ICE™ in-circuit emulator.
- **In-Circuit Debuggers** – The latest information on Microchip in-circuit debuggers. These include the PICkit™ 2, PICkit 3 and MPLAB ICD 3 in-circuit debuggers.
- **MPLAB® X IDE** – The latest information on Microchip MPLAB X IDE, the Windows® Integrated Development Environment for development systems tools. This list is focused on the MPLAB X IDE, MPLAB X IDE Project Manager, MPLAB Editor and MPLAB SIM simulator, as well as general editing and debugging features.
- **Programmers** – The latest information on Microchip programmers. These include the device (production) programmers MPLAB REAL ICE in-circuit emulator, MPLAB ICD 3 in-circuit debugger, MPLAB PM3 and development (nonproduction) programmers PICkit 2 and 3.
- **Starter/Demo Boards** – These include MPLAB Starter Kit boards, PICDEM demo boards, and various other evaluation boards.

THE MICROCHIP WEB SITE

Web Site: http://www.microchip.com

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

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

MICROCHIP FORUMS

Forums: http://www.microchip.com/forums

Microchip provides additional online support via our web forums. Currently available forums are:

- Development Tools
- 8-bit PIC MCUs
- 16-bit PIC MCUs
- 32-bit PIC MCUs

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

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

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

Technical Support: http://support.microchip.com

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

CONTACT MICROCHIP TECHNOLOGY

You can call or fax Microchip Corporate offices at the numbers below:

Voice: (480) 792-7200

Fax: (480) 792-7277

Glossary

A

Absolute Section

A GCC compiler section with a fixed (absolute) address that cannot be changed by the linker.

Absolute Variable/Function

A variable or function placed at an absolute address using the OCG compiler’s `@ address` syntax.

Access Memory

PIC18 Only – Special registers on PIC18 devices that allow access regardless of the setting of the Bank Select Register (BSR).

Access Entry Points

Access entry points provide a way to transfer control across segments to a function which may not be defined at link time. They support the separate linking of boot and secure application segments.

Address

Value that identifies a location in memory.

Alphabetic Character

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

Alphanumeric

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

ANDed Breakpoints

Set up an ANDed condition for breaking, i.e., breakpoint 1 AND breakpoint 2 must occur at the same time before a program halt. This can only be accomplished if a data breakpoint and a program memory breakpoint occur at the same time.

Anonymous Structure

16-bit C Compiler **–** An unnamed structure.

PIC18 C Compiler **–** An unnamed structure that is a member of a C union. The members of an anonymous structure may be accessed as if they were members of the enclosing union. For example, in the following code, `hi` and `lo` are members of an anonymous structure inside the union `caster`.

```
union castaway
 int intval;
 struct {
  char lo; //accessible as caster.lo
  char hi; //accessible as caster.hi
 };
} caster;
```

ANSI

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

Application

A set of software and hardware that may be controlled by a PIC® microcontroller.

Archive/Archiver

An archive/library is a collection of relocatable object modules. It is created by assembling multiple source files to object files, and then using the archiver/librarian to combine the object files into one archive/library file. An archive/library can be linked with object modules and other archives/libraries to create executable code.

ASCII

American Standard Code for Information Interchange is a character set encoding that uses 7 binary digits to represent each character. It includes upper and lower case letters, digits, symbols and control characters.

Assembly/Assembler

Assembly is a programming language that describes binary machine code in a symbolic form. An assembler is a language tool that translates assembly language source code into machine code.

Assigned Section

A GCC compiler section which has been assigned to a target memory block in the linker command file.

Asynchronously

Multiple events that do not occur at the same time. This is generally used to refer to interrupts that may occur at any time during processor execution.

Asynchronous Stimulus

Data generated to simulate external inputs to a simulator device.

Attribute

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

Attribute, Section

GCC Characteristics of sections, such as “executable”, “readonly”, or “data” that can be specified as flags in the assembler `.section` directive.

B

Binary

The base two numbering system that uses the digits 0-1. The rightmost digit counts ones, the next counts multiples of 2, then 2^2 = 4, etc.

Bookmarks

Use bookmarks to easily locate specific lines in a file.

Select Toggle Bookmarks on the Editor toolbar to add/remove bookmarks. Click other icons on this toolbar to move to the next or previous bookmark.

Breakpoint

Hardware Breakpoint: An event whose execution will cause a halt.

Software Breakpoint: An address where execution of the firmware will halt. Usually achieved by a special break instruction.

Build

Compile and link all the source files for an application.

C

C\C++

C is a general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators. C++ is the object-oriented version of C.

Calibration Memory

A special function register or registers used to hold values for calibration of a PIC microcontroller on-board RC oscillator or other device peripherals.

Central Processing Unit

The part of a device that is responsible for fetching the correct instruction for execution, decoding that instruction, and then executing that instruction. When necessary, it works in conjunction with the arithmetic logic unit (ALU) to complete the execution of the instruction. It controls the program memory address bus, the data memory address bus, and accesses to the stack.

Clean

Clean removes all intermediary project files, such as object, hex and debug files, for the active project. These files are recreated from other files when a project is built.

COFF

Common Object File Format. An object file of this format contains machine code, debugging and other information.

Command Line Interface

A means of communication between a program and its user based solely on textual input and output.

Compiled Stack

A region of memory managed by the compiler in which variables are statically allocated space. It replaces a software or hardware stack when such mechanisms cannot be efficiently implemented on the target device.

Compiler

A program that translates a source file written in a high-level language into machine code.

Conditional Assembly

Assembly language code that is included or omitted based on the assembly-time value of a specified expression.

Conditional Compilation

The act of compiling a program fragment only if a certain constant expression, specified by a preprocessor directive, is true.

Configuration Bits

Special-purpose bits programmed to set PIC microcontroller modes of operation. A Configuration bit may or may not be preprogrammed.

Control Directives

Directives in assembly language code that cause code to be included or omitted based on the assembly-time value of a specified expression.

CPU

See Central Processing Unit.

Cross Reference File

A file that references a table of symbols and a list of files that references the symbol. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

D

Data Directives

Data directives are those that control the assembler’s allocation of program or data memory and provide a way to refer to data items symbolically; that is, by meaningful names.

Data Memory

On Microchip MCU and DSC devices, data memory (RAM) is comprised of General Purpose Registers (GPRs) and Special Function Registers (SFRs). Some devices also have EEPROM data memory.

Data Monitor and Control Interface (DMCI)

The Data Monitor and Control Interface, or DMCI, is a tool in MPLAB X IDE. The interface provides dynamic input control of application variables in projects. Application-generated data can be viewed graphically using any of 4 dynamically-assignable graph windows.

Debug/Debugger

See ICE/ICD.

Debugging Information

Compiler and assembler options that, when selected, provide varying degrees of information used to debug application code. See compiler or assembler documentation for details on selecting debug options.

Deprecated Features

Features that are still supported for legacy reasons, but will eventually be phased out and no longer used.

Device Programmer

A tool used to program electrically programmable semiconductor devices such as microcontrollers.

Digital Signal Controller

A A digital signal controller (DSC) is a microcontroller device with digital signal processing capability, i.e., Microchip dsPIC DSC devices.

Digital Signal Processing\Digital Signal Processor

Digital signal processing (DSP) is the computer manipulation of digital signals, commonly analog signals (sound or image) which have been converted to digital form (sampled). A digital signal processor is a microprocessor that is designed for use in digital signal processing.

Directives

Statements in source code that provide control of the language tool’s operation.

Download

Download is the process of sending data from a host to another device, such as an emulator, programmer or target board.

DWARF

Debug With Arbitrary Record Format. DWARF is a debug information format for ELF files.

E

EEPROM

Electrically Erasable Programmable Read Only Memory. A special type of PROM that can be erased electrically. Data is written or erased one byte at a time. EEPROM retains its contents even when power is turned off.

ELF

Executable and Linking Format. An object file of this format contains machine code. Debugging and other information is specified in with DWARF. ELF/DWARF provide better debugging of optimized code than COFF.

Emulation/Emulator

See ICE/ICD.

Endianness

The ordering of bytes in a multi-byte object.

Environment

MPLAB PM3 **–** A folder containing files on how to program a device. This folder can be transferred to a SD/MMC card.

Epilogue

A portion of compiler-generated code that is responsible for deallocating stack space, restoring registers and performing any other machine-specific requirement specified in the runtime model. This code executes after any user code for a given function, immediately prior to the function return.

EPROM

Erasable Programmable Read Only Memory. A programmable read-only memory that can be erased usually by exposure to ultraviolet radiation.

Error/Error File

An error reports a problem that makes it impossible to continue processing your program. When possible, an error identifies the source file name and line number where the problem is apparent. An error file contains error messages and diagnostics generated by a language tool.

Event

A description of a bus cycle which may include address, data, pass count, external input, cycle type (fetch, R/W), and time stamp. Events are used to describe triggers, breakpoints and interrupts.

Executable Code

Software that is ready to be loaded for execution.

Export

Send data out of the MPLAB X IDE in a standardized format.

Expressions

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

Extended Microcontroller Mode

In extended microcontroller mode, on-chip program memory as well as external memory is available. Execution automatically switches to external if the program memory address is greater than the internal memory space of the PIC18 device.

Extended Mode (PIC18 MCUs)

In Extended mode, the compiler will utilize the extended instructions (i.e., `ADDFSR`, `ADDULNK`, `CALLW`, `MOVSF`, `MOVSS`, `PUSHL`, `SUBFSR` and `SUBULNK`) and the indexed with literal offset addressing.

External Label

A label that has external linkage.

External Linkage

A function or variable has external linkage if it can be referenced from outside the module in which it is defined.

External Symbol

A symbol for an identifier which has external linkage. This may be a reference or a definition.

External Symbol Resolution

A process performed by the linker in which external symbol definitions from all input modules are collected in an attempt to resolve all external symbol references. Any external symbol references which do not have a corresponding definition cause a linker error to be reported.

External Input Line

An external input signal logic probe line (TRIGIN) for setting an event based upon external signals.

External RAM

Off-chip Read/Write memory.

F

Fatal Error

An error that will halt compilation immediately. No further messages will be produced.

File Registers

On-chip data memory, including General Purpose Registers (GPRs) and Special Function Registers (SFRs).

Filter

Determine by selection what data is included/excluded in a trace display or data file.

Fixup

The process of replacing object file symbolic references with absolute addresses after relocation by the linker.

Flash

A type of EEPROM where data is written or erased in blocks instead of bytes.

FNOP

Forced No Operation. A forced NOP cycle is the second cycle of a two-cycle instruction. Since the PIC microcontroller architecture is pipelined, it prefetches the next instruction in the physical address space while it is executing the current instruction. However, if the current instruction changes the program counter, this prefetched instruction is explicitly ignored, causing a forced NOP cycle.

Frame Pointer

A pointer that references the location on the stack that separates the stack-based arguments from the stack-based local variables. Provides a convenient base from which to access local variables and other values for the current function.

Free-Standing

An implementation that accepts any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (ANSI ‘89 standard clause 7) is confined to the contents of the standard headers `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>` and `<stdint.h>`.

G

GPR

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

H

Halt

A stop of program execution. Executing Halt is the same as stopping at a breakpoint.

Heap

An area of memory used for dynamic memory allocation where blocks of memory are allocated and freed in an arbitrary order determined at runtime.

Hex Code\Hex File

Hex code is executable instructions stored in a hexadecimal format code. Hex code is contained in a hex file.

Hexadecimal

The base 16 numbering system that uses the digits 0-9 plus the letters A-F (or a-f). The digits A-F represent hexadecimal digits with values of (decimal) 10 to 15. The rightmost digit counts ones, the next counts multiples of 16, then 16^2 = 256, etc.

High Level Language

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

I

ICE/ICD

In-Circuit Emulator/In-Circuit Debugger: A hardware tool that debugs and programs a target device. An emulator has more features than an debugger, such as trace.

In-Circuit Emulation/In-Circuit Debug: The act of emulating or debugging with an in-circuit emulator or debugger.

-ICE/-ICD: A device (MCU or DSC) with on-board in-circuit emulation or debug circuitry. This device is always mounted on a header board and used to debug with an in-circuit emulator or debugger.

ICSP™

In-Circuit Serial Programming™. A method of programming Microchip embedded devices using serial communication and a minimum number of device pins.

IDE

Integrated Development Environment, as in MPLAB X IDE.

Identifier

A function or variable name.

IEEE

Institute of Electrical and Electronics Engineers.

Import

Bring data into the MPLAB X IDE from an outside source, such as from a hex file.

Initialized Data

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

```
int myVar=5;
```

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

Instruction Set

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

Instructions

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

Internal Linkage

A function or variable has internal linkage if it can not be accessed from outside the module in which it is defined.

International Organization for Standardization

An organization that sets standards in many businesses and technologies, including computing and communications. Also known as ISO.

Interrupt

A signal to the CPU that suspends the execution of a running application and transfers control to an Interrupt Service Routine (ISR) so that the event may be processed. Upon completion of the ISR, normal execution of the application resumes.

Interrupt Handler

A routine that processes special code when an interrupt occurs.

Interrupt Service Request (IRQ)

An event which causes the processor to temporarily suspend normal instruction execution and to start executing an interrupt handler routine. Some processors have several interrupt request events allowing different priority interrupts.

Interrupt Service Routine (ISR)

Language tools – A function that handles an interrupt.

MPLAB X IDE – User-generated code that is entered when an interrupt occurs. The location of the code in program memory will usually depend on the type of interrupt that has occurred.

Interrupt Vector

Address of an interrupt service routine or interrupt handler.

L

L-value

An expression that refers to an object that can be examined and/or modified. An l-value expression is used on the left-hand side of an assignment.

Latency

The time between an event and its response.

Library/Librarian

See Archive/Archiver.

Linker

A language tool that combines object files and libraries to create executable code, resolving references from one module to another.

Linker Script Files

Linker script files are the command files of a linker. They define linker options and describe available memory on the target platform.

Listing Directives

Listing directives are those directives that control the assembler listing file format. They allow the specification of titles, pagination and other listing control.

Listing File

A listing file is an ASCII text file that shows the machine code generated for each C source statement, assembly instruction, assembler directive, or macro encountered in a source file.

Little Endian

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

Local Label

A local label is one that is defined inside a macro with the LOCAL directive. These labels are particular to a given instance of a macro's instantiation. In other words, the symbols and labels that are declared as local are no longer accessible after the ENDM macro is encountered.

Logic Probes

Up to 14 logic probes can be connected to some Microchip emulators. The logic probes provide external trace inputs, trigger output signal, +5V, and a common ground.

Loop-Back Test Board

Used to test the functionality of the MPLAB REAL ICE in-circuit emulator.

LVDS

Low Voltage Differential Signaling. A low noise, low-power, low amplitude method for high-speed (gigabits per second) data transmission over copper wire.

With standard I/O signaling, data storage is contingent upon the actual voltage level. Voltage level can be affected by wire length (longer wires increase resistance, which lowers voltage). But with LVDS, data storage is distinguished only by positive and negative voltage values, not the voltage level. Therefore, data can travel over greater lengths of wire while maintaining a clear and consistent data stream.

Source: http://www.webopedia.com/TERM/L/LVDS.html.

M

Machine Code

The representation of a computer program that is actually read and interpreted by the processor. A program in binary machine code consists of a sequence of machine instructions (possibly interspersed with data). The collection of all possible instructions for a particular processor is known as its "instruction set".

Machine Language

A set of instructions for a specific central processing unit, designed to be usable by a processor without being translated.

Macro

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

Macro Directives

Directives that control the execution and data allocation within macro body definitions.

Makefile

Export to a file the instructions to Make the project. Use this file to Make your project outside of MPLAB X IDE, i.e., with a `make`.

Make Project

A command that rebuilds an application, recompiling only those source files that have changed since the last complete compilation.

MCU

Microcontroller Unit. An abbreviation for microcontroller. Also uC.

Memory Model

For C compilers, a representation of the memory available to the application. For the PIC18 C compiler, a description that specifies the size of pointers that point to program memory.

Message

Text displayed to alert you to potential problems in language tool operation. A message will not stop operation.

Microcontroller

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

Microcontroller Mode

One of the possible program memory configurations of PIC18 microcontrollers. In microcontroller mode, only internal execution is allowed. Thus, only the on-chip program memory is available in microcontroller mode.

Microprocessor Mode

One of the possible program memory configurations of PIC18 microcontrollers. In microprocessor mode, the on-chip program memory is not used. The entire program memory is mapped externally.

Mnemonics

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

Module

The preprocessed output of a source file after preprocessor directives have been executed. Also known as a translation unit.

MPASM™ Assembler

Microchip Technology's relocatable macro assembler for PIC microcontroller devices, KeeLoq® devices and Microchip memory devices.

MPLAB *Language Tool* for *Device*

Microchip's C compilers, assemblers and linkers for specified devices. Select the type of language tool based on the device you will be using for your application, e.g., if you will be creating C code on a PIC18 MCU, select the MPLAB C Compiler for PIC18 MCUs.

MPLAB® ICD

Microchip in-circuit debugger that works with MPLAB X IDE. *See* ICE/ICD.

MPLAB X IDE

Microchip's Integrated Development Environment.MPLAB X IDE comes with an editor, project manager and simulator.

MPLAB PM3

A device programmer from Microchip. Programs PIC18 microcontrollers and dsPIC digital signal controllers. Can be used with MPLAB X IDE or stand-alone. Replaces PRO MATE II.

MPLAB REAL ICE™ In-Circuit Emulator

Microchip’s next-generation in-circuit emulator that works with MPLAB X IDE. *See* ICE/ICD.

MPLAB SIM

Microchip’s simulator that works with MPLAB X IDE in support of PIC MCU and dsPIC DSC devices.

MPLIB™ Object Librarian

Microchip’s librarian that can work with MPLAB X IDE. MPLIB librarian is an object librarian for use with COFF object modules created using either MPASM assembler (mpasm or mpasmwin v2.0) or MPLAB C18 C Compiler.

MPLINK™ Object Linker

MPLINK linker is an object linker for the Microchip MPASM assembler and the Microchip C18 C compiler. MPLINK linker also may be used with the Microchip MPLIB librarian. MPLINK linker is designed to be used with MPLAB X IDE, though it does not have to be.

MRU

Most Recently Used. Refers to files and windows available to be selected from MPLAB X IDE main pull down menus.

N

Native Data Size

For Native trace, the size of the variable used in a Watch window must be of the same size as the selected device’s data memory: bytes for PIC18 devices and words for 16-bit devices.

Nesting Depth

The maximum level to which macros can include other macros.

Node

MPLAB X IDE project component.

Non-Extended Mode (PIC18 MCUs)

In Non-Extended mode, the compiler will not utilize the extended instructions nor the indexed with literal offset addressing.

Non Real Time

Refers to the processor at a breakpoint or executing single-step instructions or MPLAB X IDE being run in simulator mode.

Non-Volatile Storage

A storage device whose contents are preserved when its power is off.

NOP

No Operation. An instruction that has no effect when executed except to advance the program counter.

O

Object Code/Object File

Object code is the machine code generated by an assembler or compiler. An object file is a file containing machine code and possibly debug information. It may be immediately executable or it may be relocatable, requiring linking with other object files, e.g., libraries, to produce a complete executable program.

Object File Directives

Directives that are used only when creating an object file.

Octal

The base 8 number system that only uses the digits 0-7. The rightmost digit counts ones, the next digit counts multiples of 8, then $8^2 = 64$, etc.

Off-Chip Memory

Off-chip memory refers to the memory selection option for the PIC18 device where memory may reside on the target board, or where all program memory may be supplied by the emulator. The **Memory** tab accessed from *Options>Development Mode* provides the Off-Chip Memory selection dialog box.

Opcodes

Operational Codes. *See* Mnemonics.

Operators

Symbols, like the plus sign ‘+’ and the minus sign ‘-’, that are used when forming well-defined expressions. Each operator has an assigned precedence that is used to determine order of evaluation.

OTP

One Time Programmable. EPROM devices that are not in windowed packages. Since EPROM needs ultraviolet light to erase its memory, only windowed devices are erasable.

P

Pass Counter

A counter that decrements each time an event (such as the execution of an instruction at a particular address) occurs. When the pass count value reaches zero, the event is satisfied. You can assign the Pass Counter to break and trace logic, and to any sequential event in the complex trigger dialog.

PC

Personal Computer or Program Counter.

PC Host

Any PC running a supported Windows operating system.

Persistent Data

Data that is never cleared or initialized. Its intended use is so that an application can preserve data across a device Reset.

Phantom Byte

An unimplemented byte in the dsPIC architecture that is used when treating the 24-bit instruction word as if it were a 32-bit instruction word. Phantom bytes appear in dsPIC hex files.

PIC® MCUs

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

PICkit 2 and 3

Microchip's developmental device programmers with debug capability through Debug Express. See the Readme files for each tool to see which devices are supported.

Plug-ins

The MPLAB X IDE has both built-in components and plug-in modules to configure the system for a variety of software and hardware tools. Several plug-in tools may be found under the Tools menu.

Pod

The enclosure for an in-circuit emulator or debugger. Other names are “Puck”, if the enclosure is round, and “Probe”, not be confused with logic probes.

Power-on-Reset Emulation

A software randomization process that writes random values in data RAM areas to simulate uninitialized values in RAM upon initial power application.

Pragma

A directive that has meaning to a specific compiler. Often a pragma is used to convey implementation-defined information to the compiler. MPLAB C30 uses attributes to convey this information.

Precedence

Rules that define the order of evaluation in expressions.

Production Programmer

A production programmer is a programming tool that has resources designed in to program devices rapidly. It has the capability to program at various voltage levels and completely adheres to the programming specification. Programming a device as fast as possible is of prime importance in a production environment where time is of the essence as the application circuit moves through the assembly line.

Profile

For MPLAB SIM simulator, a summary listing of executed stimulus by register.

Program Counter

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

Program Counter Unit

16-bit assembler – A conceptual representation of the layout of program memory. The program counter increments by 2 for each instruction word. In an executable section, 2 program counter units are equivalent to 3 bytes. In a read-only section, 2 program counter units are equivalent to 2 bytes.

Program Memory

MPLAB X IDE – The memory area in a device where instructions are stored. Also, the memory in the emulator or simulator containing the downloaded target application firmware.

16-bit assembler/compiler – The memory area in a device where instructions are stored.

Project

A project contains the files needed to build an application (source code, linker script files, etc.) along with their associations to various build tools and build options.

Prologue

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

Prototype System

A term referring to a user's target application, or target board.

Psect

The OCG equivalent of a GCC section, short for program section. A block of code or data which is treated as a whole by the linker.

PWM Signals

Pulse Width Modulation Signals. Certain PIC MCU devices have a PWM peripheral.

Q

Qualifier

An address or an address range used by the Pass Counter or as an event before another operation in a complex trigger.

R

Radix

The number base, hex, or decimal, used in specifying an address.

RAM

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

Raw Data

The binary representation of code or data associated with a section.

Read Only Memory

Memory hardware that allows fast access to permanently stored data but prevents addition to or modification of the data.

Real Time

When an in-circuit emulator or debugger is released from the halt state, the processor runs in Real Time mode and behaves exactly as the normal chip would behave. In Real Time mode, the real time trace buffer of an emulator is enabled and constantly captures all selected cycles, and all break logic is enabled. In an in-circuit emulator or debugger, the processor executes in real time until a valid breakpoint causes a halt, or until the user halts the execution.

In the simulator, real time simply means execution of the microcontroller instructions as fast as they can be simulated by the host CPU.

Recursive Calls

A function that calls itself, either directly or indirectly.

Recursion

The concept that a function or macro, having been defined, can call itself. Great care should be taken when writing recursive macros; it is easy to get caught in an infinite loop where there will be no exit from the recursion.

Reentrant

A function that may have multiple, simultaneously active instances. This may happen due to either direct or indirect recursion or through execution during interrupt processing.

Relaxation

The process of converting an instruction to an identical, but smaller instruction. This is useful for saving on code size. MPLAB XC32 currently knows how to `relax` a `CALL` instruction into an `RCALL` instruction. This is done when the symbol that is being called is within +/- 32k instruction words from the current instruction.

Relocatable

An object whose address has not been assigned to a fixed location in memory.

Relocatable Section

16-bit assembler – A section whose address is not fixed (absolute). The linker assigns addresses to relocatable sections through a process called relocation.

Relocation

A process performed by the linker in which absolute addresses are assigned to relocatable sections and all symbols in the relocatable sections are updated to their new addresses.

ROM

Read Only Memory (Program Memory). Memory that cannot be modified.

Run

The command that releases the emulator from halt, allowing it to run the application code and change or respond to I/O in real time.

Run-time Model

Describes the use of target architecture resources.

Runtime Watch

A Watch window where the variables change in as the application is run. See individual tool documentation to determine how to set up a runtime watch. Not all tools support runtime watches.

S

Scenario

For MPLAB SIM simulator, a particular setup for stimulus control.

Section

The GCC equivalent of an OCG psect. A block of code or data which is treated as a whole by the linker.

Section Attribute

A GCC characteristic ascribed to a section (e.g., an `access` section).

Sequenced Breakpoints

Breakpoints that occur in a sequence. Sequence execution of breakpoints is bottom-up; the last breakpoint in the sequence occurs first.

Serialized Quick Turn Programming

Serialization allows you to program a serial number into each microcontroller device that the Device Programmer programs. This number can be used as an entry code, password or ID number.

Shell

The MPASM assembler shell is a prompted input interface to the macro assembler. There are two MPASM assembler shells: one for the DOS version and one for the Windows version.

Simulator

A software program that models the operation of devices.

Single Step

This command steps though code, one instruction at a time. After each instruction, MPLAB X IDE updates register windows, watch variables, and status displays so you can analyze and debug instruction execution. You can also single step C compiler source code, but instead of executing single instructions, MPLAB X IDE will execute all assembly level instructions generated by the line of the high level C statement.

Skew

The information associated with the execution of an instruction appears on the processor bus at different times. For example, the executed opcodes appears on the bus as a fetch during the execution of the previous instruction, the source data address and value and the destination data address appear when the opcodes is actually executed, and the destination data value appears when the next instruction is executed. The trace buffer captures the information that is on the bus at one instance. Therefore, one trace buffer entry will contain execution information for three instructions. The number of captured cycles from one piece of information to another for a single instruction execution is referred to as the skew.

Skid

When a hardware breakpoint is used to halt the processor, one or more additional instructions may be executed before the processor halts. The number of extra instructions executed after the intended breakpoint is referred to as the skid.

Source Code

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

Source File

An ASCII text file containing source code.

Special Function Registers (SFRs)

The portion of data memory (RAM) dedicated to registers that control I/O processor functions, I/O status, timers or other modes or peripherals.

SQTP

See Serialized Quick Turn Programming.

Stack, Hardware

Locations in PIC microcontroller where the return address is stored when a function call is made.

Stack, Software

Memory used by an application for storing return addresses, function parameters, and local variables. This memory is dynamically allocated at runtime by instructions in the program. It allows for reentrant function calls.

Stack, Compiled

A region of memory managed and allocated by the compiler in which variables are statically assigned space. It replaces a software stack when such mechanisms cannot be efficiently implemented on the target device. It precludes reentrancy.

MPLAB Starter Kit for *Device*

Microchip’s starter kits contains everything needed to begin exploring the specified device. View a working application and then debug and program you own changes.

Static RAM or SRAM

Static Random Access Memory. Program memory you can read/write on the target board that does not need refreshing frequently.

Status Bar

The Status Bar is located on the bottom of the MPLAB X IDE window and indicates such current information as cursor position, development mode and device, and active tool bar.

Step Into

This command is the same as Single Step. Step Into (as opposed to Step Over) follows a CALL instruction into a subroutine.

Step Over

Step Over allows you to debug code without stepping into subroutines. When stepping over a `CALL` instruction, the next breakpoint will be set at the instruction after the `CALL`. If for some reason the subroutine gets into an endless loop or does not return properly, the next breakpoint will never be reached. The Step Over command is the same as Single Step except for its handling of `CALL` instructions.

Step Out

Step Out allows you to step out of a subroutine which you are currently stepping through. This command executes the rest of the code in the subroutine and then stops execution at the return address to the subroutine.

Stimulus

Input to the simulator, i.e., data generated to exercise the response of simulation to external signals. Often the data is put into the form of a list of actions in a text file. Stimulus may be asynchronous, synchronous (pin), clocked and register.

Stopwatch

A counter for measuring execution cycles.

Storage Class

Determines the lifetime of the memory associated with the identified object.

Storage Qualifier

Indicates special properties of the objects being declared (e.g., `const`).

Symbol

A symbol is a general purpose mechanism for describing the various pieces which comprise a program. These pieces include function names, variable names, section names, file names, struct/enum/union tag names, etc. Symbols in MPLAB X IDE refer mainly to variable names, function names and assembly labels. The value of a symbol after linking is its value in memory.

Symbol, Absolute

Represents an immediate value such as a definition through the assembly `.equ` directive.

System Window Control

The system window control is located in the upper left corner of windows and some dialogs. Clicking on this control usually pops up a menu that has the items “Minimize,” “Maximize,” and “Close.”

T

Target

Refers to user hardware.

Target Application

Software residing on the target board.

Target Board

The circuitry and programmable device that makes up the target application.

Target Processor

The microcontroller device on the target application board.

Template

Lines of text that you build for inserting into your files at a later time. The MPLAB Editor stores templates in template files.

Tool Bar

A row or column of icons that you can click on to execute MPLAB X IDE functions.

Trace

An emulator or simulator function that logs program execution. The emulator logs program execution into its trace buffer which is uploaded to MPLAB X IDE trace window.

Trace Memory

Trace memory contained within the emulator. Trace memory is sometimes called the trace buffer.

Trace Macro

A macro that will provide trace information from emulator data. Since this is a software trace, the macro must be added to code, the code must be recompiled or reassembled, and the target device must be programmed with this code before trace will work.

Trigger Output

Trigger output refers to an emulator output signal that can be generated at any address or address range, and is independent of the trace and breakpoint settings. Any number of trigger output points can be set.

Trigraphs

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

U

Unassigned Section

A section which has not been assigned to a specific target memory block in the linker command file. The linker must find a target memory block in which to allocate an unassigned section.

Uninitialized Data

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

```
int myVar;
```

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

Upload

The Upload function transfers data from a tool, such as an emulator or programmer, to the host PC or from the target board to the emulator.

USB

Universal Serial Bus. An external peripheral interface standard for communication between a computer and external peripherals over a cable using bi-serial transmission. USB 1.0/1.1 supports data transfer rates of 12 Mbps. Also referred to as high-speed USB, USB 2.0 supports data rates up to 480 Mbps.

V

Vector

The memory locations that an application will jump to when either a Reset or interrupt occurs.

Volatile

A variable qualifier which prevents the compiler applying optimizations that affect how the variable is accessed in memory.

W

Warning

MPLAB X IDE – An alert that is provided to warn you of a situation that would cause physical damage to a device, software file, or equipment.

16-bit assembler/compiler – Warnings report conditions that may indicate a problem, but do not halt processing. In MPLAB C30, warning messages report the source file name and line number, but include the text ‘`warning`:’ to distinguish them from error messages.

Watch Variable

A variable that you may monitor during a debugging session in a Watch window.

Watch Window

Watch windows contain a list of watch variables that are updated at each breakpoint.

Watchdog Timer (WDT)

A timer on a PIC microcontroller that resets the processor after a selectable length of time. The WDT is enabled or disabled and set up using Configuration bits.

Workbook

For MPLAB SIM stimulator, a setup for generation of SCL stimulus.

NOTES:

Index

Worldwide Sales and Service

AMERICAS

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

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Austin, TX
Tel: 512-257-3370

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Novi, MI
Tel: 248-848-4000

Houston, TX
Tel: 281-894-5983

Indianapolis
Noblesville, IN
Tel: 317-773-8323
Fax: 317-773-5453
Tel: 317-536-2380

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608
Tel: 951-273-7800

Raleigh, NC
Tel: 919-844-7510

New York, NY
Tel: 631-435-6000

San Jose, CA
Tel: 408-735-9110
Tel: 408-436-4270

Canada - Toronto
Tel: 905-695-1980
Fax: 905-695-2078

ASIA/PACIFIC

Australia - Sydney
Tel: 61-2-9868-6733

China - Beijing
Tel: 86-10-8569-7000

China - Chengdu
Tel: 86-28-8665-5511

China - Chongqing
Tel: 86-23-8980-9588

China - Dongguan
Tel: 86-769-8702-9880

China - Guangzhou
Tel: 86-20-8755-8029

China - Hangzhou
Tel: 86-571-8792-8115

China - Hong Kong SAR
Tel: 852-2943-5100

China - Nanjing
Tel: 86-25-8473-2460

China - Qingdao
Tel: 86-532-8502-7355

China - Shanghai
Tel: 86-21-3326-8000

China - Shenyang
Tel: 86-24-2334-2829

China - Shenzhen
Tel: 86-755-8864-2200

China - Suzhou
Tel: 86-186-6233-1526

China - Wuhan
Tel: 86-27-5980-5300

China - Xian
Tel: 86-29-8833-7252

China - Xiamen
Tel: 86-592-2388138

China - Zhuhai
Tel: 86-756-3210040

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444

India - New Delhi
Tel: 91-11-4160-8631

India - Pune
Tel: 91-20-4121-0141

Japan - Osaka
Tel: 81-6-6152-7160

Japan - Tokyo
Tel: 81-3-6880- 3770

Korea - Daegu
Tel: 82-53-744-4301

Korea - Seoul
Tel: 82-2-554-7200

Malaysia - Kuala Lumpur
Tel: 60-3-7651-7906

Malaysia - Penang
Tel: 60-4-227-8870

Philippines - Manila
Tel: 63-2-634-9065

Singapore
Tel: 65-6334-8870

Taiwan - Hsin Chu
Tel: 886-3-577-8366

Taiwan - Kaohsiung
Tel: 886-7-213-7830

Taiwan - Taipei
Tel: 886-2-2508-8600

Thailand - Bangkok
Tel: 66-2-694-1351

Vietnam - Ho Chi Minh
Tel: 84-28-5448-2100

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

Finland - Espoo
Tel: 358-9-4520-820

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Garching
Tel: 49-8931-9700

Germany - Haan
Tel: 49-2129-3766400

Germany - Heilbronn
Tel: 49-7131-72400

Germany - Karlsruhe
Tel: 49-721-625370

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Germany - Rosenheim
Tel: 49-8031-354-560

Israel - Ra’anana
Tel: 972-9-744-7705

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Italy - Padova
Tel: 39-049-7625286

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Norway - Trondheim
Tel: 47-7288-4388

Poland - Warsaw
Tel: 48-22-3325737

Romania - Bucharest
Tel: 40-21-407-87-50

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

Sweden - Gothenberg
Tel: 46-31-704-60-40

Sweden - Stockholm
Tel: 46-8-5090-4654

UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820

05/14/19