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

Notice to Development Tools Customers

Important:
All documentation becomes dated, and Development Tools manuals are no exception. Our tools and documentation are constantly evolving to meet customer needs, so some actual dialogs and/or tool descriptions may differ from those in this document. Please refer to our website (www.microchip.com/) to obtain the latest version of the PDF document.

Documents are identified with a DS number located on the bottom of each page. The DS format is DS<DocumentNumber><Version>, where <DocumentNumber> is an 8-digit number and <Version> is an uppercase letter.

For the most up-to-date information, find help for your tool at onlinedocs.microchip.com/.

Table of Contents

1. Preface

MPLAB® XC32 C/C++ Compiler for PIC32C/SAM and support information is discussed in this user's guide.

1.1 Conventions Used in This Guide

The following conventions may appear in this documentation:

Table 1-1. Documentation Conventions

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	*MPLAB® XC32 C/C++ Compiler User's Guide for PIC32C/SAM MCUs*
	Emphasized text	...is the *only* compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	Select File and then Save.
Quotes	A field name in a window or dialog	"Save project before build"
Underlined, italic text with right angle bracket	A menu path	*File>Save*
Bold characters	A dialog button	Click **OK**
	A tab	Click the **Power** tab
N'Rnnnn	A number in verilog format, where N is the total number of digits, R is the radix and n is a digit.	4'b0010, 2'hF1
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
Courier New font:		
Plain Courier New	Sample source code	`#define START`
	Filenames	`autoexec.bat`
	File paths	`C:\Users\User1\Projects`
	Keywords	`static`, `auto`, `extern`
	Command-line options	`-Opa+, -Opa-`
	Bit values	`0, 1`
	Constants	`0xFF, 'A'`
Italic Courier New	A variable argument	*`file`*`.o`, where *`file`* can be any valid filename
Square brackets [ ]	Optional arguments	`xc8 [`*`options`*`]` *`files`*
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)`
		`{ ...`
		`}`

1.2 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. You can find an online copy of the Release Notes on Microchip's website.

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.

Microchip Unified Standard Library Reference Guide (DS50002895)

This guide details the functions, types, and preprocessor macros provided by the standard libraries that ship with the compiler.

32-Bit Language Tools Libraries (DS50001685)

Lists all the specialized library functions provided by the MPLAB XC32 C/C++ Compiler, with detailed descriptions of their use.

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

International Standardization Organization (ISO) and International Electrotechnical Commission (IEC) – *ISO/IEC 9899:1999 Programming languages — C*. ISO Central Secretariat, Chemin de Blandonnet 8, CP 401, 1214 Vernier, Geneva, Switzerland.

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

Important:

1. Features and options described in the following GCC and Binutils documentation might not be officially supported by MPLAB XC32. It is recommended that only features described in the Microchip XC32 documentation be used.

gcc.gnu.org/onlinedocs/

sourceware.org/binutils/

Arm Reference Manuals

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.

2. Compiler Overview

The MPLAB® XC32 C/C++ Compiler for PIC32C/SAM MCUs is defined and described in this section.

2.1 Device Support

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

2.2 Compiler Description and Documentation

The MPLAB XC32 C/C++ Compiler is a full-featured, optimizing compiler that translates standard C programs into 32-bit device assembly language code. The toolchain supports the PIC32C/SAM microcontroller families based on a variety of Arm® Cortex®-M*x* and A*x* 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's code generator.

The compiler is based on GCC, the GNU Compiler Collection, from the Free Software Foundation.

The compiler is available for several popular operating systems, including Windows®, Linux® and macOS®.

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.

2.2.1 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 C/ASM language driver for the MPLAB XC32 C/C++ Compiler package is called `xc32-gcc`. The C/C++/ASM language driver is called `xc32-g++`. The drivers and their options are discussed in 6.7. 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.

2.2.2 C Standards

The compiler is a fully validated compiler that conforms to the ISO/IEC 9899:1990 Standard (referred to as the C90 standard) as well the ISO/IEC 9899:1999 Standard (C99) for programming languages. The compiler also supports many PIC32 MCU-oriented language extensions that allow full access to the 32-bit device hardware capabilities, and affords fine control of the compiler's code generator.

2.2.3 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 PIC32C/SAM MCU-specific optimizations that take advantage of the particular features of the device architecture.

For more on optimizations, see 22. Optimizations.

2.2.4 ISO/IEC C Library Support

The MPLAB XC32 C/C++ Compiler provides libraries of functions, macros, types, and objects that can assist with your code development.

The Microchip Unified Standard Library is used for C projects. This library is C99 compliant. The same library implementation is used as the C library (`libc`) within the standard C++ library (`libstdc++`) when building C++ projects.

Several math libraries are available, handling devices with and without a hardware floating-point unit.

The Standard C libraries are described in the separate *Microchip Unified Standard Library Reference Guide* document, whose content is relevant for all MPLAB XC C compilers.

2.2.5 ISO/IEC C++ Library Support

The MPLAB XC32 C/C++ Compiler is distributed with a ISO/IEC 14882:2011 compliant Standard C++ Library (`libstdc++`) based on the GNU C++ Library. The Microchip Unified Standard Library is used as the C library (`libc`) within the standard C++ library when building C++ projects.

The Standard C libraries are described in the separate *Microchip Unified Standard Library Reference Guide* document, whose content is relevant for all MPLAB XC C compilers.

2.2.6 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.

2.3 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 (v5.50 or higher).
- The MPLAB Simulator.
- All Microchip debug tools and programmers.
- Demo boards and starter kits that support 32-bit devices.

3. 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.

3.1 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.

3.1.1 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 (see following note). But, if it is too weak, programmers would see wildly differing results within different compilers and architectures, and the standard would lose its effectiveness.

Note: 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.

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.

This compiler is a freestanding implementation that conforms to the ISO/IEC 9899:1990 Standard (referred to as the C90 standard) as well the ISO/IEC 9899:1999 Standard (C99) for programming languages, unless otherwise stated.

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.

3.1.2 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.

3.2 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 3.5.1. 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.

3.3 ANSI Standard Refinement

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

3.3.1 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.

Example

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

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

Differences

All compilers have used this character set.

Migration to the CCI

No action required.

3.3.2 The Prototype for `main`

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

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

Example

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

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

Differences

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

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.

3.3.3 Header File Specification

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

Example

The following example shows two conforming include directives.

```
#include <usb_main.h>
#include “global.h”
```

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.

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
```

3.3.4 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 be to the same directories searched when the header file is specified in angle bracket delimiters.

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

Example

If including a header file, as in the following directive:

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

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

Differences

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

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.

3.3.5 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.

Example

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

```
int stateOfPortBWhenTheOperatorHasSelectedAutomaticModeAndMotorIsRunningFast;

int stateOfPortBWhenTheOperatorHasSelectedAutomaticModeAndMotorIsRunningSlow;
```

Differences

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

Migration to the CCI

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

3.3.6 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.

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;
```

Differences

This is consistent with previous types implemented by the compiler.

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>`.

3.3.7 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.

Example

The following example

```
char foobar;
```

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

Differences

None.

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.

3.3.8 Signed Integer Representation

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

Example

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

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

Differences

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

Migration to the CCI

No action required.

3.3.9 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.

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).

Differences

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

Migration to the CCI

No action required.

3.3.10 Bitwise Operations on Signed Values

Bitwise operations on signed values act on the two’s complement representation, including the sign bit. (See also 3.3.11. Right-Shifting Signed Values.)

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.

Differences

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

Migration to the CCI

No action required.

3.3.11 Right-Shifting Signed Values

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

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).

Differences

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

Migration to the CCI

No action required.

3.3.12 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.

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;
```

Differences

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

Migration to the CCI

No action required.

3.3.13 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.

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;
};
```

Differences

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

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;
};
```

3.3.14 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.

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).

Differences

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

Migration to the CCI

No action required.

3.3.15 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.

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.

Differences

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

Migration to the CCI

No action required.

3.3.16 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).

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.

Differences

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

Migration to the CCI

No action required.

3.3.17 Floating-Point Sizes

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

Example

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

```
float outY;
```

Migration to the CCI

No migration is required for the XC32 compiler.

3.4 ANSI Standard Extensions

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

3.4.1 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.

The generic device support include file xc.h now includes a header file that contains prototypes for all XC special compiler builtins.

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>
```

Migration to the CCI

No changes required.

3.4.2 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.

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;
}
```

Differences

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

Migration to the CCI

Avoid making objects and functions absolute if possible.

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

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

to:

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

3.4.3 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.

Example

The following shows a variable qualified using `__persistent`.

```
__persistent int serialNo;
```

Differences

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

Migration to the CCI

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

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

to:

```
int __persistent tblIdx;
```

Caveats

None.

3.4.4 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.

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];
```

Differences

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

Migration to the CCI

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

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

to:

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

3.4.5 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 18. 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.

Example

The following shows a function qualified using `__interrupt`.

```
static unsigned long tick_counter;

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

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)
{
   /* ... */
}
```

Caveats

None.

3.4.6 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.

Example

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

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

Differences

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

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.

Caveats

None.

3.4.7 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.

Example

The following shows a function that uses the __deprecate keyword.

```
void __deprecate getValue(int mode)
{
```

```
//...
}
```

Differences

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

Migration to the CCI

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

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

to:

```
int __deprecate intMask;
```

Caveats

None.

3.4.8 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.

Example

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

```
int __section(“comSec”) commonFlag;
```

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.

Migration to the CCI

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

```
int __attribute__((section(“myVars”))) intMask;
```

to:

```
int __section(“myVars”) intMask;
```

Caveats

None.

3.4.9 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.

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
```

Differences

The XC32 compilers supported the use of #pragma config.

Migration to the CCI

No migration is required.

Caveats

None.

3.4.10 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 the following table.

Table 3-1. 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_ _

Example

Code conditionally compiled for a particular device family:

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

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__).

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.

Caveats

None.

3.4.11 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.

Example

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

```
asm(“NOP”);
```

Differences

This is the same syntax used by the XC32 compilers.

Migration to the CCI

No migration is required.

Caveats

None.

3.5 Compiler Features

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

3.5.1 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.

Differences

This option has never been implemented previously.

Migration to the CCI

Enable the option.

Caveats

None.

4. How To's

This section contains help and references 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 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

4.1 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?

4.1.1 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.

4.1.2 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 your 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.

4.1.3 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 4.2.4. 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\
```

4.2 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.

- 4.2.1. How Do I Compile from Within MPLAB X IDE?
- 4.2.2. How Do I Compile on the Command-line?
- 4.2.3. How Do I Compile Using a Make Utility?
- 4.2.4. How Can I Select Which Compiler I Want to Build With?
- 4.2.5. How Can I Change the Compiler's Operating Mode?
- 4.2.6. How Do I Build Libraries?
- 4.2.7. How Do I Know What Compiler Options Are Available and What They Do?
- 4.2.8. How Do I Know What the Build Options in MPLAB X IDE Do?
- 4.2.9. What is Different About an MPLAB X IDE Debug Build?
- See also 4.3.3.4. How Do I Stop the Compiler From Using Certain Memory Locations?
- See also 4.4.3. What Do I Need to Do When Compiling to Use a Debugger?
- See also 4.5.16. How Do I Use Library Files In My Project?
- See also 4.5.8. What Optimizations Will Be Applied to My Code?

4.2.1 How Do I Compile from Within MPLAB X IDE?

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

- 5.4. Project Setup - MPLAB X IDE

4.2.2 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 6.1. Invoking the Compiler. See 4.2.4. 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 6.7. Driver Option Descriptions. The files that can be passed to the driver are listed and described in 6.1.3. Input File Types.

4.2.3 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 6.2.2. Multi-Step C Compilation.

4.2.4 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.

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

The compiler’s operating mode (Free, Evaluation or PRO) purely determines the allowable level of optimization (see 22. Optimizations), which can be specified as a command line option (see 6.7.7. 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 5.4.4. xc32-g++ (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.

4.2.6 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 6.4.1. Library Files, "User-Defined Libraries" sub-section.

4.2.7 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 6.7. Driver Option Descriptions in this user’s guide. If you are compiling in MPLAB X IDE, see 5.4. Project Setup.

4.2.8 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 5.4. Project Setup for a list of options and any corresponding command-line options.

4.2.9 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, etc (see 6.7.8. 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 4.4.3. What Do I Need to Do When Compiling to Use a Debugger?.

4.3 Writing Source Code

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

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

4.3.1 C Language Specifics

This section discusses source code issues that are directly relates 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?

4.3.1.1 When Should I Cast Expressions?

Expressions can be explicitly cast using the cast operator -- a type in round brackets, for example, (`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 an `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 value on the left of the operator (`short`).This is called an implicit type conversion. The compiler typically produces 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 performs 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 works the same way; but if in the future, the type of `s` is changed to a `long`, for example, then you must remember to 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.

Only use a cast 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 yields 3 and this is 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` forces the compiler to encode a floating-point division.) The result assigned to `fl` now is 3.5.

An explicit cast can 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.

4.3.1.2 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 14.1. 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 4.3.1.1. When Should I Cast Expressions?

4.3.1.3 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 9.8. Constant Types and Formats to see how these characters can be entered using escape sequences.

4.3.1.4 How Can I Use A Variable Defined In Another Source File?

Provided the variable defined in the other source file is not `static` (see 10.2.2. Static Variables) or `auto` (see 10.3. Auto Variable Allocation and Access), adding a declaration for that variable into the current file will allow you to access it. A declaration consists of the keyword `extern` in addition to the type and the name of the variable, as specified in its definition, for example,

```
extern int systemStatus;
```

This is part of the C language. Your favorite C textbook will give you more information.

The position of the declaration in the current file determines the scope of the variable. That is, if you place the declaration inside a function, it will limit the scope of the variable to that function. If you place it 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 then they are `#include`d into the C source code (see 23.3. Pragma Directives).

4.3.1.5 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 3. Common C Interface.

4.3.2 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.

- 4.3.2.1. How Do I Set the Configuration Bits?
- 4.3.2.2. How Do I Access SFRS?
- 4.3.3.4. How Do I Stop the Compiler From Using Certain Memory Locations?

See also the following linked information in other sections.

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

4.3.2.1 How Do I Set the Configuration Bits?

These should be set in your code using either a macro or pragma. Earlier verions of the 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 8.3. Configuration Bit Access for more information on the config pragma.

4.3.2.2 How Do I Access SFRS?

The compiler ships with header files that define variables that are mapped over the top of memory-mapped Special Function Registers (SFRs). Some device documentation refers to these as peripheral registers. Since these mapped variables are regular C variables, they can be used like any other object, and no new syntax is required to access the underlying register.

The name assigned to the variable is usually the same as the name specified in the device data sheet. See 4.3.2.3. How Do I Find The Names Used To Represent SFRs And Bits? if these names are not recognized.

4.3.2.3 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, 4.3.2.2. 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.

4.3.3 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?

4.3.3.1 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 9.11. Variable Attributes) or the `__at()` CCI construct (see 3.4.2. 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 10.3. Auto Variable Allocation and Access for information on moving auto variables, 10.2.1. Non-auto Variable Allocation for moving non-auto variables and 10.4. Variables in Program Memory for moving program-space variables.

4.3.3.2 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 17.2.1. 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.

4.3.3.3 How Do I Place Variables in Program Memory?

The `const` qualifier implies that the qualified object is read only. An object qualified with const might also be placed in program memory, but its exact location will depend on the use of options such as `-fdata-sections`, and `-mpure-code`, any linker-script customizations, and whether the variable is initialized. To ensure that objects are placed in program memory, use the `space(prog)` attribute in addition to the `const` qualifer.

4.3.3.4 How Do I Stop the Compiler From 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:

9.11. Variable Attributes

6.7.1. Options Specific to PIC32C/SAM Devices

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

4.3.4 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?

4.3.4.1 Why Are My Floating-Point Results Not Quite What I Am Expecting?

Make sure that if you are watching floating-point variables in MPLAB IDE that the type and size of these match how they are defined. In MPLAB XC32, the `float` type is a 32-bit floating-point type, while `double` and `long double` types are a 64-bit floating-point type.

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.

4.3.4.2 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, respectively, test to see if bit number, `bit`, in the integer, `var`, is set; 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 9.5.2. 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`).

4.3.4.3 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 this number is and it varies from compiler to compiler. For MPLAB XC32, no limit is imposed, but for CCI there is a limit (see 3.3.5. 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.

4.3.5 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?

4.3.5.1 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 fewer 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 17.5. Function Size Limits. The smaller the functions, the easier it is for the linker to allocate them to memory without errors.

4.3.5.2 How Do I Stop An Unused Function Being Removed?

The `__attribute__((keep,used))` may be applied to a function. The `keep` attribute tells the compiler to put the function into a section that won't be removed by the linker, even if the `--gc-sections` option is used to perform garbage collection. The `used` attribute stops the compiler from applying optimizations that would remove the function. It also suppresses warnings related to unused functions.

4.3.5.3 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; 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)++;
}
```

For a function qualified 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 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. Enable optimization level `–O1` or greater to enable function inlining.

4.3.6 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?

4.3.6.1 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 18.1. Interrupt Operation.

Prior to any interrupt occurring, your program must ensure that peripherals are correctly configured and that interrupts are enabled. For details, see 18.7. Enabling/Disabling Interrupts.

For all other interrupt related tasks, including specifying the interrupt vector, context saving, nesting and other considerations, consult 18. Interrupts.

4.3.7 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?

4.3.7.1 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 21.1. 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 21.2. Using Inline Assembly Language.

Macros are provided which in-line several simple instructions, as discussed in 21.3. Predefined Macro. More complex in-line assembly that changes register contents and the device state should be used with caution.

See 15.1. Register Usage for those registers used by the compiler.

4.3.7.2 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 aforementioned user's guide.

4.3.7.3 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 21.1. 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 9.9.2. Volatile Type Qualifier). Specifying the `volatile` qualifier in C code is preferred as it makes it clear that external code may access the object.

4.3.7.4 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.

4.3.7.5 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 in-line 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 15.1. Register Usage, registers). No assumptions are made for in-line assembly (see 21.1. 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.

4.4 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?

4.4.1 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 9.9.2. 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.

4.4.2 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.

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

You can use debuggers, such as the PICkit™ 5 in-circuit debugger or the MPLAB ICD 5 in-circuit debugger, to debug code built with the MPLAB XC32 compiler. When a debug Run is requested, a debug executive is automatically downloaded with your program image. This debug executive might use some of the device memory for its own purposes, making this memory off-limits to your program. These resources will not be used if you are not debugging and you perform a regular Run, Build Project, or Clean and Build.

Typically, no memory will be required by the debug executive when the debugger is connected to PIC32C/SAM devices (although this is not necessarily true for PIC32M devices). It is important that no memory used by the debugger is also used by your program.

Any memory locations used by hardware tools when debugging are attributes of MPLAB X IDE, which contains and downloads the debug executive executed on the targte device. The IDE uses the `-mreserve` option to reserve any memory required by the debug executive when it builds for debugging. If you move a project to a new version of the IDE, the resources required might 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. Check the Reserved Resources information, available in the MPLAB X IDE's **Start page**, under **Learn and Discover**, for full information on which resources are used by the debugger.

See also 4.5.14. Why are Some Objects Positioned into Memory that I Reserved?.

4.4.4 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 mis-read by the program. The `volatile` qualifier (see 9.9.2. 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 18.7. Enabling/Disabling Interrupts for more information.

4.4.5 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 8.3. Configuration Bit Access and 3.4.9. Specifying Configuration Bits for CCI as well as 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.

4.4.6 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 and 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 standard `printf` function, see the *Microchip Unified Standard Library Reference Guide*.

To have the compiler statically analyze what format strings are passed to the `printf` function, you may use the `-msmart-io` option (6.7.1. Options Specific to PIC32C/SAM Devices). Also you may 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 6.7.5. 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 17.2.1. Function Attributes.

4.4.7 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.

4.4.8 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 3.3.10. Bitwise Operations on Signed Values and 3.3.11. Right-Shifting Signed Values if you will be using signed variables.

For the following example C code:

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

the compiler may generate assembly instructions similar to the following:

```
rotate_left:
  rsb r1, r1, #32
  rors r0, r0, r1
  bx lr
```

4.5 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?
- 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?

4.5.1 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 22. 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.

4.5.2 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 (which also reduces RAM usage). For examples, a `short` integer type exists for this compiler. See 9. 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, i.e., the following:

```
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. If you have a Pro edition, you can use the `-Os` option (see 6.7.7. 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.

4.5.3 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 10.3. 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 10.4. Variables in Program Memory). This frees up precious RAM, but slows execution.

4.5.4 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 4.5.2. How Can I Make My Code Smaller? and 4.5.6. 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, you may be able to use the `-O3` option (see 6.7.7. 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.

4.5.5 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 under 9.11. Variable Attributes.

4.5.6 How Can I Make My Interrupt Routine Faster?

Consider suggestions made in 4.5.2. 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. This is executed immediately after an interrupt occurs and immediately before the interrupt returns, so 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 9.9.2. Volatile Type Qualifier for variables shared by the interrupt and main-line code, see 4.4.4. How Do I Share Data Between Interrupt and Main-Line Code?.

If your target device supports tightly coupled memory (see 8.5. Tightly-Coupled Memories) it can help improve the determinism and performance of an ISR.

4.5.7 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 10.4.1. Size Limitations of const Variables. Objects in data memory are broadly grouped into autos and non-autos and the size limitations of these objects, respectively, are discussed in 10.2.1. Non-auto Variable Allocation and 10.2.3. Non-Auto Variable Size Limits.

4.5.8 What Optimizations Will Be Applied to My Code?

Code optimizations available depend on the edition of your compiler (see 22. Optimizations). A description of optimization options can be found under 6.7.7. Options for Controlling Optimization.

4.5.9 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.

You can typically add support for a new device without updating the compiler once a device family pack (DFP) becomes available for that device. DFPs can be downloaded from within the MPLAB X IDE. See also 6.7.1.5. Dfp Option.

4.5.10 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).

4.5.11 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 4.5.10. How Do I Know What Code the Compiler Is Producing? for more on creating an assembly listing file.

4.5.12 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 4.5.10. 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 10.3. Auto Variable Allocation and Access.

4.5.13 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).

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

Most variables and function 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.

4.5.15 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.

Additionally, a Memory Report by Module, showing the memory usage (text, data, and bss sections) per object file, is provided. This report indicates the size in the final ELF file attributable to each input object. A miscellaneous entry also shows those sections whose input object file cannot be determined or the type of memory cannot be established.

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.

4.5.16 How Do I Use Library Files In My Project?

See 4.2.6. 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.

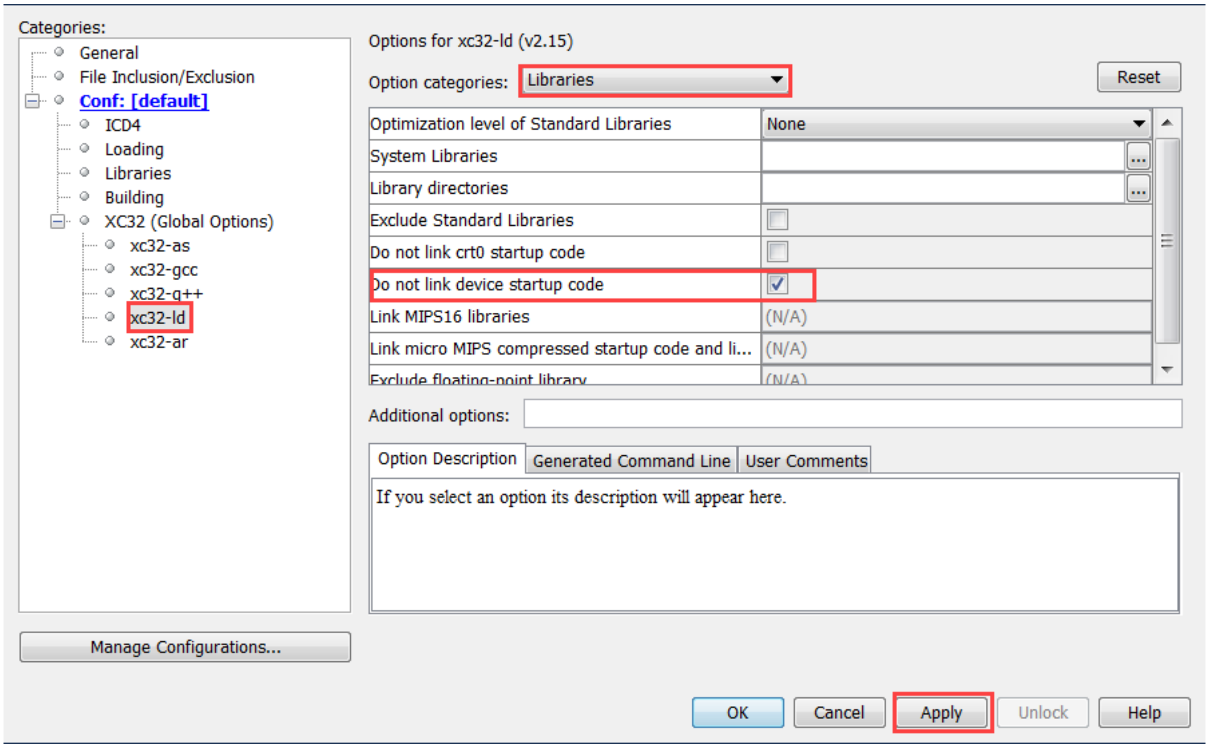
4.5.17 How Do I Customize the C Runtime Startup Code?

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>/startup_<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 4-1. Startup Code Properties Setting



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:

If you select an option its description will appear here.

4.6 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?

4.6.1 How Do I Set Up Warning/Error Messages?

To control message output, see 6.7.5. Options for Controlling Warnings and Errors.

4.6.2 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 in some situations even though that 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 5. 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.

4.6.3 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 6.7.5. Options for Controlling Warnings and Errors for details.

4.6.4 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 8.3. 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 8.3. 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.

4.6.5 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 4.4.4. How Do I Share Data Between Interrupt and Main-Line Code? for more information.

5. 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

5.1 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.

Attention: This version of the C compiler requires MPLAB X IDE v5.50 or higher.

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 - `xc32-`*`utility`*`.exe`

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.

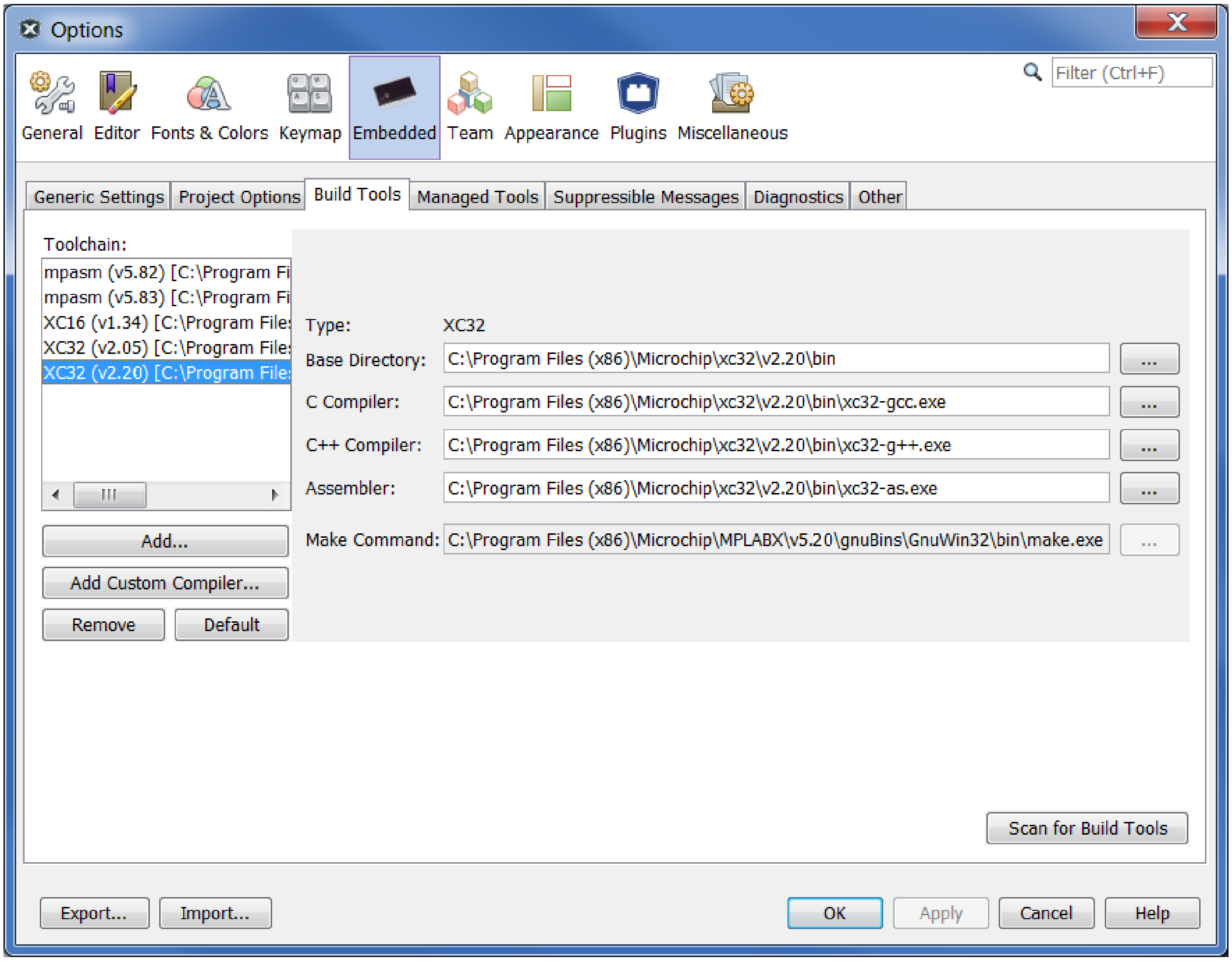
Code examples are located in the examples directory.

5.2 MPLAB X IDE Setup

Once the MPLAB X IDE is installed on your host machine, 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. Ensure the MPLAB XC32 compiler version(s) you have installed are listed under **Toolchain:**.
3. Select the desired XC32 tool and ensure that the paths are correct for your installation.
4. Click **OK**.

Figure 5-1. XC32 Toolsuite Locations in Windows® OS



Toolchain:

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

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

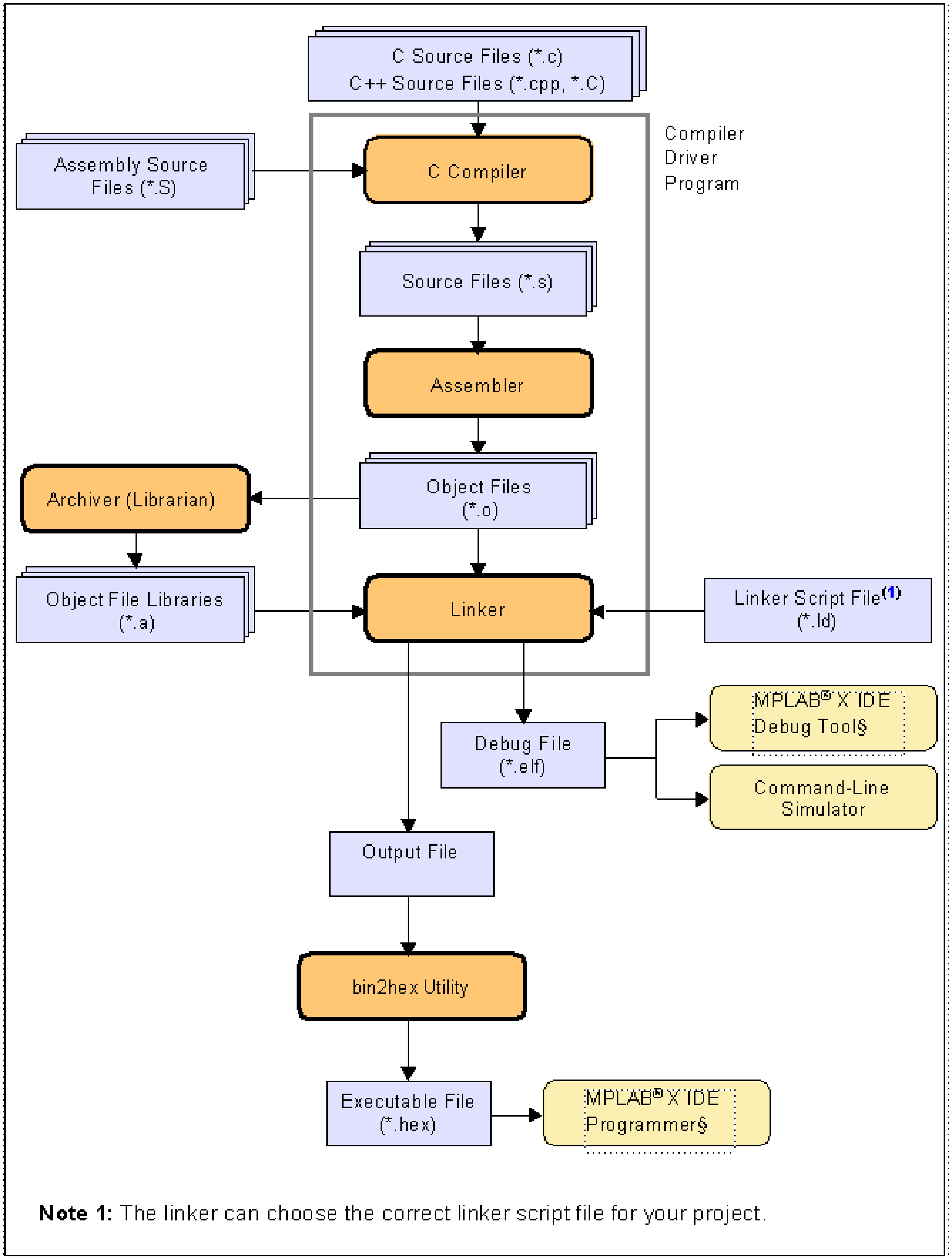
Assembler: C:\Program Files (x86)\Microchip\xc32\v2.20\bin\xc32-as.exe

Make Command: C:\Program Files (x86)\Microchip\MPLABX\v5.20\gnuBins\GnuWin32\bin\make.exe

5.3 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 5-2. 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.

5.4 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.

5.4.1 XC32 (Global Options)

Set up global options for all 32-bit language tools. See also “5.4.7. Options Page Features”.

Table 5-1. XC32 (Global Options) All Options Category

Option	Description	Command Line

Stack Smashing Protector	Selects those functions protected from stack overflow.	`-fstack-protector`
		`-fstack-protector-strong`
		`-fstack-protector-all`

Override default device support	The **Do not override** selection will let the MPLAB X IDE provide a list of its own DFPs that can be selected. The **Compiler location** will use the DFPs that ship with the compiler rather than the IDE.	`-mdfp`
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`*
Data TCM size in bytes	Enable data Tightly Coupled Memory with the specified size.	`-mdtcm=`*`n`* where *`n`* is the size specified by user
Instruction TCM size in bytes	Enable instruction Tightly Coupled Memory with the specified size.	`-mitcm=`*`n`*

...........continued

Option	Description	Command Line
Locate stack data in TCM	Locate the stack in data Tightly Coupled Memory.	`-mstack-in-tcm`

5.4.2 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. See also "5.4.7. Options Page Features".

Table 5-2. xc32-as General 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` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Preprocessor macro definitions	Project-specific preprocessor macro defines passed via the compiler's `-D` option.	`-Dmacro[=`*`defn`*`]` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Assembler symbols	Define symbol 'sym' to a given 'value'.	`--defsym` *`sym=value`* See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Preprocessor Include directories	Relative paths are from MPLAB X project directory.	`-I` *`dir`* See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*
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` *`dir`* See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.

Table 5-3. xc32-as Other Options Category

Option	Description	Command Line
Diagnostics level	Select warnings to display in the Output window. Select "Generate warnings" to have the usual warnings issued by the compiler; "Suppress warnings"to have only errors displayed, and "Fatal Warnings" to have the assembler treat warnings as if they were errors.	`--[no-]warn` `--fatal-warnings` See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*
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` See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*
Expand macros	Check to expand macros in a listing. Uncheck to collapse macros.	`-am` See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*
Include false conditionals	Check to include false conditionals (`.if, .ifdef`) in a listing. Uncheck to omit false conditionals.	`-ac` See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*

...........continued

Option	Description	Command Line
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` See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*
Include assembly	Check for an assembly listing. This `-a` suboption may be used with other suboptions. Uncheck to exclude an assembly listing.	`-al` See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*
List symbols	Check for a symbol table listing. Uncheck to exclude the symbol table from the listing.	`-as` See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*
Omit debugging directives	Check to omit debugging directives from a listing. This can make the listing cleaner. Uncheck to included debugging directives.	`-ad` See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*
List to file	Use this option if you want the assembly listing for any assembly source files in the project. They will have the same basename as the source, with a `.lst` extension.	`-a=`*`file`*`.lst` See the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*

5.4.3 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 5.4.7. Options Page Features.

Table 5-4. xc32-gcc General Category

Option	Description	Command Line
Enable 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-ARMv6 and all ARMv6-M architectures and enabled for all other architectures. If unaligned access is not enabled then words in packed data structures are accessed one byte at a time.	`-m[no-]unaligned-access`
Have symbols in production build	Adds debugging information to the generated ELF file.	`-g`
Isolate each function in a section	This option is often used with the linker's `--gc-sections` option to remove unreferenced functions (see the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*). 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 be slower executing. Uncheck to place multiple functions in a section.	`-ffunction-sections`

...........continued

Option	Description	Command Line
Place data into its own section	This option is often used with the linker's `--gc-sections` option to remove unreferenced statically-allocated variables (see the *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*). 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`
Enable toplevel reordering	Allow reordering of top-level functions, variables, and `asm` statements, such that they might not be output in the same order in which they appear in source files. When this feature is disabled, using the `-fno-toplevel-reorder` option, unreferenced static variables will not be removed. Use this options with optimization level 1 or greater.	`-f[no-]toplevel-reorder`
Use indirect calls	Enable full-range calls, which are typically required when calling a function in another memory region.	`-mlong-calls`

Note that some of the compiler options specified by fields in Project Property Categories other than Optimization can affect the size and execution speed of your project. Consider using the Compiler Advisor, accessible via the MPLAB X IDE **Tools** > **Analysis** > **Compiler Advisor** menu item, to compare the size of your project when built with different combination of compiler options.

Table 5-5. 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 for size. This level enables all `-O2` optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.	`-O0 -O1 -O2 -O3 -Os`
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	Attempts to reorder instructions to eliminate instruction stalls. Select "Default for optimization level" for this feature to be controlled purely by the `-O` level selection.	`-f[no-]schedule-insns`

...........continued

Option	Description	Command Line
Post-optimization instruction scheduling	Attempts to reorder instructions to eliminate instruction stalls. Select "Default for optimization level" for this feature to be controlled purely by the `-O` level selection.	`-f[no-]schedule-insns2`
Use Common Tentative Definition	Controls the placement of global variables defined without an initializer, known as tentative definitions in the C standard. Tentative definitions are distinct from declarations of a variable with the `extern` keyword, which do not allocate storage. When the feature is enabled, using the `-fcommon` option, storage for uninitialized global objects will be allocated in a common block. This allows the linker to resolve tentative and non-tentative definitions of the same object across translation units. Disabling the feature, using the `-fno-common` option, will have the compiler place uninitialized global objects in the data section of the corresponding object file. Doing so will prevent the linker from associating tentative definitions across translation units, resulting in multiple-definition errors if such definitions are encountered.	`-f[no-]common`.

Table 5-6. xc32-gcc Preprocessing and Messages Category

Option	Description	Command Line
Preprocessor macros	Project-specific preprocessor macro defines passed via the compiler's `-D` option.	`-D`*`macro`*`[=`*`defn`*`]`
Include directories	Search these directories for project-specific include files.	`-I`*`path`*
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`
Enable address warning attribute	Emit a warning for all uses of the `address` attribute. This option is for engineering-support use only.	`-Waddress-attribute-use`
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 3. Common C Interface).	`-mcci`

5.4.4 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. See also 5.4.7. Options Page Features.

Table 5-7. 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.	`-f[no-]rtti`

...........continued

Option	Description	Command Line
Enable C++ exception handling	Enable exception handling. Generates extra code needed to propagate exceptions.	`-f[no-]exceptions`
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. Using this option may increase code size in production builds.	`-fenforce-eh-specs`

Table 5-8. xc32-g++ General Category

Option	Description	Command Line
Enable 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-ARMv6 and all ARMv6-M architectures and enabled for all other architectures. If unaligned access is not enabled then words in packed data structures are accessed one byte at a time.	`-m[no-]unaligned-access`
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`
Enable toplevel reordering	Allow reordering of top-level functions, variables, and `asm` statements, such that they might not be output in the same order in which they appear in source files. When this feature is disabled, using the `-fno-toplevel-reorder` option, unreferenced static variables will not be removed. Use this options with optimization level 1 or greater.	`-f[no-]toplevel-reorder`
Use indirect calls	Enable full-range calls, which are typically required when calling a function in another memory region.	`-mlong-calls`

Table 5-9. xc32-g++ Optimization Category

Option	Description	Command Line
Optimization Level	Select an optimization level. Your compiler edition may support only some optimizations. Equivalent to `-O`*n* 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).	`-O0 -O1 -O2 -O3 -Os`

...........continued

Option	Description	Command Line
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	Attempts to reorder instructions to eliminate instruction stalls. Select "Default for optimization level"for this feature to be controlled purely by the `-O` level selection.	`-f[no-]schedule-insns`
Post-optimization instruction scheduling	Attempts to reorder instructions to eliminate instruction stalls. Select "Default for optimization level"for this feature to be controlled purely by the `-O` level selection.	`-f[no-]schedule-insns2`

Table 5-10. xc32-g++ Preprocessing and Messages Category

Option	Description	Command Line
Preprocessor macros	Project-specific preprocessor macro defines passed via the compiler's `-D` option.	`-Dmacro[=defn]`
Include directories	Search these directories for project-specific include files.	`-I dir`
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`
Enable Address-attribute warning	Emit a warning for all uses of the `address()` attribute. This option is for engineering-support use only.	`-Waddress-attribute-use`
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 (3. Common C Interface).	`-mcci`

5.4.5 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. See also 5.4.7. Options Page Features.

Table 5-11. 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>` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
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. 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>` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.

...........continued

Option	Description	Command Line
Allow overlapped sections	Check to ignore overlaps of section addresses. This feature is provided for diagnostic purposes and should be used only to diagnose a linker error related to section-allocation issues. Uncheck to check for overlaps.	`--[no-]check-sections` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Remove unused sections	Check to enable garbage collection of unused input sections (on some targets). This is typically used with the controls that place data/functions into their own section. Uncheck to disable garbage collection.	`--[no-]gc-sections` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
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.	
Write start linear address record	Use the ELF file's entry-point field to write a Start Linear Address (SLA) record (type 0x05) to the hex file. The hex record may be useful for a bootloader that needs to determine the entry point to the application. You can set the value using the `ENTRY` command line the linker script.	`--write-sla` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Additional driver options	Type here any additional driver options that do not have dedicated GUI widgets in the Project Properties dialog. The string entered here will be emitted verbatim with the other driver options.	

Table 5-12. xc32-ld Libraries Category

Option	Description	Command Line
Optimization level of Standard Libraries	Select the optimization level with which libraries were built. Your compiler edition may support only some library optimizations. Equivalent to `-On` option emitted for the link stage of compilation, 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 for size. This level enables all `-O2` optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.	`-On` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
System Libraries	Add libraries to be linked with the project files. You may add more than one.	`--library=name` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Library directories	Add a library directory to the library search path. You may add more than one.	`--library-path="name"` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.

...........continued

Option	Description	Command Line
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` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Do not link crt0 startup code	Exclude the default startup code because the project provides application-specific startup code.	`-nostartfiles` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Do not link device startup code	Do not link the default device-specific startup code (for example, `startup_`*`device`*`.c`)	`-mno-device-startup-code` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.

Table 5-13. xc32-ld Diagnostics Category

Option	Description	Command Line
Generate map file	Create a map file.	`-Map="`*`file`*`"` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Display memory usage	Check to print memory usage report. Uncheck to not print a report.	`--report-mem` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Generate cross-reference file	Check to create a cross-reference table. Uncheck to not create this table.	`--cref` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Warn on section realignment	Check to have warnings printed if the start of a section changes due to alignment. This feature is provided for diagnostic purposes and should be used only to diagnose section-alignment issues. Uncheck to inhibit warning.	`--warn-section-align` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Trace Symbols	Add/remove trace symbols.	`-Y `*`symbol`* or `--trace-symbol `*`symbol`* See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.

Table 5-14. 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` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Preprocessor macro definitions	Define preprocessor macros for use in linker scripts (when the option is passed to xc32-gcc directly).	`-Dmacro` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.
Symbols	Specify symbol information in the output.	`-S` or `--strip-debug`; `-s` or `--strip-all` See *MPLAB® XC32 Assembler, Linker and Utilities User's Guide*.

5.4.6 Analysis

Select a category and then set up analysis options.

See also 5.4.7. Options Page Features.

Table 5-15. Analysis General Options Category

Option	Description	Command Line
Code coverage instrumentation	Enable the code coverage feature	`-mcodecov`
Stack guidance	Check to enable the stack guidance feature, which gives an estimate of stack usage.	`-mchp-stack-usage`

5.4.7 Options Page Features

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

Table 5-16. 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.

5.5 Project Example

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

5.5.1 Run the Project Wizard

In MPLAB X IDE, select *File>New Project* 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 ATSAME70N20B. 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.

5.5.2 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.

5.5.3 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.

5.5.4 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 5.4.1. XC32 (Global Options).

5.5.5 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 Projec*t 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 “Make and Program Device Project” button on the debug toolbar. Please see MPLAB X IDE documentation concerning this control.

6. Command-line Driver

The MPLAB XC32 C/C++ Compiler command-line driver, (`xc32-gcc` or `xc32-g++`), can be invoked to perform all aspects of compilation, including C/C++ code generation, assembly and link steps. Its use is the recommended way to invoke the compiler, as it hides the complexity of all the internal applications and provides a consistent interface for all compilation steps. Even if an IDE is used 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.

This chapter describes the steps that the driver takes during compilation, the files that the driver can accept and produce, as well as the command-line options that control the compiler's operation.

6.1 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.

6.1.1 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 6.7. 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 6.1.3. 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.

6.1.2 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 6-1. 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.`
`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.

6.1.3 Input File Types

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

Table 6-2. 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.

6.2 The C Compilation Sequence

6.2.1 Single-Step C Compilation

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

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/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. Some documentation uses the term "peripheral registers" to describe these device registers.

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, 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`.

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.

6.2.2 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 6.7.2. 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 6.2.1. Single-Step C Compilation subsection "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
```

6.3 The C++ Compilation Sequence

6.3.1 Single-Step C++ Compilation

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

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/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), or as they are sometimes called, peripheral registers, on the target device. The `<iostream>` header file 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`.

6.3.2 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++, a heap is provided, 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.

6.4 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

6.4.1 Library Files

The Microchip Unified Standard C library contains a standardized collection of functions, such as string, math and input/output routines. The Microchip Unified Standard Libraries will be linked with your project when it contains C-only source and is built with the `xc32-gcc` driver. When you build a project that contains C++ code with the `xc32-g++` driver, the Microchip Unified Standard Libraries and libstdc++ libraries will be linked.

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 a multilib library appropriate for the selected options. You do not normally need to specify the search path for the standard libraries, nor manually include library files into your project. How to use these functions is described in 20. Libraries.

The multilib startup libraries are located under directories within the `lib/gcc/pic32c/`*`gcc-version`* directory of your compiler distribution, and the target-specific libraries are stored under directories in the `pic32c/lib` directory.

For all devices, the Thumb ISA library version can be selected for projects by specifying the `-mthumb` option to `xc32-gcc` at link time. For ARM9, Cortex-A5 and Cortex-A7 based devices, the ARM ISA library version can be additionally selected by specifying the `-marm` option at link time.

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

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

The C++ target libraries that are distributed with the compiler are built using combinations of the following command-line options:

- Exception support (`-fno-exceptions`, `-fno-rtti`)

The following examples provide details on which of the multilibs subdirectories are chosen. To ensure correct program operation and optimal performance, the options that determine the selected library should be used consistently when compiling all modules and at link time.

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.

6.4.2 Peripheral Library Functions

For All PIC32 devices, see 20.3. Using Library Routines.

6.4.3 Start-Up and Initialization

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

The source code for the startup modules, which initialize the runtime environment, is platform specific and can be found in files matching `startup_`*`device`*`.c` or `startup_`*`device`*`.S` within the directories `pic32c/lib/proc/`*`DEVICE`*`/` in your compiler's distribution directory. In general, the default device-specific startup code is written in C for microcontrollers (MCU) and assembly for microprocessors (MPU), since the microcprocessor initialization code requires some instructions that aren't normally available from standard C code.

Note: The device-specific runtime startup source code is also packaged in the Device Family Pack (DFP) relevant for your target device. However, some frameworks like MPLAB Harmony v3, provide their own application specific startup code rather than the default provided in a DFP.

Prebuilt startup object files are found in architecture specific directories under the `lib/gcc/pic32c/<gcc-version>/` directory of your compiler distribution.

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 19.2. Runtime Start-up Code.

6.5 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.

6.5.1 Output Files

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

Table 6-3. File Names

Extensions	Definition
`file.hex`	Intel HEX executable file
`file.elf (a.out)`	ELF debug file
`file.o`	Object file (intermediate file)
`file.s`	Assembly code file (intermediate file)

...........continued

Extensions	Definition
`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`.

If no `-o` option is given to rename the final output, the compiler writes to a file called `a.out`. This is an ELF file. Typically, ELF files are manually assigned a `.elf` extension.

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 6.7. Driver Option Descriptions.

6.5.2 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 6.7.9. 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 6.2.2. 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 6.7.10. Options for Linking.

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

6.6 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 Message**s 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 6.7.5. Options for Controlling Warnings and Errors.

6.7 Driver Option Descriptions

Most aspects of the compilation process can be controlled using options passed to the command-line driver, `xc32-gcc`.

The GCC compiler on which the MPLAB XC32 C/C++ Compiler is based provides many options in addition to those discussed in this document. It is recommended that you avoid any option that has not been documented here, especially those that control the generation or optimization of code.

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

Use the `--help` option to obtain a brief description of accepted options on the command line.

If you are compiling from within the MPLAB X IDE, it will issue explicit options to the compiler that are based on the selections in the project's **Project Properties** dialog. The default project options might be different to the default options used by the compiler when running on the command line, so you should review these to ensue that they are acceptable.

6.7.1 Options Specific to PIC32C/SAM Devices

Table 6-4. PIC32C/SAM Device-specific Options

Option (links to explanatory section)	Definition
`-f[no-]stack-protector-f[no-]stack-protector-strong-f[no-]stack-protector-all`	Controls stack protection.
`-m[no-]allow-partial-config-words`	Specify that a program can be built with only some Configuration Words defined.
`-mchp-stack-usage`	Generate stack usage information and warnings.
`-mcodecov=`*`options`*	Instrument the output to provide code coverage information.
`-mdfp=`*`path`*	Specifies which device family pack to use.
`-mdtcm=`*`size`*	Specifies the size (in bytes) of the data tightly couple memory.
`-mfloat-abi=`*`name`*	Specifies which floating-point ABI to use.
`-mitcm=`*`size`*	Specifies the size (in bytes) of the instruction tightly couple memory.
`-m[no-]long-calls`	Specifies whether functions are called using an address stored in a register.
`-mprint-builtins`	Print a list of enabled builtin functions.
`-mprocessor`	Selects the device for which to compile.

...........continued

Option (links to explanatory section)	Definition
-m[no-]pure-code	This option ensures the compiler will not place const-qualified data into any section that also contain executable code.
-msmart-io=[0|1|2]	Controls the feature set of the IO library linked in.
-mtcm=*size*	Specifies the size (in bytes) of the single-area tightly couple memory.
-mthumb	Generate code using the Thumb instruction set.
-m[no-]unaligned-access	Controls access of 16- and 32-bit values from addresses that are not 16- or 32-bit aligned.
--nofallback	Require an MPLAB XC32 Pro license and do not fall back to a lesser license.

6.7.1.1 Stack-protector Options

The -fstack-protector option enables stack protection for vulnerable functions that contain:

- A character array larger than 8 bytes
- An 8-bit integer array larger than 8 bytes
- A call to alloca() with either a variable size or a constant size bigger than 8 bytes

The -fstack-protector-strong form of this option enables stack protection for vulnerable functions that contain:

- An array of any size and type
- A call to alloca()
- An automatic local variable that has its address taken

The -fstack-protector-all form of this option adds stack protection to all functions regardless of their vulnerability.

The -fno-stack-protector form of this option disables stack protection and is the default action if no option is specified.

Note: If you specify more than one of these options, the last option that is specified takes effect.

6.7.1.2 Allow-partial-config-words Option

The -mallow-partial-config-words option allows a program to be built with only some of the Configuration Words specified using the config pragma. Using this form of the option with caution, as the combination of configuration bits that have been specified and those left in their default state may render the device inoperable.

The -mno-allow-partial-config-words form of this option will prevent the program from being built if the config pragma has been used to specify only some of the Configuration Words. This is the default action if no option is specified. When using this option, a program can be built with no configuration words specified, but if any words are specified, they must all be specified within a single translation unit.

6.7.1.3 Stack Guidance Option

The -mchp-stack-usage option analyzes the program and reports on the estimated maximum depth of any stack used by a program. The option can only be enabled with a PRO license.

See 16.4. Stack Guidance for more information on the stack guidance reports that are produced by the compiler.

6.7.1.4 Codecov Option

The -mcodecov=*suboptions* option embeds diagnostic code into the program's output, allowing analysis of the extent to which the program's source code has been executed. See 8.6. Code Coverage for more information.

A suboption must be specified and at this time, the only available suboption is ram.

6.7.1.5 Dfp Option

The `-mdfp=path` option indicates that device-support for the target device (indicated by the –`mprocessor` option) should be obtained from the contents of a Device Family Pack (DFP), where *`path`* is the path to the `XC32` sub-directory of the DFP.

When this option has not been used, the `xc32-gcc` driver will where possible use the device-specific files provided in the compiler distribution.

The Microchip development environments automatically uses this option to inform the compiler of which device-specific information to use. Use this option on the command line if additional DFPs have been obtained for the compiler.

A DFP might contain such items as device-specific header files, configuration bit data and libraries, letting you take advantage of features on new devices without you having to otherwise update the compiler. DFPs never contain executables or provide bug fixes or improvements to any existing tools or standard library functions.

6.7.1.6 Dtcm Option

The `-mdtcm=size` option specifies the size (in bytes) of the data tightly coupled memory for devices where this is implemented separately to instruction TCM. See 8.5. Tightly-Coupled Memories.

6.7.1.7 Float-abi Option

The `-mfloat-abi=name` option specifies which floating-point ABI to use. Permissible values are: `soft`, `softfp` and `hard`.

Specifying `soft` as the argument causes XC32 to generate output containing library calls for floating-point operations. The `softfp` argument allows the generation of code using hardware floating-point instructions, but still uses the soft-float calling conventions. The `hard` argument 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.

6.7.1.8 Itcm Option

The `-mitcm=size` option specifies the size (in bytes) of the instruction tightly coupled memory for devices where this is implemented separately to data TCM. See 8.5. Tightly-Coupled Memories.

6.7.1.9 Long-calls Option

The `-mlong-calls` option 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 addressing range of the offset-based version of subroutine call instruction. This range various with device and ISA. It is typically required when calling a function in another memory region. For instance, when calling a ram-allocated function from a flash-allocated function, the target address may be out of range of a short call.

Even if this feature 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 does not affect how the compiler generates code to indirectly call functions via a pointer.

The `-mno-long-calls` form of this option will not use long calls. This is the default action if no option is specified.

6.7.1.10 Print-builtins Option

The `-mprint-builtins` option prints a list of enabled built-in functions and then stops compilation.

6.7.1.11 Processor Option

The `-mprocessor=device` option selects the target device for which to compile. A list of all supported devices can be found the compiler release notes. Note that the name must be in upper case, for example, `-mprocessor=ATSAME70N20B`.

6.7.1.12 Pure-code Option

The `-mpure-code` option ensures that the compiler will not place `const`-qualified data into any section that also contain executable code. This is a requirement if you intend to place code into eXecute-Only Memory (XOM). Additionally, this option marks sections in the generated ELF file containing executable code with the `SHF_ARM_PURECODE` flag to indicate that they purely contain code and no data. Placement of pure code sections into XOM is performed separately (see 17.4. Changing the Default Function Allocation).

This option is usable with Arm Cortex-M23 and Cortex-M0+ based target devices. Consult your specific target device's data sheet to determine whether your target device supports XOM and for more information on its usage. Using this option with the MPLAB Code Coverage feature is not supported and will trigger an error from the compiler.

Using this option might affect the code size and execution speed of the program. Data qualified `const` is often placed into program memory and read when it is needed. This cannot be performed when this option is in effect, and the compiler will need to generate more/longer instructions to load the constant as an immediate operand. For example, to load an `int const`-qualified constant stored into memory will require the 4 bytes of constant data and typically a 2-byte `ldr` instruction to load that constant into a register. To instead load this constant using a `movw` + `movt` instruction sequence with immediate operands would require two 4-byte instructions. For devices such as the Cortex M0+ that cannot use such instruction sequences, the code to load the constant byte-by-byte can be significantly larger. When enabling this option, be sure to verify that any increase in code size or execution time still meets your application's timing requirements.

Precompiled libraries provided with MPLAB XC32, such as the Standard C library, are not built for Pure Code. Mapping these functions to an XOM region in your application linker script might result in code failure.

When using both the `-mpure-code` and `-ffunction-sections` options, the generated input section will be named according to the rules of the `-ffunction-sections` option, making it more difficult to map code to XOM in a custom linker script. In this case, you can't rely on the `SHF_ARM_PURECODE` section flag, and you must map the section by name instead.

The same is true when combining the `-mpure-code` option with function attributes that directly or indirectly affect function placement. Such attributes include `tcm`, `ramfunc`, `address`, `always_inline`, etc. Again, map the section by name in the custom linker script rather than using the `SHF_ARM_PURECODE` section flag to ensure your code is fully protected.

Procedural abstraction optimizations `(-mpa)` might abstract code out of sections designated to be pure code when using the `-mpure-code` option, and this abstracted code will need to be appropriately mapped to XOM in the custom linker script.

The `-mno-pure-code` form of this option will not separate executable code and `const`-qualified data. This is the default action if no option is specified.

6.7.1.13 Smart-io Option

The `-msmart-io=level` option in conjunction with the IO format string conversion specifications detected in your program control the feature set (hence size) of the library code that is linked in to perform formatted IO through functions like `printf`. See 20.1. Smart IO Routines for more information on how the smart IO feature operates.

A numerical level of operation can be specified and these have the meaning shown in the following table.

Table 6-5. Smart IO Implementation Levels

Level	Smart IO features; linked library
0	Disabled; Full-featured library (largest code size)
1	Enabled; Minimal-featured library (smallest code size)
2	Manual control; Integer-only library

When the smart IO feature is disabled (`-msmart-io=0`), a full implementation of the IO functions will be linked into your program. All features of the IO library functions will be available, and these may consume a significant amount of the available program and data memory on the target device.

The default setting is for smart IO to be enabled with a minimal feature set. This can be made explicit by using either the `-msmart-io=1` or `-msmart-io` option. When thus enabled, the compiler will link in the least complex variant of the IO library that implements all of the IO functionality required by the program, based on the conversion specifications detected in the program's IO function format strings. This can substantially reduce the memory requirements of your program, especially if you can eliminate in your program the use of floating-point features.

If the format string in a call to an IO function is not a string literal, the compiler will not be able to detect the exact usage of the IO function and a full-featured variant of the IO library will be linked into the program image, even with smart IO enabled.

These options should be used consistently across all program modules to ensure an optimal selection of the library routines included in the program image.

6.7.1.14 Stack-in-tcm Option

The `-mstack-in-tcm` option places the runtime stack into the data tightly coupled memory (TCM) region. The runtime startup code will transfer the stack from System SRAM to DTCM before calling your `main()` function. This options must be used at both compile and link times.

6.7.1.15 Tcm Option

The `-mtcm=`*`size`* option specifies the size (in bytes) of the tightly couple memory for devices where this is implemented as a single memory area. See 8.5. Tightly-Coupled Memories.

6.7.1.16 Thumb Option

The `-mthumb` option informs the compiler to generate code using the 16-bit Arm Thumb instruction set. This might result in more optimal code for your application. Use the `-marm` option to explicitly indicate that you do not want to have code generated using the Arm Thumb instruction set.

6.7.1.17 Unaligned-access Option

The `-munaligned-access` option enables reading and writing of 16- and 32-bit values from addresses that are not 16- or 32-bit aligned. This option does not affect instruction fetches.

The `-mno-unaligned-access` form of this option accesses 16- and 32-bit values in data structures one byte at a time.

If no option is specified, unaligned access is disabled for all pre-Arm v6, Arm v6-M, and Arm v8-M architectures, and enabled for all other architectures.

6.7.1.18 Nofallback Option

The `--nofallback` option can be used to ensure that the compiler is not inadvertently executed with optimizations below the that specified by the `-O` option.

For example, if an unlicensed compiler was requested to run with level `s` optimizations, without this option, it would normally revert to a lower optimization level and proceed. With this option, the compiler will instead issue an error and compilation will terminate. Thus, this option can ensure that builds are performed with a properly licensed compiler.

6.7.2 Options for Controlling the Kind of Output

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

Table 6-6. Kind-of-Output Control Options

Option (links to explanatory section)	Definition
`-c`	Stop compilation before the link step, producing an intermediate file.
`-E`	Stop compilation after preprocessing, producing a preprocessed file.
`-o` *`file`*	Place the output in a file with the specified name.
`-S`	Stop compilation before the assembly step, producing an assembly file output.
`-specs=`*`file`*	Overrides the standard specs file.
`-v`	Print the commands executed during each stage of compilation.
`-x`	Specify the language of a source file regardless of its file extension.
`--help`	Print a description of the command line options.

6.7.2.1 C: Compile To Intermediate File

The `-c` option is used to generate an intermediate file for each source file listed on the command line.

For all source files, compilation will terminate after executing the assembler, leaving behind relocatable object files with a `.o` extension.

This option is often used to facilitate multi-step builds using a make utility.

6.7.2.2 E: Preprocess Only

The `-E` option is used to generate preprocessed C/C++ source files (also called modules or translation units).

The preprocessed output is printed to `stdout`, but you can use the `-o` option to redirect this to a file.

You might check the preprocessed source files to ensure that preprocessor macros have expanded to what you think they should. The option can also be used to create C/C++ source files that do not require any separate header files. This is useful when sending files to a colleague or to obtain technical support without sending all the header files, which can reside in many directories.

6.7.2.3 O: Specify Output File

The `-o` option specifies the base name and directory of the output file.

The option `-o main.elf`, for example, will place the generated output in a file called `main.elf`. The name of an existing directory can be specified with the file name, for example `-o build/main.elf`, so that the output file will appear in that directory.

You cannot use this option to change the type (format) of the output file.

6.7.2.4 S: Compile To Assembly

The `-S` option is used to generate an assembly file for each source file listed on the command line.

When this option is used, the compilation sequence will terminate early, leaving behind assembly files with the same basename as the corresponding source file and with a `.s` extension.

For example, the command:

```
xc32-gcc -mprocessor=ATSAME70N20B -S test.c io.c
```

will produce two assembly file called `test.s` and `io.s`, which contain the assembly code generated from their corresponding source files.

This option might be useful for checking assembly code output by the compiler without the distraction of line number and opcode information that will be present in an assembly list file. The assembly files can also be used as the basis for your own assembly coding.

6.7.2.5 Specs Option

The `-specs=file` option allows the standard specs file to be overridden.

Spec files are plain-text files used to construct spec strings, which in turn control which programs the compiler should invoke and which command-line options they need to be passed. The compiler has a standard specs file which defines the options with which the compiler's internal applications will execute.

After the compiler reads in the standard specs file, it processes *file* specified with this option in order to override the defaults that the `xc32-gcc` driver program uses when building the command-line options to pass to `xc32-as`, `xc32-ld`, etc. More than one `-specs` option can be specified on the command line, and they are processed in order, from left to right.

6.7.2.6 V: Verbose Compilation

The `-v` option specifies verbose compilation.

When this option is used, the name and path of the internal compiler applications will be displayed as they are executed, followed by the command-line arguments that each application was passed.

You might use this option to confirm that your driver options have been processed as you expect, or to see which internal application is issuing a warning or error.

6.7.2.7 X: Specify Source Language Option

The `-x language` option specifies the language for the sources files that follow on the command line.

The compiler usually uses the extension of an input file to determine the file’s content. This option allows you to have the language of a file explicitly stated. The option remains in force until another language is specified with a `-x` option, or the `-x none` option, which turns off the language specification entirely for subsequent files. The allowable languages are tabulated below.

Table 6-7. Source file Language

Language	File language
`assembler`	Assembly source
`assembler-with-cpp`	Assembly with C preprocessor directives
`c`	C source
`c++`	C++ source
`cpp-output`	Preprocessed C source
`c-header`	C header file
`none`	Based entirely on the file’s extension

The `-x assembler-with-cpp` language option ensures assembly source files are preprocessed before they are assembled, thus allowing the use of preprocessor directives, such as `#include`, and C-style comments with assembly code. By default, assembly files not using a `.S` extension are not preprocessed.

You can create precompiled header files with this option, for example:

```
xc32-gcc -mprocessor=ATSAME70N20B -x c-header init.h
```

will create the precompiled header called `init.h.gch`.

6.7.2.8 Help

The --help option displays information on the xc32-gcc compiler options, then the driver will terminate.

For example:

```
xc32-gcc --help
Microchip Language Tool Shell Version 4.20 (Build date: Sep 16 2022).
Copyright (c) 2012-2017 Microchip Technology Inc. All rights reserved

  -omf=elf        Select elf object module format
Usage: pic32m-gcc [options] file...
Options:
  -pass-exit-codes         Exit with highest error code from a phase.
  --help                   Display this information.
  --target-help            Display target specific command line options.
  --help={common|optimizers|params|target|warnings|[^]{joined|separate|undocumented}}[,...].
                           Display specific types of command line options.
  (Use '-v --help' to display command line options of sub-processes).
  --version                Display compiler version information.
  -dumpspecs               Display all of the built in spec strings.
  -dumpversion             Display the version of the compiler.
  -dumpmachine             Display the compiler's target processor.
  -print-search-dirs       Display the directories in the compiler's search path.
  -print-libgcc-file-name  Display the name of the compiler's companion library.
```

6.7.3 Options for Controlling the C Dialect

The options tabulated below define the type of C dialect used by the compiler and are discussed in the sections that follow.

Table 6-8. C Dialect Control Options

Option (links to explanatory section)	Definition
-ansi	Support all (and only) ANSI-standard C programs.
-aux-info *filename*	Generates function prototypes from a C module, placing the output in a file with the specified name.
-f[no-]freestanding	Asserts that the project will be built for a freestanding (as opposed to a hosted) environment.
-f[no-]asm	Restricts the recognition of certain keywords, freeing them to be used as identifiers.
-fno-builtin -fno-builtin-*function*	Don't recognize built-in functions that do not begin with __builtin_ as prefix.
-f[no-]unsigned-bitfields-f[no-]signed-bitfields	Changes the signedness of a plain int bit-field.
-f[no-]signed-char-f[no-]unsigned-char	Changes the signedness of a plain char type.

6.7.3.1 Ansi Option

The -ansi option ensures the C program strictly conforms to the C90 standard.

When specified, this option disables certain GCC language extensions when compiling C source. Such extension include C++ style comments, and keywords, such as asm and inline. The macro __STRICT_ANSI__ is defined when this option is in use. See also -Wpedantic for information on ensuring strict ISO compliance.

6.7.3.2 Aux-info Option

The -aux-info option generates function prototypes from a C module.

The -aux-info main.pro option, for example, prints to main.pro prototyped declarations for all functions declared and/or defined in the module being compiled, including those in header files. Only one source file can be specified on the command line when using this option so that the output file is not overwritten. This option is silently ignored in any language other than C.

The output file also indicates, using comments, the source file and line number of each declaration, whether the declaration was implicit, prototyped or unprototyped. This is done by using the codes `I` or `N` for new-style and `O` for old-style (in the first character after the line number and the colon) and whether it came from a declaration or a definition using the codes `C` or `F` (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.

For example, compiling with the command:

```
xc32-gcc -mprocessor=ATSAME70N20B -aux-info test.pro test.c
```

might produce `test.pro` containing the following declarations, which can then be edited as necessary:

```
/* test.c:2:NC */ extern int add (int, int);
/* test.c:7:NF */ extern int rv (int a); /* (a) int a; */
/* test.c:20:NF */ extern int main (void); /* () */
```

6.7.3.3 Enforce-eh-specs Option

The `-fenforce-eh-specs` option generates code to check for violation of exception specifications at runtime. This is the default action if no option is specified.

The `-fno-enforce-eh-specs` form of this option omits generation of verification code, which violates the C++ standard, but doing so might reduce the code size in production builds of validated source. Even when specifying this option, the compiler will still optimize exception code based on the specifications, so throwing an unexpected exception will result in undefined behavior.

6.7.3.4 Freestanding Option

The `-ffreestanding` option asserts that the project will be built for a freestanding (as opposed to a hosted) environment.

A freestanding environment is one in which the standard library may not be fully implemented, and program start-up and termination are implementation defined.

This option is identical to the `-fno-hosted` option and implies the `-fno-builtin` option.

The `-fno-freestanding` option is identical to the `-fhosted` option and indicates that the project will be built for a hosted environment.

6.7.3.5 Asm Option

The `-fasm` option reserves the use of `asm`, `inline` and `typeof` as keywords, preventing them from being defined as identifiers. This is the default action if no option is specified.

The `-fno-asm` form of this option restricts the recognition of these keywords. You can, instead, use the keywords `__asm__`, `__inline__` and `__typeof__`, which have identical meanings.

The `-ansi` option implies `-fno-asm`.

6.7.3.6 No-builtin Option

The `-fbuiltin` option has the compiler produce specialized code that avoids a function call for many built-in functions. The resulting code is often smaller and faster, but since calls to these functions no longer appear in the output, you cannot set a breakpoint on those calls, nor can you change the behavior of the functions by linking with a different library. This is the default behavior if no option is specified.

The `-fno-builtin` option will prevent the compiler from producing special code for built-in functions that are not prefixed with `__builtin_`.

The `-fno-builtin-`*`function`* form of this option allows you to prevent a built-in version of the named function from being used. In this case, *`function`* must not begin with `__builtin_`. There is no `-fbuiltin-`*`function`* form of this option.

6.7.3.7 Signed-bitfields Option

The -fsigned-bitfield option control the signedness of a plain int bit-field type.

By default, the plain int type, when used as the type of a bit-field, is equivalent to signed int. This option specifies the type that will be used by the compiler for plain int bit-fields. Using -fsigned-bitfield or the -fno-unsigned-bitfield option forces a plain int bit-field to be signed.

Consider explicitly stating the signedness of bit-fields when they are defined, rather than relying on the type assigned to a plain int bit-field type.

6.7.3.8 Signed-char Option

The -fsigned-char option forces plain char objects to have a signed type.

By default, the plain char type is equivalent to signed char, unless the -mext=cci option has been used, in which case it is equivalent to unsigned char.

The -fsigned-char (or -fno-unsigned-char option) makes it explicit that plain char types are to be treated as signed integers.

Consider explicitly stating the signedness of char objects when they are defined, rather than relying on the type assigned to a plain char type by the compiler.

Attention:
The *ARM C Language Extensions* and the *ARM Procedure Call Standard* specifications define a plain char to be equivalent to an unsigned char. When using the -fsigned-char option, the generated code will not be compliant with these specifications.

6.7.3.9 Unsigned-bitfields Option

The -funsigned-bitfield option control the signedness of a plain int bit-field type.

By default, the plain int type, when used as the type of a bit-field, is equivalent to signed int. This option specifies the type that will be used by the compiler for plain int bit-fields. Using the -funsigned-bitfield or the -fno-signed-bitfield option forces a plain int to be unsigned.

Consider explicitly stating the signedness of bit-fields when they are defined, rather than relying on the type assigned to a plain int bit-field type.

6.7.3.10 Unsigned-char Option

The -funsigned-char option forces a plain char objects to have an unsigned type.

By default, the plain char type is equivalent to signed char, unless the -mext=cci option has been used, in which case it is equivalent to unsigned char. The -funsigned-char (or -fno-signed-char option) makes this type explicit.

Consider explicitly stating the signedness of char objects when they are defined, rather than relying on the type assigned to a plain char type by the compiler.

6.7.4 Options for Controlling the C++ Dialect

The options tabulated below define the type of C++ dialect used by the compiler and are discussed in the sections that follow. Note that the sections for those options also relevant for the C dialect are located in the 6.7.3. Options for Controlling the C Dialect section.

Table 6-9. C++ Dialect Control Options

Option (links to explanatory section)	Definition
-ansi	Support all (and only) ANSI-standard C++ programs.

...........continued

Option (links to explanatory section)	Definition
`-aux-info` *`filename`*	Generates function prototypes from a C module, placing the output in a file with the specified name.
`-f[no-]check-new`	Check that the pointer returned by operator `new` is non-null.
`-f[no-]enforce-eh-specs`	Specify generation of code to check for violation of exception specifications at runtime.
`-f[no-]freestanding`	Asserts that the project will be built for a freestanding (as opposed to a hosted) environment.
`-f[no-]asm`	Restricts the recognition of certain keywords, freeing them to be used as identifiers.

`-fno-builtin`
`-fno-builtin-`*`function`*
	Don't recognize built-in functions that do not begin with `__builtin_` as prefix.

`-f[no-]rtti`	Specifies if code for Run Time Type Identification features should be generated.

`-f[no-]signed-bitfields`
`-f[no-]unsigned-bitfields`
	Changes the signedness of a plain int bit-field.

`-f[no-]signed-char-f[no-]unsigned-char`	Changes the signedness of a plain `char` type.

6.7.4.1 Check-new Option

The `-fcheck-new` option checks that the pointer returned by `operator new` is not `NULL` before attempting to modify the storage allocated.

Typically, this check is not necessary. You can ensure that no checks are made by using the `-fno-check-new` form of this option. This is the default action if no option is specified.

6.7.4.2 Rtti Option

The `-frtti` option enables generation of information about every class with virtual functions for use by the C++ Run Time Type Identification (RTTI) features, such as `dynamic_cast` and `typeid` operators.

The `-fno-rtti` form of this option disables generation of this information. If you don't use the RTTI features of the language, code size might be reduced by using this option. Note that exception handling uses the same information, but it will generate this information as needed. The `dynamic_cast` operator can still be used for casts that do not require RTTI, that is, casts to `void *` or to unambiguous base classes.

Ensure that the same form of this option is specified at compile time for all modules and at link time.

6.7.5 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 options shown in the tables below are the more commonly used warning options that control messages issued by the compile. In the (linked) sections that follow, the options that affect warnings generally are discussed.

Table 6-10. Warning and Error Options Implied by All Warnings

Option (links to explanatory section)	Definition
-fsyntax-only	Check the code for syntax, but don't do anything beyond that.
-pedantic	Issue all the warnings demanded by strict ANSI C. Reject all programs that use forbidden extensions.
-pedantic-errors	Like -pedantic, except that errors are produced rather than warnings.
-w	Inhibit all warning messages.
-Wall	Enable all warnings about constructions that some users consider questionable, and that are easy to avoid, even in conjunction with macros.
-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.
-Warray-bounds	When combined with -02 or greater, warns for many instances out-of-bounds array indices and pointer offsets. This feature cannot detect all cases. Enabled with -Wall command.
-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 } };

-Wmultistatement-macros	Warns for unsafe macro expansion as a body of a statement such as if, else, while, switch, or for. Enabled with -Wall command.

-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.

...........continued

Option (links to explanatory section)	Definition
-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.
-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. 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.
-Wsizeof-pointer-div	Warns for suspicious divisions of the size of a pointer by the size of the elements it points to, which looks like the usual way to compute the array size but won't work out correctly with pointers.
-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.

...........continued

Option (links to explanatory section)	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 6-11. Warning And Error Options Not Implied By All Warnings

Option	Definition
-Wextra (formerly) -W	Enable extra warning flags that are not enabled by -Wall.
-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, and conversions changing the width or signedness of a fixed point argument, except when the same as the default promotion. Also, warn if a negative integer constant expression is implicitly converted to an unsigned type. For example, warn about the assignment x = -1 if x is unsigned. But do not warn about explicit casts like (unsigned) -1.
-Werror	Make all warnings into errors.
-Winline	Warn if a function can not be inlined, and either it was declared as inline, or else the -finline-functions option was given.
-Wlarger-than-len	Warn whenever an object of larger than len bytes is defined.
-Wlong-long -Wno-long-long	Warn if long long type is used. This is default. To inhibit the warning messages, use -Wno-long-long. Flags -Wlong-long and -Wno-long-long are taken into account only when -pedantic flag is used.
-Wmissing-declarations	Warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a prototype.

...........continued

Option	Definition
-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.
-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.

6.7.5.1 Syntax-only Option

The -fsyntax-only option checks the C source code for syntax errors, then terminates the compilation.

6.7.5.2 Pedantic Option

The -pedantic option ensures that programs do not use forbidden extensions and that warnings are issued when a program does not follow ISO C.

6.7.5.3 Pedantic-errors Option

The -pedantic-errors option works in the same way as the -pedantic option, only errors, instead of warnings, are issued when a program is not ISO compliant.

6.7.5.4 W: Disable all Warnings Option

The -w option inhibits all warning messages, and thus should be used with caution.

6.7.5.5 Wall Option

The -Wall option enables all the warnings about easily avoidable constructions that some users consider questionable, even in conjunction with macros.

Note that some warning flags are not implied by -Wall. Of these warnings, some relate to constructions that users generally do not consider questionable, but which you might occasionally wish to check. 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 these warnings can be enabled using the -Wextra option, but many of them must be enabled individually.

6.7.5.6 Wextra Option

The -Wextra option generates extra warnings in the following situations.

- 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 an 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 won'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 };
```

6.7.6 Options for Debugging

The options tabulated below control the debugging output produced by the compiler and are discussed in the sections that follow.

Table 6-12. Debugging Options

Option (links to explanatory section)	Definition
`-f[no-]eliminate-unused-debug-symbols`	Eliminate debug information associated with any C/C++ symbol that has not been used in the program.
`-g`	Produce debugging information.
`-Q`	Print function names and statistics from each pass.
`-save-temps[=`*`dir`*`]`	Don’t delete intermediate files.

6.7.6.1 Eliminate-unused-debug-symbols Option

The `-feliminate-unused-debug-symbols` option eliminates the debug information associated with any C/C++ symbol that has not been used in the program. This is also the default action if no option has been specified. Eliminating this information will reduce the size and load time of output files.

Use the `-fno-eliminate-unused-debug-symbols` form of this option if you want debug information generated for all symbols.

6.7.6.2 G: Produce Debugging Information Option

The `-g` option instructs the compiler to produce additional information, which can be used by hardware tools to debug your program.

The option can be used 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.

6.7.6.3 Q: Print Function Information Option

The `-Q` option instructs the compiler to print out each function name as it is compiled, and print statistics about each pass when it finishes.

6.7.6.4 Save-temps Option

The `-save-temps` option instructs the compiler to keep temporary files after compilation has finished. You might find the generated `.i` and `.s` temporary files in particular useful for troubleshooting, and they are often used by the Microchip Support team when you enter a support ticket.

The intermediate files will be placed in the current directory and have a name based on the corresponding source file. Thus, compiling `foo.c` with `-save-temps` would produce `foo.i`, `foo.s` and the `foo.o` object file.

The `-save-temps=cwd` option is equivalent to `-save-temps`.

The `-save-temps=obj` form of this option is similar to `-save-temps`, but if the `-o` option is specified, the temporary files are placed in the same directory as the output object file. If the `-o` option is not specified, the `-save-temps=obj` switch behaves like `-save-temps`.

The following example will create `dir/xbar.i`, `dir/xbar.s`, since the `-o` option was used.

```
xc32-gcc -save-temps=obj -c bar.c -o dir/xbar.o
```

6.7.7 Options for Controlling Optimization

The options tabulated below control compiler optimizations and are described in the sections that follow.

Use of an option is permitted and has an effect with an unlicensed compiler (Free edition) if that option is enabled by default when using optimization level 2 or lower. Any form of option that disables an optimization is available unconditionally.

Table 6-13. General Optimization Options

Option (links to explanatory section)	Edition	Controls
-O0	All	Do not optimize.
-O -O1	All	Optimization level 1.
-O2	All	Optimization level 2.
-O3	PRO only	Speed-orientated optimizations.
-Og	All	Less optimizations for better debugging.
-Os	PRO only	Size-orientated optimizations.
-falign-functions[=*n*] -f[no-]align-functions	All	Alignment of the start of functions.
-falign-labels[=*n*] -f[no-]align-labels	All	Alignment of all branch targets.
-falign-loops[=*n*] -f[no-]align-loops	All	Alignment of loops.
-f[no-]caller-saves	All	Allocation into registers clobbered by function calls.
-f[no-]cse-follow-jumps	All	Customization of common subexpression elimination optimizations.
-f[no-]cse-skip-blocks	All	Customization of common subexpression elimination optimizations.
-f[no-]data-sections	All	Placement of objects into a section in the output file.
-f[no-]defer-pop	All	When arguments to function calls are popped.
-f[no-]expensive-optimizations	All	Minor optimizations that are relatively expensive.
-f[no-]function-cse	All	Placement of function addresses.
-f[no-]function-sections	All	Placement of functions intoa section in the output file.
-f[no-]gcse	All	The global common subexpression elimination pass.
-f[no-]gcse-lm	All	Customization of common subexpression elimination dealing with loads.
-f[no-]gcse-sm	All	Customization of common subexpression elimination dealing with stores.
-finline	All	Controls processing of functions marked with the`inline` keyword.
-f[no-]inline-functions	All	Integratation of all simple functions into their callers.
-f[no-]inline-limit=n	All	The size limit for inlining functions.
-f[no-]keep-inline-functions	All	Output of a separate run-time-callable version of inlined functions.
-f[no-]keep-static-consts	All	Output of `static const` objects.
-f[no-]lto	PRO only	The standard link-time optimizer.

...........continued

Option (links to explanatory section)	Edition	Controls
-f[no-]omit-frame-pointer	All	Reservation of the Frame Pointer in a register for functions that don't need one.
-f[no-]optimize-sibling-calls	All	Optimization of sibling and tail recursive calls.
-m[no-]pa	PRO	Procedural Abstraction optimizations.
-f[no-]peephole -f[no-]peephole2	All	Machine-specific peephole optimizations.
-f[no-]rename-registers	All	Use of free registers to avoid false dependencies in scheduled code.
-f[no-]rerun-cse-after-loop	All	When common subexpression elimination optimizations should be performed.
-f[no-]rerun-loop-opt	All	Legacy option and is ignored.
-f[no-]schedule-insns -f[no-]schedule-insns2	All	Reordering instructions to eliminate instruction stalls.
-f[no-]strength-reduce	All	Legacy Option and is ignored.
-f[no-]strict-aliasing	All	Assumption of the strictest aliasing rules applicable to the language being compiled.
-f[no-]thread-jumps	All	Jump-to-branch optimizations.
-f[no-]toplevel-reorder	All	Reordering of top-level functions, variables, and `asm` statements.
-f[no-]unroll-loops -f[no-]unroll-all-loops	All	Loop unrolling.

6.7.7.1 O0: Level 0 Optimizations

The `-O0` option performs only rudimentary optimization. This is the default optimization level if no `-O` option is specified.

With this optimization level selected, the compiler’s goal is to reduce the cost of compilation and to make debugging produce the expected results.

Statements are independent when compiling with this optimization level. 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.

6.7.7.2 O1: Level 1 Optimizations

The `-O1` or `-O` options request level 1 optimizations.

The optimizations performed when using `-O1` aims to reduce code size and execution time, but still allows a reasonable level of debugability.

This level is available for unlicensed as well as licensed compilers.

6.7.7.3 O2: Level 2 Optimizations Option

The `-O2` option requests level 2 optimizations.

At this level, the compiler performs nearly all supported optimizations that do not involve a space-speed trade-off.

This level is available for unlicensed as well as licensed compilers.

6.7.7.4 O3: Level 3 Optimizations Option

The `-O3` option requests level 3 optimizations.

This option requests all supported optimizations that reduces execution time but which might increase program size.

This level is available only for licensed compilers.

6.7.7.5 Og: Better Debugging Option

The `-Og` option disables optimizations that severely interfere with debugging, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience.

This option selects the best optimization level for the standard edit-compile-debug cycle. It enables all level 1 optimization flags except for those that may interfere with debugging, and those optimizations disabled cannot be re-enabled by their corresponding option. This level is a better choice than level 0 optimizations for producing debuggable code, because some compiler passes that collect debug information are disabled at level 0.

6.7.7.6 Os: Level s Optimizations Option

The `-Os` option requests space-orientated optimizations.

This option requests all supported optimizations that do not typically increase code size.

This level is available only for licensed compilers.

6.7.7.7 Align-functions Option

The `-falign-functions=`*`n`* option aligns the start of functions to the next power-of-two greater than *`n`*, skipping up to *`n`* bytes. For PIC32C/SAM devices, `n`must not exceed 64.

For instance, `-falign-functions=32` aligns functions to the next 32-byte boundary; `-falign-functions=24` aligns functions to the next 32-byte boundary but only if this can be done by skipping no more than 23 bytes.

This option is automatically enabled at optimization levels `-O2` and `-O3`.

The `-fno-align-functions` form of this option is equivalent to `-falign-functions=1` and implies that functions are not aligned.

The `-falign-functions` form of this option (with no argument) performs no additional alignment other than the usual alignment by 4 for code using the ARM instruction set or by 2 for code using the Thumb instruction set.

6.7.7.8 Align-labels Option

The `-falign-labels=`*`n`* option aligns all branch targets to the next power-of-two greater than *`n`*, skipping up to *`n`* bytes. For instance, `-falign-labels=8` aligns functions to the next 8-byte boundary; `-falign-functions=9` aligns functions to the next 16-byte boundary but only if this can be done by skipping no more than 9 bytes.

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 the options `-falign-loops` or `-falign-jumps` have been used and either of their arguments are greater than *`n`*, then their argument values are used to determine the label alignment instead.

This option is automatically enabled at optimization levels `-O2` and `-O3`.

The `-fno-align-labels` form of this option is equivalent to `-falign-labels=1` and implies that functions are not aligned. The `-falign-labels` form of this option (with no argument) also performs no alignment.

6.7.7.9 Align-loops Option

The `-falign-loops=n` option aligns loops to the next power-of-two greater than *n*, skipping up to *n* bytes. For instance, `-falign-loops=32` aligns loops to the next 32-byte boundary; `-falign-loops=24` aligns loops to the next 32-byte boundary but only if this can be done by skipping no more than 23 bytes.

The `-fno-align-loops` form of this option is equivalent to `-falign-loops=1` and implies that functions are not aligned. The `-falign-loops` form of this option (with no argument) also performs no alignment.

The hope is that the loop is executed many times, which makes up for any execution of the dummy operations.

6.7.7.10 Caller-saves Option

The `-fcaller-saves` option allows the compiler to allocate values to registers that are clobbered by function calls. If allocation to these registers takes place, extra code to save and restore the clobbered registers is generated around function calls. Allocation is performed only when better performing code is expected than would otherwise be produced.

This option is automatically enabled at optimization levels `-O2`, `-O3`, and `-Os`.

The `-fno-caller-saves` form of this option never allocates values to registers that are clobbered by functions.

6.7.7.11 Cse-follow-jumps Option

The `-fcse-follow-jumps` option instructs the common subexpression elimination (CSE) optimizations to 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.

This option is automatically enabled at optimization levels `-O2`, `-O3`, and `-Os`.

The `-fno-cse-follow-jumps` form of this option does not perform this scan.

6.7.7.12 Cse-skip-blocks Option

The `-fcse-skip-blocks` option performs a similar task to the `-fcse-follow-jumps` option but instructs the common subexpression elimination (CSE) optimizations to follow jumps that conditionally skip over blocks. When CSE encounters a simple `if()` statement with no `else` clause, it will follow the jump around the body of the `if()`.

This option is automatically enabled at optimization levels `-O2`, `-O3`, and `-Os`.

The `-fno-cse-skip-blocks` form of this option does not perform this scan.

6.7.7.13 Data-sections Option

The `-fdata-sections` option places each object in its own section to assist with garbage collection and potential code size reductions.

When enabled, this option has each object placed in its own section named after the object. When used in conjunction with garbage collection performed by the linker (enabled using the`-Wl,--gc-sections` driver option) the final output might be smaller. It can, however, negatively impact other code generation optimizations, so confirm whether this option is of benefit with each project.

This option has no effect when building with Link-Time Optimizations (`-flto`). Use the `section` attribute with objects that need to be placed into a named section.

The `-fno-data-sections` form of this option does not force each object to a unique section. This is the default action if no option is specified.

6.7.7.14 Defer-pop Option

The `-fdefer-pop` option specifies that the compiler should let function arguments accumulate on the stack for several function calls before they are popped off the stack in one step.

This option is automatically enabled at optimization levels -O1, -O2, -O3, and -Os.

The -fno-defer-pop form of this option has the compiler pop the arguments to each function call as soon as that function returns.

6.7.7.15 Expensive-optimizations Option

The -fexpensive-optimizations options has the compiler perform a number of optimizations that have minimal effect but that are relatively expensive to perform.

This option is automatically enabled at optimization levels -O2, -O3, and -Os.

The -fno-expensive-optimizations form of this option does not perform these optimizations.

6.7.7.16 Function-cse Option

The -ffunction-cse form of this option allows the address of called functions to be held in registers. This is the default action if no option is specified.

The -fno-function-cse form of this option will not place function addresses in registers. 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.

6.7.7.17 Function-sections Option

The -ffunction-sections option places each function in its own section to assist with garbage collection and potential code size reductions.

When enabled, this option has each function placed in its own section named after the function. When used in conjunction with garbage collection performed by the linker (enabled using the -Wl,--gc-sections driver option) the final output might be smaller. The -ffunction-sections option can, however, hamper other code generation optimizations, so confirm whether this option is of benefit with each project.

Note also that if functions are removed, their debugging information will remain in the ELF file, potentially imperiling the debuggability of code. When displaying source code information, the debugger looks for associations between addresses and the functions whose code resides at those locations, and it might wrongly consider that an address belongs to a removed function. As removed functions are not allocated memory, their "assigned" address will not change from being 0, thus the debugger is more likely to show references to a removed function when interpreting addresses closer to 0.

This option has no effect when building with Link-Time Optimizations (-flto). Use the section attribute with functions that need to be placed into a named section.

The -fno-function-sections form of this option does not force each function into a unique section. This is the default action if no option is specified.

6.7.7.18 Gcse Option

The -fgcse option perform a global common subexpression elimination pass. This pass also performs global constant and copy propagation.

This option is automatically enabled at optimization levels -O2, -O3, and -Os.

The -fno-gcse form of this option does not perform this pass.

6.7.7.19 Gcse-lm Option

The -fgcse-lm option attempts to have the global common subexpression elimination optimizations move load sequences that 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.

The option is enabled when -fgcse is enabled.

The -fno-gcse-lm form of this option does not move these sequences.

6.7.7.20 Gcse-sm Option

The -fgcse-sm option runs a store motion pass after the global common subexpression elimination optimizations. 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.

The option is not automatically enabled at any optimization level.

The -fno-gcse-sm form of this option does not move these sequences.

6.7.7.21 Inline Option

The -finline option allows the compiler to expand functions using the inline keyword. This is the default action if no option is specified.

The -fno-inline form of this option will never inline functions, even if they have been marked as inline.

6.7.7.22 Inline-functions Option

The -finline-functions option considers all functions for inlining, even if they are not declared inline.

If all calls to a given function are inlined, and the function is declared static, then the function is normally not output as a stand-alone assembler routine.

This option is automatically enabled at optimization levels -O3 and -Os.

The -fno-inline-functions form of this option will never inline function that have not been marked as inline.

6.7.7.23 Inline-limit Option

The -finline-limit=*n* option controls the inlining size limit for functions marked inline. The argument, *n*, is a number of pseudo instructions (not counting parameter handling) being an abstract measurement of a function's size. In no way does this value represent a count of assembly instructions and as such, its exact meaning might change from one release of the compiler to an another. The default value of *n* is 10000.

Increasing the inlining limit can result in more inlined code at the cost of compilation time and memory consumption.

6.7.7.24 Keep-inline-functions Option

The -fkeep-inline-functions option ensures that separate run-time-callable assembly code for functions is output, even if all calls to a given function are inlined and the function is declared static. This switch does not affect the output of extern inline functions.

The -fno-keep-inline-functions form of this option will not emit output for static functions where all calls to it have been inlined. This is the default action if no option has been specified.

6.7.7.25 Keep-static-consts Option

The -fkeep-static-consts option emits static const objects when the optimizers are disabled, even if the objects are not referenced. This is the default action if no option is specified.

The -fno-keep-static-consts form of this option forces the compiler to emit the object only when referenced, regardless of whether the optimizers are enabled.

6.7.7.26 Lto Option

This -flto 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. Link time optimizations do not require the presence of the whole program to operate.

To use the link-timer optimizer, specify -flto both at compile time and during the final link. For example:

```
xc32-gcc -c -O3 -flto -mprocessor=ATSAME70N20B foo.c
xc32-gcc -c -O3 -flto -mprocessor=ATSAME70N20B bar.c
xc32-gcc -o myprog.elf -flto -O3 -mprocessor=ATSAME70N20B foo.o bar.o
```

Another (simpler) way to enable link-time optimization is:

```
xc32-gcc -o myprog.elf -flto -O3 -mprocessor=ATSAME70N20B foo.c bar.c
```

The bytecode files are versioned and there is a strict version check, so bytecode files generated in one version of the XC32 compiler might not work with a different version.

The -fno-lto form of this option does not run the standard link-time optimizer. This is the default action if this option is not specified.

6.7.7.27 Omit-frame-pointer Option

The -fomit-frame-pointer option instructs the compiler to directly use the stack pointer to access objects on the stack and, if possible, omit code that saves, initializes, and restores the frame register. It is enabled automatically at all non-zero optimization levels.

Negating the option, using -fno-omit-frame-pointer, might assist debugging optimized code; however, this option does not guarantee that the frame pointer will always be used.

6.7.7.28 Optimize-sibling-calls Option

The -foptimize-sibling-calls option optimizes sibling calls (where the last action of a function is a call to another function whose stack footprint is compatible with the callee) and tail recursive calls (where the last action of a function is a call to itself).

This option is automatically enabled at optimization levels -O2, -O3, and -Os.

The -foptimize-sibling-calls form of this option will not enable these optimizations.

6.7.7.29 Pa Option

The -mpa option enables Procedural Abstraction optimizationsfor Cortex®-M devices and the Thumb/2 instruction sets. This optimization is available only for licensed compilers.

Procedural Abstraction finds common blocks of assembly code instructions and factors them out into a new callable procedures. It then replaces each instance of the block of instructions with a call to this new procedure. This is essentially a reverse inlining process.

Important:
Ensure that the same form of this option is specified at compile time for all modules. Additionally, the option must be specified at link time.

This option is useable when linking with response files and is compatible with link-time optimizations (-flto).

Important: Procedural Abstraction is an aggressive code-size optimization. As such, there is a negative impact to both debug-ability and performance of the compiled application.

Procedural abstraction can be disabled on a per-function basis by using the nopa attribute when this option is in effect.

The -mno-pa option disables procedural abstraction optimizations. This is the default action if no option is specified.

6.7.7.30 Peephole/peephole2 Options

The -fpeephole option enables machine-specific peephole optimizations. Peephole optimizations occur at various points during the compilation. This is the default action if no option is specified.

The -fpeephole2 form of this option enables high-level peephole optimizations. This option is enabled at optimization levels -O2, -O3, and -Os.

The -fno-peephole and -fno-peephole2 form of these options disables standard and high-level peephole optimizations, respectively. Both options should be used to entirely disable peephole optimizations.

6.7.7.31 Rename-registers Option

The -frename-registers option attempts to avoid false dependencies in scheduled code by making use of registers leftover after register allocation. This optimization most benefits processors with many registers, but it can reduce the debugability of code, since variables might no longer be allocated to a “home register.”

This option is automatically enabled when using -funroll-loops.

The -fno-rename-registers form of this option does not use leftover registers. This is the default action if no option has been specified and the -funroll-loops option has not been used.

6.7.7.32 Rerun-cse-after-loop Option

The -frerun-cse-after-loop option reruns common subexpression elimination (CSE) optimizations after loop optimizations has been performed.

This option is automatically enabled at optimization levels -O2, -O3, and -Os.

The -fno-rerun-cse-after-loop form of this option does not rerun the CSE optimizations.

6.7.7.33 Rerun-loop-opt Option

The -frerun-loop-opt option no longer has any effect and is ignored for compatibility with legacy projects.

6.7.7.34 Schedule-insns/schedule-insns2 Options

The -fschedule-insns option attempts to reorder instructions to eliminate instruction stalls due to required data being unavailable.

This option is automatically enabled at optimization levels -O2 and -O3.

The -fschedule-insns2 form of this option is similar to -fschedule-insns, but it requests an additional pass of instruction scheduling after register allocation has been performed.

This option is automatically enabled at optimization levels -O2, -O3, and -Os.

The -fno-schedule-insns and -fno-schedule-insns2 forms of these options do not reorder instructions.

6.7.7.35 Strength-reduce Option

The -fstrength-reduce option no longer has any effect and is ignored for compatibility with legacy projects.

6.7.7.36 Strict-aliasing Option

The -fstrict-aliasing option 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. 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;
}
```

This option is automatically enabled at optimization levels `-O2`, `-O3`, and `-Os`.

`-fno-strict-aliasing` form of this option relaxes the aliasing rules, preventing some optimizations from being performed.

6.7.7.37 Thread-jumps Option

The `-fthread-jumps` option enables checks to determine if the branch destinations of comparisons are further comparisons that have been subsumed by the first. 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 second branch condition is known to be true or false.

This option is automatically enabled at optimization levels `-O2`, `-O3`, and `-Os`.

The `-fno-thread-jumps` form of this option does not perform these checks.

6.7.7.38 Toplevel-reorder Option

The `-ftoplevel-reorder` option allows the compiler to reorder top-level functions, variables, and `asm` statements, in which case, they might not be output in the same order that they appear in the input file. This option also allows the compiler to remove unreferenced `static` variables.

Enabled at level `-O0`. When disabled explicitly using the `-fno-toplevel-reorder` option, it also implies `-fno-section-anchors`, which is otherwise enabled at `-O0` on some targets.

6.7.7.39 Unroll-loops/unroll-all-loops Options

The `-funroll-loops` and `-funroll-all-loops` options control speed-orientated optimizations that attempt to remove branching delays in loops. Unrolled loops typically increase the execution speed of the generated code, at the expense of larger code size.

The `-funroll-loops` option unrolls loops where the number of iterations can be determined at compile time or when code enters the loop. This option is enabled with `-fprofile-use`. The `-funroll-all-loops` option is more aggressive, unrolling all loops, even when the number of iterations is unknown. It is typically less effective at improving execution speed than the `-funroll-loops` option.

Both options imply `-frerun-cse-after-loop`, `-fweb` and `-frename-registers`.

The `-fno-unroll-loops` and `-fno-unroll-all-loops` forms of these options do not unroll any loops and are the default actions if no options are specified.

6.7.8 Options for Controlling the Preprocessor

The options tabulated below control the preprocessor and are discussed in the sections that follow.

Table 6-14. Preprocessor Options

Option **(links to explanatory section)**	**Definition**
`-C`	Preserve comments.
`-d`*`letters`*	Make debugging dumps of preprocessor macros.
`-Dmacro[=defn]`	Define a macro .
`-H`	Print the name of each header file used.
`-imacros file`	Include file macro definitions only.
`-include file`	Process file as input before processing the regular input file.
`-M`	Generate make rule.
`-MD`	Write dependency information to file.
`-MF file`	Specify dependency file.
`-MG`	Ignore missing header files.
`-MM`	Generate make rule for quoted headers.
`-MMD`	Generate make rule for user headers.
`-MP`	Add phony target for dependency.
`-MQ`	Change rule target with quotes.
`-MT` *`target`*	Change rule target.
`-nostdinc`	Omit system directories from header search.
`-P`	Don't generate #line directives.
`-fno-show-column`	Omit column numbers in diagnostics.
`-trigraphs`	Support ANSI C trigraphs.
`-Umacro`	Undefine a macro.
`-undef`	Do not predefine any nonstandard macros.

6.7.8.1 C: Preserve Comments Option

The `-C` option tells the preprocessor not to discard comments from the output. Use this option with the `-E` option to see commented yet preprocessed source code.

6.7.8.2 d: Preprocessor Debugging Dumps Option

The `-d`*`letters`* option has the preprocessor produce debugging dumps during compilation as specified by *`letters`*. This option should be used in conjunction with the `-E` option.

The `-dM` option generates a list of `#define` directives for all the macros defined during the execution of the preprocessor, including predefined macros. These will indicate the replacement string if one was defined. For example, the output might include:

```
#define _CP0_BCS_TAGLO(c,s) _bcsc0(_CP0_TAGLO, _CP0_TAGLO_SELECT, c, s)
#define PORTE PORTE
#define _LATE_LATE1_LENGTH 0x00000001
#define _IPC22_w_POSITION 0x00000000
```

You can use this option to show those macros which are predefined by the compiler or header files.

The acceptable letter arguments to `-d` are tabulated below.

Table 6-15. Preprocessor Debugging Information

Letter	Produces
`D`	Similar output to `-dM` but lacking predefined macros. The normal preprocessor output is also included in this output.

...........continued

Letter	Produces
`M`	Full macro output, as described in the main text before this table.
`N`	Similar output to `-dD` but only the macro names are output with each definition.

6.7.8.3 D: Define a Macro

The `-D`*`macro`* option allows you to define a preprocessor macro and the `-D`*`macro`*`=`*`text`* form of this option additionally allows a user-define replacement string to be specified with the macro. A space may be present between the option and macro name.

When no replacement text follows the macro name, the `-D`*`macro`* option defines a preprocessor macro called *`macro`* and specifies its replacement text as `1`. Its use is the equivalent of placing `#define` *`macro`* `1` at the top of each module being compiled.

The `-D`*`macro`*`=`*`text`* form of this option defines a preprocessor macro called *`macro`* with the replacement text specified. Its use is the equivalent of placing `#define` *`macro text`* at the top of each module being compiled.

Either form of this option creates an identifier (the macro name) whose definition can be checked by `#ifdef` or `#ifndef` directives. For example, when using the option, `-DMY_MACRO` (or `-D MY_MACRO`) and building the following code:

```
#ifdef MY_MACRO
int input = MY_MACRO;
#endif
```

the definition of the `int` variable `input` will be compiled, and the variable assigned the value `1`.

If the above example code was instead compiled with the option `-DMY_MACRO=0x100`, then the variable definition that would ultimately be compiled would be: `int input = 0x100;`

See 23.1.1. Preprocessor Arithmetic for clarification of how the replacement text might be used.

Defining macros as C string literals requires escaping the quote characters (`" "`) used by the string. If a quote is intended to be included and passed to the compiler, it should be escaped with a backslash character (`\`). If a string includes a space character, the string should have additional quotes around it.

For example, to pass the C string, `"hello world"`, (including the quote characters and the space in the replacement text), use `-DMY_STRING="\"hello world\""`. You could also place quotes around the entire option: `"-DMY_STRING=\"hello world\""`. These formats can be used on any platform. Escaping the space character, as in `-DMY_STRING=\"hello\ world\"` is only permitted with macOS and Linux systems and will not work under Windows, and hence it is recommended that the entire option be quoted to ensure portability.

All instances of `-D` on the command line are processed before any `-U` options.

6.7.8.4 H: Print Header Files Option

The `-H` option prints to the console the name of each header file used, in addition to other normal activities.

6.7.8.5 Imacros Option

The `-imacros` *`file`* option processes the specified file in the same way the `-include` option would, except that any output produced by scanning the file is thrown away. The macros that the file defines remain defined during processing. Because the output generated from the file is discarded, the only effect of this option 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 an `-imacros` option, regardless of the order in which they are placed. All the `-include` and `-imacros` options are processed in the order in which they are written.

6.7.8.6 Include Option

The `-include` *`file`* option processes *`file`* as if `#include "file"` appeared as the first line of the primary source file. In effect, the contents of *`file`* are compiled first. Any `-D` and `-U` options on the command line are always processed before the `-include` option, 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.

6.7.8.7 M: Generate Make Rule

The `-M` option tells the preprocessor to output a rule suitable for `make` that describes 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 header files it includes. This rule may be a single line or may be continued with a backslash-newline sequence if it is lengthy.

The list of rules is printed on standard output instead of the preprocessed C program.

The `-M` option implies `-E`.

6.7.8.8 MD: Write Dependency Information To File Option

The `-MD` option writes dependency information to a file.

This option is similar to `-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.

6.7.8.9 MF: Specify Dependency File Option

The `-MF` *`file`* option specifies a file in which to write the dependencies for the `-M` or `-MM` options. If no `-MF` option 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.

6.7.8.10 MG: Ignore Missing Header Files Option

The `-MG` option treats missing header files as generated files and adds them to the dependency list without raising an error.

The option assumes the missing files live in the same directory as the source file. If `-MG` is specified, then either `-M` or `-MM` must also be specified. This option is not supported with `-MD` or `-MMD`.

6.7.8.11 MM: Generate Make Rule For Quoted Headers Option

The `-MM` option performs the same tasks as `-M`, but system headers are not included in the output.

6.7.8.12 MMD: Generate Make Rule For User Headers Option

The `-MMD` option performs the same tasks as `-MD`, but only user header files are included in the output.

6.7.8.13 MP: Add Phony Target For Dependency Option

The `-MP` option instructs the preprocessor to add a phony target for each dependency other than the main file, causing each to depend on nothing. These MP rules work around make errors if you remove header files without updating the make-file to match.

This is typical output:

```
test.o: test.c test.h
test.h:
```

6.7.8.14 MQ: Change Rule Target With Quotes Option

The -MQ option is similar to -MT, but it quotes any characters which are considered special by make.

```
-MQ '$(objpfx)foo.o' gives $$(objpfx)foo.o: foo.c
```

The default target is automatically quoted, as if it were given with -MQ.

6.7.8.15 MT: Change Rule Target Option

The -MT target option changes the target of the rule emitted by dependency generation.

By default, the preprocessor 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
```

6.7.8.16 Nostdinc Option

The -nostdinc option prevents the standard system directories for header files being searched by the preprocessor. Only the directories you have specified with -I options (and the current directory, if appropriate) are searched.

By using both -nostdinc and -iquote, the include-file search path can be limited to only those directories explicitly specified.

6.7.8.17 P: Don't Generate #line Directives Option

The -P option tells the preprocessor not to generate #line directives in the preprocessed output. Used with the -E option.

6.7.8.18 No-show-column Option

The -fno-show-column option controls whether column numbers will be printed in diagnostics.

This option may be necessary if diagnostics are being scanned by a program that does not understand the column numbers, such as DejaGnu.

6.7.8.19 Trigraphs Option

The -trigraphs option turns on support for ANSI C trigraphs. The -ansi option also has this effect.

6.7.8.20 U: Undefine Macros

The -U*macro* option undefines the macro macro.

Any builtin macro or macro defined using -D will be undefined by the option. All -U options are evaluated after all -D options, but before any -include and -imacros options.

6.7.8.21 Undef Option

The -undef option prevents any system-specific or GCC-specific macros being predefined (including architecture flags).

6.7.9 Options for Assembling

The option tabulated below control assembler operations and is discussed in the section that follows.

Table 6-16. Assembly Options

Option (links to explanatory section)	Definition
`-Wa,`*`option`*	Pass `option` to the assembler.

6.7.9.1 Wa: Pass Option To The Assembler, Option

The `-Wa,`*`option`* option passes its *`option`* argument directly to the assembler. If *`option`* contains commas, it is split into multiple options at the commas. For example `-Wa,-a` will pass the `-a` option to the assembler, requesting that an assembly list file be produced.

6.7.10 Options for Linking

The options tabulated below control linker operations and are discussed in the sections that follow. If any of the options `-c`, `-S` or `-E` are used, the linker will not be executed.

Table 6-17. Linking Options

Option (links to explanatory section)	Definition
`--dinit-compress=`*`level`*	Applies optimizations of the specified level to the data initialization template, which initializes objects and `ramfunc` functions in RAM.
`-llibrary`	Search the library named *`library`* when linking.
`-nodefaultlibs`	Do not use the standard system libraries when linking.
`-nostdlib`	Do not use the standard system start-up files or libraries when linking.
`-s`	Remove all symbol table and relocation information from the output.
`-u symbol`	Add an undefined symbol that will be present at link stage.
`-Wl,`*`option`*	Pass options to the linker.
`-Xlinker `*`option`*	Pass system-specific options to the linker.

6.7.10.1 Dinit-compress Option

The `-dinit-compress=`*`level`* option enables optimization to the specified level of the data initialization template, which initializes objects and `ramfunc`-attributed functions in RAM. See 19.2.5. Initialize Objects and RAM Functions for more information on the operation of the data initialization template and record formats.

When the argument *`level`* is set to 0, the compiler uses the unoptimized legacy `.dinit` format used by v4.10 and prior compilers.

Level 1 merges those initialized objects in contiguous memory under the same `.dinit` record.

Level 2 performs level 1 optimizations but additionally groups the same (non-zero) initial value in a record of format #2 or #3 (as described in 19.2.5. Initialize Objects and RAM Functions), specifying respectively a single 16- or 32-bit initial value that will be copied multiple times. A format #2 record is chosen when there is a 16-bit repeated value in the initial value sequence (e.g. 0x13571357...); a format #3 record is chosen when there is a 32-bit repeated value in the initial value sequence (e.g. 0x6701238067012380...). This option adds size to runtime startup code.

Level 3 can perform any of the optimizations available in the lower levels and can additionally perform a PackBits-based run-length encoding compression to objects and `ramfunc` functions, storing this in record format #4. The pack-bits decompression algorithm will only be used if it saves an amount of space equal to or larger than the size of the decompression algorithm that needs to be additionally linked in; however, any repeated values initialized (as described above) might add to the size of the runtime startup routine. This is the default level if no option is specified.

6.7.10.2 L: Specify Library File Option

The `-l`*`library`* option scans the named library file for unresolved symbols when linking.

When this option is used, the linker will search a standard list of directories for the library with the name *library*.a. The directories searched include the standard system directories, plus any that you specify with the -L option.

The linker processes libraries and object files in the order they are specified, so it makes a difference where you place this option in the command. The options (and in this order), foo.o -llibz bar.o search library libz.a after file foo.o but before bar.o. If bar.o refers to functions in libz.a, those functions may not be loaded.

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., mylib.a) is that the compiler will search for a library specified using -l in several directories, as specified by the -L option.

By default the linker is directed to search <install-path>/lib for libraries specified with the -l option. This behavior can be overridden using environment variables.

See also the INPUT and OPTIONAL linker script directives.

6.7.10.3 Nodefaultlibs Option

The -nodefaultlibs option will prevent the standard system libraries being linked into the project. Only the libraries you specify are passed to the linker.

The compiler may generate calls to memcmp, memset and memcpy, even when this option is specified. As these symbols are usually resolved by entries in the standard compiler libraries, they should be supplied through some other mechanism when this option is specified.

6.7.10.4 Nostdlib Option

The -nostdlib option will prevent the standard system startup files and libraries being linked into the project. No startup 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.

Additionally, you must provide your own implementations of the __pic32c_data_initialization() and __libc_init_array() functions, which are called by the default device startup code calls .

6.7.10.5 S: Remove Symbol Information Option

The -s option removes all symbol table and relocation information from the output.

6.7.10.6 U: Add Undefined Symbol Option

The -u *symbol* option adds an undefined symbol that will be present at the link stage. To resolve the symbol, the linker will search library modules for its definition, thus this option is useful if you want to force a library module to be linked in. It is legitimate to use this option multiple times with different symbols to force loading of additional library modules.

6.7.10.7 Wl: Pass Option To The Linker, Option

The -Wl,*option* option passes *option* to the linker application where it will be interpreted as a linker option. If *option* contains commas, it is split into multiple options at the commas. Any linker option specified will be added to the default linker options passed by the driver and these will be executed before the default options by the linker.

6.7.10.8 Xlinker Option

The `-Xlinker` *`option`* option pass *`option`* to the linker where it will be interpreted as a linker option. You can use this to supply system-specific linker options that the compiler does not know how to recognize.

6.7.11 Options for Directory Search

The options tabulated below control directories searched operations and are discussed in the sections that follow.

Table 6-18. Directory Search Options

Option (links to explanatory section)	Definition
`-B`*`prefix`*	This option specifies where to find executables, libraries, include files and compiler data files.
`-I` *`dir`*	Add directory to search for header files
`-Idirafter` *`dir`*	Add directory to search for header files after all other paths are exhausted
`-Iquote` *`dir`*	Add directory to search for “quoted” header files before processing -I option directories
`-L`*`dir`*	Specify additional library search directories.

6.7.11.1 B: Specify Compiler Component Search Path Option

The `-B`*`prefix`* option specifies a path where executables, libraries, include files and compiler data files can be found.

The compiler driver runs one or more of the internal applications `xc32-cpp`, `xc32-as` and `xc32-ld`. It tries *`prefix`* as a prefix for each application it tries to run.

For each application to be run, the compiler driver first tries adding *`prefix`*. If the application cannot be found, the driver searches the search paths specified by the `PATH` environment variable for the application.

If *`prefix`* specifies a directory name, this path also applies to libraries used by the linker, because the driver translates this into `-L` options for the linker. This also applies to include file search paths, because the compiler translates these options into `-isystem` options for the preprocessor. In this case, the compiler appends `include` to the prefix.

6.7.11.2 I: Specify Include File Search Path Option

The `-I`*`dir`* option adds the directory *`dir`* to the head of the list of directories to be searched for header files. A space may be present between the option and directory name.

The option can specify either an absolute or relative path and it can be used more than once if multiple additional directories are to be searched, in which case they are scanned from left to right. The standard system directories are searched after scanning the directories specified with this option.

Under the Windows OS, the use of the directory backslash character may unintentionally form an escape sequence. To specify an include file path that ends with a directory separator character and which is quoted, use `-I "E:\\"`, for example, instead of `-I "E:\"`, to avoid the escape sequence `\"`. Note that MPLAB X IDE will quote any include file path you specify in the project properties and that search paths are relative to the output directory, not the project directory.

Note: Do not use this option to specify any MPLAB XC32 system include paths. The compiler drivers automatically select the default language libraries and their respective include-file directory for you. Manually adding a system include path may disrupt this mechanism and cause the incorrect 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.

6.7.11.3 Idirafter Option

The -idirafter *dir* option adds the directory *dir* list of directories to be searched for header files during preprocessing. The directory is searched only after all other search paths (including the standard search directories as well as those specified by the -I and -Iquote options) have been searched. If this option is used more than once, the directories they specify are searched in a left-to-right order as they appear on the command line.

6.7.11.4 Iquote Option

The -iquote *dir* option adds the directory *dir* to the list of directories to be searched for header files during preprocessing. Directories specified with this option apply only to the quoted form of the directive, for example #include "file", and the directory is searched only after the current working directory has been searched. If this option is used more than once, the directories they specify are searched in a left-to-right order as they appear on the command line.

6.7.11.5 L: Specify Library Search Path Option

The -L*dir* option allows you to specify an additional directory to be searched for library files that have been specified by using the -l option. The compiler will automatically search standard library locations, so you only need to use this option if you are linking in your own libraries.

6.7.12 Options for Code Generation Conventions

The options tabulated below control machine-independent conventions used when generating code and are discussed in the sections that follow.

Table 6-19. Code Generation Convention Options

Option (links to explanatory section)	Definition
-fargument-alias -fargument-noalias -fargument-noalias-global	Specify the possible relationships among parameters and between parameters and global data.
-fcall-saved-*reg*	Treat *reg* as an allocatable register saved by functions.
-fcall-used-*reg*	Treat *reg* as an allocatable register that is clobbered by function calls.
-f[no-]common	Controls the placement of global variables defined without an initializer.
-f[no-]exceptions	Generate extra code to propagate exceptions.
-ffixed-*reg*	Treat*reg* as a fixed register that will not be used in code.
-f[no-]ident	Ignore the #ident directive.
-fpack-struct	Pack all structure members together without memory holes.
-f[no-]pcc-struct- return	Return short struct and union values in memory, rather than in registers.
-f[no-]short-enums	Specify the size of enum types.
-f[no-]verbose-asm	Put extra commentary information in the generated assembly code to make it more readable.

6.7.12.1 Argument-alias Options

This set of options specify the possible relationships among parameters, and between parameters and global data.

The -fargument-alias option specifies that parameters might alias each other and might alias global storage.

The -fargument-noalias option specifies that parameters do not alias each other, but they might alias global storage.

The -fargument-noalias-global option specifies that parameters do not alias each other and do not alias global storage.

Each language automatically uses the appropriate option is required by the language standard, so you should not need to use these options yourself.

6.7.12.2 Call-saved-reg Option

The `-fcall-saved-`*`reg`* option will have the compiler treat the register named *`reg`* as an allocatable register that is saved by functions. Functions compiled with this option in effect save and restore the specified register if they use it.

The register specified in this option might be allocated for objects or temporary objects whose duration extends over a function call.

It is an error to specify the Frame Pointer or Stack Pointer registers with this option. Use of this option to specify other registers that have fixed pervasive roles in the machine's execution model might result in code failure. The same is also true if this option specifies a register in which function values are returned.

This option should be used consistently with all project modules.

6.7.12.3 Call-used-reg Option

The `-fcall-used-`*`reg`* option will have the compiler treat the register named *`reg`* as an allocatable register that is clobbered by functions. Functions compiled with this option in effect will not save and restore the specified register if they use it, meaning that the register's content might be lost.

The register specified in this option might be allocated for objects or temporary objects whose duration extends over a function call.

It is an error to specify the Frame Pointer or Stack Pointer registers with this option. Use of this option to specify other registers that have fixed pervasive roles in the machine's execution model might result in code failure. The same is also true if this option specifies a register in which function values are returned.

This option should be used consistently with all project modules.

6.7.12.4 Common Option

The `-fcommon` option tells the compiler to allow merging of tentative definitions by the linker. This is the default action if no option is specified.

The definition for a file-scope object without a storage class specifier or initializer is known as a tentative definition in the C standard. Such definitions are treated as external references when the `-fcommon` option is specified and will be placed in a common block. As such, if another compilation unit has a full definition for an object with the same name, the definitions are merged and storage allocated. If no full definition can be found, the linker will allocate unique memory for the object tentatively defined. Tentative definitions are thus distinct from declarations of a variable with the `extern` keyword, which never allocate storage.

In the following code example:

```
extra.c
int a = 42;  /* full definition */

main.c
int a;       /* tentative definition */
int main(void) {
  ...
}
```

The object `a` is defined in `extra.c` and tentatively defined in `main.c`. Such code will build when using the `-fcommon` option, since the linker will resolve the tentative definition for `a` with the full definition in `extra.c`.

The `-fno-common` form of this option inhibits the merging of tentative definitions by the linker, treating tentative definitions as a full definition if the end of the translation unit is reached and no definition has appeared with an initializer. Building the above code, for example, would result in a multiple definition of 'a' error, since the tentative definition and initialized definition would both attempt to allocate storage for the same object. If you are using this form of the option, the definition of a in main.c should be written extern int a; to allow the program to build.

6.7.12.5 Exceptions option

The -fexceptions option generates extra code needed to propagate exceptions. This option might need to be specified when compiling C code that needs to inter-operate properly with exception handlers written in C++.

The -fno-exceptions form of this option disables exception handling. This is the default action if no option is specified.

Ensure that the same form of this option is specified at compile time for all modules and at link time. When used additionally at link time, the option allows the linker to select a variant of the libraries that are built with C++ exceptions disabled, reducing code size.

6.7.12.6 Fixed-reg Option

The -ffixed-reg option has the compiler treat the register named *reg* as a fixed register. Code generated by the compiler will never make use of this register (except in cases where the register has some fixed role, like as a Stack Pointer or Frame Pointer).

The register name can be a register number, for example -ffixed-3 to refer to r3.

6.7.12.7 Ident option

The -fident option forces the compiler to process any occurrence of the #ident directive, which places the string constant argument to the directive into a special segment of the object file. This is the default action if no option is specified.

The -fno-ident form of this option forces the compiler to ignore any occurrence of this directive.

6.7.12.8 Pack-struct Option

The -fpack-struct option packs all structure members together without padding bytes between them. Typically, this option is not used, since the structure members might become unaligned and the code required to access them becomes sub-optimal. Additionally, the offsets of members within such structures won’t agree with those for unpacked (aligned) structures in system libraries.

Code must take special care accessing packed structure members via a pointer. Unaligned access is not permitted on some Cortex-M devices and attempting such access will cause a hardfault exception.

The -fno-pack-struct option chooses a structure arrangement that will produce optimal code, potentially placing padding bytes between members to ensure they are properly aligned for the device. This is the default action if no option is specified.

6.7.12.9 Pcc-struct-return Option

The -fpcc-struct-return option forces the compiler to return small struct and union objects (whose size and alignment match that of an integer type) in memory, rather than in registers. This convention is less efficient but allows compatibility with files compiled using other compilers.

The -fno-pcc-struct-return form of this option returns small struct and union objects in registers. This is the default action no option is specified.

6.7.12.10 Short-enums Option

The -fshort-enums option allocates the smallest possible integer type (with a size of 1, 2, or 4 bytes) to an enum such that the range of possible values can be held.

The -fno-short-enums form of this option forces each enum type to be 4-bytes wide (the size of the int type). This is the default action if no option is specified. When using this option, generated code is not binary compatible with code generated without the option.

6.7.12.11 Verbose-asm Option

The `-fno-verbose-asm` option places extra commentary information in the generated assembly code to make it more readable.

The `-fno-verbose-asm` form of this option omits this extra information and is useful when comparing two assembler files. This is the default action if no option is specified.

7. C Standard Issues

This compiler is a freestanding implementation that conforms to the ISO/IEC 9899:1990 Standard (referred to as the C90 standard) as well the ISO/IEC 9899:1999 Standard (C99) for programming languages, unless otherwise stated.

7.1 Divergence from the C99 Standard

The C language implemented on MPLAB XC32 C Compiler diverges from the C99 Standard, as described in the following sections.

7.1.1 Library Divergences

The standard pragmas, for example `#pragma STDC FP_CONTRACT`, are not supported by MPLAB XC32.

Additionally, the default runtime startup code does not call `exit()` after returning from `main()`. This is to reduce code size; however, custom startup code that does call `exit()` can be provided if necessary.

7.2 Extensions to the C99 Standard

The compiler permits the use of certain source constructs, detailed in the following sections, that are in addition to those provided by the C standard.

7.2.1 Keyword Extensions

The compiler supports several keywords and attributes in addition to those provided by the C standard. These 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 – 9.11. Variable Attributes
- Specifying Attributes of Functions – 17.2.1. Function Attributes
- Inline Functions – 17.9. Inline Functions
- Variables in Specified Registers – 9.11. Variable Attributes
- Complex Numbers – 9.7. Complex Data Types
- Double-Word Integers – 9.3. Integer Data Types
- Referring to a Type with `typeof` – 14.2. Type References

7.2.2 Statement and Expression Extensions

The compiler allows expressions using features not supported by the C standard. These features 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 – 14.3. Labels as Values
- Conditionals with Omitted Operands – 14.4. Conditional Operator Operands
- Case Ranges – 14.5. Case Ranges
- Binary constants – 9.8. Constant Types and Formats.

7.3 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 25. Implementation-Defined Behavior.

8. 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 ROM-based applications. This chapter documents the special language features which are specific to these devices.

8.1 Device Support

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

8.2 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 8.4. Using SFRS From C Code for further information.

8.3 Configuration Bit Access

The `#pragma config` directive specifies the Configuration Words to be programmed into the device running the application.

The PIC32 devices have several Configuration Words. The bits within these words control 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 `config` pragma has the following general form:

```
#pragma config setting = value
```

where *`setting`* is a configuration word bit-field name, e.g., `WDT_ENABLE`, and *`value`* can be either a textual description of the desired state, e.g., `CLEAR`, or a numerical value, as required by the setting. Multiple pragmas may be used to incrementally specify the complete device state; however, more than one comma-separated setting-value pair may be specified with each pragma.

The following example shows some pragmas relevant for a SAME54P20A device. You can find the configuration word bit-field names used in the setting descriptor from the **Configuration Bits** view in the MPLAB X IDE.

```
#pragma config BOD33_DIS = SET
#pragma config BOD33USERLEVEL = 0x1c, BOD33_ACTION = RESET
#pragma config WDT_PER = CYC8192
```

The `#pragma config` directives should be specified in only a single translation unit, or module. They should be placed outside of a function definition, as they do not define executable code.

The compiler verifies that the configuration settings specified are valid for the target device. 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. It is recommended that all Configuration Words are explicitly programmed to ensure correct device operation.

The `config` pragma can be used with those devices whose configuration locations must be addressed at the byte level instead of the (4-byte) word level.

In cases where the same configuration word bit-field name is used by more than one configuration setting in different Configuration Words, a more qualified name can be used. The configuration word bit-field (e.g. `BOD33_DIS`) can be prefixed with the register name (e.g. `USER_WORD_0`) separated by an underscore character, as in, `USER_WORD_0_BOD33_DIS`. If further clarification is required, the setting can be additionally prefixed with the register-group address space (e.g. `fuses`) and an underscore character, as in, fuses_USER_WORD_0_BOD33_DIS. The pragmas in the above example could be rewritten using longer forms of the setting descriptor if desired and as follows (although such names are not necessary when using this device):

```
#pragma config USER_WORD_0_BOD33_DIS = SET
#pragma config fuses_USER_WORD_0_BOD33USERLEVEL = 0x1c, fuses_USER_WORD_0_BOD33_ACTION = RESET
#pragma config USER_WORD_1_WDT_PER = CYC8192
```

The compiler will emit an error and make suggestions if a fully qualified name was not used but is required.

8.4 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 8.2. 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. 8.4.1. 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 can be found in architecture-specific directories under the pic32c/include/proc directory of your compiler distribution; however they should not be explicitly included into your code, as this will reduce portability.

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.

8.4.1 SFR Register Definitions

In this section we describe the conventions of the SFR interface. In what follows we 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, e.g., _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.

8.5 Tightly-Coupled Memories

The compiler supports attributes and options that allow the placement of code and objects into Tightly Coupled Memory (TCM). Accessing code and data in this memory avoids the intricacies of a system bus matrix and provides fixed latency and hence deterministic program execution.

Some Cortex-M7 devices implement TCM in SRAM, as separate instruction TCM (or ITCM), used to hold code, and data TCM (DTCM), used to hold data. These memories can be enabled and configured through independent device registers. The internal organization and size of the TCM areas vary greatly across devices, and this will be discussed in the data sheet for your selected device.

The `tcm` attribute places objects or functions into the appropriate TCM, for example `uint32_t __attribute__((tcm)) var;`, which will place the object `var` into DTCM.

In device families where separate ITCM and DTCM are present, independent driver options allow you to set the size of these memory areas, those being `-mitcm=`*`size`* and `-mdtcm=`*`size`*, which set the size (in bytes) of the instruction and data TCM respectively. The *`size`* specified must be one permitted by the device and can be specified as a decimal number or as a hexadecimal number with a leading 0x. When an invalid size has been specified, check the error message to see the sizes acceptable by that device. Use of these options will ensure that the device-specific runtime startup code and linker script enable and initialize TCM before your main() function is called. Once enabled, the preprocessor macros __XC32_ITCM_LENGTH and/or __XC32_DTCM_LENGTH will be defined and equated to the size of the respective TCM area.

By default, the vector table is allocated to TCM if possible. The option -mno-vectors-in-tcm keeps the vector table out of TCM, which might be desirable if the selected device has limited memory available.

The data stack can be moved to TCM by ensuring that the -mstack-in-tcm driver option is issued at both 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.

8.5.1 CMCC-based Tightly-Coupled Memory

Some SAM microcontrollers based on the Cortex®-M4, such as the SAME54 or SAMD51, feature a ARM Cortex-M Cache Controller (CMCC) peripheral. With CMCC, a part of the cache can be used as Tightly-Coupled Memory (TCM) for deterministic code performance by loading the critical code in a WAY and locking it.

To have functions and objects placed into CMCC-based TCM, use the tcm attribute with those functions and objects which need to be repositioned, then use the -mtcm=*size* option to set the size (in bytes) of the TCM and ensure that the device-specific runtime startup code and linker script configure the TCM. The *size* specified must be one permitted by the device and can be specified as a decimal number or as a hexadecimal number with a leading 0x. When an invalid size has been specified, check the error message to see the sizes acceptable by that device. Once enabled, the preprocessor macro __XC32_TCM_LENGTH will be defined and equated to the size of the TCM area.

8.6 Code Coverage

After purchase of the Analysis Tool Suite License (SW006027-2), the compiler's code coverage feature can be used to facilitate analysis of the extent to which a project's source code has been executed.

When enabled, this feature instruments the project's program image with small assembly sequences. When the program image is executed, these sequences record the execution of the code that they represent in reserved areas of device RAM. The records stored in the device can be later analyzed to determine which parts of a project's source code have been executed. Compiler-supplied library code is not instrumented.

When code coverage is enabled, the compiler will execute an external tool called xc-ccov to determine the most efficient way to instrument the project. The tool considers the program's basic blocks, which can be considered as sequences of one or more instructions with only one entry point, located at the start of the sequence and only one exit located at the end. Not all of these blocks need to be instrumented, with the tool determining the minimum set of blocks that will allow the program to be fully analyzed.

Use the -mcodecov option to enable code coverage in the compiler. The preprocessor macro __CODECOV will be defined once the feature is enabled.

All compiler options you use to build the project, when using code coverage, are significant, as these will affect the program image that is ultimately instrumented. To ensure that the analysis accurately reflects the shipped product, the build options should be the same as those that will be used for the final release build.

If code coverage is enabled, there will be 1 bit of RAM allocated per instrumented basic block, which will increase the data memory requirement of the project.

There is a small sequence of assembly instructions inserted into each instrumented basic block to set the corresponding coverage bit.

The instrumented project code must be executed for code coverage data to be generated and this execution will be fractionally slower due to the added assembly sequences. Provide the running program with input and stimulus that should exercise all parts of the program code, so that execution of all parts of the program source can be recorded.

Code coverage data can be analyzed in the MPLAB X IDE. Information in the ELF file produced by the compiler will allow the plugin to locate and read the device memory containing the code coverage results and display this in a usable format. See Microchip's Analysis Tool Suite License webpage for further information on the code coverage feature and other analysis tools.

9. 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 10. Memory Allocation and Access.

9.1 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.

9.2 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

9.3 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
`signed char`	8	-128	127
`char, 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

9.3.1 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.

9.3.2 `limits.h`

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

Table 9-1. LIMITS.H HEADER FILE

Macro Name	Value	Description
`CHAR_BIT`	8	The size, in bits, of the smallest non-bit field object.

...........continued

Macro Name	Value	Description
`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 9.3.1. Signed and Unsigned Character Types)	The minimum value possible for an object of type `char`.
`CHAR_MAX`	127 (or 255, see 9.3.1. 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`.
`ULLONG_MAX`	$2^{64}-1$	The maximum value possible for an object of type `unsigned long long`.

9.4 Floating-Point Data Types

The compiler uses 32- and 64-bit forms of the IEEE-754 floating-point format to store floating-point values. The implementation limits applicable to a translation unit are contained in the `float.h` header.

Attention: Some target PIC32C/SAM devices implement a Floating-point Unit (FPU). The compiler implements certain features, described in this guide, to support this hardware.

The table below shows the data types and their corresponding size and arithmetic type.

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 the table below, 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:

$(-1)^{sign}$ x $2^{(exponent-127)}$ x 1. mantissa

and for 64-bit values

$(-1)^{sign}$ x $2^{(exponent-1023)}$ x 1. mantissa.

In the following table, examples of the 32- and 64-bit IEEE 754 formats are shown. Note that the Most Significant bit of the mantissa column (that is, 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 number is zero).

Table 9-2. Floating-Point Format Example IEEE 754

Format	Number	Biased Exponent	1.mantissa	Decimal
32-bit	0x7DA6B69C	11111011b (251)	1.01001101011011010011100b (1.3024477959)	2.770000117e+37 —
64-bit	0x47B4D6D37131A DE	10001111011b (1147)	1.0100110101101101001101110001001100011010011011011110b (1.3024477407110946)	2.77e+37 —

The example in the table can be calculated manually as follows.

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

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

which becomes:

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

which is approximately equal to:

2.77000*e*+37

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

So, for example, if you are using a 32-bit wide floating-point type, it can exactly store the value 95000.0. However, the next highest number it can represent is (approximately) 95000.00781 and it is impossible to represent any value in between these two in such a type as it will be rounded.

This implies that C/C++ code which compares floating-point type may not behave as expected. For example:

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

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

The characteristics of the floating-point formats are summarized in Table 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. So, for example, `FLT_MAX` represents the maximum floating-point value of the float type. `DBL_MAX` represents the same values for the `double` type. As the size and format of floating-point data types are not fully specified by the ANSI Standard, these macros allow for more portable code which can check the limits of the range of values held by the type on this implementation.

Table 9-3. 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	2.2204460492503131e-16

9.5 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.

9.5.1 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};
```

9.5.2 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;
    // ...}
```

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

9.6 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.

9.6.1 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.

9.6.2 Data Pointers

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

9.6.3 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.

9.6.4 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 PIC32C/SAM® 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.

9.7 Complex Data Types

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

9.8 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 9-4. 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. So, for example, if the suffix `l` is specified and the value is a decimal constant, the compiler will assign the type `long int`, if that type will hold the constant; otherwise, it will assigned `long long int`. If the constant was specified as an octal or hexadecimal constant, then unsigned types are also considered.

Table 9-5. 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`

...........continued

Suffix	Decimal	Octal or Hexadecimal
`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”`.

9.9 Standard Type Qualifiers

Type qualifiers provide additional information regarding how an object may be used. The MPLAB XC32 C/C++ Compiler supports both standard C qualifiers and additional, special qualifiers that are useful for embedded applications and that take advantage of the PIC32C/SAM architecture.

9.9.1 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.

9.9.2 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.

9.10 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.

9.11 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 an attribute, and declared `extern` in file B without the same attribute, then a link error might 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. Also, see the description for the `space` attribute.

`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.

`externally_visible`

This attribute when used with a global object ensures the object remains visible outside the current compilation unit. This might prevent certain optimizations from being performed on the object.

`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.

`persistent`

This attribute prevents the startup code's data-initialization code from initializing the variable. This behavior can be useful when you want to preserve the value across a soft reset. *Caveat:* Note that on microcontrollers featuring an L1 cache, cache reinitialization may cause the runtime value of the variable to change. For devices with an L1 cache and a Memory Protection Unit (MPU), you may choose to use the Memory Protection Unit to create an uncached region of RAM and place the persistent variable into that region.

`section ("`*`section-name`*`")`

Place the variable into the named section.

For example,

```
unsigned int dan __attribute__ ((section (".quixote")))
```

Variable `dan` will be placed in section `.quixote`.

The `-fdata-sections` command line option has no effect on variables defined with a `section` attribute unless `unique_section` is also specified.

`space(`*`memory-space`*`)`

The `space` attribute can be used to direct the compiler to allocate a variable in a specific memory space. Valid memory spaces are `prog` for program memory and `data` for data memory. The `data` space is the default space for non-`const` variables.

The `prog` and `data` spaces normally correspond to the `rom`, `ram` 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)`, 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)` 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;
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.

used

Indicate to the compiler that the object is always used and storage must be allocated for the object, even if the compiler cannot see a reference to it. For example, if inline assembly is the only reference to an object.

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.

10. 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.

10.1 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.

10.2 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 9.9.1. 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.

10.2.1 Non-auto Variable Allocation

Non-`auto` variables (those with permanent storage duration) are located by the compiler into any of the available data memory. 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` This section 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` This section 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.

[1] The device can be considered a Harvard architecture in terms of its internal bus arrangement.

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.

10.2.2 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 6.4.2. Peripheral Library Functions. Static variables are located in the same sections as their non-`static` counterparts.

10.2.3 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 9.5. Structures and Unions. There are no theoretical limits as to how large these objects can be made.

10.2.4 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 9.9.1. 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 9.11. 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 9.11. Variable Attributes.

10.2.5 Data Memory Allocation Macros

The `sys/attribs.h` header file provides many macros for commonly used attributes in order to enhance code readability.

`__ramfunc__`	Locate the attributed function in the RAM function code section.
`__longramfunc__`	Locate the attributed function in the RAM function code section and apply the `long_call` attribute.

10.3 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.

10.3.1 Local Variable Size Limits

There is no theoretical maximum size for auto variables.

10.4 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
```

10.4.1 Size Limitations of const Variables

There is no theoretical maximum size for const variables.

10.4.2 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 9.11. Variable Attributes.

If you cannot have const-qualified objects placed in a section that also contain executable code, use the –mpure–code option, described in 6.7.1. Options Specific to PIC32C/SAM Devices.

10.5 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.

As indicated in 15.2. 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`

Example 10-1. Variables in Registers

```
volatile unsigned int special;
unsigned int example (void)
{
  register unsigned int my_reg __asm__("V1");
  my_reg += special;
  return my_reg;
}
```

10.6 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.

11. Chip-Level Security and Arm® TrustZone® Technology

SAM L11 MCUs integrate chip-level security and Arm TrustZone technology to help protect from both physical and remote attacks.

XC32 v2.20 and higher support the Arm v8-M CMSE security extension using the `–mcmse` option. Linking TrustZone applications, both secure and non-secure, requires passing extra arguments to the linker to describe memory region sizes. Secure applications must also pass linker options to create the secure gateway veneers. Applications written that take advantage of TrustZone are really two applications, compiled and linked separately.

11.1 ARMv8-M Security Extensions

On TrustZone enabled devices, memory is partitioned into two zones: secure and non-secure. Access to secure memory in the non-secure state is done by way of function calls. Such a call originates in non-secure memory and transitions to executing in secure memory. The transition is done by way of a veneer found in another special zone of memory: non-secure callable (NSC). Non-secure callable has various restrictions on how it must be constructed and is populated by the linker. It is always found at the end of the secure memory area.

The `–mcmse` option enables the ARMv8-M Security Extensions as described in the "*ARMv8-M Security Extensions: Requirements on Development Tools - Engineering Specification*" (developer.arm.com/documentation/ecm0359818/latest).

As part of the Security Extensions XC32 implements two new function attributes: `cmse_nonsecure_entry` and `cmse_nonsecure_call`. XC32 also implements the intrinsics below. FPTR is used here to mean any function pointer type.

`cmse_address_info_t cmse_TT (void *)`	Generates a TT instruction
`cmse_address_info_t cmse_TT_fptr (FPTR)`	Generates a TT instruction. The argument FPTR can be any function pointer type.
`cmse_address_info_t cmse_TTT (void *)`	Generates a TT instruction with the T flag.
`cmse_address_info_t cmse_TTT_fptr (FPTR)`	Generates a TT instruction with the T flag. The argument FPTR can be any function pointer type.
`cmse_address_info_t cmse_TTA (void *)`	Generates a TT instruction with the A flag.
`cmse_address_info_t cmse_TTA_fptr (FPTR)`	Generates a TT instruction with the A flag. The argument FPTR can be any function pointer type.
`cmse_address_info_t cmse_TTAT (void *)`	Generates a TT instruction with the T and A flag.
`cmse_address_info_t cmse_TTAT_fptr (FPTR)`	Generates a TT instruction with the T and A flag. The argument FPTR can be any function pointer type.
`void * cmse_check_address_range (void *, size_t, int)`	Perform permission checks on C objects
`typeof(p) cmse_nsfptr_create (FPTR)`	Returns the value of p with its LSB cleared. The argument FPTR can be any function pointer type.
`intptr_t cmse_is_nsfptr (FPTR)`	Returns non-zero if p has LSB unset, zero otherwise. The argument FPTR can be
`int cmse_nonsecure_caller (void)`	Returns non-zero if entry function is called from non-secure state and zero otherwise.

11.2 Linker Macros Controlling Memory Regions for TrustZone®

Linking either the secure or non-secure application requires that the linker know the address and length of secure, non-secure, and non-secure callable memory regions. The following are the preprocessor definitions that are used to set and control memory zones for TrustZone when linking TrustZone applications. They are passed as `-Wl,-DNAME[=value]`.

When building a non-secure application, the following preprocessor definitions affect the linker script.

- `BOOTPROT=`*`size`* (optional): Defines the boot protections size in bytes. The default value is 0 if not provided.
- `AS=`*`size`* (recommended): Defines the flash secure application size, in bytes. Defaults to 50% of ROM if not provided.
- `RS=`*`size`* (recommended): Defines the size of secure RAM, in bytes. Defaults to 50% of RAM if not provided.

When building a secure application, the following preprocessor definitions affect the linker script.

- `SECURE` (required): Use the memory layout for a secure application.
- `BOOTPROT=`*`size`* (optional): Defines the boot protections size in bytes. The default value is 0 if not provided.
- `AS=`*`size`* (recommended): Defines the flash application secure size, in bytes. Defaults to 50% of ROM is not provided.
- ANSC=*`size`* (recommended): Defines the flash application non-secure callable size, in bytes. Defaults to 0 if not provided.
- `RS=`*`size`* (recommended): Defines the size of secure RAM, in bytes. Defaults to 50% of RAM if not provided.

In all cases, if a recommended value is not provided it will result in a warning message from the linker preprocessor.

Important: The memory region sizes must match those programmed in the device configuration bits in NVM User Row (UROW) and NVM Boot Configuration Row (BOCOR). These can be set as configuration bits via #pragma config in the source code. If that is done, the values in the config bits must match up with the values in the linker scripts dictating the memory regions to facilitate proper operation. See the normal config pragma documentation for the names of the fields and values.

12. Floating-point Support

Some PIC32C/SAM devices implement a 1985 IEEE-754 compliant Floating-point Unit (FPU) that supports single and double precision data types.

12.1 Floating-point Calling Conventions

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 a specific FPU calling convention is required, you can specify the following options. These must be issued 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.

13. Fixed-Point Arithmetic Support

The MPLAB XC32 C/C++ Compiler supports fixed-point arithmetic. This, according to ISO/IEC TR 18037:2008//the N1169 draft of ISO/IEC TR 18037, the ISO C99 technical report on Embedded C. It is available at [via]: http://www.open-std.org/JTC1/SC22/WG14/www/projects#18037. or standards.iso.org/ittf/PubliclyAvailableStandards/c051126_ISO_IEC_TR_18037_2008.zip

This chapter describes the implementation-specific details of the types and operations supported by the compiler under this draft standard.

Because of the sensitivity of DSP applications to performance, application developers have historically tended to write functions in assembly. However, the XC32 compiler reduces, and may even eliminate, the need to write assembly code. This chapter describes coding styles and usage tips that can help you to obtain the best optimizations for your DSP application.

13.1 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. The `<stdfix.h>` header may be included to provide convenient definitions as described in 13.2. Data Types.

13.2 Data Types

All 12 of the primary fixed-point types and their aliases, described in Section 4.1 “Overview and principles of the fixed-point data types” of ISO/IEC TR 18037:2008, are supported. Fixed-point data values contain fractional and optional integral parts. The format of fixed-point data in XC32 are as specified in the table below.

In the formats shown, `s` is the sign bit for a signed type (there is no sign bit for an unsigned type). The period character (`.`) is the specifier that separates the integral part and the fractional part. The numeric digits represent the number of bits in the integral part or in the fractional part.

Table 13-1. Fixed Point Formats

Type	Format	Description
`short _Fract`	s0.7	1 sign bit, no integer bits, 7 fractional bits
`unsigned short _Fract`	0.8	8 fractional bits only
`_Fract`	s0.15	1 sign bit, 15 fractional bits
`unsigned _Fract`	0.16	16 fractional bits only
`long _Fract`	s0.31	1 sign bit, no integer bits, 31 fractional bits
`unsigned long _Fract`	0.32	32 fractional bits only
`long long _Fract`	s0.63	1 sign bit, no integer bits, 63 fractional bits
`unsigned long long _Fract`	0.64	64 fractional bits only
`short _Accum`	s8.7	1 sign bit, 8 integer bits, 7 fractional bits
`unsigned short _Accum`	8.8	8 integer bits, 8 fractional bits
`_Accum`	s16.15	1 sign bit, 16 integer bits, 15 fractional bits
`unsigned _Accum`	16.16	16 integer bits, 16 fractional bits
`long _Accum`	s32.31	1 sign bit, 32 integer bits, 31 fractional bits
`unsigned long _Accum`	32.32	32 integer bits, 32 fractional bits
`long long _Accum`	s32.31	1 sign bit, 32 integer bits, 31 fractional bits
`unsigned long long _Accum`	32.32	32 integer bits, 32 fractional bits

The _Sat type modifier may be used with any type in the above table to indicate that values are saturated, as described in ISO/IEC TR 18037:2008. For example, _Sat short _Fract is the saturating form of short _Fract. Signed types saturate at the most negative and positive numbers representable in the corresponding format. Unsigned types saturate at zero and the most positive number representable in the format.

The MPLAB XC32 C compiler provides an include file, stdfix.h, which provides various pre-processor macros related to fixed-point support. These include those show in the following table.

<stdfix.h> alias	Type
fract	_Fract
accum	_Accum
sat	_Sat

Usage example:

```
#include <stdfix.h>
void main () {
  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;
  /* Calculate dot product of a[] and b[] */
  for (i=0; i<5; i++) {
      dp += a[i] * b[i];
  }
}
```

The default behavior of overflow on signed or unsigned types is saturation. The pragmas described in Section 4.1.3 "Rounding and Overflow" of ISO/IEC TR 18037:2008 to control the rounding and overflow behavior are not supported.

The following table describes the fixed-point literal suffixes supported to form fixed-point literals of each type.

Table 13-2. 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

13.3 Fixed-point Library Functions

The fixed-point functions described in Section 4.1.7 in ISO/IEC TR 18037:2008 (rounding, conversion functions, etc.) are not provided in the current MPLAB XC32 standard C libraries.

13.4 Operations on Fixed-Point Variables

Support for fixed-point types includes:

- 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

13.5 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 18037:2008, 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 18037:2008 are not provided by the current MPLAB XC32 standard C libraries.

14. 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 25. 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.

14.1 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 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.

14.2 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`.

14.3 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.

14.4 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.

14.5 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:
```

15. Register Usage

This chapter examines registers used by the compiler to generate assembly from C/C++ source code.

15.1 Register Usage

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.

15.2 Register Conventions

The 16 general-purpose registers in the Arm® Cortex®-Mx core of PIC32C devices are shown in the table below. 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 15-1. Register Conventions

Register Number	Special Name	Use
R0-R3		General purpose registers/function arguments and return value registers.
R4-R8		General purpose registers/variable registers.
R9		General purpose registers/platform register.
R10		General purpose registers/variable 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.

16. Stack

The software stack used with the PIC32C/SAM devices is discussed in this chapter.

16.1 Software Stack

The PIC32C/SAM 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 PIC32C/SAM devices use a dedicated stack pointer register `sp` (register r13) 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 at runtime, 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 -mprocessor=ATSAME70N20B foo.c -Wl,--defsym=__min_stack_size=2048
```

Two working registers are used to manage the stack:

- Register r13 (`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.

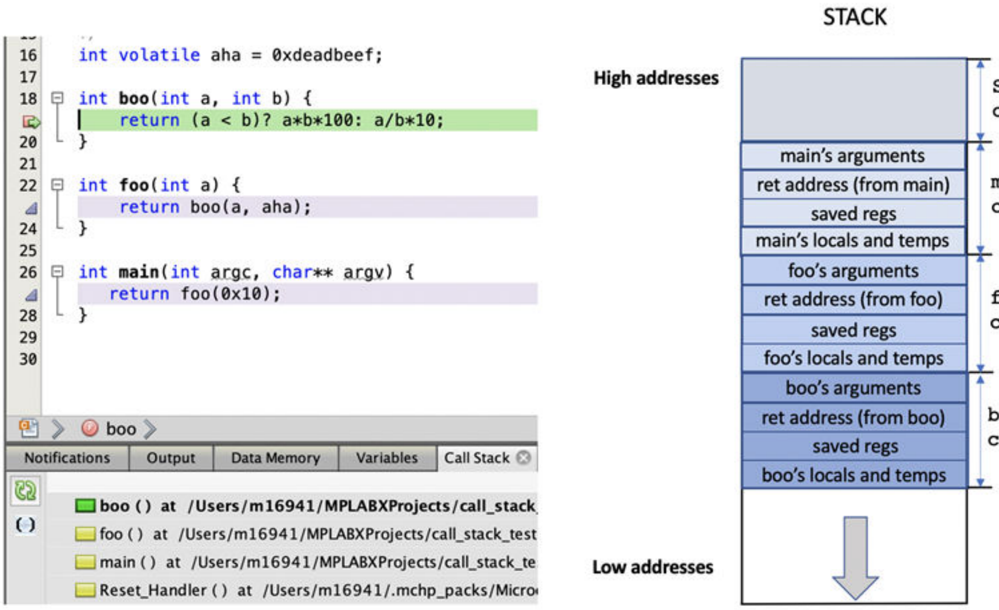
The C/C++ run-time start-up module initializes the stack pointer during the start-up and initialization sequence, see 19.2.1. Initialize Stack Pointer and Heap.

16.2 Call Frame

The stack is a memory region that grows dynamically at runtime. It starts at high address and grows to lower address (stack grows downward).

The stack has call frames, each one containing data corresponding to an active function call.

Figure 16-1. Call Frame Diagram



```
int volatile aha = 0xdeadbeef;

int boo(int a, int b) {
    return (a < b)? a*b*100: a/b*10;
}

int foo(int a) {
    return boo(a, aha);
}

int main(int argc, char** argv) {
    return foo(0x10);
}
```

```
boo () at /Users/m16941/MPLABXProjects/call_stack
foo () at /Users/m16941/MPLABXProjects/call_stack_test
main () at /Users/m16941/MPLABXProjects/call_stack_te
Reset_Handler () at /Users/m16941/.mchp_packs/Micro
```

A call frame contains:

- return address
- callee saved registers
- parameters passed through stack
- local variables
- temporary variables (inserted by compiler)
- interrupt context

16.3 Stack Smashing Protector (SSP) for Target Devices with an Arm Cortex Core

Background

“Smashing the stack” is a term coined by Elias Levy (a.k.a. Aleph One) in a 1996 article in *Phrack Magazine* titled “Smashing the Stack for Fun and Profit.” This article is still available from various sources online including duckduckgo.com/?q=Smashing+the+Stack+for+Fun+and+Profit+by+aleph+one&ia=web.

In many applications written in C, attackers can corrupt the execution stack by writing past the end of a local buffer array using the automatic storage class. An attack overwrites or “smashes” critical data on the stack to enable nefarious actions used in a security exploit. The details of a stack-smashing attack are beyond the scope of this document.

The Stack Smashing Protector (SSP) compiler feature helps detect a stack buffer overrun through runtime detection of changes to a secret value, known as the stack canary, located on the stack. The feature helps the application developer to identify such application bugs and vulnerabilities and it also helps mitigate against stack-smashing security attacks. To help detect these types of problems, the compiler instruments the function’s prologue code to store the secret stack-canary value, to the stack before other automatic variables. Then, before returning from the function, the function’s epilogue code checks the stored stack-canary value. If the function’s epilogue code determines that the value has been modified, a stack buffer overflow has occurred and a failure callback, `__stack_chk_fail()`, is invoked as described in detail below.

Important: The Stack Smashing Protector (SSP) feature merely helps to detect stack buffer overruns. It does not prevent them. In addition, defeating the detection mechanism is possible by preparing the input such that the stack canary is overwritten with the correct secret value and thus does not offer perfect protection. Therefore, your first line of defense must be to fix your application code to eliminate the possibility of buffer overruns. **Do not** rely on this feature to catch all cases of buffer overruns.

Since this feature adds code to both the function prologue and epilogue, it increases code size and execution time of every affected function. The compiler options listed below allow you to specify which types of functions to instrument.

16.3.1 Stack Protector Function Attribute

`__attribute__((stack_protect))`

This attribute adds stack protection code to the function if flags `-fstack-protector`, `-fstack-protector-strong` or `-fstack-protector-explicit` are set.

Optimizations can affect stack protection:

- Function inlining can affect whether a function is protected.
- Removal of an unused variable can prevent a function from being protected.

Example 16-1. Usage (test_stack_protect.c)

```
#include <stdint.h>
int32_t __attribute__((stack_protect)) func1()
{
    char ana[]="Test canary use";
    int32_t i;

    if (ana[1] == ana[14])
      return 1;

    return 0;
}

int main()
{
    return func1();
}
```

Compiled with this command line:

```
xc32-gcc -fstack-protector -O2 test_stack_protect.c -o test_p.elf -
mprocessor=ATSAME70J19
```

In this example, function `func1()` would be instrumented with the stack protection code. The stack protector feature adds one stack-canary variable on the call stack of `func1()` and, at the end of the function, it checks the variable value. If the value is modified, the stack was corrupted, and the generated code calls the `__stack_chk_fail()` function.

16.3.2 Stack Callback Function

When the checking code detects that the guard variable on the stack has been modified, it notifies the run-time environment by calling the function: `void __stack_chk_fail(void);`

You must provide an implementation for this function. suitable for your application. Normally, such a function terminates the program, possibly after reporting a fault.

XC32 provides a default, weak implementation for `__stack_chk_fail()` with a minimal footprint. `__stack_chk_fail()` is called when an overflow is detected, as due to the canary being overwritten. This function should not return. If the overflow was caused by an attack, the suspended execution is a good way to block it.

```
void __attribute__((noreturn)) __stack_chk_fail (void)
{
    // XC32 default weak implementation
    while(1)
    {
        // Replace this code with application-specific code
        __builtin_software_breakpoint();
    }
}
```

16.3.3 Customizing the Stack Canary Value

The Stack Smashing Protector implementation relies on a *secret* value that is placed on a protected function's call stack. This value, called the canary, is determined at compile time. Your application code can (and should) define an application-specific value. The value must be kept secret from any potential attacker.

```
extern unsigned long __stack_chk_guard;
void __attribute__((constructor, optimize("-fno-stack-protector")))
my_stack_guard_setup(void)
{
    __stack_chk_guard = 0x12345678; // the new secret value for the canary
}
```

- The custom setter function must be a constructor to ensure it is called before executing the application.
- The attribute `__optimize__ ("-fno-stack-protector")))` disable stack protector option for this function to avoid a runtime error in case `--fstack-protector-all` is used.

16.3.4 Stack Smash Protector at Work

Let’s see an example id when Stack Smash Protector demonstrates its usefulness:

```
extern unsigned long __stack_chk_guard;
void __attribute__((constructor, optimize("-fno-stack-protector")))
my_stack_guard_setup(void)
{
    __stack_chk_guard = 0xDEADBEEF; // the new value for the canary
}

void __attribute__((noreturn)) __stack_chk_fail(void)
{
    // handle the failure in a way that is appropriate for your application
    while(1)
    {
      // A software breakpoint is not likely to be an appropriate response to an attack!
      __builtin_software_breakpoint();
    }
}

void foo (char* ptr)
{
  int retval;
  char buffer[10];       // buffer located on the stack
  char *dest = buffer;

  // A buffer overrun here will cause a stack smash
  // The SSP feature helps you to identify & catch vulnerabilities in your application code.
  // It does not prevent them.
  while (*ptr != 0)
  {
      *dest++ = *ptr++;
  }

  return;
}

int main (void)
{
    // In a real application, this data may come from an external source,
    // making the application vulnerable to an attack.
    char string[] = "hello world!";

    foo(string);

    while(1);
}
```

The example is built successfully with `-fstack-protector` and debugged on a target device, execution should stop at the software breakpoint in the `__stack_chk_fail()` function.

16.4 Stack Guidance

Available with a PRO compiler license, the compiler's stack guidance feature can be used to estimate the maximum depth of any stack used by a program.

Runtime stack overflows cause program failure and can be difficult to track down, especially when the program is complex and interrupts are being used. The compiler's stack guidance feature constructs and analyzes the call graph of a program, determines the stack usage of each function, and produces a report, from which the depth of stacks used by the program can be inferred. Monitoring a program's stack usage during its development will mitigate the possibility of stack overflow situations.

This feature is enabled by the `-mchp-stack-usage` command-line option.

Once enabled, the operation of the stack guidance feature is fully automatic. For command-line execution of the compiler, a report will be displayed directly to the console after a successful build. When building in the MPLAB X IDE, this same report will be displayed in the build view in the **Output** window.

A more detailed and permanent record of the stack usage information will be available in the map file, should one be requested using the `-Wl,-Map=`*`mapfile`* command-line option or the equivalent control in the MPLAB X IDE project properties.

16.4.1 What is a Stack Overflow?

A stack overflow can occur if the stack pointer exceeds the RAM address range reserved for the call stack. When the application uses more space than is reserved for the stack, it can overwrite and corrupt other data such as statically allocated variables and the heap. In addition, when those other variables are accessed, they can corrupt values on the stack. This data corruption can be challenging to debug and leads to unpredictable runtime behavior of the application.

16.4.2 Estimating Stack Usage

In a bare-metal embedded application, the application developer must determine an appropriate minimum amount of data RAM to reserve for the stack and pass that value to the XC32 linker. The linker then uses this value to ensure that sufficient RAM is reserved for the stack.

However, due to reasons described later, the exact stack requirements of an application can be determined only at runtime. At link time, XC32 can use static analysis to ***estimate*** the maximum stack size required by the application and provide guidance in a human-readable report. It does this by computing the stack usage of each function and then using application's call graph to find the largest stack usage.

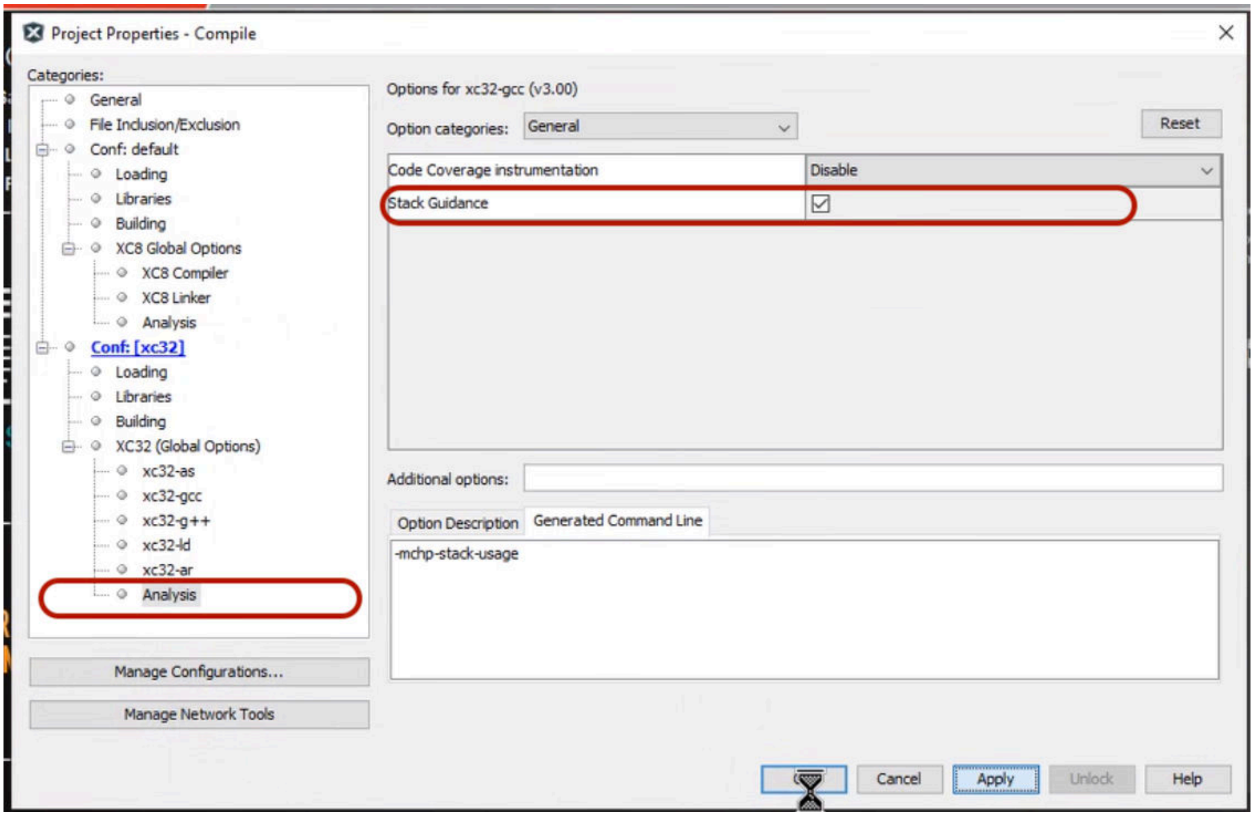
Generally speaking, this report cannot provide you with an exact value, but it provides you with information that you can use to determine an appropriate size for your stack reservation. *Only you understand the precise system-level, runtime behavior of your application.*

16.4.3 Enabling the Stack Usage Report

The feature can be used only when a PRO XC32 license is present. The stack-usage report is enabled through the `-mchp-stack-usage` command-line option, which is passed to XC32 when the Stack Guidance checkbox is enabled in the MPLAB X Project Properties.

To enable Stack Usage Analysis from MPLAB X, set Stack Guidance in *Project Properties> XC32>Analysis*

Figure 16-2. Project Properties for Stack Guidance



Options for xc32-gcc (v3.00)

Option categories: General

Code Coverage instrumentation	Disable
Stack Guidance	☑

Additional options:

Generated Command Line:

```
-mchp-stack-usage
```

or pass `-mchp-stack-usage` in command line.

```
xc32/<version>/bin/xc32-gcc -mprocessor=ATSAME70N20B -mchp-stack-usage -O0 test.c
```

16.4.4 Interpreting the Report

The stack-usage report is written:

- directly to standard error (either on terminal for command-line execution or in the MPLAB X output window)
- to the map file

The same information is available in both places, but the map file can serve as a record for later review.

The report contains:

- the largest stack usage that could be determined by static analysis
- a list of reasons that the maximum stack usage couldn't be determined, such as recursion or variable adjustment of the stack

- a list of disjoint and/or interrupt handler functions, which are not called directly by the main flow of the program

To determine the appropriate stack size for your application:

First, start with the initial value from the reset-handler call graph. In this example, the report shows 72 bytes required for the reset-handler call graph.

Stack Usage Report

```
============= STACK USAGE GUIDANCE =============
In the call graph beginning at Reset_Handler,
72 bytes of stack are required.
```

Second, add the listed stack allowances based on the system-level, runtime behavior of your application.

Stack Usage Report

```
However, the following cautions exists:
1. The following functions cannot be connected to the main call graph.
This is usually caused by some indirection:
frame_dummy uses 8 bytes
Dummy_Handler uses 0 bytes
__libc_init_array uses 40 bytes
You must add stack allowances for those functions.
================================================
```

Third, add stack allowances for your Interrupt handlers (interrupt service routines) as described next.

16.4.4.1 Cortex-M Interrupts

Stack Usage Report prints a list of the interrupt handlers defined in the application along with their stack consumption, so that the stack overhead generated by interrupts can be estimated.

In this context, use the `interrupt` function attribute to allow the compiler to identify and analyze the function as an interrupt handler and report it as such.

without __attribute__((interrupt))	with __attribute__((interrupt))

`1. The following functions cannot be connected to the main call graph.`
`This is usually caused by some indirection:`
`FstHandler uses 408 bytes`
`SndHandler uses 408 bytes`
	`1. The following functions are interrupt functions:`
	`FstHandler uses 408 bytes`
	`SndHandler uses 408 bytes`

Important:
When interrupts occur on a Cortex-M target, the stack estimate should be adjusted by adding the appropriate space for interrupt context saving as follows:

- with software floating-point support - 32 bytes (4 bytes * 7 caller saved registers: R0-R3, R12, LR, PC and the address of the next instruction to be execute when exiting an interrupt handler.
- with a hardware Floating-Point Unit (FPU) - 164 bytes (above 32 bytes + 4 bytes * 33 floating-point registers) - assuming Lazy Context Save is disabled.

Be sure to add the appropriate value to account for interrupt context saving when estimating stack usage. Interrupt context is saved by the core, not by the compiler.

16.4.4.2 Cortex-A Interrupts

When interrupts occur on a Cortex-A target, the stack guidance should be adjusted by 56 bytes (R0-R12 and PC x 4 bytes) in the worst-case scenario.

- It is highly recommended to use `interrupt` function attribute for interrupt handlers, as the Vector Table can be relocated and it is difficult to find ISRs based on addresses. Beside this, the Vector Table can be set/modified at runtime.
- If nested interrupts are enabled, the nesting cost should be considered.

16.4.5 Stack Usage Limitations

Static estimation of the maximum stack usage is inaccurate if the application contains functions falling in at least one of the following categories:

- functions for which the stack usage information is not available (not compiled/assembled with `-mchp-stack-usage`)
- functions containing indirect calls to other functions (the callees can not be identified)
- functions containing variable stack adjustments, inline assembly, or stack-usage information generated by the assembler (which might be inaccurate)
- recursive functions (direct or indirect) - only if at least one of the functions in the cycle has non-zero stack consumption

For each of these cases, a list of the involved functions is provided.

Be aware that the stack-usage estimation for assembly files may be inaccurate if your assembly code adjusts the stack. Be sure to take these adjustments into account when estimating your stack usage.

16.4.6 Stack Example

For this simple example:

test.c

```
int max(int a, int b){
        if (a < b)
                return b;
        return a;
}

int main() {
        int a[10], i, j;

        for (i =0; i < 10; i++)
                a[i] = i;
        for (i = 0; i < 10; i++)
                a[i] = max(a[i], a[10-i/2]);
        return a[0];
}
```

when built with

```
xc32-gcc -mprocessor=ATSAMD51N20A -mchp-stack-usage -O0 test.c
```

the stack-usage report shows

Stack Usage Report

```
============= STACK USAGE GUIDANCE =============
In the call graph beginning at Reset_Handler,
    72 bytes of stack are required.

However, the following cautions exists:

1. Indeterminate stack adjustment has been detected:
    _init uses 24 bytes
    _fini uses 24 bytes
No stack usage predictions can be made.

2. The following functions cannot be connected to the main call graph.
This is usually caused by some indirection:
    _init uses 24 bytes
    _fini uses 24 bytes
    __do_global_dtors_aux uses 8 bytes
    frame_dummy uses 8 bytes
    Dummy_Handler uses 0 bytes
You must add stack allowances for those functions.
================================================
```

17. 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.

17.1 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 functions is `static`. Interrupt functions are defined with the use of the `interrupt` attribute, see 17.2. 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 10.2.2. Static Variables. This specifier does not change the way the function is encoded.

17.2 Function Attributes And Specifiers

17.2.1 Function Attributes

The compiler keyword `__attribute__` allows you to specify special attributes of functions. 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., `__noreturn__` instead of `noreturn`). 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__ ((address(0x100), keep))
```

The attribute specifier can be placed either before or after the function's return type.

Note: It is important to use function attributes consistently throughout a project. For example, if a function is defined in file A with the an attribute, and declared `extern` in file B without the same attribute, then a link error might result.

`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`, `.bss` or `.ramfunc` sections. By not mapping these sections in the linker script, we allow these sections to be allocated using the best-fit allocator rather than the sequential allocator. Sections that are unmapped in the linker script can flow around absolute sections, whereas sections that are linker-script mapped are grouped together and allocated sequentially, potentially causing conflicts with absolute sections.

`alias ("`*`symbol`*`")`

Indicates that the function is an alias for another symbol. For example:

```
void foo (void) { /* stuff */ }
__attribute__ ((alias("foo"))) void bar (void);
```

In the above example, the 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.

`externally_visible`

This attribute when used with a function ensures the function remains visible outside the current compilation unit. This might prevent certain optimizations from being performed on the function.

`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, as described in 18. Interrupts. For Cortex-A devices, the *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. Any argument specifed is ignore when building for Cortex-M devices.

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.

nopa

Indicate to the compiler that the assembly code generated for this function should not be considered when performing procedural abstraction.

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(*optimization*)

The `optimize` attribute allows a function to be built with optimizations that differ to what has been specified on the command line and which will be applied to the rest of the program. The *optimization* argument can either be a number or a string. String arguments represent the command-line option used to control an optimization, so for example, to enable peephole optimizations (`-fpeephole`), use `optimize("peephole")`. The `-f` option prefix does not need to be specified with the argument. If you want to specify more than one optimization, separate the arguments with commas but with no space characters. Arguments that begin with `O` are assumed to be an optimization level option, for example `optimize("O1,unroll-loops")` turns on level 1 optimizations and the unroll-loops optimizations (controlled by the `-funroll-loops` command-line option). A numerical argument is also assumed to be an optimization level, for example optimize(3) turns on level 3 optimizations and is equivalent to the full usage of the attribute in the following example.

```
int __attribute__((optimize("O3"))) pandora (void) {
  if (maya > axton)
    return 1;
  return 0;
}
```

This feature can be used, for instance, to have frequently executed functions compiled with more aggressive optimization options that produce faster and larger code, while other functions can be called with less aggressive options. Typically, however, it is not used for production builds.

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.

ramfunc

The ramfunc attribute locates the attributed function in RAM rather than in Flash memory on devices that normally execute code from Flash.

The compiler's default runtime startup code uses the data-initialization template to copy the code associated with functions using this attribute from Flash to RAM at program startup. For example:

```
__attribute__((ramfunc))
unsigned int myramfunct(void)
{ /* code */ }
```

The <sys/attribs.h> header file provides a __ramfunc__ macro that is useful for this purpose. For example:

```
#include <sys/attribs.h>
__ramfunc__ unsigned int myramfunct(void)
{ /* code */ }
```

The branch/call instruction BL has a limited range (32MB for ARM instructions, 16MB for Thumb2 and 4MB for Thumb). As a result, RAM might not be within range of calls made from code in Flash. To allow such calls, additionally use the long_call attribute with the definition of the function located in RAM when it must be called from functions in Flash. For example:

```
__attribute__((ramfunc, long_call))
unsigned int myramfunct(void)
{ /* code */ }
```

The <sys/attribs.h> header file provides a __longramfunc__ macro that will specify both the ramfunc and long_call attributes. For example:

```
#include <sys/attribs.h>
__longramfunc__ unsigned int myramfunct(void)
{ /* code */ }
```

If a function in Flash is called by a function located in RAM, the Flash-based function definition must include the long_call attribute.

```
__attribute__ ((long_call))
unsigned int myflashfunct(void)
{ /* code */ }

__attribute__((ramfunc))
unsigned int myramfunct(void)
{
        return myflashfunct();
}
```

Having functions located in RAM rather than Flash can increase code size as well as startup time, as the attributed functions must be copied from Flash to RAM at startup. It may also have an impact on runtime performance, depending on your target device. Verify that your application meets your startup timing constraints. Also verify that the increased runtime performance meets your application's timing requirements and that your application does not have hidden dependencies on timing that is negatively impacted by this design selection.

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.

short_call

Always call the function using an absolute call instruction, even when the `-mlong-calls` command line option is specified.

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.

stack_protect

This attribute adds stack protection code to the function if one of the following options is set: `-fstack-protector`, `-fstack-protector-strong`, or `-fstack-protector-explicit`.

Optimizations can affect stack protection:

- Function inlining can affect whether a function is protected.
- Removal of an unused variable can prevent a function from being protected.

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 8.5. 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.

17.3 Allocation of Function Code

Code associated with C/C++ functions is normally placed in the program Flash memory of the target device. Functions may be alternatively located in and executed from RAM rather than Flash by using the __ramfunc__ or __longramfunc__ macros or the ram_func and long_call attributes that these macros represent. See 17.2.1. Function Attributes for more information.

Another alternative to Flash memory is to place functions in Tightly Coupled Memory on those devices if available. See 8.5. Tightly-Coupled Memories for information on this memory.

17.4 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 9.11. 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 9.11. Variable Attributes.

To place executable code into the eXecute-Only Memory (XOM) memory region, use a custom linker script to map the text input sections to a new output section. Next, map the output section to the desired XOM region. This must be performed in conjunction with the -mpure-code option (see 6.7.1. Options Specific to PIC32C/SAM Devices). This option ensures that text sections do not contain read-only data, which would be inaccessible if such data was present in XOM, and places a special flag on text sections to indicate that they contain purely code and no data. The use of XOM can help protect firmware from being stolen or reverse engineered by third parties. For example, add the following to your linker script.

```
SECTIONS
{
 .purecode :
 {
 INPUT_SECTION_FLAGS (SHF_ARM_PURECODE) *(.text .text.*)
 } > purecode_memory
}
```

This change will ensure that input sections whose name matches .text or .text.* and which are marked with the SHF_ARM_PURCODE section flag (set by the -mpure-code option), will be placed in the .purecode output section. That .purecode output section will be mapped to a memory region named purecode_memory, which must have previously been defined using the MEMORY command to associate it with the desired XOM locations on your target device. For more information on linker scripts, see the *MPLAB XC32 Assembler, Linker, and Utilities User's Guide* (DS50002186).

17.5 Function Size Limits

There are no theoretical limits as to how large functions can be made.

17.6 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 4-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 `R0–R3` (see the table below for how many registers are required for each data type).
- When calling a function:

  Registers `R0–R3` are used for passing arguments to functions. Values in these registers are not preserved across function calls.

  Registers `R0–R3`, `R12`, `R14`, and `R15` are caller-saved registers. The calling function must push these values onto the stack for the registers' values to be saved.

  Registers `R4–R7` are callee-saved registers. The function being called must save any of these registers it modifies.

  Register `R11` is a saved register if the optimizer eliminates its use as the Frame Pointer. This is a reserved register otherwise.

  Register `R14` contains the return address of a function call.

Table 17-1. Registers Required

Data Type	Number of Registers Required
`char`	1
`short`	1
`int`	1
`long`	1
`long long`	2
`float`	1
`double`	2
`long double`	2
`structure`	Up to 4, depending on the size of the struct.

17.7 Function Return Values

Function return values are returned in registers.

Integral or pointer value are placed in register `R0`, extending up to `R3` is necessary. This is true also for floating-point values.

If a function needs to return an aggregate value that is too large to fit in the return registers, it is placed in stack space allocated by the caller.

17.8 Calling Functions

By default, functions are called using the direct form of the call (`bl`) instruction. 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. Long calls are typically used when calling between memory regions, such as calling a function located in RAM from a function located in Flash and vice versa.

The -mlong-calls option, see 6.7.1. 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 long_call attribute can be used with a function definition to always enforce the longer call sequence for that particular function. The short_call 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.

17.9 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.

18. 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 interrupts occur, the normal flow of software execution is suspended, and special functions are invoked to process the event. At the completion of interrupt processing, previous context information is restored and normal execution resumes.

PIC32 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.

18.1 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 usually the main part of the program that executes after Reset.

18.2 Writing an Interrupt Service Routine

An interrupt service routine takes no arguments and returns no results, that is, its argument list is `void`, as is its return type. This pattern is not enforced but executing an ISR that does not follow it will result in unpredictable behavior.

On Cortex-M cores, the hardware takes care of preserving and restoring context. 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. Regardless, a function should still be marked as an interrupt handler function using the `interrupt` attribute so that the compiler knows that it is used. When seeing 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.

When targeting Cortex-A devices, the compiler must generate application code to save and restore the device context. The function acting as the interrupt handler, therefore, must use the `interrupt` attribute so that the compiler can ensure the correct context switch is generated.

See the appropriate architecture reference manual for full details on that device's exception entry and exit behavior.

18.2.1 Interrupt Attribute

`__attribute__((interrupt))`

The `interrupt` attribute tells the compiler that the function is an interrupt handler. Use of this attribute is optional when building interrupt handlers for Cortex-M devices, but it is recommended to ensure that the compiler generates code to align the stack pointer on an 8 byte boundary upon function entry. This attribute is mandatory when writing interrupt handlers for Cortex-A devices, as the compiler must generate context switch code for interrupt handlers on such devices.

Note: The `interrupt` attribute can take arguments, as described in the GCC manual, but are ignored when targeting devices with a Cortex-M core.

18.3 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/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 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 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.
}
```

18.4 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.

18.4.1 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 19. Main, Runtime Start-up and Reset for details.

18.4.2 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;
```

18.4.3 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.

18.4.4 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.

18.4.5 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;
```

18.4.6 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;
```

18.5 Interrupt Service Routine Context Switching

When using Cortex-M devices, 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 when the interrupt occurs and returns. Other registers must be preserved by the interrupt handler function, as with any other function. The compiler will do this automatically for C functions. If you write your own handler in assembly, be sure to follow the procedure call standard.

It is possible for other exceptions to occur during the context switch. The work for this is handled in hardware. Details of this can be found in the appropriate architecture reference manual, but in essence, the context switching does not occur twice. Instead, the higher priority interrupt is run with the other interrupt set to pending.

When targeting Cortex-A devices, the compiler must generate application code to save and restore the device context; however the Floating-point Unit (FPU) registers are never saved by interrupt handlers. Applications on these devices that need to use the FPU from an interrupt context require bespoke interrupt handlers. Consider writing the interrupt handler in assembly code, having them push all the registers that need to be saved, call the function for the handler, restore the registers, and then return. This assembly wrapper function would need to be applied to every interrupt handler (or at least those that use FPU registers). In this case, C functions should not set the `interrupt` attribute. Registers r0-r12, r14, d0-d7, d16-d32, and the status registers must be saved if they have been used in the handler. This is potentially a large number of registers that need to be saved, which would require a significant amount of stack space to have them preserved and a significant amount of time for that preservation code to execute.

18.6 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 Cortex-M 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.

On Cortex-A devices, the time taken by context saving is dependent on the number of registers saved on entry, which is determined by the code making up the function. This time has to be determined on a case-by-case basis, if required.

To determine the value of the interrupt latency, see the data sheet for your device and the appropriate technical reference manual.

18.7 Enabling/Disabling Interrupts

The following functions from the CMSIS API are used to manipulate the interrupt state of the CPU:

```
__enable_irq()
```

```
__disable_irq()
```

When using a Cortex-M device, the Nested Vector Interrupt Controller (NVIC), which controls the aspects of nearly all internal and external interrupts, can be manipulated using the CMSIS API. See the *Interrupts and Exceptions (NVIC)* reference section of the API for more information. The NVIC API functions are made available when you include the `<xc.h>` header.

With Cortex-A devices, the CMSIS API can be used to control aspects of interrupts. See the *Interrupts and Exceptions* reference section. These functions are made available when you include the `<xc.h>` header.

18.8 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. Additionally, avoid using the hardware floating-point unit (FPU) from an interrupt context, as its use will increase interrupt latency, since additional FPU registers will need to be preserved.

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.

If you are building with link-time optimizations (the `-flto` option), you might need to take special steps to ensure that code associated with interrupts is not removed. These optimizations for the most part work on the whole program. When the whole program is analyzed, interrupt functions will be found to be not called by any other function, so the compiler believes it can remove them. To prevent this from occurring, interrupt functions must be marked with the `used` attribute to inform the compiler that they are not redundant. The same attribute should be used with data objects that are used by interrupt functions. As an alternative, consider building source files containing interrupt functions with link-time optimizations disabled.

19. 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.

19.1 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.

19.2 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. For Cortex-M devices, ensure the stack pointer register is initialized to point to the top of the system stack; for Cortex-A devices, ensure the stack pointer registers are initialized to point to the top of the appropriate 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. Initialize the data, bss, and ramfunc sections using the linker-generated data-initialization template.
6. Configure the instruction and/or data Tightly-Coupled Memory (TCM), if present and enabled. Also see 8.5. Tightly-Coupled Memories.
7. 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.
8. If requested, relocate the stack to data TCM.
9. For Cortex-M devices only, initialize the `VTOR` (Vector Table on Reset) register to the address of the interrupt vector table.
10. Perform the initialization for the standard C library.
11. Call the function `void _on_bootstrap()` if it is defined by the application.
12. Call the application `main()` function.
13. On return from `main()`, enter an infinite loop.

Each step will be described in further detail in the following sections.

19.2.1 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 19.2.7. Relocate Stack to TCM and 8.5. Tightly-Coupled Memories.

Although the stack pointer is initialized by hardware on Cortex-M devices, it can be initialized from the runtime startup code in cases where the application is bootloaded by other code. Note in the following example that the code is guarded by the preprocessor macro `__REINIT_STACK_POINTER`.

```
#if defined (__REINIT_STACK_POINTER)
  /* Initialize SP from linker-defined _stack symbol. */
  __asm__ volatile ("ldr sp, =_stack" : : : "sp");
#ifdef SCB_VTOR_TBLOFF_Msk
  /* Buy stack for locals */
  __asm__ volatile ("sub sp, sp, #8" : : : "sp");
 #endif
  __asm__ volatile ("add r7, sp, #0" : : : "r7");
 #endif
```

For Cortex-A devices, the runtime startup code sets the stack for the following modes: FIQ, IRQ, Abort, Undefined, System, and Supervisor. Once finished, the system is in Supervisor mode.

19.2.2 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.

19.2.3 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.

19.2.4 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.

19.2.5 Initialize Objects and RAM Functions

Objects accessed from RAM and functions executed from RAM must have their allocated RAM initialized before execution of the `main()` function can commence, and this is a significant task performed by the runtime startup code.

Those non-`auto` objects that are uninitialized must be cleared (assigned the value 0) before execution of `main()` begins. Such objects are not assigned a value in their definition, for example `output` in the following example:

```
int output;
int main(void) { ...
```

There is only one `.bss` section for PIC32C/SAM devices, which contains all uninitialized objects.

Another task of the runtime start-up code is to ensure that any initialized objects contain their initial value before the program begins execution. Initialized objects are those that are not `auto` objects and that are assigned an initial value in their definition, for example `input` in the following example:

```
int input = 0x88;
int main(void) { ...
```

Such initialized objects have two components: their initial value (0x0088 in the above example) stored in program memory (that is, placed in the HEX file), and space reserved in RAM, where the objects will reside and be accessed during program execution (runtime).

There is only one `.data` section, which contains all initialized objects.

The runtime start-up code will copy all the blocks of initial values from program memory to RAM so the objects will contain the correct values before `main()` is executed.

Since `auto` objects are dynamically created, they require code to be positioned in the function in which they are defined to perform their initialization. It is possible that the initial value of an `auto` object may change on each instance of the function and so the initial values cannot be stored in program memory and copied. As a result, initialized `auto` objects are not considered by the runtime start-up code, but are instead initialized by assembly code in each function output.

Note: Initialized `auto` objects can impact code performance, particularly if the objects are large in size. Consider using global or `static` objects instead.

Objects whose contents should be preserved over a Reset should be qualified with the `persistent` attribute, see 9.9. Standard Type Qualifiers. Such objects are linked at a different area of memory and are not altered by the runtime start-up code in any way.

Any functions that use the `ramfunc` attribute are copied from program memory to RAM before they are executed. This is also performed by the runtime startup code, in much the same way that initialized objects have their initial value copied before they are accessed.

19.2.5.1 Data-initialization Template

In order to clear or initialize all the data and RAM function sections, the linker creates a data-initialization template, which is loaded into an output section named `.dinit` and allocated space in program memory. The code in the C/C++ start-up module contained in `libpic32c.a` interprets this template, which indicates how the appropriate sections must be initialized.

The sections initialized by this template includes those holding intialized objects (such as the `.data` section) as well as sections containing `ramfunc` attributed functions, all of which must have values copied from the template in program to data memory where the objects and functions will be accessed at runtime. Other data sections holding unititialized objects (such as the `.bss` section) are cleared by the template before the `main()` function is called. The persistent data section (`.pbss`) is not considered by the runtime startup code. When the application's main program takes control, all objects and RAM functions in data memory will have been initialized.

The data initialization template contains one record for each output section that needs initializing. Each record has one of several formats, represented by a format code within the record. The record formats specify how the data values are stored in the record itself and how they should be used to initialize the corresponding section. The `--dinit-compress` option (see 6.7.10.1. Dinit-compress Option) controls which of these records can be utilized by the template. The numerical format codes and the type of initialization they represent are as follows:

#0 Fill the RAM defined by the corresponding output section with zeros. No data bytes are stored in the record. Used by bss sections.

#1 Copy each byte of data from the record's array to the RAM associated with the output section. Used by data sections, and sections associated with `ramfunc` functions.

#2 Copy the same 16-bit value into the RAM associated with the output section multiples times. Used by data sections whose initial values are a repeating sequence.

#3 Copy the same 32-bit value into the RAM associated with the output section multiple times. Used by data sections whose initial values are a repeating sequence.

#4 Copy and decompress a simplified version of PackBits encoded data_record. Used by data sections, and sections associated with `ramfunc` functions which contain large numbers of consecutive zero bytes.

The data contained in each record type can be represented by the equivalent C structures that are presented below. The first element of the record is a pointer to the section in data memory. The second element is the section length or repeat count. The third element is the format code, which indicates the type of the record (listed above) and hence how the corresponding section should be initialized. The forth element is used for either alignment padding or an initial value. A fifth element, if present, is an array of data bytes.The template is terminated by two null instruction words.

```
/* For format values of 0 */
struct data_record_bss {
    uint32_t *dst;     /* destination address */
    uint32_t len;      /* length in bytes */
    uint16_t format;   /* format code */
    uint16_t padding;  /* padding for alignment */
};

/* For format values of 1 and 3 (also identical to format value 4) */
struct data_record_standard {
    uint32_t *dst;     /* destination address */
    uint32_t len;      /* length in bytes */
    uint16_t format;   /* format code */
    uint16_t padding;  /* padding for alignment or a 16-bit initialization value */
    uint32_t dat[0];   /* object-length data - holding initialization data */
};

/* For format values of 2 - objects are initialised with the same 16-bit value */
struct data_record_short_standard {
    uint32_t *dst;     /* destination address */
    uint32_t count;    /* count in bytes */
    uint16_t format;   /* format code */
    uint16_t dat       /* 16-bit repeated value data */
};

/* For format values of 4 - A simplified PackBits data compression is applied, where
each run of zeros is replaced by two 8-bit characters in the compressed array:
zero followed by the number of zeros in the original run. */
struct data_record_compressed {
    uint32_t *dst;               /* destination address */
    uint32_t count;              /* count in bytes */
    uint16_t format;             /* format code */
    uint16_t padding;            /* 16-bit repeated value data */
    uint32_t compressed_data[0]; /* compressed intialized data */
};
```

19.2.6 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 8.5. Tightly-Coupled Memories.

19.2.7 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.

19.2.8 Set VTOR Register

Relevant for Cortex-M devices only, the `VTOR`, or Vector Table Offset Register, 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. Cortex-A devices have no special initialization requirements for the exception handlers.

19.2.9 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.

19.2.10 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.

19.2.11 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()`.

19.2.12 Exception Handlers

An exception table is defined by the compiler. For devices based on Arm Cortex-M cores, that table will contain the initial stack pointer and program start address (e.g. the `Reset_Handler()` function address) as well as the interrupt service routine (ISR) vector. A similar table is created when using Cortex-A devices, only the stack pointer is not defined. This table is placed in the `.vectors` section.

The device-specific start-up code defines a default vector table `exception_table`, as well as a default ISR named `Dummy_Handler()`. When using Cortex-M devices, all pointers in `exception_table` apart from the `Reset_Handler()`, are initialized to point to `Dummy_Handler()`, which simply enters an infinite loop. Not all handlers default to `Dummy_Handler()` when using Cortex-A devices, and additionally, the FIQ and IRQ interrupt handlers might have device-specific implementations. 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.

20. Libraries

The MPLAB XC32 C/C++ Compiler provides C90- and C99-validated libraries of functions, macros, types, and objects that can assist with your code development.

The Standard C libraries are described in the separate *Microchip Unified Standard Library Reference Guide* document, whose content is relevant for all MPLAB XC C compilers.

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.

In addition to the libraries supplied with the compiler, you can create your own libraries from source code you have written.

20.1 Smart IO Routines

The library code associated with the print and scan families of IO functions can be customized by the compiler with each compilation, based on compiler options and how you use these functions in your project. This can reduce the amount of redundant library code being linked into the program image, hence can reduce the program memory and data memory used by a program.

The smart output (print family) functions are:

`printf`	`fprintf`	`snprintf`	`sprintf`
`vfprintf`	`vprintf`	`vsnprintf`	`vsprintf`

The smart input (scan family) functions are:

`scanf`	`fscanf`	`sscanf`
`vfscanf`	`vscanf`	`vsscanf`

When this feature is enabled, the compiler analyzes your project's C source code every time you build, searching for calls to any of the smart IO functions. The conversion specifications present in the format strings are collated across all calls, and their presence triggers inclusion of library routines with the associated functionality in the program image output.

For example, if a program contained only the following call:

```
printf("input is: %d\n", input);
```

when smart IO is enabled, the compiler will note that only the `%d` placeholder has been used by the `printf` function in the program, and the linked library routine defining `printf` will thus contain a basic functionality that can at least handle the printing of decimal integers. If the following call was added to the program:

```
printf("input is: %f\n", ratio);
```

the compiler will then see that both the `%d` and `%f` placeholders were used by `printf`. The linked library routine would then have additional functionality to ensure that all the requirements of the program can be met.

Specific details of how the smart IO feature operates for this compiler are detailed in the following section. The syntax and usage of all IO functions, which are part of the `<stdio.h>` header, are described in the *Microchip Unified Standard Library Reference Guide*.

20.1.1 Smart IO For PIC32C/SAM Devices

When using MPLAB XC32 C/C++ Compiler, multiple IO library variants, representing increasingly complex subsets of IO functionality, are available and are linked into your program based on the `-msmart-io` option and how you use the smart IO functions in your project's source code.

When the smart IO feature is disabled (`-msmart-io=0`), a full implementation of the IO functions will be linked into your program. All features of the IO library functions will be available, and these may consume a significant amount of the available program and data memory on the target device.

When the smart IO feature is enabled (`-msmart-io=1` or `-msmart-io`), the compiler will link in the least complex variant of the IO library that implements all of the IO functionality required by the program, based on the conversion specifications detected in the program's IO function format strings. This can substantially reduce the memory requirements of your program, especially if you can eliminate in your program the use of floating-point features in calls to smart IO functions. This is the default setting.

The compiler analyzes the usage of each IO function independently, so while the code for a particular program might require that the `printf` function be full featured, only a basic implementation of the `snprintf` function might be required, for example.

If the format string in a call to an IO function is not a string literal, the compiler will not be able to detect the exact usage of the IO function, and a full-featured variant of the IO library will be linked into the program image, even with smart IO enabled. In this instance, the `-msmart-io=2` form of the option can be used. This has the compiler assume that no floating-point has been used by formatted IO functions and that it is safe to link in integer-only format IO libraries. You must ensure that your program only uses the indicated conversion specifications; otherwise, IO functions may not work as expected.

For example, consider the following four calls to smart IO functions.

```
vscanf("%d:%li", va_list1);
vprintf("%-s%d", va_list2);
vprintf(fmt1, va_list3); // ambiguous usage
vscanf(fmt2, va_list4);  // ambiguous usage
```

When processing the last two calls, the compiler cannot deduce any usage information from either of the format strings. If it is known that the format strings pointed to by `fmt1` and `fmt2` collectively use only the `%d`, `%i` and `%s` conversion specifiers, the `-msmart-io=2` form of the option can be used.

These options should be used consistently across all program modules to ensure an optimal selection of the library routines included in the program image.

20.2 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.

20.3 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.

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 C++ library 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.

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 Harmony web site (www.microchip.com/mplab/mplab-harmony) to download MPLAB Harmony.

21. 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.

21.1 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 15.2. 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.
- The table in 15.2. Register Conventions describes which registers must be saved across non-interrupt function calls.
- Interrupt functions must preserve all registers. Unlike a normal function call, an interrupt may occur at any point during the execution of a program. When returning to the normal program, all registers must be as they were before the interrupt occurred.
- Variables or functions declared within a separate assembly file that will be referenced by any C source file should be declared as global using the assembler directive `.global`. Undeclared symbols used in assembly files will be treated as externally defined.

The following example shows how to use variables and functions in both assembly language and C regardless of where they were originally defined.

The file `ex1.c` defines `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 21-1. 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.

21.2 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 the following tables.

Table 21-1. Register Constraint Letters Supported by the Compiler

Letter	Constraint
r	Any general register
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 21-2. 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 21-3. 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

21.2.1 Examples:

- Insert Bit-field
- Multiple Assembler Instructions

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
```

Multiple Assembler Instructions

This example demonstrates how tot 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
```

21.2.2 Equivalent Assembly Symbols

C/C++ symbols can be accessed directly with no modification in extended assembly code.

21.3 Predefined Macro

There is one predefined macro available once you include `<xc.h>`. It is `_nop()`. This macro inserts a `nop` instruction.

22. Optimizations

Activation of a compiler license controls which code optimizations are available.

An unlicensed compiler operating in Free mode allows access to only basic optimizations. You can purchase a PRO compiler license at any time. Activating the compiler with a PRO license unlocks all available speed- and space-orientated optimizations. Prior to purchase, you can, if desired, obtain and activate a free 60-day PRO license evaluation, which also permits full optimization of source code and shows the benefits that a PRO license would deliver.

Visit www.microchip.com/mplab/compilers for more information on C and C++ licenses.

MPLAB XC32 C/C++ Compiler license types are Free, EVAL and PRO. The initial compiler download begins as an Evaluation (EVAL) license allows 60 days to evaluate the compiler as a Professional (PRO) license with the most optimizations. The Free license has minimal 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 www.microchip.com/mplab/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.

22.1 Optimization Feature Summary

Licensing your compiler entitles you to optimizations that are not available with the Free product. Those optimizations available with the Free and Licensed XC32 compiler are tabulated below. The optimization names are derived from the option that can typically be used to enable or disable them, for example the "Defer pop" optimization can be manually enabled using the `-fdefer-pop` option if it is not already enabled at the selected optimization level, or disabled using `-fno-defer-pop`. For details on compiler options used to set optimizations, see 6.7.7. Options for Controlling Optimization.

Table 22-1. License Optimization Features

Free	PRO License
• Align functions • Align labels • Align loops • Caller saves • Cse follow jumps • Cse skip blocks • Data sections • Defer pop • Expensive optimizations • Function cse • Function sections • Gcse • Gsce lm • Gsce sm • Inline • Inline functions • Inline limit • Keep inline functions • Keep static consts • Omit frame pointer • Optimize sibling calls • Peephole/2 • Rename registers • Rerun cse after loop • Rerun loop opt • Schedule insns/2 • Strength reduce • Strict aliasing • Thread jumps • Toplevel reorder • Unroll/all loops	All Free optimizations, plus: • Lto • Pa

23. 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 6.7.8. Options for Controlling the Preprocessor.

23.1 Preprocessor Directives

The XC32 accepts several specialized preprocessor directives, in addition to the standard directives. All of these are tabulated below.

Table 23-1. 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 filename for listing	`#line 3 final`
#*nn* filename	(where *nn* is a number, and *filename* is the name of the source file) the following content originated from the specified file and line number.	`#20 init.c`
`#pragma`	Compiler specific options.	See the Pragma Directives section in this guide.
`#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; 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 can require further expansion. Remember, that once a macro identifier has been expanded, it will not be expanded again if it appears after concatenation.

23.1.1 Preprocessor Arithmetic

Preprocessor macro replacement expressions are textual and do not utilize types. Unless they are part of the controlling expression to the inclusion directives (discussed below), macros are not evaluated by the preprocessor. Once macros have been textually expanded and preprocessing is complete, the expansion forms a C expression which is evaluated by the code generator along with other C code. Tokens within the expanded C expression inherit a type, with values then subject to integral promotion and type conversion in the usual way.

If a macro is part of the controlling expression to a conditional inclusion directive (#if or #elif), the macro must be evaluated by the preprocessor. The result of this evaluation is often different to the C-domain result for the same sequence. The preprocessor assigns sizes to literal values in the controlling expression that are equal to the largest integer size accepted by the compiler, as specified by the size of intmax_t defined in <stdint.h>.

For the MPLAB XC32 C/C++ Compiler, this size is 64 bits.

23.2 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
```

23.3 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.

23.3.1 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.

23.3.2 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" ...)`

The `#pragma GCC optimize ("`*`string`*`"...)` pragma sets default optimization attributes for function declarations and definitions that do not otherwise specify these attributes. All functions after the pragma will be optimized accordingly. The parentheses are optional. 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., `-O`*`string`*
- Otherwise, *`string`* should be an option that can be used with a `-f` prefix.

The `#pragma GCC reset_options` pragma clears the default optimizations, so that the optimization of subsequent functions is not controlled by `optimize` pragma.

`#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.

23.3.3 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 8.3. Configuration Bit Access for a description of the syntax for the `config` options.

`#pragma default_function_attributes`

The `#pragma default_function_attributes = [@ "`*`section`*`"]` pragma sets default section placement attributes for function declarations and definitions that do not otherwise specify this attribute. All functions after the pragma will be placed in the section whose name is quoted after the `@` token. Using the `#pragma default_function_attributes =` form of this pragma will reset the section specification, so that subsequent functions are not placed in any special section.

`#pragma default_variable_attributes`

The `#pragma default_variable_attributes = [@ "`*`section`*`"]` pragma sets default section placement attributes for variable declarations and definitions that do not otherwise specify this attribute. All variables after the pragma will be placed in the section whose name is quoted after the `@` token. Using the `#pragma default_variable_attributes =` form of this pragma will reset the section specification, so that subsequent variables are not placed in any special section.

23.4 Predefined Macros

The compiler provides a number of macro definitions that characterize the various target-specific options and other aspects of the compiler and host environment. Some of these are listed in the table below.

See also the device-specific include files (`pic32c/include/proc/`*`family`*`/`*`device`*`.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.

You can also run the compiler with the `-E` and `-dM` options to have it print out those macros defined for that particular project and set of compiler options.

Table 23-2. Predefined Macro Definitions

Macro	Meaning
`__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, when using `-mprocessor=32CX0525SG12144`, the compiler will define `__32CX05255SG12144__`.
`__XC`	Always defined to indicate that this is a Microchip XC compiler.
`__XC32`	Always defined to indicate this the XC32 compiler.
`__VERSION__`	The `__VERSION__` macro expands to a string constant describing the compiler in use. Do not rely on its contents having any particular form, but it should contain at least the release number. Use the `__XC32_VERSION` macro for a numeric version number.

...........continued

Macro	Meaning
`__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.
`__XC32_DTCM_LENGTH`	Defined to be the size (in bytes) of the data tightly coupled memory specified by the `-mdtcm` option.
`__XC32_ITCM_LENGTH`	Defined to be the size (in bytes) of the instruction tightly coupled memory specified by the `-mitcm` option.
`__XC32_TCM_LENGTH`	Defined to be the size (in bytes) of the combined tightly coupled memory specified by the `-mtcm` option.

24. 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.

24.1 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 9.11. 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.

24.2 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 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.

25. Implementation-Defined Behavior

25.1 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.

25.2 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.

25.3 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.

25.4 Identifiers

ISO Standard:	“Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).”
Implementation:	None.

ISO Standard:	“The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).”
Implementation:	All characters are significant.

25.5 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.

25.6 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.

25.7 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.

25.8 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.

25.9 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.

25.10 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.

25.11 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 */`

25.12 Declarators

ISO Standard:	“The maximum number of declarators that may modify an arithmetic, structure or union type (C90 6.5.4).”
Implementation:	No limit.

25.13 Statements

ISO Standard:	“The maximum number of case values in a switch statement (C90 6.6.4.2).”
Implementation:	No limit.

25.14 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 9.11. 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.

25.15 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`, `strfxtime`, `wcsftime`, and `wcsfxtime` functions in the `“C”` locale (7.23.3.5, 7.23.3.6, 7.24.5.1, 7.24.5.2).”
Implementation:	Unimplemented.
ISO Standard:	“Whether or when the trigonometric, hyperbolic, base-*e* exponential, base-*e* logarithmic, error, and log gamma functions raise the inexact exception in an IEC 60559 conformant implementation (F.9).”
Implementation:	No.
ISO Standard:	“Whether the inexact exception may be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (F.9).”

Implementation:	No.
ISO Standard:	“Whether the underflow (and inexact) exception may be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (F.9).”
Implementation:	No.
ISO Standard:	“Whether the functions honor the Rounding Direction mode (F.9).”
Implementation:	The Rounding mode is not forced.

25.16 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 9.3.2. 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 9.2. 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 9.2. Data Representation.

26. 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.

26.1 Built-in Function Descriptions

This section describes the programmer interface to the compiler built-in functions. Since the functions are "built in", there are no header files associated with them. Similarly, there are no command-line switches associated with the built-in functions – they are always available. The built-in function names are chosen such that they belong to the compiler's namespace (they all have the prefix `__builtin_`), so they will not conflict with function or variable names in the programmer's namespace.

26.1.1 void __builtin_nop(void)

Emit a no-operation instruction.

Prototype

`void __builtin_nop(void);`

Argument

None.

Return Value

None.

Assembler Operator/ Machine Instruction

`nop`

Error Messages

None.

26.1.2 __builtin_software_breakpoint Built-in Function

Insert a software breakpoint.

Note that the `__conditional_software_breakpoint()` macro defined in `<assert.h>` provides a lightweight variant of `assert(exp)` that causes only a software breakpoint when the assertion fails rather than printing a message. This macro is disabled if, at the moment of including `<assert.h>`, a macro with the name `NDEBUG` has already been defined of if a macro with the name `__DEBUG` has not been defined. For example:

```
__conditional_software_breakpoint(myPtr!=NULL);
```

Prototype

```
void __builtin_software_breakpoint(void)
```

Argument

None.

Return Value

None.

Assembler Operator/ Machine Instruction

```
bkpt
```

Error Messages

None.

27. ASCII Character Set

Table 27-1. 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

28. Document Revision History

28.1 Document Revision History

Revision A (June 2019)

- Initial revision of the document.

Revision B (October 2021)

- Added code coverage information
- Added stack guidance
- Reformatted and renumbered

Revision C (April 2023)

- Updated library information to reflect recent changes, including the Microchip Unified Standard Library
- Corrected examples of using the `__pack` specifier
- Removed 'How To' on the cause or reset, which was not appropriate for PIC32C/SAM devices
- Updated the size of floating-point types to reflect recent compiler changes
- Updated information on how to ensure a function is not removed
- Updated information relating to using a debugger
- Updated information on the compiler option controls available in the MPLAB X IDE
- Updated information on the runtime startup code, including information on the data initialization template
- Provided dedicated sections on all compiler options in addition to a summary table
- Added new information on `-mdfp`, `-mpure-code`, `-feliminate-unused-debug-symbols`, `-Og`, and `--dinit-compress` options
- Some options and description were not relevant for PIC32/SAM devices and were removed
- Added divergences from the C99 language standard
- Expanded configuration bit access information
- Improved information on tightly coupled memories
- Added `used` variable attribute
- Added `externally_visible` and `nopa` function attributes
- Added information on placing code in eXecute Only Memory (XOM)
- Updated information on interrupt context switch
- Improved information on smart IO features
- Many general corrections and improvements

The Microchip Website

Microchip provides online support via our website at www.microchip.com/. This website is used to make files and information easily available to customers. Some of the content available includes:

- **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 design partner 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

Product Change Notification Service

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

To register, go to www.microchip.com/pcn and follow the registration instructions.

Customer Support

Users of Microchip products can receive assistance through several channels:

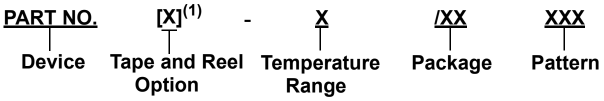
- Distributor or Representative
- Local Sales Office
- Embedded Solutions Engineer (ESE)
- Technical Support

Customers should contact their distributor, representative or ESE for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in this document.

Technical support is available through the website at: www.microchip.com/support

Product Identification System

To order or obtain information, e.g., on pricing or delivery, refer to the factory or the listed sales office.



Device:	PIC16F18313, PIC16LF18313, PIC16F18323, PIC16LF18323	
Tape and Reel Option:	Blank	= Standard packaging (tube or tray)
	T	= Tape and Reel[1]
Temperature Range:	I	= -40°C to +85°C (Industrial)
	E	= -40°C to +125°C (Extended)
Package:[2]	JQ	= UQFN
	P	= PDIP
	ST	= TSSOP
	SL	= SOIC-14
	SN	= SOIC-8
	RF	= UDFN
Pattern:	QTP, SQTP, Code or Special Requirements (blank otherwise)	

Examples:

- PIC16LF18313- I/P Industrial temperature, PDIP package
- PIC16F18313- E/SS Extended temperature, SSOP package

Notes:

1. Tape and Reel identifier only appears in the catalog part number description. This identifier is used for ordering purposes and is not printed on the device package. Check with your Microchip Sales Office for package availability with the Tape and Reel option.
2. Small form-factor packaging options may be available. Please check www.microchip.com/packaging for small-form factor package availability, or contact your local Sales Office.

Microchip Devices Code Protection Feature

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

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

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

Legal Notice

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

Trademarks

The Microchip name and logo, the Microchip logo, Adaptec, AnyRate, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, CryptoMemory, CryptoRF, dsPIC, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

AgileSwitch, APT, ClockWorks, The Embedded Control Solutions Company, EtherSynch, Flashtec, Hyper Speed Control, HyperLight Load, IntelliMOS, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet- Wire, SmartFusion, SyncWorld, Temux, TimeCesium, TimeHub, TimePictra, TimeProvider, TrueTime, WinPath, and ZL are registered trademarks of Microchip Technology Incorporated in the U.S.A.

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, Augmented Switching, BlueSky, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, Espresso T1S, EtherGREEN, GridTime, IdealBridge, In-Circuit Serial Programming, ICSP, INICnet, Intelligent Paralleling, Inter-Chip Connectivity, JitterBlocker, Knob-on-Display, maxCrypto, maxView, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, NVM Express, NVMe, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, RTAX, RTG4, SAM-ICE, Serial Quad I/O, simpleMAP, SimpliPHY, SmartBuffer, SmartHLS, SMART-I.S., storClad, SQI, SuperSwitcher, SuperSwitcher II, Switchtec, SynchroPHY, Total Endurance, TSHARC, USBCheck, VariSense, VectorBlox, VeriPHY, ViewSpan, WiperLock, XpressConnect, and ZENA are trademarks of Microchip Technology Incorporated in the U.S.A. and other countries.

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

The Adaptec logo, Frequency on Demand, Silicon Storage Technology, Symmcom, and Trusted Time are registered trademarks of Microchip Technology Inc. in other countries.

GestIC is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries.

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

ISBN: 978-1-6683-2352-6

Quality Management System

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

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:
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: 45-4485-5910
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: 47-72884388

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