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

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

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

Contact us

Tel: +86-755-8981 8866 Fax: +86-755-8427 6832
Email & Skype: info@chipsmall.com Web: www.chipsmall.com
Address: A1208, Overseas Decoration Building, #122 Zhenhua RD., Futian, Shenzhen, China

MPLAB C32 C COMPILER USER'S GUIDE

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Table of Contents

MPLAB® C32 C COMPILER USER'S GUIDE

Preface

NOTICE TO CUSTOMERS

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

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

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

INTRODUCTION

This chapter contains general information that will be useful to know before using the MPLAB C32 C Compiler. Items discussed in this chapter include:

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

DOCUMENT LAYOUT

This document describes how to use the MPLAB C32 C Compiler as a development tool to emulate and debug firmware on a target board. The document layout is as follows:

- **Chapter 1. Language Specifics** – discusses command line usage of the MPLAB C32 C compiler, attributes, pragmas, and data representation
- **Chapter 2. Library Environment** – discusses using the MPLAB C32 C libraries
- **Chapter 3. Interrupts** – presents an overview of interrupt processing
- **Chapter 4. Low Level Processor Control** – discusses access to the low level registers and configuration of the PIC32MX devices
- **Chapter 5. Compiler Runtime Environment** – discusses the MPLAB C32 C compiler runtime environment
- **Appendix A. Implementation Defined Behavior** – discusses the choices for implementation defined behavior in MPLAB C32 C compiler
- **Appendix B. Open Source Licensing** – gives a summary of the open source licenses used for portions of the MPLAB C32 C compiler package

CONVENTIONS USED IN THIS GUIDE

This manual uses the following documentation conventions:

DOCUMENTATION CONVENTIONS

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	*MPLAB® IDE User's Guide*
	Emphasized text	...is the *only* compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	select Enable Programmer
Quotes	A field name in a window or dialog	"Save project before build"
Underlined, italic text with right angle bracket	A menu path	*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:\mcc18\h`
	Keywords	`_asm, _endasm, static`
	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	`mcc18 [options]` *`file`* `[options]`
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; an OR selection	`errorlevel {0|1}`
Ellipses...	Replaces repeated text	`var_name [, var_name...]`
	Represents code supplied by user	`void main (void) { ... }`

RECOMMENDED READING

This user's guide describes how to use MPLAB C32 C Compiler. Other useful documents are listed below. The following Microchip documents are available and recommended as supplemental reference resources.

Readme Files

For the latest information on Microchip tools, read the associated Readme files (HTML files) included with the software.

Device-Specific Documentation

The Microchip website contains many documents that describe 16-bit device functions and features. Among these are:

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

MPLAB® C32 C Compiler Libraries (DS51685)

Reference guide for MPLAB C32 libraries and precompiled object files. Lists all library functions provided with the MPLAB C32 C compiler with detailed descriptions of their use.

PIC32MX Configuration Settings

Lists the Configuration Bit Settings for the Microchip PIC32MS devices supported by the MPLAB C32 C compiler's `#pragma config`.

C Standards Information

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

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

C Reference Manuals

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

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

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

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

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

GCC Documents

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

http://sourceware.org/binutils/

THE MICROCHIP WEB SITE

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

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

DEVELOPMENT SYSTEMS CUSTOMER CHANGE NOTIFICATION SERVICE

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

To register, access the Microchip web site at www.microchip.com, click on Customer Change Notification and follow the registration instructions.

The Development Systems product group categories are:

- **Compilers** – The latest information on Microchip C compilers and other language tools. These include the MPLAB C18, MPLAB C30 and MPLAB C32 C compilers; MPASM™ and MPLAB ASM30 assemblers; MPLINK™ and MPLAB LINK30 object linkers; and MPLIB™ and MPLAB LIB30 object librarians.
- **Emulators** – The latest information on Microchip in-circuit emulators. This includes the MPLAB REAL ICE™ and MPLAB ICE 2000 in-circuit emulators.
- **In-Circuit Debuggers** – The latest information on the Microchip in-circuit debuggers. These include MPLAB ICD 2 and PICkit™ 2.
- **MPLAB® IDE** – The latest information on Microchip MPLAB IDE, the Windows® Integrated Development Environment for development systems tools. This list is focused on the MPLAB IDE, MPLAB IDE Project Manager, MPLAB Editor and MPLAB SIM simulator, as well as general editing and debugging features.
- **Programmers** – The latest information on Microchip programmers. These include the MPLAB PM3 device programmer and the PICSTART® Plus, PICkit™ 1 and PICkit™ 2 development programmers.

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

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

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

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

DOCUMENT REVISION HISTORY

Revision A (October 2007)

- Initial Release of this document.

NOTES:

Chapter 1. Language Specifics

1.1 INTRODUCTION

This chapter discusses command line usage of the MPLAB C32 C compiler, attributes, pragmas and data representation.

1.2 HIGHLIGHTS

Items discussed in this chapter are:

- Overview
- File Naming Conventions
- Data Storage
- Predefined Macros
- Attributes and Pragmas
- Command Line Options
- Compiling a Single File on the Command Line
- Compiling Multiple Files on the Command Line

1.3 OVERVIEW

The compilation driver program (`pic32-gcc`) compiles, assembles and links C and assembly language modules and library archives. Most of the compiler command line options are common to all implementations of the GCC toolset. A few are specific to the MPLAB C32 C compiler.

The basic form of the compiler command line is:

```
pic32-gcc [options] files
```

> **Note:** Command line options and file name extensions are case sensitive.

The available options are described in **Section 1.8 "Command Line Options"**.

For example, to compile, assemble and link the C source file `hello.c`, creating the absolute executable `hello.out`.

```
pic32-gcc -o hello.out hello.c
```

1.4 FILE NAMING CONVENTIONS

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

TABLE 1-1: FILE NAMES

Extensions	Definition
file.c	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.s	An assembly language source file.

TABLE 1-1: FILE NAMES (CONTINUED)

Extensions	Definition
file.S	An assembly language source file that must be preprocessed.
other	A file to be passed to the linker.

1.5 DATA STORAGE

1.5.1 Storage Endianness

MPLAB C32 C compiler stores multi-byte 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

1.5.2 Integer Representation

Integer values in MPLAB C32 C compiler are represented in 2's complement and vary in size from 8 to 64 bits. These values are available in compiled code via `limits.h`.

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

1.5.3 Signed and Unsigned Character Types

By default, values of type plain `char` are signed values. This behavior is implementation-defined by the C standard, and some environments[1] define a plain char value to be unsigned. The command line option `-funsigned-char` can be used to set the default type to unsigned for a given translation unit.

1.5.4 Floating-Point Representation

MPLAB C32 C compiler uses the IEEE-754 floating-point format. Detail regarding the implementation limits is available to a translation unit in `float.h`.

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

1.5.5 Pointers

Pointers in MPLAB C32 C compiler are all 32 bits in size.

1. Notably, PowerPC and ARM

1.5.6 limits.h

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

Macro name	Value	Description
CHAR_BIT	8	The size, in bits, of the smallest non-bitfield object.
SCHAR_MIN	-128	The minimum value possible for an object of type signed char.
SCHAR_MAX	127	The maximum value possible for an object of type signed char.
UCHAR_MAX	255	The maximum value possible for an object of type unsigned char.
CHAR_MIN	-128 (or 0, see Signed and Unsigned Character Types)	The minimum value possible for an object of type char.
CHAR_MAX	127 (or 255, see 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.

1.6 PREDEFINED MACROS

1.6.1 MPLAB C32 C Compiler Macros

MPLAB C32 C compiler defines a number of macros, most with the prefix "_MCHP_," which characterize the various target specific options, the target processor and other aspects of the host environment.

_MCHP_SZINT	32 or 64, depending on command line options to set the size of an integer (-mint32 -mint64).
_MCHP_SZLONG	32 or 64, depending on command line options to set the size of an integer (-mlong32 -mlong64).
_MCHP_SZPTR	32 always since all pointers are 32 bits.
__mchp_no_float	Defined if -mno-float specified.
__NO_FLOAT	Defined if -mno-float specified.
__SOFT_FLOAT	Defined if -mno-float not specified. Indicates that floating-point is supported via library calls.
__PIC__ __pic__	The translation unit is being compiled for position independent code.
__PIC32MX __PIC32MX__	Always defined.
PIC32MX	Defined if -ansi is not specified.
__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.
__*processor*__	Where "processor" is the capitalized argument to the -mprocessor option. E.g., -mprocessor=32mx12f3456 will define __32MX12F3456__.

1.6.2 SDE Compatibility Macros

The MIPS® SDE (Software Development Environment) defines a number of macros, most with the prefix "_MIPS_," which characterize various target specific options, some determined by command line options (e.g., -mint64). Where applicable, these macros will be defined by the MPLAB C32 C compiler in order to ease porting applications and middleware from the SDE to MPLAB C32 C compiler.

_MIPS_SZINT	32 or 64, depending on command line options to set the size of an integer (-mint32 -mint64).
_MIPS_SZLONG	32 or 64, depending on command line options to set the size of an integer (-mlong32 -mlong64).
_MIPS_SZPTR	32 always since all pointers are 32 bits.
__mips_no_float	Defined if -mno-float specified.

__mips__ _mips _MIPS_ARCH_PIC32MX _MIPS_TUNE_PIC32MX _R3000 __R3000 __R3000__ __mips_soft_float __MIPSEL __MIPSEL__ _MIPSEL	Always defined.
R3000 MIPSEL	Defined if -ansi is not specified.
_mips_fpr	Defined as 32.
__mips16 __mips16e	Defined if -mips16 or -mips16e specified.
__mips	Defined as 32.
__mips_isa_rev	Defined as 2.
_MIPS_ISA	Defined as _MIPS_ISA_MIPS32.
__mips_single_float	Defined if -msingle-float specified.

1.7 ATTRIBUTES AND PRAGMAS

1.7.1 Function Attributes

always_inline

If the function is declared inline, always inline the function, even if no optimization level was specified.

longcall

Always invoke the function by first loading its address into a register and then using the contents of that register. This allows calling a function located beyond the 28 bit addressing range of the direct call instruction.

far

Functionally equivalent to longcall.

near

Always invoke the function with an absolute call instruction, even when the -mlong-calls command line option is specified.

mips16

Generate code for the function in the MIPS16 instruction set.

nomips16

Always generate code for the function in the MIPS32 instruction set, even when compiling the translation unit with the -mips16 command line option.

interrupt

Generate prologue and epilogue code for the function as an interrupt handler function. See **Chapter 3. "Interrupts"** and **Section 3.5 "Exception Handlers"**.

vector

Generate a branch instruction at the indicated exception vector which targets the function. See **Chapter 3. "Interrupts"** and **Section 3.5 "Exception Handlers"**.

at_vector

Place the body of the function at the indicated exception vector address. See **Chapter 3. "Interrupts"** and **Section 3.5 "Exception Handlers"**.

naked

Generate no prologue or epilogue code for the function.

section ("name")

Place the function into the named section.

For example,

```
void __attribute__ ((section (".wilma"))) baz () {return;}
```

Function `baz` will be placed in section `.wilma`.

The `-ffunction-sections` command line option has no effect on functions defined with a `section` attribute.

unique_section

Place the function in a uniquely named section, just as if `-ffunction-sections` had been specified. If the function also has a `section` attribute, use that section name as the prefix for generating the unique section name.

For example,

```
void __attribute__ ((section (".fred"), unique_section) foo (void) {return;}
```

Function `foo` will be placed in section `.fred.foo`.

noreturn

Indicate to the compiler that the function will never return. 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 as `noreturn` should always have a return type of `void`.

noinline

The function will never be considered for inlining.

pure

If a function has no side effects other than its return value, and the return value is dependent only on parameters and/or (nonvolatile) global variables, the compiler can perform more aggressive optimizations around invocations of that function. Such functions can be indicated with the `pure` attribute.

const

If a pure function determines its return value exclusively from its parameters (i.e., does not examine any global variables), it may be declared `const`, allowing for even more aggressive optimization. Note that a function which de-references a pointer argument is not `const` since the pointer de-reference uses a value which is not an parameter, even though the pointer itself is a parameter.

format (type, format_index, first_to_check)

The `format` attribute indicates that the function takes a `printf`, `scanf`, `strftime`, or `strfmon` style format string and arguments and that the compiler should type check those arguments against 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 which function parameter is the format string. Function parameters are numbered from the left-most parameter, starting from 1.

The `first_to_check` parameter specifies which parameter is the first to check against the format string. If `first_to_check` is zero, type checking is not performed and the compiler only checks the format string for consistency (e.g., `vfprintf`).

`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 function attribute which is a format string is identified by `index`.

`nonnull (index, ...)`

Indicate to the compiler that one or more pointer arguments to the function must be non-null. If the compiler determines that a null pointer is passed as a value to a non-null argument, and the `-Wnonnull` command line option was specified, a warning diagnostic is issued.

If no arguments are give to the `nonnull` attribute, all pointer arguments of the function are marked as non-null.

`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 see a reference to the function. For example, if inline assembly is the only reference to a static function.

`deprecated`

When a function specified as `deprecated` is used, a warning is generated.

`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.

`malloc`

Any non-null pointer return value from the indicated function will not alias any other pointer which is live at the point when the function returns. This allows the compiler to improve optimization.

`alias ("symbol")`

Indicates that the function is an alias for another symbol. For example,

```
void foo (void) { /* stuff */ }
void bar (void) __attribute__ ((alias("foo")));
```

Symbol `bar` is considered to be an alias for symbol `foo`.

1.7.2 Variable Attributes

`aligned (n)`

The attributed variable will 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.

`deprecated`

When a variable specified as `deprecated` is used, a warning is generated.

`packed`

The attributed variable or structure member will have the smallest possible alignment. That is, no alignment padding storage will be allocated for the declaration. Used in combination with the `aligned` attribute, `packed` can be used to set an arbitrary alignment restriction, greater or lesser than the default alignment for the type of the variable or structure member.

`section ("name")`

Place the function 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.

`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`.

`transparent_union`

When a function parameter of union type has the `transparent_union` attribute attached, corresponding arguments are passed as if the type were the type of the first member of the union.

`unused`

Indicate to the compiler that the variable may not be used. The compiler will not issue a warning for this variable if it is not used.

`weak`

A weak symbol indicates that if another version of the same symbol is available, that version should be used instead.

1.7.3 Pragmas

`#pragma interrupt`

Mark a function as an interrupt handler. The prologue and epilogue code for the function will perform more extensive context preservation. See **Chapter 3. "Interrupts"** and **Section 3.5 "Exception Handlers"**.

`#pragma vector`

Generate a branch instruction at the indicated exception vector which targets the function. See **Chapter 3. "Interrupts"** and **Section 3.5 "Exception Handlers"**.

`#pragma config`

The #pragma config directive specifies the processor-specific configuration settings (i.e., configuration bits) to be used by the application. See **Chapter 4. "Low Level Processor Control"**.

1.8 COMMAND LINE OPTIONS

MPLAB C32 C compiler has many options for controlling compilation, all of which are case sensitive.

- Options Specific to PIC32MX Devices
- Options for Controlling the Kind of Output
- Options for Controlling the C Dialect
- Options for Controlling Warnings and Errors
- Options for Debugging
- Options for Controlling Optimization
- Options for Controlling the Preprocessor
- Options for Assembling
- Options for Linking
- Options for Directory Search
- Options for Code Generation Conventions

1.8.1 Options Specific to PIC32MX Devices

TABLE 1-2: PIC32MX DEVICE-SPECIFIC OPTIONS

Option	Definition
`-m`*`processor`*	Selects the device for which to compile (e.g., `-m`*`processor`*=32MX360F512L)
`-mips16` `-mno-mips16`	Generate (do not generate) MIPS16 code.
`-mno-float`	Don't use floating-point libraries.
`-msingle-float`	Assume that the floating-point coprocessor only supports single-precision operations.
`-mdouble-float`	Assume that the floating-point coprocessor supports double-precision operations. This is the default.
`-mlong64`	Force `long` types to be 64 bits wide. See `-mlong32` for an explanation of the default and the way that the pointer size is determined.
`-mlong32`	Force `long`, `int`, and pointer types to be 32 bits wide. The default size of `int`s, `long`s and `pointer`s is 32 bits.
`-G` *`num`*	Put global and static items less than or equal to *`num`* bytes into the small data or bss section instead of the normal data or bss section. This allows the data to be accessed using a single instruction. All modules should be compiled with the same `-G` *`num`* value.

TABLE 1-2: PIC32MX DEVICE-SPECIFIC OPTIONS (CONTINUED)

Option	Definition
`-membedded-data` `-mno-embedded-data`	Allocate variables to the read-only data section first if possible, then next in the small data section if possible, otherwise in data. This gives slightly slower code than the default, but reduces the amount of RAM required when executing, and thus may be preferred for some embedded systems.
`-muninit-const-in-rodata` `-mno-uninit-const-in-rodata`	Put uninitialized `const` variables in the read-only data section. This option is only meaningful in conjunction with `-membedded-data`.
`-mcheck-zero-division` `-mno-check-zero-division`	Trap (do not trap) on integer division by zero. The default is `-mcheck-zero-division`.
`-mmemcpy` `-mno-memcpy`	Force (do not force) the use of `memcpy()` for non-trivial block moves. The default is `-mno-memcpy`, which allows GCC to inline most constant-sized copies.
`-mlong-calls` `-mno-long-calls`	Disable (do not disable) use of the `jal` instruction. Calling functions using `jal` is more efficient but requires the caller and callee to be in the same 256 megabyte segment. This option has no effect on abicalls code. The default is `-mno-long-calls`.
`-mno-peripheral-libs`	Do not use the standard peripheral libraries when linking.

1.8.2 Options for Controlling the Kind of Output

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

TABLE 1-3: KIND-OF-OUTPUT CONTROL OPTIONS

Option	Definition
`-c`	Compile or assemble the source files, but do not link. The default file extension is `.o`.
`-E`	Stop after the preprocessing stage, i.e., before running the compiler proper. The default output file is `stdout`.
`-o` *file*	Place the output in *file*.
`-S`	Stop after compilation proper (i.e., before invoking the assembler). The default output file extension is `.s`.
`-v`	Print the commands executed during each stage of compilation.

`-x`	You can specify the input language explicitly with the `-x` option:
	-x language
	Specify explicitly the language for the following input files (rather than letting the compiler choose a default based on the file name suffix). This option applies to all following input files until the next `-x` option. The following values are supported by MPLAB C32 C compiler:
	`c`
	`c-header`
	`cpp-output`
	`assembler`
	`assembler-with-cpp`
	
	`-x none`
	Turn off any specification of a language, so that subsequent files are handled according to their file name suffixes. This is the default behavior but is needed if another `-x` option has been used. For example:
	`pic32-gcc -x assembler foo.asm bar.asm -x none main.c mabonga.s`
	
	Without the `-x none`, the compiler assumes all the input files are for the assembler.

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

1.8.3 Options for Controlling the C Dialect

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

TABLE 1-4: C DIALECT CONTROL OPTIONS

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

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

`-funsigned-char`	Let the type `char` be unsigned, like `unsigned char`.
`-fwritable-strings`	Store strings in the writable data segment and don't make them unique.

1.8.4 Options for Controlling Warnings and Errors

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

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

The following options control the amount and kinds of warnings produced by the MPLAB C32 C Compiler.

TABLE 1-5: WARNING AND ERROR OPTIONS IMPLIED BY `-WALL`

Option	Definition
`-fsyntax-only`	Check the code for syntax, but don't do anything beyond that.
`-pedantic`	Issue all the warnings demanded by strict ANSI C. Reject all programs that use forbidden extensions.
`-pedantic-errors`	Like `-pedantic`, except that errors are produced rather than warnings.
`-w`	Inhibit all warning messages.
`-Wall`	All of the `-W` options listed in this table combined. This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
`-Wchar-subscripts`	Warn if an array subscript has type `char`.
`-Wcomment` `-Wcomments`	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.)**
`-Werror-implicit-function-declaration`	Give an error whenever a function is used before being declared.
`-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 } };`

TABLE 1-5: WARNING AND ERROR OPTIONS IMPLIED BY `-WALL` (CONTINUED)

Option	Definition

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

`-Wparentheses`	Warn if parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, or when operators are nested whose precedence people often find confusing.
`-Wreturn-type`	Warn whenever a function is defined with a return-type that defaults to `int`. Also warn about any `return` statement with no return-value in a function whose return-type is not `void`.
`-Wsequence-point`	Warn about code that may have undefined semantics because of violations of sequence point rules in the C standard. The C standard defines the order in which expressions in a C program are evaluated in terms of sequence points, which represent a partial ordering between the execution of parts of the program: those executed before the sequence point and those executed after it. These occur after the evaluation of a full expression (one which is not part of a larger expression), after the evaluation of the first operand of a `&&`, `||`, `? :` or `,` (comma) operator, before a function is called (but after the evaluation of its arguments and the expression denoting the called function), and in certain other places. Other than as expressed by the sequence point rules, the order of evaluation of subexpressions of an expression is not specified. All these rules describe only a partial order rather than a total order, since, for example, if two functions are called within one expression with no sequence point between them, the order in which the functions are called is not specified. However, the standards committee has ruled that function calls do not overlap. It is not specified, when, between sequence points modifications to the values of objects take effect. Programs whose behavior depends on this have undefined behavior, The C standard specifies that "Between the previous and next sequence point, an object shall have its stored value modified, at most once, by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored." If a program breaks these rules, the results on any particular implementation are entirely unpredictable. Examples of code with undefined behavior are `a = a++;`, `a[n] = b[n++]` and `a[i++] = i;`. Some more complicated cases are not diagnosed by this option, and it may give an occasional false positive result, but in general it has been found fairly effective at detecting this sort of problem in programs.

TABLE 1-5: WARNING AND ERROR OPTIONS IMPLIED BY `-WALL` (CONTINUED)

Option	Definition
`-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 MPLAB C32 C 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 MPLAB C32 C compiler. If this command line option is used, warnings are 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.
`-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.