MPLAB® C Compiler For PIC32 MCUs 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.

Trademarks

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

MPLAB® C COMPILER FOR PIC32 MCUs USER'S GUIDE

Table of Contents

MPLAB® C Compiler for PIC32 MCUs User's Guide

MPLAB® C COMPILER FOR PIC32 MCUs 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 32-bit 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 32-bit 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 compiler, attributes, pragmas, and data representation
- **Chapter 2. Library Environment** – discusses using the compiler 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 Run-time Environment** – discusses the compiler run-time environment
- **Appendix A. Implementation Defined Behavior** – discusses the choices for implementation defined behavior in compiler
- **Appendix B. Open Source Licensing** – gives a summary of the open source licenses used for portions of the compiler package

---

MPLAB® C Compiler for PIC32 MCUs User's Guide

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 32-bit 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 web site 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

32-Bit Language Tools Libraries (DS51685)

Lists all library functions provided with the MPLAB C Compiler for PIC32 MCUs with detailed descriptions of their use.

PIC32MX Configuration Settings

Lists the Configuration Bit Settings for the Microchip PIC32MX devices supported by the MPLAB C Compiler for PIC32 MCUs'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/

MPLAB® C Compiler for PIC32 MCUs User's Guide

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, assemblers, linkers and other language tools. These include all MPLAB C compilers; all MPLAB assemblers (including MPASM™ assembler); all MPLAB linkers (including MPLINK™ object linker); and all MPLAB librarians (including MPLIB™ object librarian).
- **Emulators** – The latest information on Microchip in-circuit emulators. This includes the MPLAB REAL ICE™ 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, PICkit™ 2 and PICkit™ 3 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.

Revision B (July 2009)

- Changed product name from MPLAB C32 C Compiler to MPLAB C Compiler for PIC32 MCUs throughout the document. Also changed references to C32 Libraries to 32-Bit Language Tools Libraries.
- Added `__VERSION__` and `__C32_VERSION__` to **Section 1.6.1 "32-Bit C Compiler Macros"**
- Added `-msmart-io=[0|1|2]` and `-mapio-debug` to **Section 1.8.1 "Options Specific to PIC32MX Devices"**, Table 1-2
- Added **Section 1.11 "Binary Constants"**
- Added `__ISR_SINGLE__` and `__ISR_SINGLE_AT_VECTOR__` to **Section 2.5.1 "sys/attribs.h"**
- Revised **Section 3.3.2.1 "Interrupt Attribute"**
- Added **Section 3.4.4 "__ISR Macros"**
- Added **Section 3.4.4.3 "Interrupt-Vector Macros"**

MPLAB® C Compiler for PIC32 MCUs User's Guide

NOTES:

Chapter 1. Language Specifics

1.1   INTRODUCTION

This chapter discusses command line usage of the 32-bit 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
- Binary Constants

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 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.

TABLE 1-1: FILE NAMES (CONTINUED)

Extensions	Definition
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.

1.5 DATA STORAGE

1.5.1 Storage Endianness

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

1.5.2 Integer Representation

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

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

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

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

Type	Bits
float	32
double	64
long double	64

1.5.5 Pointers

Pointers in the 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-bit field object.
SCHAR_MIN	-128	The minimum value possible for an object of type `signed char`.
SCHAR_MAX	127	The maximum value possible for an object of type `signed char`.
UCHAR_MAX	255	The maximum value possible for an object of type `unsigned char`.
CHAR_MIN	-128 (or 0, see 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 32-Bit C Compiler Macros

The 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.
`__PIC32_FEATURE_SET__`	The compiler predefines a macro based on the features available for the selected device. These macros are intended to be used when writing code to take advantage of features available on newer devices while maintaining compatibility with older devices. Examples: PIC32MX795F512L would use `__PIC32_FEATURE_SET__ == 795`, and PIC32MX340F128H would use `__PIC32_FEATURE_SET__ == 340`.
`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__`.
`__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 `__C32_VERSION__` macro for a numeric version number.

__C32_VERSION__	The C compiler defines the constant __C32_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 __C32_VERSION__ definition of 103. This macro can be used, in conjunction with standard preprocessor comparison statements, to conditionally include/exclude various code constructs.

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 compiler in order to ease porting applications and middleware from the SDE to the 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.

MPLAB® C Compiler for PIC32 MCUs User's Guide

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 a 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 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.

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"**.

MPLAB® C Compiler for PIC32 MCUs User's Guide

1.8   COMMAND LINE OPTIONS

The 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
`-mprocessor`	Selects the device for which to compile. (e.g., `-mprocessor`=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.
`-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`.

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

Option	Definition
-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.
-msmart-io=[0|1|2]	This option attempts to statically analyze format strings passed to printf, scanf and the 'f' and 'v' variations of these functions. Uses of nonfloating-point format arguments will be converted to use an integer-only variation of the library function. For many applications, this feature can reduce program-memory usage. -msmart-io=0 disables this option, while -msmart-io=2 causes the compiler to be optimistic and convert function calls with variable or unknown format arguments. -msmart-io=1 is the default and will convert only when the compiler can prove that floating-point support is not required.
-mappio-debug	Enable the APPIN/APPOUT debugging library functions for the MPLAB ICD 3 debugger and MPLAB REAL ICE emulator. This feature allows you to use the DBPRINTF and related functions and macros as described in the 32-bit Language Tool Libraries document (DS51685). Enable this option only when using a target PIC32 device that supports the APPIN/APPOUT feature.

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 the 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 start-up may not necessarily be at main. The most obvious example is an OS kernel. This is equivalent to `-fno-hosted`.
`-fno-asm`	Do not recognize `asm`, `inline` or `typeof` as a keyword, so that code can use these words as identifiers. You can use the keywords `__asm__`, `__inline__` and `__typeof__` instead. `-ansi` implies `-fno-asm`.
`-fno-builtin` `-fno-builtin-`*function*	Don't recognize built-in functions that do not begin with `__builtin_` as prefix.
`-fsigned-char`	Let the type `char` be signed, like `signed char`. **(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 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 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.
-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.

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

Option	Definition
`-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 1-6:   WARNING AND ERROR OPTIONS NOT IMPLIED BY -WALL

Option	Definition

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

-Waggregate-return	Warn if any functions that return structures or unions are defined or called.
-Wbad-function-cast	Warn whenever a function call is cast to a non-matching type. For example, warn if int foof() is cast to anything *.

TABLE 1-6: **WARNING AND ERROR OPTIONS NOT IMPLIED BY -WALL (CONTINUED)**

Option	Definition
-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.
-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. Actually, 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.

TABLE 1-6: **WARNING AND ERROR OPTIONS NOT IMPLIED BY
-WALL (CONTINUED)**

Option	Definition
-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.

1.8.5 Options for Debugging

The following options are used for debugging.

TABLE 1-7: DEBUGGING OPTIONS

Option	Definition
-g	Produce debugging information. The compiler supports the use of -g with -O making it possible to debug optimized code. The shortcuts taken by optimized code may occasionally produce surprising results: • Some declared variables may not exist at all; • Flow of control may briefly move unexpectedly; • Some statements may not be executed because they compute constant results or their values were already at hand; • Some statements may execute in different places because they were moved out of loops. Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.
-Q	Makes the compiler print out each function name as it is compiled, and print some statistics about each pass when it finishes.
-save-temps	Don't delete intermediate files. Place them in the current directory and name them based on the source file. Thus, compiling foo.c with -c -save-temps would produce the following files: foo.i (preprocessed file) foo.s (assembly language file) foo.o (object file)

1.8.6   Options for Controlling Optimization

The following options control compiler optimizations.

TABLE 1-8:   GENERAL OPTIMIZATION OPTIONS

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

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

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

TABLE 1-9:   SPECIFIC OPTIMIZATION OPTIONS

Option	Definition
-falign-functions -falign-functions=*n*	Align the start of functions to the next power-of-two greater than *n*, skipping up to *n* bytes. For instance, -falign-functions=32 aligns functions to the next 32-byte boundary, but -falign-functions=24 would align to the next 32-byte boundary only if this can be done by skipping 23 bytes or less. -fno-align-functions and -falign-functions=1 are equivalent and mean that functions are not aligned. The assembler only supports this flag when *n* is a power of two, so *n* is rounded up. If *n* is not specified, use a machine-dependent default.
-falign-labels -falign-labels=*n*	Align all branch targets to a power-of-two boundary, skipping up to *n* bytes like -falign-functions. This option can easily make code slower, because it must insert dummy operations for when the branch target is reached in the usual flow of the code. If -falign-loops or -falign-jumps are applicable and are greater than this value, then their values are used instead. If *n* is not specified, use a machine-dependent default which is very likely to be 1, meaning no alignment.
-falign-loops -falign-loops=*n*	Align loops to a power-of-two boundary, skipping up to *n* bytes like -falign-functions. The hope is that the loop is executed many times, which makes up for any execution of the dummy operations. If *n* is not specified, use a machine-dependent default.
-fcaller-saves	Enable values to be allocated in registers that are clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.
-fcse-follow-jumps	In common subexpression elimination, scan through jump instructions when the target of the jump is not reached by any other path. For example, when CSE encounters an if statement with an else clause, CSE follows the jump when the condition tested is false.
-fcse-skip-blocks	This is similar to -fcse-follow-jumps, but causes CSE to follow jumps which conditionally skip over blocks. When CSE encounters a simple if statement with no else clause, -fcse-skip-blocks causes CSE to follow the jump around the body of the if.
-fexpensive- optimizations	Perform a number of minor optimizations that are relatively expensive.
-ffunction-sections -fdata-sections	Place each function or data item into its own section in the output file. The name of the function or the name of the data item determines the section's name in the output file. Only use these options when there are significant benefits for doing so. When you specify these options, the assembler and linker may create larger object and executable files and is also slower.
-fgcse	Perform a global common subexpression elimination pass. This pass also performs global constant and copy propagation.

TABLE 1-9: **SPECIFIC OPTIMIZATION OPTIONS (CONTINUED)**

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

TABLE 1-9:   SPECIFIC OPTIMIZATION OPTIONS (CONTINUED)

Option	Definition

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

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

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

TABLE 1-10: MACHINE-INDEPENDENT OPTIMIZATION OPTIONS

Option	Definition
`-fforce-mem`	Force memory operands to be copied into registers before doing arithmetic on them. This produces better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load. The `-O2` option turns on this option.
`-finline-functions`	Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated, and the function is declared `static`, then the function is normally not output as assembler code in its own right.
`-finline-limit=n`	By default, the compiler limits the size of functions that can be inlined. This flag allows the control of this limit for functions that are explicitly marked as inline (i.e., marked with the `inline` keyword). `n` is the size of functions that can be inlined in number of pseudo instructions (not counting parameter handling). The default value of `n` is 10000. Increasing this value can result in more inlined code at the cost of compilation time and memory consumption. Decreasing usually makes the compilation faster and less code is inlined (which presumably means slower programs). This option is particularly useful for programs that use inlining. **Note:** Pseudo instruction represents, in this particular context, an abstract measurement of function's size. In no way does it represent a count of assembly instructions and as such, its exact meaning might change from one release of the compiler to an another.
`-fkeep-inline-functions`	Even if all calls to a given function are integrated, and the function is declared `static`, output a separate run time callable version of the function. This switch does not affect `extern` inline functions.
`-fkeep-static-consts`	Emit variables declared static const when optimization isn't turned on, even if the variables are not referenced. The compiler enables this option by default. If you want to force the compiler to check if the variable was referenced, regardless of whether or not optimization is turned on, use the `-fno-keep-static-consts` option.
`-fno-function-cse`	Do not put function addresses in registers. Make each instruction that calls a constant function contain the function's address explicitly. This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

TABLE 1-10: MACHINE-INDEPENDENT OPTIMIZATION OPTIONS

Option	Definition
-fno-inline	Do not pay attention to the inline keyword. Normally this option is used to keep the compiler from expanding any functions inline. If optimization is not enabled, no functions can be expanded inline.
-fomit-frame-pointer	Do not keep the Frame Pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore Frame Pointers. It also makes an extra register available in many functions.
-foptimize-sibling-calls	Optimize sibling and tail recursive calls.

1.8.7 Options for Controlling the Preprocessor

The following options control the compiler preprocessor.

TABLE 1-11: PREPROCESSOR OPTIONS

Option	Definition

-A*question* (*answer*)	Assert the answer *answer* for question *question*, in case it is tested with a preprocessing conditional such as #if #*question*(*answer*). -A- disables the standard assertions that normally describe the target machine.
	For example, the function prototype for main might be declared as follows:
	#if #environ(freestanding)
	int main(void);
	#else
	int main(int argc, char *argv[]);
	#endif
	A -A command line option could then be used to select between the two prototypes. For example, to select the first of the two, the following command line option could be used:
	-Aenviron(freestanding)

-A -*predicate* =*answer*	Cancel an assertion with the predicate *predicate* and answer *answer*.
-A *predicate* =*answer*	Make an assertion with the predicate *predicate* and answer *answer*. This form is preferred to the older form -A *predicate*(*answer*), which is still supported, because it does not use shell special characters.
-C	Tell the preprocessor not to discard comments. Used with the -E option.
-dD	Tell the preprocessor to not remove macro definitions into the output, in their proper sequence.
-D*macro*	Define macro *macro* with the string 1 as its definition.
-D*macro*=*defn*	Define macro *macro* as *defn*. All instances of -D on the command line are processed before any -U options.
-dM	Tell the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing. Used with the -E option.
-dN	Like -dD except that the macro arguments and contents are omitted. Only #define name is included in the output.
-fno-show-column	Do not print column numbers in diagnostics. This may be necessary if diagnostics are being scanned by a program that does not understand the column numbers, such as dejagnu.
-H	Print the name of each header file used, in addition to other normal activities.

TABLE 1-11:   PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
-I-	Any directories you specify with -I options before the -I- options are searched only for the case of #include "*file*". They are not searched for #include <*file*>. If additional directories are specified with -I options after the -I-, these directories are searched for all #include directives. (Ordinarily all -I directories are used this way.) In addition, the -I- option inhibits the use of the current directory (where the current input file came from) as the first search directory for #include "*file*". There is no way to override this effect of -I-. With -I. you can specify searching the directory that was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory. -I- does not inhibit the use of the standard system directories for header files. Thus, -I- and -nostdinc are independent.
-I*dir*	Add the directory *dir* to the head of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one -I option, the directories are scanned in left-to-right order. The standard system directories come after.
-idirafter *dir*	Add the directory *dir* to the second include path. The directories on the second include path are searched when a header file is not found in any of the directories in the main include path (the one that -I adds to).
-imacros *file*	Process file as input, discarding the resulting output, before processing the regular input file. Because the output generated from the file is discarded, the only effect of -imacros *file* is to make the macros defined in file available for use in the main input. Any -D and -U options on the command line are always processed before -imacros *file*, regardless of the order in which they are written. All the -include and -imacros options are processed in the order in which they are written.
-include *file*	Process file as input before processing the regular input file. In effect, the contents of file are compiled first. Any -D and -U options on the command line are always processed before -include *file*, regardless of the order in which they are written. All the -include and -imacros options are processed in the order in which they are written.
-iprefix *prefix*	Specify *prefix* as the prefix for subsequent -iwithprefix options.
-isystem *dir*	Add a directory to the beginning of the second include path, marking it as a system directory, so that it gets the same special treatment as is applied to the standard system directories.
-iwithprefix *dir*	Add a directory to the second include path. The directory's name is made by concatenating prefix and *dir*, where prefix was specified previously with -iprefix. If a prefix has not yet been specified, the directory containing the installed passes of the compiler is used as the default.
-iwithprefixbefore *dir*	Add a directory to the main include path. The directory's name is made by concatenating prefix and *dir*, as in the case of -iwithprefix.

TABLE 1-11: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
`-M`	Tell the preprocessor to output a rule suitable for `make` describing the dependencies of each object file. For each source file, the preprocessor outputs one make-rule whose target is the object file name for that source file and whose dependencies are all the `#include` header files it uses. This rule may be a single line or may be continued with `\-newline` if it is long. The list of rules is printed on standard output instead of the preprocessed C program. `-M` implies `-E` (see **Section 1.8.2 "Options for Controlling the Kind of Output"**).
`-MD`	Like `-M` but the dependency information is written to a file and compilation continues. The file containing the dependency information is given the same name as the source file with a `.d` extension.
`-MF` *file*	When used with `-M` or `-MM`, specifies a file in which to write the dependencies. If no `-MF` switch is given, the preprocessor sends the rules to the same place it would have sent preprocessed output. When used with the driver options, `-MD` or `-MMD`, `-MF`, overrides the default dependency output file.
`-MG`	Treat missing header files as generated files and assume they live in the same directory as the source file. If `-MG` is specified, then either `-M` or `-MM` must also be specified. `-MG` is not supported with `-MD` or `-MMD`.
`-MM`	Like `-M` but the output mentions only the user header files included with `#include "`*file*`"`. System header files included with `#include <`*file*`>` are omitted.
`-MMD`	Like `-MD` except mention only user header files, not system header files.

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

`-MQ`	Same as `-MT`, but it quotes any characters which are special to `make`. `-MQ '$(objpfx)foo.o'` gives `$$(objpfx)foo.o: foo.c` The default target is automatically quoted, as if it were given with `-MQ`.
`-MT` *target*	Change the target of the rule emitted by dependency generation. By default, CPP takes the name of the main input file, including any path, deletes any file suffix such as `.c`, and appends the platform's usual object suffix. The result is the target. An `-MT` option sets the target to be exactly the string you specify. If you want multiple targets, you can specify them as a single argument to `-MT`, or use multiple `-MT` options. For example: `-MT '$(objpfx)foo.o'` might give `$(objpfx)foo.o: foo.c`

TABLE 1-11: PREPROCESSOR OPTIONS (CONTINUED)

Option	Definition
-nostdinc	Do not search the standard system directories for header files. Only the directories you have specified with -I options (and the current directory, if appropriate) are searched. (See **Section 1.8.10 "Options for Directory Search"**) for information on -I. By using both -nostdinc and -I-, the include-file search path can be limited to only those directories explicitly specified.
-P	Tell the preprocessor not to generate #line directives. Used with the -E option (see **Section 1.8.2 "Options for Controlling the Kind of Output"**).
-trigraphs	Support ANSI C trigraphs. The -ansi option also has this effect.
-Umacro	Undefine macro *macro*. -U options are evaluated after all -D options, but before any -include and -imacros options.
-undef	Do not predefine any nonstandard macros (including architecture flags).

1.8.8 Options for Assembling

The following options control assembler operations.

TABLE 1-12: ASSEMBLY OPTIONS

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

1.8.9   Options for Linking

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

TABLE 1-13:   LINKING OPTIONS

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

-l*library*	Search the library named *library* when linking.
	The linker searches a standard list of directories for the library, which is actually a file named lib*library*.a. The linker then uses this file as if it had been specified precisely by name.
	It makes a difference where in the command you write this option. The linker processes libraries and object files in the order they are specified. Thus, foo.o -lz bar.o searches library z after file foo.o but before bar.o. If bar.o refers to functions in libz.a, those functions may not be loaded.
	The directories searched include several standard system directories, plus any that you specify with -L.
	Normally the files found this way are library files (archive files whose members are object files). The linker handles an archive file by scanning through it for members which define symbols that have so far been referenced but not defined. But if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between using an -l option (e.g., -lmylib) and specifying a file name (e.g., libmylib.a) is that -l searches several directories, as specified.
	By default the linker is directed to search:
	`<install-path>\lib`
	for libraries specified with the -l option. For a compiler installed into the default location, this would be:
	`c:\Program Files\Microchip\MPLAB C32\lib`
	This behavior can be overridden using the environment variables.

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

1.8.10   Options for Directory Search

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

TABLE 1-14:   DIRECTORY SEARCH OPTIONS

Option	Definition
-B*prefix*	This option specifies where to find the executables, libraries, include files and data files of the compiler itself. The compiler driver program runs one or more of the sub-programs `pic32-cpp`, `pic32-cc1`, `pic32-as` and `pic32-ld`. It tries *prefix* as a prefix for each program it tries to run. For each sub-program to be run, the compiler driver first tries the -B prefix, if any. Lastly, the driver searches the current `PATH` environment variable for the subprogram. -B prefixes that effectively specify directory names also apply to libraries in the linker, because the compiler translates these options into -L options for the linker. They also apply to include files in the preprocessor, because the compiler translates these options into -isystem options for the preprocessor. In this case, the compiler appends `include` to the prefix.
-specs=*file*	Process file after the compiler reads in the standard `specs` file, in order to override the defaults that the `pic32-gcc` driver program uses when determining what switches to pass to `pic32-cc1`, `pic32-as`, `pic32-ld`, etc. More than one -specs=*file* can be specified on the command line, and they are processed in order, from left to right.

1.8.11   Options for Code Generation Conventions

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

TABLE 1-15:   CODE GENERATION CONVENTION OPTIONS

Option	Definition
-fargument-alias -fargument-noalias -fargument- noalias-global	Specify the possible relationships among parameters and between parameters and global data. -fargument-alias specifies that arguments (parameters) may alias each other and may alias global storage. -fargument-noalias specifies that arguments do not alias each other, but may alias global storage. -fargument-noalias-global specifies that arguments do not alias each other and do not alias global storage. Each language automatically uses whatever option is required by the language standard. You should not need to use these options yourself.
-fcall-saved-*reg*	Treat the register named *reg* as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way saves and restores the register *reg* if they use it. It is an error to used this flag with the Frame Pointer or Stack Pointer. Use of this flag for other registers that have fixed pervasive roles in the machine's execution model produces disastrous results. A different sort of disaster results from the use of this flag for a register in which function values are returned. This flag should be used consistently through all modules.
-fcall-used-*reg*	Treat the register named *reg* as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way do not save and restore the register *reg*. It is an error to use this flag with the Frame Pointer or Stack Pointer. Use of this flag for other registers that have fixed pervasive roles in the machine's execution model produces disastrous results. This flag should be used consistently through all modules.
-ffixed-*reg*	Treat the register named *reg* as a fixed register. Generated code should never refer to it (except perhaps as a Stack Pointer, Frame Pointer or in some other fixed role). *reg* must be the name of a register (e.g., -ffixed-$0).

TABLE 1-15: CODE GENERATION CONVENTION OPTIONS (CONTINUED)

Option	Definition

-finstrument-functions	Generate instrumentation calls for entry and exit to functions. Just after function entry and just before function exit, the following profiling functions are called with the address of the current function and its call site.
	`void __cyg_profile_func_enter`
	`  (void *this_fn, void *call_site);`
	`void __cyg_profile_func_exit`
	`  (void *this_fn, void *call_site);`
	The first argument is the address of the start of the current function, which may be looked up exactly in the symbol table. The profiling functions should be provided by the user.
	Function instrumentation requires the use of a Frame Pointer. Some optimization levels disable the use of the Frame Pointer. Using -fno-omit-frame-pointer prevents this.
	This instrumentation is also done for functions expanded inline in other functions. The profiling calls indicates where, conceptually, the inline function is entered and exited. This means that addressable versions of such functions must be available. If all your uses of a function are expanded inline, this may mean an additional expansion of code size. If you use extern inline in your C code, an addressable version of such functions must be provided.
	A function may be given the attribute no_instrument_function, in which case this instrumentation is not done.

-fno-ident	Ignore the #ident directive.
-fpack-struct	Pack all structure members together without holes. Usually you would not want to use this option, since it makes the code sub-optimal, and the offsets of structure members won't agree with system libraries.
-fpcc-struct-return	Return short struct and union values in memory like longer ones, rather than in registers. This convention is less efficient, but it has the advantage of allowing capability between 32-bit compiled files and files compiled with other compilers. Short structures and unions are those whose size and alignment match that of an integer type.
-fno-short-double	By default, the compiler uses a double type equivalent to float. This option makes double equivalent to long double. Mixing this option across modules can have unexpected results if modules share double data either directly through argument passage or indirectly through shared buffer space. Libraries provided with the product function with either switch setting.
-fshort-enums	Allocate to an enum type only as many bytes as it needs for the declared range of possible values. Specifically, the enum type is equivalent to the smallest integer type which has enough room.
-fverbose-asm -fno-verbose-asm	Put extra commentary information in the generated assembly code to make it more readable. -fno-verbose-asm, the default, causes the extra information to be omitted and is useful when comparing two assembler files.
-fvolatile	Consider all memory references through pointers to be volatile.
-fvolatile-global	Consider all memory references to external and global data items to be volatile. The use of this switch has no effect on static data.
-fvolatile-static	Consider all memory references to static data to be volatile.

Language Specifics

1.9   COMPILING A SINGLE FILE ON THE COMMAND LINE

This section demonstrates how to compile and link a single file. For the purpose of this discussion, it is assumed the compiler is installed on your `c:` drive in a directory called `Program Files\Microchip\MPLAB C32`. Therefore the following applies:

- `c:\Program Files\Microchip\MPLAB C32\pic32mx\include` – Include directory for standard C header files.
- `c:\Program Files\Microchip\MPLAB C32\pic32mx\include\proc` – Include directory for PIC32MX device-specific header files.
- `c:\Program Files\Microchip\MPLAB C32\pic32mx\lib` – Library directory structure for standard libraries and start-up files.
- `c:\Program Files\Microchip\MPLAB C32\pic32mx\include\peripheral` – Include directory for PIC32MX peripheral library include files.
- `c:\Program Files\Microchip\MPLAB C32\pic32mx\lib\proc` – Directory for device-specific linker script fragments, register definition files and configuration data may be found.
- `c:\Program Files\Microchip\MPLAB C32\bin` – Directory where the top level tools executables are located. The `PATH` environment variable may include this directory.

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`.

```
#include <p32xxxx.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 first line of the program includes the header file `p32xxxx.h,` which provides definitions for all Special Function Registers (SFRs) on that part. For more information on processor header files, see **Chapter 4. "Low-Level Processor Control"**.

Compile the program by typing the following at a DOS prompt:

```
C:\> pic32-gcc -o ex1.out ex1.c
```

The command line option `-o ex1.out` 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 loaded into the MPLAB IDE.

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

```
C:\> pic32-bin2hex ex1.out
```

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

1.10 COMPILING MULTIPLE FILES ON THE COMMAND LINE

Move the `Add()` function into a file called `add.c` to demonstrate the use of multiple files in an application. That is:

File 1
```c
/* ex1.c */
#include <p32xxxx.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
```c
/* add.c */
#include <p32xxxx.h>
unsigned int
add(unsigned int a, unsigned int b)
{
  return(a+b);
}
```

Compile both files by typing the following at a DOS prompt:

```
C:\> pic32-gcc -o ex1.out 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.out` is created.

1.11 BINARY CONSTANTS

A sequence of binary digits preceded by `0b` or `0B` (the numeral '`0`' followed by the letter '`b`' or '`B`') is taken to be a binary integer. The binary digits consist of the numerals '`0`' and '`1`'. For example, the (decimal) number 255 can be written as `0b11111111`. Like other integer constants, a binary constant may be suffixed by the letter '`u`' or '`U`', to specify that it is unsigned. A binary constant may also be suffixed by the letter '`l`' or '`L`', to specify that it is long. Similarly, the suffix '`ll`' or '`LL`' denotes a long long binary constant.

> **Note:** This binary-constant syntax may not be accepted by other C compilers.

Chapter 2. Library Environment

2.1 INTRODUCTION

This chapter discusses using the 32-bit language tools libraries.

2.2 HIGHLIGHTS

Items discussed in this chapter are:

- Standard I/O
- Weak Functions
- "Helper" Header Files
- Multilibs

2.3 STANDARD I/O

The standard input/output library functions support two modes of operation, Simple and Full. The Simple mode supports I/O via a two function interface on a single character device used for `stdout`, `stdin` and `stderr`. The Full mode supports the complete set of standard I/O functions. The library will use Full mode if the application calls `fopen`, otherwise Simple mode is used.

Simple mode performs I/O using four functions, `_mon_puts`, `_mon_write`, `_mon_getc` and `_mon_putc`, to perform the raw device I/O. The default implementation of `_mon_getc` always returns failure (i.e., by default, character input is not available). The default implementation of `_mon_putc` writes a character to UART2. It is assumed that the application has performed any necessary initialization of the UART. The default implementations of `_mon_puts` and `_mon_write` both simply call `_mon_putc` iteratively. All four functions are defined as weak functions, and so may be overridden by the user application if different functionality is desired. See the "*32-Bit Language Tools Libraries*" for detailed information on these functions.

An application using Full mode must supply the standard low-level POSIX I/O functions `open`, `read`, `write`, `lseek` and `close`. No default implementations are provided. See the "*32-Bit Language Tools Libraries*" (DS51685) for detailed information on these functions.

2.4 WEAK FUNCTIONS

The standard library provides a number of weak function implementations of low-level interfaces. User applications which use this functionality will often implement more full featured versions of these functions. For details of the specific functions, see the "*32-Bit Language Tools Libraries Libraries*" (DS51685).

As described above, the standard I/O library functions utilize a set of weak functions for simple output: `_mon_write`, `_mon_putc`, `_mon_puts`, and `_mon_getc`.

The standard start-up code (See **Section 5.7 "Start-up and Initialization"**) invokes a number of weak functions directly and provides weak handlers for bootstrap exceptions and general exceptions: `_on_reset`, `_nmi_handler`, `_bootstrap_exception_handler`, `_general_exception_handler`, and `_on_bootstrap`.

The standard library function `exit` calls the weak function `_exit` prior to returning.

The standard library functions for signals, `signal` and `raise`, are implemented as weak functions which always fail.

The standard library functions for locales, `setlocale` and `localeconv`, are implemented as weak functions which do nothing.

The standard library function for accessing environment variables, `getenv`, is implemented as a weak function which always returns null.

2.5 "HELPER" HEADER FILES

2.5.1 sys/attribs.h

Macros are provided for many commonly used attributes in order to enhance user code readability.

`__section__(s)`	Apply the `section` attribute with section name `s`.
`__unique_section__`	Apply the `unique_section` attribute.
`__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 `longcall` attribute.
`__longcall__`	Apply the `longcall` attribute.
`__ISR(v,ipl)`	Apply the `interrupt` attribute with priority level `ipl` and the `vector` attribute with vector number `v`.
`__ISR_AT_VECTOR(v,ipl)`	Apply the `interrupt` attribute with priority level `ipl` and the `at_vector` attribute with vector number `v`.
`__ISR_SINGLE__`	Specifies a function as an Interrupt Service Routine in single-vector mode. This places a jump at the single-vector location to the interrupt handler.
`__ISR_SINGLE_AT_VECTOR__`	Places the entire single-vector interrupt handler at the vector 0 location. When used, ensure that the vector spacing is set to accommodate the size of the handler.

2.5.2 sys/kmem.h

System code may need to translate between virtual and physical addresses, as well as between kernel segment addresses. Macros are provided to make these translations easier and to determine the segment an address is in.

`KVA_TO_PA(v)`	Translate a kernel virtual address to a physical address.
`PA_TO_KVA0(pa)`	Translate a physical address to a KSEG0 virtual address.
`PA_TO_KVA1(pa)`	Translate a physical address to a KSEG1 virtual address.
`KVA0_TO_KVA1(v)`	Translate a KSEG0 virtual address to a KSEG1 virtual address.
`KVA1_TO_KVA0(v)`	Translate a KSEG1 virtual address to a KSEG0 virtual address.
`IS_KVA(v)`	Evaluates to 1 if the address is a kernel segment virtual address, zero otherwise.
`IS_KVA0(v)`	Evaluate to 1 if the address is a KSEG0 virtual address, zero otherwise.
`IS_KVA1(v)`	Evaluate to 1 if the address is a KSEG1 virtual address, zero otherwise.
`IS_KVA01(v)`	Evaluate to 1 if the address is either a KSEG0 or a KSEG1 virtual address, zero otherwise.

2.6   MULTILIBS

2.6.1   What are Multilibs?

With multilibs, target libraries are built multiple times with a permutated set of options. Multilibs are the resulting set of target libraries that are built with these options. When the compiler shell is called to compile and link an application, the shell chooses the version of the target library that has been built with the same options.

2.6.2   What Multilibs are Available for 32-Bit Language Tools?

The target libraries that are distributed with the compiler are built for the following options:

- Size versus speed (`-Os` vs. `-O3`)
- 16-bit versus 32-bit (`-mips16` vs. `-mno-mips16`)
- Software floating-point versus no floating-point support (`-msoft-float` vs. `-mno-float`)

By default the 32-bit language tools compile for `-O0`, `-mno-mips16`, and `-msoft-float`. Therefore, the options that we are concerned about are `-Os` or `-O3`, `-mips16`, and `-mno-float`. Libraries built with the following command line options are made available:

1.   Default command line options
2.   `-Os`
3.   `-O3`
4.   `-mips16`
5.   `-mno-float`
6.   `-mips16 -mno-float`
7.   `-Os -mips16`
8.   `-Os -mno-float`
9.   `-Os -mips16 -mno-float`
10. `-O3 -mips16`
11. `-O3 -mno-float`
12. `-O3 -mips16 -mno-float`

2.6.3   Where are the Multilibs Directories?

By default, the 32-bit language tools use the directory
`<install-directory>/lib/gcc/` to store the specific libraries and the directory
`<install-directory>/<pic32mx>/lib` to store the target-specific libraries. Both
of these directory structures contain subdirectories for each of the multilib combinations
specified above. These subdirectories, respectively, are as follows:

```
1.  .
2.  ./size
3.  ./speed
4.  ./mips16
5.  ./no-float
6.  ./mips16/no-float
7.  ./size/mips16
8.  ./size/no-float
9.  ./size/mips16/no-float
10../speed/mips16
11../speed/no-float
12../speed/mips16/no-float
```

2.6.4   Which Multilib Directory Are Selected?

This section looks at examples and provide details on which of the multilibs
subdirectories are chosen.

1. `pic32-gcc foo.c`

   For this example, no command line options have been specified (i.e., the default
   command line options are being used). In this case, the `.` subdirectories are
   used.

2. `pic32-gcc -Os foo.c`

   For this example, the command line option for optimizing for size has been
   specified (i.e., `-Os` is being used). In this case, the `./size` subdirectories are
   used.

3. `pic32-gcc -O2 foo.c`

   For this example, the command line option for optimizing has been specified,
   however, this command line option optimizes for neither size nor space (i.e., `-O2`
   is being used). In this case, the `.` subdirectories are used.

4. `pic32-gcc -Os -mips16 foo.c`

   For this example, the command line options for optimizing for size and for
   MIPS16 code have been specified (i.e., `-Os` and `-mips16` are being used). In
   this case, the `./size/mips16` subdirectories are used.

Chapter 3. Interrupts

3.1   INTRODUCTION

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.

PIC32MX 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 or inline assembly code. This chapter presents an overview of interrupt processing.

3.2   HIGHLIGHTS

Items discussed in this chapter are:

• Specifying an Interrupt Handler Function
• Associating a Handler Function with an Exception Vector
• Exception Handlers

3.3   SPECIFYING AN INTERRUPT HANDLER FUNCTION

An interrupt handler function handles the context save and restore to ensure that upon return from interrupt, the program context is maintained.

3.3.1   Handler Function Context Saving

The standard calling convention for C functions will already preserve `zero`, `s0-s7`, `gp`, `sp`, and `fp`. `k0` and `k1` are used by the compiler to access and preserve non-GPR context, but are always accessed atomically (i.e., in sequences with global interrupts disabled), so they need not be preserved actively. In addition to the standard registers, a handler function will actively preserve the `a0-a3`, `t0-t9`, `v0`, `v1` and `ra` registers.

An interrupt handler function will also actively save and restore processor status registers that are utilized by the handler function. Specifically, the `EPC`, `SR`, `hi` and `lo` registers are preserved as context.

Handler functions specified as priority level 7 (highest priority) will use a shadow register set to preserve the General Purpose Registers, enabling lower latency entry into the application code of the handler function.

3.3.2   Marking a Function as an Interrupt Handler

A function is marked as a handler function via either the interrupt attribute or the interrupt pragma[1]. Each method is functionally equivalent to the other. The interrupt is specified as handling interrupts of a specific priority level or for operating in single vector mode.

---

1. Note that pre-processor macros are not expanded in pragma directives.

3.3.2.1 INTERRUPT ATTRIBUTE

```
__attribute__((interrupt([IPLn[SRS|SOFT|AUTO]]))))
```

Where *n* is in the range of 0..7, inclusive.

In early PIC32 devices, the shadow register set is hard coded to priority level 7 (IPL7). For these devices, we specify `__attribute__((interrupt(IPLn)))`, where *n* is the priority level. For IPL7, the compiler generates function prologue and epilogue code utilizing the shadow register set for context saving. The compiler generates software context-saving code when the user specifies lower IPL values.

Later PIC32 devices allow us to select, via configuration settings, which priority will use the shadow register set. (Refer to the device data sheet to determine if your PIC32 device supports this feature.) This means we must specify which context-saving mechanism to use for each Interrupt Service Routine. The compiler will generate interrupt prologue and epilogue code utilizing shadow register context saving for IPL*n*SRS. It will use software context saving for IPL*n*SOFT.

> **Note:** Application code is responsible for applying the correct value to the matching handler routine.

The compiler also supports an IPLnAUTO IPL specifier that uses the run-time value in SRSCTL to determine whether it should use software or SRS context-saving code. The compiler defaults to using IPLnAUTO when the IPL specifier is omitted from the `interrupt()` attribute.

> **Note:** SRS has the shortest latency and SOFT has a longer latency due to registers saved on the stack. AUTO adds a few cycles to test if SRS or SOFT should be used.

3.3.2.2 INTERRUPT PRAGMA

 # pragma interrupt *function-name* IPL*n*[AUTO|SOFT|SRS] [**vector** [**@**]*vector-number* [, *vector-number-list*]]
 # pragma interrupt *function-name* **single** [**vector** [**@**] 0

Where *n* is in the range of 0..7, inclusive. The IPL*n* specifier may be all uppercase or all lowercase.

The function definition for a handler function indicated by an interrupt pragma must follow in the same translation unit as the pragma itself.

The `interrupt` attribute will also indicate that a function definition is an interrupt handler. It is functionally equivalent to the interrupt pragma.

For example, the definitions of `foo` below both indicate that it is an interrupt handler function for an interrupt of priority 4 that uses software context saving.

```
#pragma interrupt foo IPL4SOFT
void foo (void)
```

is functionally equivalent to

```
void __attribute__ ((interrupt(IPL4SOFT))) foo (void)
```

3.4 ASSOCIATING A HANDLER FUNCTION WITH AN EXCEPTION VECTOR

There are 64 exception vectors, numbered 0..63 inclusive. Each interrupt source is mapped to an exception vector as specified in the device data sheet. By default, four words of space are reserved at each vector address for a dispatch to the handler function for that exception source.

An interrupt handler function can be associated with an interrupt vector either as the target of a dispatch function located at the exception vector address, or as being located directly at the exception vector address. A single handler function can be the target of multiple dispatch functions.

The association of a handler function to one or more exception vector addresses is specified via a clause of the interrupt pragma, a separate vector pragma, or a vector attribute on the function declaration.

3.4.1   Interrupt Pragma Clause

The interrupt pragma has an optional `vector` clause following the priority specifier.

 # pragma interrupt *function-name ipl-specifier* [**vector**
[**@**]*vector-number* [, *vector-number-list*]]

A dispatch function targeting the specified handler function will be created at the exception vector address for the specified vector numbers. If the first vector number is specified with a preceding "@" symbol, the handler function itself will be located there directly.

For example, the following pragma specifies that function `foo` will be created as an interrupt handler function of priority four. `foo` will be located at the address of exception vector 54. A dispatch function targeting `foo` will be created at exception vector address 34.

```
#pragma interrupt foo ipl4 vector @54, 34
```

The following pragma specifies that function `bar` will be created as an interrupt handler function of priority five. `bar` will be located in general purpose program memory (.text section). A dispatch function targeting `bar` will be created at exception vector address 23.

```
#pragma interrupt bar ipl5 vector 23
```

3.4.2   Vector Pragma

The `vector` pragma creates one or more dispatch functions targeting the indicated function. For target functions specified with the `interrupt` pragma, this functions as if the vector clause had been used. The target function of a `vector` pragma can be any function, including external functions implemented in assembly or by other means.

 # pragma vector *function-name* **vector** *vector-number* [,
vector-number-list]

The following pragma defines a dispatch function targeting `foo` at exception vector address 54.

```
#pragma vector foo 54
```

3.4.3   Vector Attribute

A handler function can be associated with one or more exception vector addresses via an attribute. The `at_vector` attribute indicates that the handler function should itself be placed at the exception vector address. The `vector` attribute indicates that a dispatch function should be created at the exception vector address(es).

For example, the following declaration specifies that function `foo` will be created as an interrupt handler function of priority four. `foo` will be located at the address of exception vector 54.

```
void __attribute__ ((interrupt(ipl4))) __attribute__ ((at_vector(54)))
foo (void)
```

The following declaration specifies that function `foo` will be created as an interrupt handler function of priority four. Define dispatch functions targeting `foo` at exception vector addresses 52 and 53.

```
void __attribute__ ((interrupt(ipl4))) __attribute__ ((vector(53,
52))) foo (void)
```

3.4.4   __ISR Macros

The `<sys/attribs.h>` header file provides macros intended to simplify the application of attributes to functions. There are also vector macros defined in the processor header files. (See the appropriate header file in the compiler install directory.)

3.4.4.1   __ISR(v, ipl)

Use the `__ISR(v, ipl)` macro to assign the vector-number location and associate it with the software-assigned interrupt priority. This will place a jump to the interrupt handler at the associated vector location. This macro also applies the nomips16 attribute since ISR functions are required to be MIPS32.

EXAMPLE 3-1:   CORE TIMER VECTOR, IPL2SOFT

```
void __ISR(_CORE_TIMER_VECTOR, IPL2SOFT) CoreTimerHandler(void);
```

Example 3-1 creates an interrupt handler function for the core timer interrupt that has an interrupt priority level of two. The compiler places a dispatch function at the associated vector location. To reach this function, the core timer interrupt flag and enable bits must be set, and the interrupt priority should be set to a level of two. The compiler generates software context-saving code for this handler function.

EXAMPLE 3-2:   CORE SOFTWARE 0 VECTOR, IPL3SRS

```
void __ISR(_CORE_SOFTWARE_0_VECTOR,IPL3SRS)
_CoreSoftwareInt0Handler(void);
```

Example 3-2 creates an interrupt handler function for the core software interrupt 0 that has an interrupt priority level of three. The compiler places a dispatch function at the associated vector location. To reach this function, the core software interrupt flag and enable bits must be set, and the interrupt priority should be set to a level of three. The device configuration fuses must assign Shadow Register Set 1 to interrupt priority level three. The compiler generates code that assumes that register context will be saved in SRS1.

EXAMPLE 3-3:   CORE SOFTWARE 1 VECTOR, IPL0AUTO

```
void __ISR(_CORE_SOFTWARE_1_VECTOR, IPL0AUTO)
_CoreSoftwareInt1Handler(void);
```

Example 3-3 creates an interrupt handler function for the core software interrupt 1 that has an interrupt priority level of zero. The compiler places a dispatch function at the associated vector location. To reach this function, the core software interrupt 1 flag and enable bits must be set, and the interrupt priority should be set to a level of zero. The compiler generates code that determines at run time whether software context saving is required.

EXAMPLE 3-4:   CORE SOFTWARE 1 VECTOR, DEFAULT

```
void __ISR(_CORE_SOFTWARE_1_VECTOR) _CoreSoftwareInt1Handler(void);
```

Example 3-4 is functionally equivalent to Example 3. Because the IPL specifier is omitted, the compiler assumes `IPL0AUTO`.

3.4.4.2   __ISR_AT_VECTOR(v, ipl)

Use the `__ISR_AT_VECTOR(v, ipl)` to place the entire interrupt handler at the vector location and associate it with the software-assigned interrupt priority. Application code is responsible for making sure that the vector spacing is set to accommodate the size of the handler. This macro also applies the nomips16 attribute since ISR functions are required to be MIPS32.

EXAMPLE 3-5:   CORE TIMER VECTOR, IPL2SOFT

```
void __ISR_AT_VECTOR(_CORE_TIMER_VECTOR, IPL2SOFT)
CoreTimerHandler(void);
```

Example 3-5 creates an interrupt handler function for the core timer interrupt that has an interrupt priority level of two. The compiler places the entire interrupt handler at the vector location. It does not use a dispatch function. To reach this function, the core timer interrupt flag and enable bits must be set, and the interrupt priority should be set to a level of two. The compiler generates software context-saving code for this handler function.

3.4.4.3   INTERRUPT-VECTOR MACROS

Each processor-support header file provides a macro for each interrupt-vector number (for example, `\pic32mx\include\proc\p32mx360f512l.h`. See the appropriate header file in the compiler install directory). When used in conjunction with the `__ISR()` macro provided by the `sys\attribs.h` header file, these macros help make an Interrupt Service Routine easier to write and maintain.

EXAMPLE 3-6:   INTERRUPT-VECTOR WITH HANDLER

```
void __ISR (_TIMER_1_VECTOR, IPL7SRS) Timer1Handler (void);
```

Example 3-6 creates an interrupt handler function for the Timer 1 interrupt that has an interrupt priority level of seven. The compiler places a dispatch function at the vector location associated with macro `_TIMER_1_VECTOR` as defined in the device-specific header file. To reach this function, the Timer 1 interrupt flag and enable bits must be set, and the interrupt priority should be set to a level of seven. For devices that allow assignment of shadow registers to specific IPL values, the device Configuration bit settings must assign Shadow Register Set 1 to interrupt priority level seven. The compiler generates code that assumes that register context will be saved in SRS1.

3.5   EXCEPTION HANDLERS

The PIC32MX devices also have two exception vectors for non-interrupt exceptions. These exceptions are grouped into bootstrap exceptions and general exceptions.

3.5.1   Bootstrap Exception

A reset exception is any exception which occurs while bootstrap code is running ($Status_{BEV}=1$). All reset exceptions are vectored to `0xBFC00380`.

At this location the 32-bit toolchain places a branch instruction targeting a function named `_bootstrap_exception_handler()`. In the standard library, a default weak version of this function is provided which merely goes into an infinite loop. If the user application provides an implementation of `_bootstrap_exception_handler()`, that implementation will be used instead.

3.5.2 General Exception

A general exception is any non-interrupt exception which occurs during program execution outside of bootstrap code ($Status_{BEV}=0$). General exceptions are vectored to offset `0x180` from `EBase`.

At this location the 32-bit toolchain places a branch instruction targeting a function named `_general_exception_context()`. The provided implementation of this function saves context, calls an application handler function, restores context and performs a return from the exception instruction. The context saved is the `hi` and `lo` registers and all General Purpose Registers except `s0-s8`, which are defined to be preserved by all called functions and so are not necessary to actively save again here. The values of the `Cause` and `Status` registers are passed to the application handler function (`_general_exception_handler()`). If the user application provides an implementation of `_general_exception_context()`, that implementation will be used instead.

```
void _general_exception_handler (unsigned cause, unsigned status);
```

A weak default implementation of `_general_exception_handler()` is provided in the standard library which merely goes into an infinite loop. If the user application provides an implementation of `_general_exception_handler()`, that implementation will be used instead.

Chapter 4. Low-Level Processor Control

4.1 INTRODUCTION

This chapter discusses access to the low-level registers and configuration of the PIC32MX devices.

4.2 HIGHLIGHTS

Items discussed in this chapter are:

- Generic Processor Header File
- Processor Support Header Files
- Peripheral Library Functions
- Special Function Register Access
- CP0 Register Access
- Configuration Bit Access

4.3 GENERIC PROCESSOR HEADER FILE

The generic processor header file is a C file that includes the correct processor-specific header file based on the processor specified with the `-mprocessor` command line option. The generic processor header file is located in
`c:\Program Files\Microchip\MPLAB C32\pic32mx\include`, where
`c:\Program Files\Microchip\MPLAB C32` is the directory in which the 32-bit toolchain was installed. Besides including the correct processor-specific header file, the generic processor header file also provides `#define`s which allow the use of conventional register names from within assembly language files.

To include the generic processor header file, use the following from within your source code:

```
#include <p32xxxx.h>
```

Inclusion of the generic processor header file allows your source code to be compiled for any of the processors supported by the 32-bit toolchain without having to change the file which is being included.

4.4 PROCESSOR SUPPORT HEADER FILES

The processor-specific header files are files that contain external declarations for the Special Function Registers (SFRs) for use in either C or assembly. By convention, each SFR is named using the same name that appears in the data sheet – for example, `WDTCON` for the Watchdog Timer Control register. If the register has individual bits that may be of interest, there is also be a structure `typedef` defined for that SFR, where the name of the structure `typedef` is the name of the register with `bits_t` appended – for example, `__WDTCONbits_t`. The individual bits (or bit fields) are named in the structure using the names in the data sheet. For example, in the PIC32MX360F512L processor-specific header file, the `WDTCON` register for use with C is declared as:

```
extern volatile unsigned int WDTCON __attribute__((section("sfrs")));
typedef union {
  struct {
    unsigned WDTCLR:1;
    unsigned :1;
    unsigned SWDTPS0:1;
    unsigned SWDTPS1:1;
    unsigned SWDTPS2:1;
    unsigned SWDTPS3:1;
    unsigned SWDTPS4:1;
    unsigned :8;
    unsigned ON:1;
  };
  struct {
    unsigned :2;
    unsigned WDTPSTA:5;
    unsigned :1;
    unsigned PWRTPSTA:3;
  };
  struct {
    unsigned w:32;
  };

} __WDTCONbits_t;
extern volatile __WDTCONbits_t WDTCONbits asm ("WDTCON") __attribute__((section("sfrs")));
```

> **Note:** The symbols `WDTCON` and `WDTCONbits` refer to the same register and
> resolve to the same address as can be seen by the declaration for
> `WDTCONbits`.

For use with assembly, the WDTCON register is declared as: `.extern WDTCON`.

The processor-specific header files are located in
`c:\Program Files\Microchip\MPLAB C32\pic32mx\include\proc`, where
`c:\Program Files\Microchip\MPLAB C32` is the directory in which the 32-bit
toolchain was installed. To include a processor-specific header file, it is recommended
that you include the generic processor header file (see **Section 4.3 "Generic
Processor Header File"**), however, if you would like to specifically call out the
processor-specific header file, use the following from your source file (example
assumes inclusion of the processor-specific header file for the PIC32MX360F512L):

`#include <proc/p32mx360f512l.h>`

4.5   PERIPHERAL LIBRARY FUNCTIONS

Many of the peripherals of the PIC32MX devices are supported by the peripheral library
functions provided with the compiler tools. See the "*32-Bit Language Tools Libraries*"
(DS51685) for details on the functions provided.

4.6   SPECIAL FUNCTION REGISTER ACCESS

There are three steps to follow when using SFRs in an application.

1. Include either the generic processor header file (i.e., `p32xxxx.h`) or the processor-specific header file for the appropriate device (e.g., `proc/p32mx360f512l.h`).
    ```
    #include <p32xxxx.h>
    ```
2. Access SFRs like any other C variables. The source code can write to and/or read from the SFRs. For example, the following statement clears all the bits to zero in the Special Function Register for Timer 1:
    ```
    TMR1 = 0;
    ```

    The next statement enables the Watchdog Timer:
    ```
    WDTCONbits.ON = 1;
    ```
3. Link with the default linker script or include the `processor.o` file for the appropriate processor in your project.

4.7   CP0 REGISTER ACCESS

4.7.1   CP0 Register Definitions Header File

The CP0 register definitions header file (`cp0defs.h`) is a file that contains definitions for the CP0 registers and their fields. In addition, it contains macros for accessing the CP0 registers. The CP0 register definitions header file is located in `c:\Program Files\Microchip\MPLAB C32\pic32mx\include`, where `c:\Program Files\Microchip\MPLAB C32` is the directory in which the 32-bit toolchain was installed. The CP0 register definitions header file was designed to work with either Assembly or C files.

The CP0 register definitions header file is dependent on macros defined within the processor generic header file (See **Section 4.3 "Generic Processor Header File"**). To include the CP0 register definitions header file, use the following from within your source code:

```
#include <p32xxxx.h>
```

4.7.2   CP0 Register Definitions

When the CP0 register definitions header file is included from an Assembly file, the CP0 registers are defined as:

```
#define _CP0_REGISTER_NAME $register_number, select_number
```

For example, the `IntCtl` register is defined as:

```
#define _CP0_INTCTL $12, 1
```

When the CP0 register definitions header file is included from a C file, the CP0 registers and selects are defined as:

```
#define _CP0_REGISTER_NAME register_number
#define _CP0_REGISTER_NAME_SELECT select_number
```

For example, the `IntCtl` register is defined as:

```
#define _CP0_INTCTL 12
#define _CP0_INTCTL_SELECT 1
```

4.7.3   CP0 Register Field Definitions

When the CP0 register definitions header file is included from either an Assembly or a C file, three `#defines` exist for each of the CP0 register fields.

`_CP0_REGISTER_NAME_FIELD_NAME_POSITION` – the starting bit location

_CP0_*REGISTER_NAME_FIELD_NAME*_MASK – the bits that are part of this field are set

_CP0_*REGISTER_NAME_FIELD_NAME*_LENGTH – the number of bits that this field occupies

For example, the vector spacing field of the `IntCtl` register has the following defines:

```
#define _CP0_INTCTL_VS_POSITION 0x00000005
#define _CP0_INTCTL_VS_MASK     0x000003E0
#define _CP0_INTCTL_VS_LENGTH   0x00000005
```

4.7.4   CP0 Access Macros

When the CP0 register definitions header file is included from a C file, CP0 access macros are defined. Each CP0 register may have up to six different access macros defined:

`_CP0_GET_`*`REGISTER_NAME`*` ()`	Returns the value for register, *REGISTER_NAME*.
`_CP0_SET_`*`REGISTER_NAME`*` (val)`	Sets the register, *REGISTER_NAME*, to val, and returns void. Only defined for registers that contain a writable field.
`_CP0_XCH_`*`REGISTER_NAME`*` (val)`	Sets the register, *REGISTER_NAME*, to val, and returns the previous register value. Only defined for registers that contain a writable field.
`_CP0_BIS_`*`REGISTER_NAME`*` (set)`	Sets the register, *REGISTER_NAME*, to (reg |= set), and returns the previous register value. Only defined for registers that contain writable bit fields.
`_CP0_BIC_`*`REGISTER_NAME`*` (clr)`	Sets the register, *REGISTER_NAME*, to (reg &= ~clr), and returns the previous register value. Only defined for registers that contain writable bit fields.
`_CP0_BCS_`*`REGISTER_NAME`*` (clr, set)`	Sets the register, *REGISTER_NAME*, to (reg = (reg & ~clr) | set), and returns the previous register value. Only defined for registers that contain writable bit fields.

4.8   CONFIGURATION BIT ACCESS

4.8.1   #pragma config

The `#pragma config` directive specifies the processor-specific configuration settings (i.e., Configuration bits) to be used by the application. Refer to the "PIC32MX Configuration Settings" online help (found under *MPLAB IDE>Help>Topics>Language Tools*) for more information. (If using the compiler from the command line, this help file is located at the default location at `C:\Program Files\Microchip\MPLAB C32\doc\hlpPIC32MXConfigSet.chm`.).

Configuration settings may be specified with multiple `#pragma config` directives. The compiler verifies that the configuration settings specified are valid for the processor for which it is compiling. If a given setting in the Configuration Word has not been specified in any `#pragma config` directive, the bits associated with that setting default to the unprogrammed value.

For each Configuration Word for which a setting is specified with the `#pragma config` directive, the compiler generates a read-only data section named `.config_`*address*, where *address* is the hexadecimal representation of the address of the Configuration Word. For example, if a configuration setting was specified for the Configuration Word located at address `0xBFC02FFC`, a read-only data section named `.config_BFC02FFC` would be created.

4.8.1.1   SYNTAX

pragma-config-directive:

      # pragma config *setting-list*

setting-list:

   setting

 | *setting-list*, *setting*

setting:

   setting-name **=** *value-name*

The *setting-name* and *value-name* are device specific and can be determined by utilizing the *PIC32MX Configuration Settings* document.

4.8.1.2   EXAMPLE

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

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

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

NOTES:

Chapter 5. Compiler Run-time Environment

5.1 INTRODUCTION

This chapter discusses the compiler run-time environment.

5.2 HIGHLIGHTS

Items discussed in this chapter are:

- Register Conventions
- Stack Usage
- Heap Usage
- Function Calling Convention
- Start-up and Initialization
- Contents of the Default Linker Script
- RAM Functions

5.3 REGISTER CONVENTIONS

TABLE 5-1:   REGISTER CONVENTIONS

Register Name	Software Name	Use
$0	zero	Always 0 when read.
$1	at	Assembler temporary variable.
$2-$3	v0-v1	Return value from functions.
$4-$7	a0-a3	Used for passing arguments to functions.
$8-$15	t0-t7	Temporary registers used by compiler for expression evaluation. Values not saved across function calls.
$16-$23	s0-s7	Temporary registers whose values are saved across function calls.
$24-$25	t8-t9	Temporary registers used by compiler for expression evaluation. Values not saved across function calls.
$26-$27	k0-k1	Reserved for interrupt/trap handler.
$28	gp	Global Pointer.
$29	sp	Stack Pointer.
$30	fp or s8	Frame Pointer if needed. Additional temporary saved register if not.
$31	ra	Return address for functions.

5.4   STACK USAGE

The compiler dedicates General Purpose Register 29 as the software Stack Pointer. All processor stack operations, including function call, interrupts and exceptions use the software stack. The stack grows downward from high addresses to low addresses.

By default, the size of the stack is 1024 bytes. The size of the stack may be changed by specifying the size on the linker command line using the `--defsym_min_stack_size` linker command line option. An example of allocating a stack of 2048 bytes using the command line is:

```
pic32-gcc foo.c -Wl,--defsym,_min_stack_size=2048
```

The run-time stack grows downward from higher addresses to lower addresses (see Figure 5-1). The compiler uses two working registers to manage the stack:

- Register 29 (`sp`) – This is the Stack Pointer. It points to the next free location on the stack.
- Register 30 (`fp`) – This is the Frame Pointer. It points to the current function's frame. Each function, if required, creates a new frame from which automatic and temporary variables are allocated. Compiler optimization may eliminate Stack Pointer references via the Frame Pointer to equivalent references via the Stack Pointer. This optimization allows the Frame Pointer to be used as a General Purpose Register.

FIGURE 5-1:   STACK FRAME
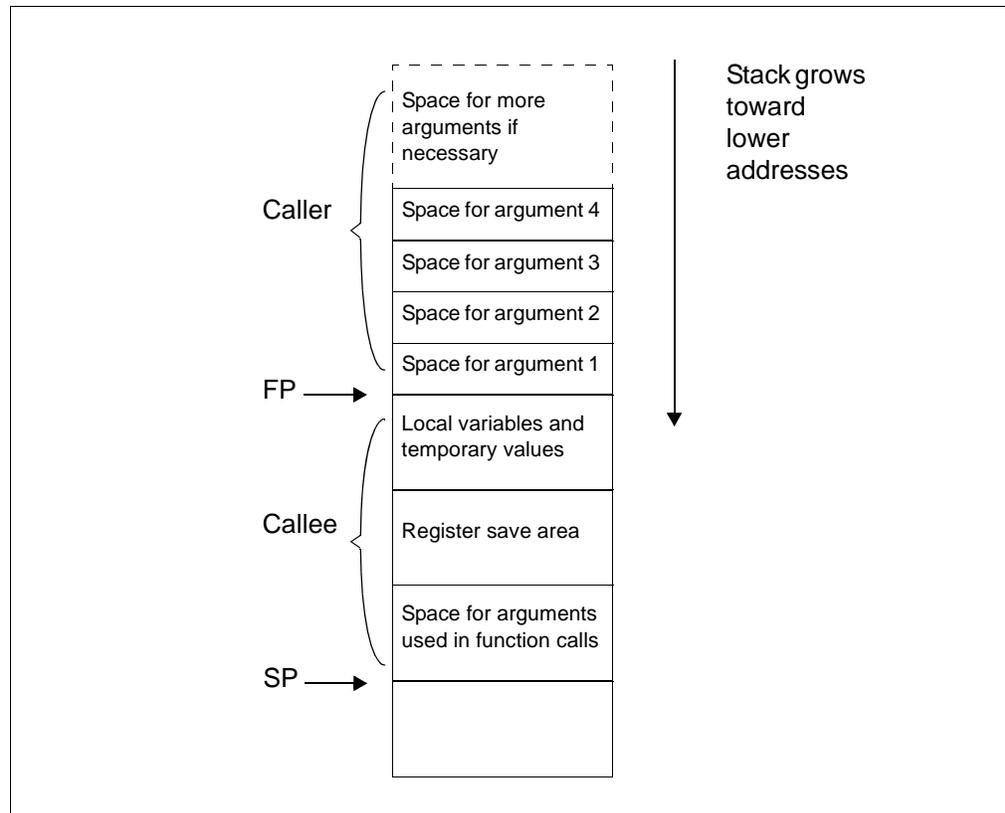
5.5   HEAP USAGE

The C 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`. 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.

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:

```
pic32-gcc foo.c -Wl,--defsym,_min_heap_size=512
```

The linker allocates the heap immediately before the stack.

5.6   FUNCTION CALLING CONVENTION

The Stack Pointer is always aligned on a 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 `a0-a3` (see Table 5-2 for how many registers are required for each data type).
- Although some arguments may be passed in registers, space is still allocated on the stack for all arguments to be passed to a function (see Figure 5-2).
- When calling a function:
  - Registers `a0-a3` are used for passing arguments to functions. Values in these registers are not preserved across function calls.
  - Registers `t0-t7` and `t8-t9` are caller saved registers. The calling function must push these values onto the stack for the registers' values to be saved.
  - Registers `s0-s7` are called saved registers. The function being called must save any of these registers it modifies.
  - Register `s8` is a saved register if the optimizer eliminates its use as the Frame Pointer. `s8` is a reserved register otherwise.
  - Register `ra` contains the return address of a function call.

TABLE 5-2:   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.

MPLAB® C Compiler for PIC32 MCUs User's Guide

FIGURE 5-2:   **PASSING ARGUMENTS**

Example 1:

```
int add (int, int)
a= add (5, 10);
```

SP + 4	undefined	a0	5
SP	undefined	a1	10
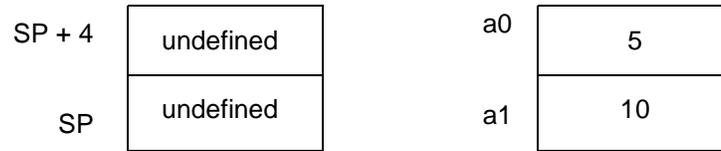
Example 2:

```
void foo (double, double)
call= foo (10.5, 20.1);
```

SP + 12	undefined	a0	10.5
SP + 8		a1	
SP + 4	undefined	a2	20.1
SP		a3	

Example 3:

```
void calculate (double, double, int)
calculate (50.3, 100.0, .10);
```

	.10		
SP + 16			
SP + 12	undefined	a0	50.3
SP + 8		a1	
SP + 4	undefined	a2	100.0
SP		a3	
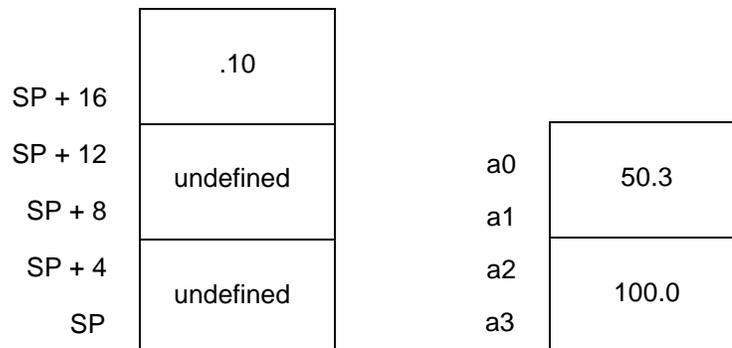
5.7   START-UP AND INITIALIZATION

5.7.1   Provisions

The following provisions are made regarding the run-time model:

- Kernel mode only
- KSEG1 only
- RAM functions are attributed with `__ramfunc__` or `__longramfunc__`, meaning that all RAM functions end up in the `.ramfunc` section

5.7.2   PIC32MX Start-up Code

The PIC32MX start-up code must perform the following:

1. Jump to NMI Handler if an NMI Occurred
2. Initialize Stack Pointer and Heap
3. Initialize Global Pointer
4. Call "On Reset" Procedure
5. Clear Uninitialized Data Sections
6. Copy Initialized Data from Program Flash to Data Memory
7. Copy RAM Functions from Program Flash to Data Memory
8. Initialize Bus Matrix Registers
9. Initialize CP0 Registers
10. Trace Control 2 Register (`TraceControl2` – CP0 Register 23, Select 2)
11. Call "On Bootstrap" Procedure
12. Change Location of Exception Vectors
13. Call Main

5.7.2.1   JUMP TO NMI HANDLER IF AN NMI OCCURRED

If an NMI caused entry to the Reset vector, a jump to an NMI handler procedure (`_nmi_handler`) occurs. A weak version of the NMI handler procedure is provided that performs an `ERET`. The `_nmi_handler` function must be attributed with `nomips16` [e.g., `__attribute__((nomips16))`] since the start-up code jumps to this function.

5.7.2.2   INITIALIZE STACK POINTER AND HEAP

The Stack Pointer (`sp`) register must be initialized in the start-up code. To enable the start-up code to initialize the `sp` register, the linker script must initialize a variable which points to the end of KSEG1 data memory[1]. This variable is named `_stack`. The user can change the minimum amount of stack space allocated by providing the command line option `--defsym _min_stack_size=N` to the linker. `_min_stack_size` is provided by the linker script with a default value of `1024`.

On a similar note, the user may wish to utilize a heap with their application. While the start-up code does not need to initialize the heap, the standard C libraries (`sbrk`) must be made aware of the heap location and its size. The linker script creates a variable to identify the beginning of the heap. The location of the heap is the end of the utilized KSEG1 data memory. This variable is named `_heap`. The user can change the minimum amount of heap space allocated by providing the command line option `--defsym _min_heap_size=M` to the linker. `_min_heap_size` is provided by the

---

1. The end of data memory are different based on whether RAM functions exist. If RAM functions exist, then part of the DRM must be configured for kernel program to contain the RAM functions, and the Stack Pointer is located one word prior to the beginning of the DRM kernel program boundary address. If RAM functions do not exist, then the Stack Pointer is located at the true end of DRM.

linker script with a default value of `0`. If the heap is used when the heap size is set to zero, the behavior is the same as when the heap usage exceeds the minimum heap size. Namely, it overflows into the space allocated for the stack.

The heap and the stack use the unallocated KSEG1 data memory, with the heap starting from the end of allocated KSEG1 data memory and growing upwards towards the stack while the stack starts at the end of KSEG1 data memory and grows downwards towards the heap. If enough space is not available based on the minimum amount of heap size and stack size requested, the linker issues an error.

FIGURE 5-3:   STACK AND HEAP LAYOUT
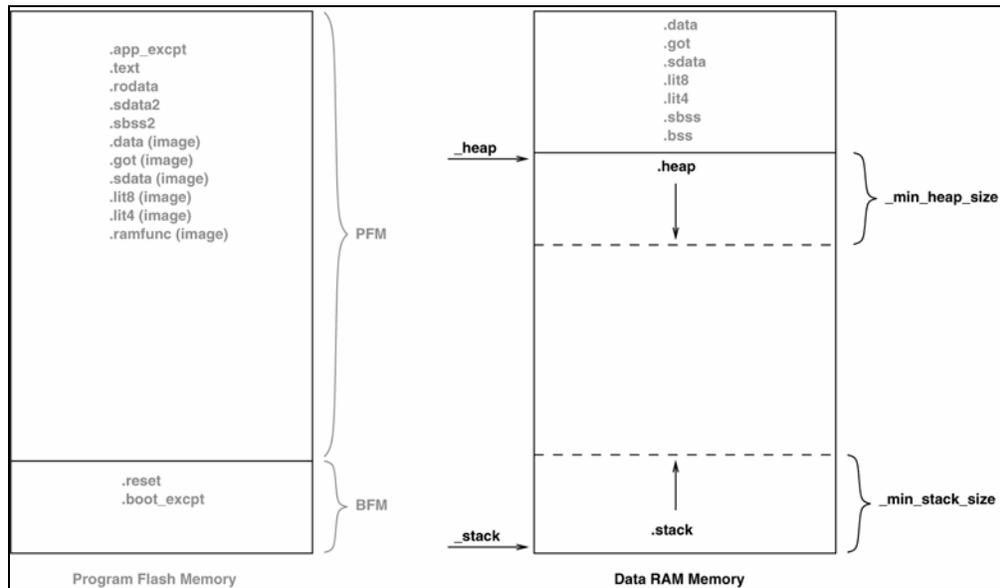
FIGURE 5-4:   STACK AND HEAP LAYOUT WITH RAM FUNCTIONS
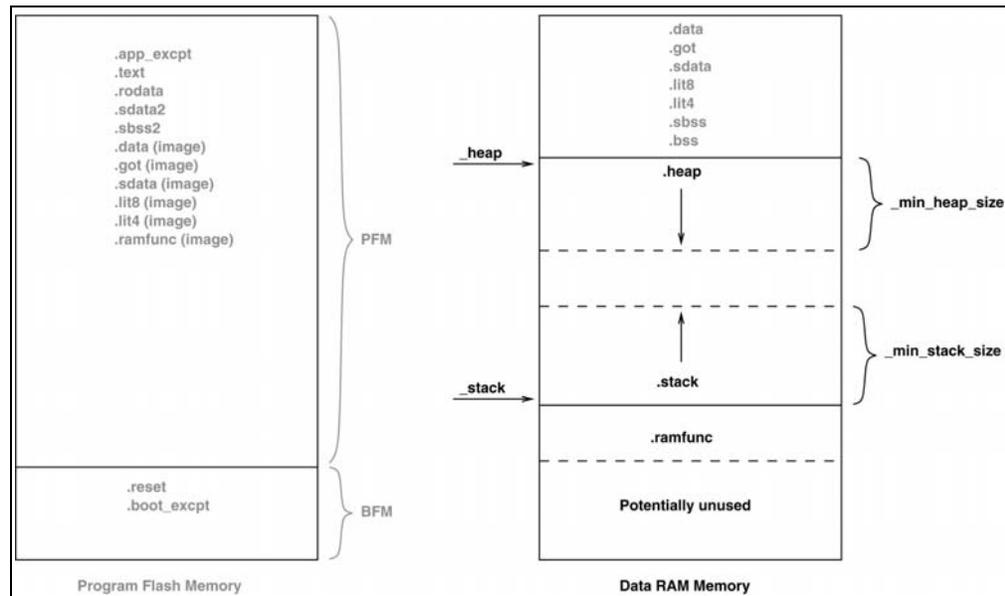
5.7.2.3   INITIALIZE GLOBAL POINTER

The compiler toolchain supports Global Pointer (`gp`) relative addressing. Loads and stores to data lying within 32KB of either side of the address stored in the `gp` register can be performed in a single instruction using the `gp` register as the base register.

Without the Global Pointer, loading data from a static memory area takes two instructions – one to load the Most Significant bits of the 32-bit constant address computed by the compiler/linker and one to do the data load.

To utilize `gp`-relative addressing, the compiler and assembler must group all of the "small" variables and constants into one of the following sections:

- `.lit4.`
- `lit8`
- `.sdata.`
- `sbss`
- `.sdata.*`
- `sbss.*`
- `.gnu.linkonce.s.*`
- `.gnu.linkonce.sb.*`

The linker must then group all of the above input sections together. The run-time start-up code must initialize the `gp` register to point to the "middle" of this output section. To enable the start-up code to initialize the `gp` register, the linker script must initialize a variable which is 32 KB from the start of the output section containing the "small" variables and constants. This variable is named `_gp` (to match core linker scripts). Besides being initialized in the standard GPR set, the Global Pointer must also be initialized in the register shadow set.

FIGURE 5-5: GLOBAL POINTER LOCATION
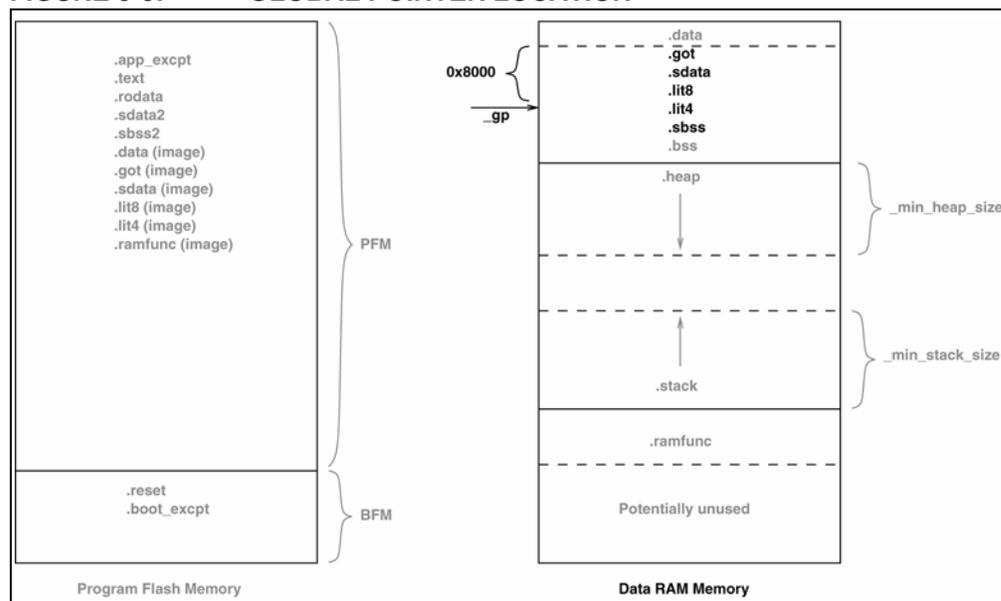
5.7.2.4 CALL "ON RESET" PROCEDURE

A procedure is called after initializing a minimum 'C' context. This procedure allows users to perform actions almost immediately on Reset of the device. An empty weak version of this procedure (`_on_reset`) is provided with the start-up code. Special consideration needs to be taken by the user if this procedure is written in 'C'. Most importantly, statically allocated variables are not initialized (with either the specified initializer or a zero as required for uninitialized variables).

5.7.2.5 CLEAR UNINITIALIZED DATA SECTIONS

There are two uninitialized data sections—`.sbss` and `.bss`. The `.sbss` section is a data segment containing uninitialized variables less than or equal to n bytes where n is determined by the `-G`n command line option. The `.bss` section is a data segment containing uninitialized variables not included in `.sbss`.

The C standard requires that the uninitialized data sections be initialized to 0 on start-up. In order to initialize these sections, the linker script must allocate these sections contiguously and initialize two variables – one for the start address of the uninitialized data section and one for the end address of the uninitialized data section. The start-up code clears all data memory locations between these two addresses. These variables are named `_bss_begin` and `_bss_end`, respectively.

FIGURE 5-6:   UNINITIALIZED DATA
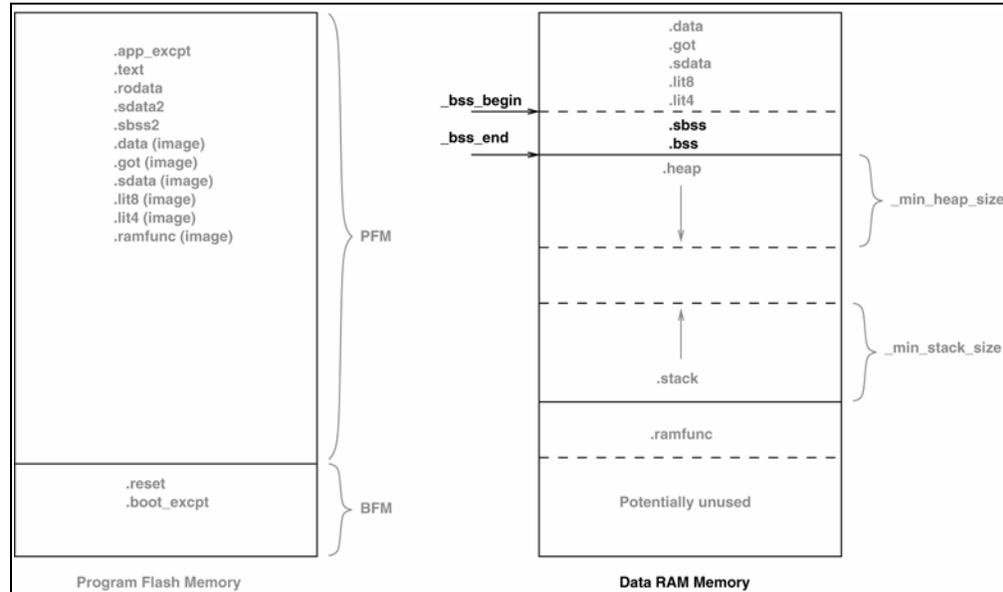
5.7.2.6   COPY INITIALIZED DATA FROM PROGRAM FLASH TO DATA MEMORY

Similar to uninitialized data sections, four initialized data sections exist: `.sdata`, `.data`, `.lit4`, and `.lit8`. The `.sdata` section is a data segment containing initialized variables less than or equal to n bytes where n is determined by the `-Gn` command line option. The `.data` section is a data segment containing initialized variables not included in `.sdata`. The `.lit4` and `.lit8` sections contain constants (usually floating-point) which the assembler decides to store in memory rather than in the instruction stream.

On start-up, a copy of the initialized data exists in the program Flash. This data must be copied to data memory. To facilitate this, the linker script must initialize three variables—one for the start address of the image in program Flash, one for the start address of the section in data memory, and one for the end address of the section in data memory. The start-up code copies all data memory locations from program Flash image to data memory using these variables. These variables are named `_data_image_begin`, `_data_begin`, and `_data_end`, respectively.

FIGURE 5-7:   INITIALIZED DATA
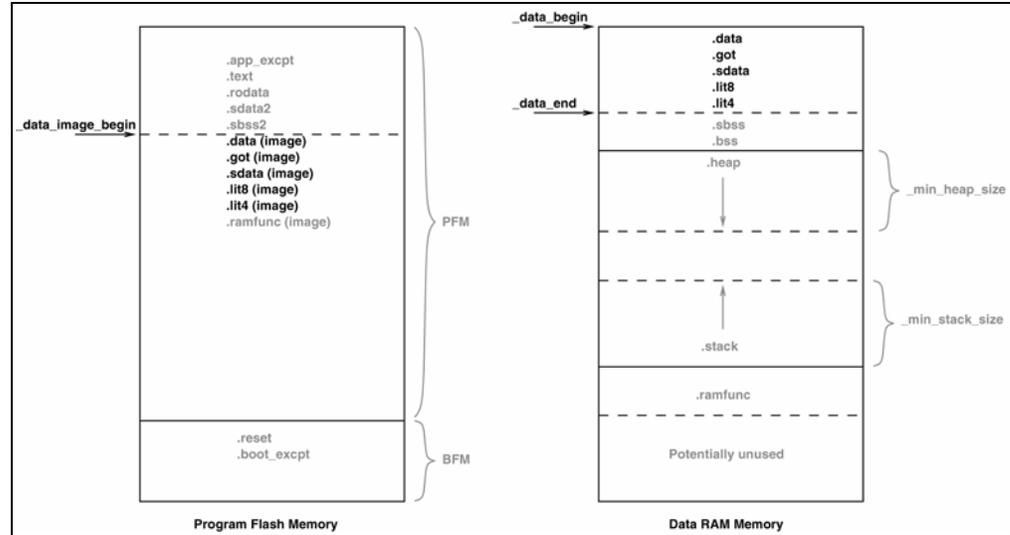
5.7.2.7   COPY RAM FUNCTIONS FROM PROGRAM FLASH TO DATA MEMORY

RAM functions are similar to initialized data, except that the data that exists in the program Flash represents functions instead of initial values for symbols. Similar to the way that initialized data is copied from program Flash to data memory, the linker script must initialize three variables—one for the start address of the image in program Flash, one for the start address of the section in data memory and one for the end address of the section in data memory. The start-up code copies the memory locations from the program Flash image to the data memory using these variables. These variables are named `_ramfunc_image_begin`, `_ramfunc_begin`, and `_ramfunc_end`, respectively.

FIGURE 5-8:   RAM FUNCTIONS
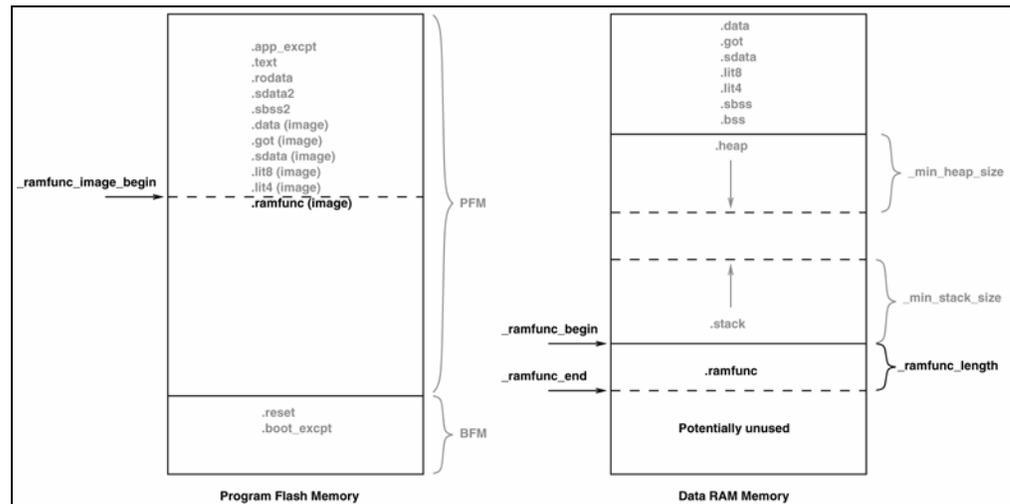
5.7.2.8   INITIALIZE BUS MATRIX REGISTERS

The bus matrix registers (`BMXDKPBA`, `BMXDUDBA`, `BMXDUPBA`) should be initialized by the start-up code if any RAM functions exist, otherwise, these registers should not be modified. To determine whether any RAM functions exist in the application, the linker script provides a variable that contains the length of the `.ramfunc` section[1]. This

---

1. All functions attributed with `__ramfunc__` or `__longramfunc__` are placed in the `.ramfunc` section.

variable is named _ramfunc_length. In addition, the linker script provides three variables that contain the address of the bus matrix registers. These variables are named _bmxdkpba_address, _bmxdudba_address, and _bmxdupba_address. The following calculations are used to calculate these addresses:

```
_bmxdkpba_address = _ramfunc_begin -
                    ORIGIN(${DATA_MEMORY_LOCATION}) ;
_bmxdudba_address = LENGTH(${DATA_MEMORY_LOCATION}) ;
_bmxdupba_address = LENGTH(${DATA_MEMORY_LOCATION}) ;
```

The linker script ensures that RAM functions are aligned to a 2K alignment boundary as is required by the BMXDKPBA register.

FIGURE 5-9:   BUS MATRIX INITIALIZATION
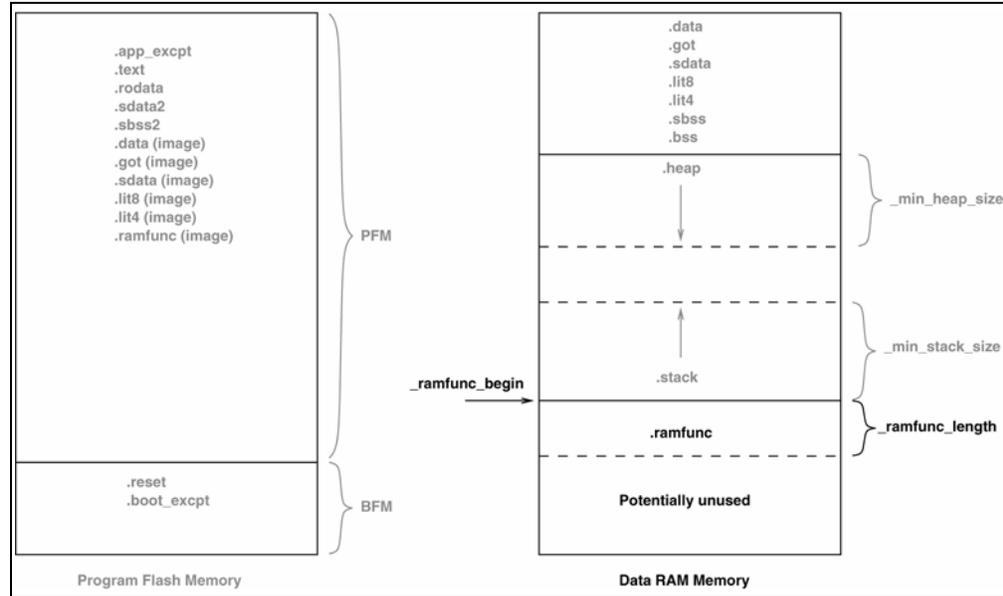
5.7.2.9   INITIALIZE CP0 REGISTERS

The CP0 registers are initialized in the following order:

1. Count register
2. Compare register
3. EBase register
4. IntCtl register
5. Cause register
6. Status register

5.7.2.9.1   Hardware Enable Register (HWREna – CP0 Register 7, Select 0)

This register contains a bit mask that determines which hardware registers are accessible via the RDHWR instruction. Privileged software may determine which of the hardware registers are accessible by the RDHWR instruction. In doing so, a register may be virtualized at the cost of handling a Reserved Instruction Exception, interpreting the instruction, and returning the virtualized value. For example, if it is not desirable to provide direct access to the Count register, access to the register may be individually disabled and the return value can be virtualized by the operating system.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.9.2 Bad Virtual Address Register (`BadVAddr` – CP0 Register 8, Select 0)

This register is a read-only register that captures the most recent virtual address that caused an Address Error exception (`AdEL` or `AdES`).

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.9.3 Count Register (`Count` – CP0 Register 9, Select 0)

This register acts as a timer, incrementing at a constant rate, whether or not an instruction is executed, retired, or any forward progress is made through the pipeline. The counter increments every other clock, if the `DC` bit in the `Cause` register is 0. The `Count` register can be written for functional or diagnostic purposes, including at Reset or to synchronize processors. By writing the $Count_{DM}$ bit in the `Debug` register, it is possible to control whether the `Count` register continues incrementing while the processor is in Debug mode.

This register is cleared in the PIC32MX start-up code.

5.7.2.9.4 Compare Register (`Compare` – CP0 Register 11, Select 0)

This register acts in conjunction with the `Count` register to implement a timer and timer interrupt function. The timer interrupt is an output of the core. The `Compare` register maintains a stable value and does not change on its own. When the value of the `Count` register equals the value of the `Compare` register, the `SI_TimerInt` pin is asserted. This pin remains asserted until the `Compare` register is written. The `SI_TimerInt` pin can be fed back into the core on one of the interrupt pins to generate an interrupt. For diagnostic purposes, the `Compare` register is a read/write register. In normal use, however, the `Compare` register is write-only. Writing a value to the `Compare` register, as a side effect, clears the timer interrupt.

This register is set to `0xFFFFFFFF` in the PIC32MX start-up code.

5.7.2.9.5 Status Register (`Status` – CP0 Register 12, Select 0)

This register is a read/write register that contains the operating mode, Interrupt Enabling, and the diagnostic states of the processor. Fields of this register combine to create operating modes for the processor.

The following settings are initialized by the PIC32MX start-up code (0b000**000**000**x**0**xx**0**?**0**00000000**000**00**000**):

- Access to Coprocessor 0 not allowed in User mode (`CU0 = 0`)
- User mode uses configured endianess (`RE = 0`)
- No change to exception vectors location (`BEV = no change`)
- No change to flag bits that indicate reason for entry to the Reset exception vector (`SR, NMI = no change`)
- If CorExtend User Defined Instructions have been implemented ($Config_{UDI}$ == 1), CorExtend is enabled (`CEE = 1`), otherwise, CorExtend is disabled (`CEE = 0`).
- Interrupt masks are cleared to disable any pending interrupt requests (`IM7..IM2 = 0, IM1..IM0 = 0`)
- Interrupt priority level is 0 (`IPL = 0`)
- Base mode is Kernel mode (`UM = 0`)
- Error level is normal (`ERL = 0`)
- Exception level is normal (`EXL = 0`)
- Interrupts are disabled (`IE = 0`)

5.7.2.9.6   Interrupt Control Register (`IntCtl` – CP0 Register 12, Select 1)

This register controls the expanded interrupt capability added in Release 2 of the Architecture, including vectored interrupts and support for an external interrupt controller.

This register contains the vector spacing for interrupt handling. The vector spacing portion of this register (bits 9..5) is initialized with the value of the `_vector_spacing` symbol by the PIC32MX start-up code. All other bits are set to '1'.

5.7.2.9.7   Shadow Register Control Register (`SRSCtl` – CP0 Register 12, Select 2)

This register controls the operation of the GPR shadow sets in the processor.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.9.8   Shadow Register Map Register (`SRSMap` – CP0 Register 12, Select 3)

This register contains eight 4-bit fields that provide the mapping from a vector number to the shadow set number to use when servicing such an interrupt. The values from this register are not used for a non-interrupt exception, or a non-vectored interrupt ($Cause_{IV}$ = 0 or $IntCtl_{VS}$ = 0). In such cases, the shadow set number comes from $SRSCtl_{ESS}$. If $SRSCtl_{HSS}$ is zero, the results of a software read or write of this register are UNPREDICTABLE. The operation of the processor is UNDEFINED if a value is written to any field in this register that is greater than the value of $SRSCtl_{HSS}$. The `SRSMap` register contains the shadow register set numbers for vector numbers 7..0. The same shadow set number can be established for multiple interrupt vectors, creating a many-to-one mapping from a vector to a single shadow register set number.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.9.9   Cause Register (`Cause` – CP0 Register 13, Select 0)

This register primarily describes the cause of the most Recent exception. In addition, fields also control software interrupt requests and the vector through which interrupts are dispatched. With the exception of the `DC`, `IV`, and `IP1..IP0` fields, all fields in the `Cause` register are read-only. Release 2 of the Architecture added optional support for an External Interrupt Controller (EIC) interrupt mode, in which `IP7..IP2` are interpreted as the Requested Interrupt Priority Level (`RIPL`).

The following settings are initialized by the PIC32MX start-up code:

- Enable counting of `Count` register (`DC = no change`)
- Use the special exception vector (16#200) (`IV = 1`)
- Disable software interrupt requests (`IP1..IP0 = 0`)

5.7.2.9.10   Exception Program Counter (`EPC` – CP0 Register 14, Select 0)

This register is a read/write register that contains the address at which processing resumes after an exception has been serviced. All bits of the `EPC` register are significant and must be writable. For synchronous (precise) exceptions, the `EPC` contains one of the following:

- The virtual address of the instruction that was the direct cause of the exception
- The virtual address of the immediately preceding branch or jump instruction, when the exception causing instruction is a branch delay slot and the `Branch Delay` bit in the `Cause` register is set.

On new exceptions, the processor does not write to the `EPC` register when the `EXL` bit in the `Status` register is set, however, the register can still be written via the `MTC0` instruction.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.9.11   Processor Identification Register (`PRid` – CP0 Register 15, Select 0)

This register is a 32-bit read-only register that contains information identifying the manufacturer, manufacturer options, processor identification, and revision level of the processor.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.9.12   Exception Base Register (`EBase` – CP0 Register 15, Select 1)

This register is a read/write register containing the base address of the exception vectors used when $Status_{BEV}$ equals 0, and a read-only CPU number value that may be used by software to distinguish different processors in a multi-processor system. The `EBase` register provides the ability for software to identify the specific processor within a multi-processor system, and allows the exception vectors for each processor to be different, especially in systems composed of heterogeneous processors. Bits 31..12 of the `EBase` register are concatenated with zeros to form the base of the exception vectors when $Status_{BEV}$ is 0. The exception vector base address comes from fixed defaults when $Status_{BEV}$ is 1, or for any EJTAG Debug exception. The reset state of bits 31..12 of the `EBase` register initialize the exception base register to `16#80000000`, providing backward compatibility with Release 1 implementations. Bits 31..30 of the `EBase` register are fixed with the value `2#10` to force the exception base address to be in KSEG0 or KSEG1 unmapped virtual address segments.

If the value of the exception base register is to be changed, this must be done with $Status_{BEV}$ equal 1. The operation of the processor is UNDEFINED if the Exception Base field is written with a different value when $Status_{BEV}$ is 0.

Combining bits 31..30 with the Exception Base field allows the base address of the exception vectors to be placed at any 4K byte page boundary. If vectored interrupts are used, a vector offset greater than 4K byte can be generated. In this case, bit 12 of the Exception Base field must be zero. The operation of the processor is UNDEFINED if software writes bit 12 of the Exception Base field with a 1 and enables the use of a vectored interrupt whose offset is greater than 4K bytes from the exception base address.

This register us initialized with the value of the `_ebase_address` symbol by the PIC32MX start-up code. `_ebase_address` is provided by the linker script with a default value of the start of KSEG1 program memory. The user can change this value by providing the command line option `--defsym _ebase_address=A` to the linker.

5.7.2.9.13   Config Register (`Config` – CP0 Register 16, Select 0)

This register specifies various configuration and capabilities information. Most of the fields in the `Config` register are initialized by hardware during the Reset exception process, or are constant.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.9.14   Config1 Register (`Config1` – CP0 Register 16, Select 1)

This register is an adjunct to the `Config` register and encodes additional information about the capabilities present on the core. All fields in the `Config1` register are read-only.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.9.15   Config2 Register (`Config2` – CP0 Register 16, Select 2)

This register is an adjunct to the `Config` register and is reserved to encode additional capabilities information. `Config2` is allocated for showing the configuration of level 2/3 caches. These fields are reset to 0 because L2/L3 caches are not supported on the core. All fields in the `Config2` register are read-only.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.9.16   Config3 Register (`Config3` – CP0 Register 16, Select 3)

This register encodes additional capabilities. All fields in the `Config3` register are read-only.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.9.17   Debug Register (`Debug` – CP0 Register 23, Select 0)

This register is used to control the debug exception and provide information about the cause of the debug exception and when re-entering at the debug exception vector due to a normal exception in Debug mode. The read-only information bits are updated every time the debug exception is taken or when a normal exception is taken when already in Debug mode. Only the `DM` bit and the $EJTAG_{ver}$ field are valid when read from non-Debug mode. The values of all other bits and fields are UNPREDICTABLE. Operation of the processor is UNDEFINED if the `Debug` register is written from non-Debug mode.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.9.18   Trace Control Register (`TraceControl` – CP0 Register 23, Select 1)

This register provides control and status information. The `TraceControl` register is only implemented if the EJTAG Trace capability is present.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.10   TRACE CONTROL 2 REGISTER (`TraceControl2` – CP0 REGISTER 23, SELECT 2)

This register provides additional control and status information. The `TraceControl2` register is only implemented if the EJTAG Trace capability is present.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.10.1   User Trace Data Register (`UserTraceData` – CP0 Register 23, Select 3)

When this register is written to, a trace record is written indicating a type 1 or type 2 user format. This type is based on the `UT` bit in the `TraceControl` register. This register cannot be written in consecutive cycles. The trace output data is UNPREDICTABLE if this register is written in consecutive cycles. The `UserTraceData` register is only implemented if the EJTAG Trace capability is present.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.10.2   TraceBPC Register (`TraceBPC` – CP0 Register 23, Select 4)

This register is used to control start and stop of tracing using an EJTAG hardware breakpoint. The hardware breakpoint would then be set as a triggered source and optionally also as a Debug exception breakpoint. The `TraceBPC` register is only implemented if both the hardware breakpoints and the EJTAG Trace cap are present.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.10.3   Debug2 Register (`Debug2` – CP0 Register 23, Select 5)

This register holds additional information about Complex Breakpoint exceptions. The `Debug2` register is only implemented if complex hardware breakpoints are present.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.10.4   Debug Exception Program Counter (`DEPC` – CP0 Register 24, Select 0)

This register is a read/write register that contains the address at which processing resumes after a debug exception or Debug mode exception has been serviced. For synchronous (precise) debug and Debug mode exceptions, the `DEPC` contains either:

- The virtual address of the instruction that was the direct cause of the debug exception, or
- The virtual address of the immediately preceding branch or jump instruction, when the debug exception causing instruction is in a branch delay slot, and the Debug Branch Delay (DBD) bit in the Debug register is set.

For asynchronous debug exceptions (debug interrupt, complex break), the DEPC contains the virtual address of the instruction where execution should resume after the debug handler code is executed.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.10.5 Error Exception Program Counter (ErrorEPC – CP0 Register 30, Select 0)

This register is a read/write register, similar to the EPC register, except that it is used on error exceptions. All bits of the ErrorEPC are significant and must be writable. It is also used to store the program counter on Reset, Soft Reset, and non-maskable interrupt (NMI) exceptions. The ErrorEPC register contains the virtual address at which instruction processing can resume after servicing an error. This address can be:

- The virtual address of the instruction that caused the exception, or
- The virtual address of the immediately preceding branch or jump instruction when the error causing instruction is a branch delay slot.

Unlike the EPC register, there is no corresponding branch delay slot indication for the ErrorEPC register.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.10.6 Debug Exception Save Register (DeSave – CP0 Register 31, Select 0)

This register is a read/write register that functions as a simple memory location. This register is used by the debug exception handler to save on of the GPRs that is then used to save the rest of the context to a pre-determined memory area (such as in the EJTAG Probe). This register allows the safe debugging of exception handlers and other types of code where the existence of a valid stack for context saving cannot be assumed.

No initialization is performed on this register in the PIC32MX start-up code.

5.7.2.11 CALL "ON BOOTSTRAP" PROCEDURE

A procedure is called after initializing the CP0 registers. This procedure allows users to perform actions during bootstrap (i.e., while $Status_{BEV}$ is set) and before entering into the main routine. An empty weak version of this procedure (_on_bootstrap) is provided with the start-up code. This procedure may be used for performing hardware initialization and/or for initializing the environment required by an RTOS.

5.7.2.12 CHANGE LOCATION OF EXCEPTION VECTORS

Immediately before calling the applications main routine, the $Status_{BEV}$ is cleared to change the location of the exception vectors from the bootstrap location to the normal location.

5.7.2.13 CALL MAIN

The last thing that the start-up code performs is a call to the main routine. If the user returns from main, the start-up code goes into an infinite loop.

5.7.3   Exceptions

In addition, two weak general exception handlers are provided that can be overridden by the application—one to handle exceptions when $Status_{BEV}$ is 1 (_bootstrap_exception_handler) and one to handle exceptions when $Status_{BEV}$ is 0 (_general_exception_handler). Both the weak Reset exception handler and the weak general exception handler provided with the start-up code enters an infinite loop. The start-up code arranges for a jump to the reset exception handler to be located at 0xBFC00380 and a jump to the general exception handler to be located at EBASE + 0x180.

Both handlers must be attributed with the nomips16 [e.g., __attribute__ ((nomips16))] since the start-up code jumps to these functions.

FIGURE 5-10:   EXCEPTIONS
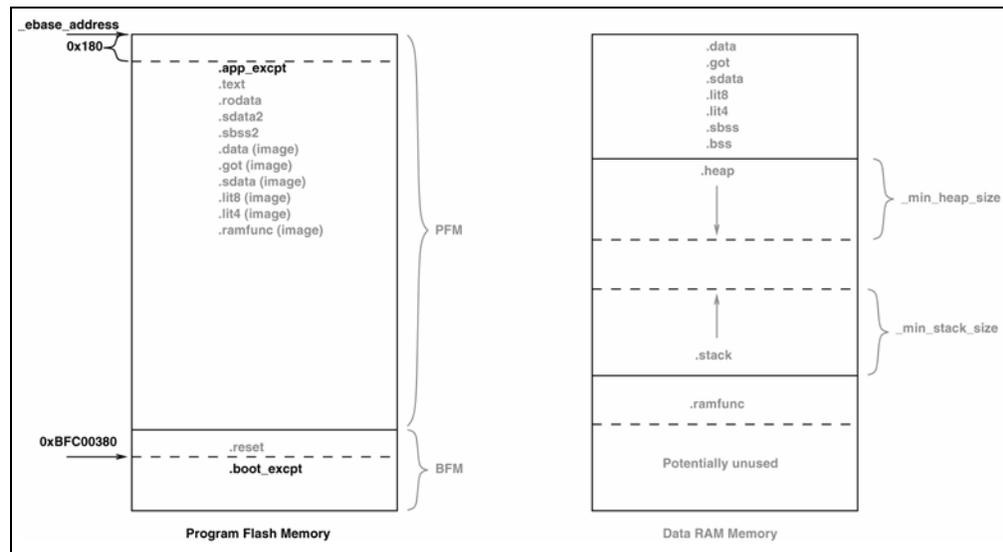
5.7.4   Symbols Required by Start-up Code and C Library

This section details the symbols that are required by the start-up code and C library. Currently the default linker script defines these symbols. If an application provides a custom linker script, the user must ensure that all of the following symbols are provided in order for the start-up code and C library to function:

Symbol Name	Description
_bmxdkpba_address	The address to place into the BMXDKPBA register if _ramfunc_length is greater than 0.
_bmxdudba_address	The address to place into the BMXDUDBA register if _ramfunc_length is greater than 0.
_bmxdupba_address	The address to place into the BMXDUPBA register if _ramfunc_length is greater than 0.
_bss_begin	The starting location of the uninitialized data. Uninitialized data includes both .sbss and .bss.
_bss_end	The end location of the uninitialized data. Uninitialized data includes both .sbss and .bss.
_data_begin	The starting location of the initialized data. Initialized data includes .data, .got, .sdata, .lit8, and .lit4.
_data_end	The end location of the initialized data. Initialized data includes .data, .got, .sdata, .lit8, and .lit4.
_data_image_begin	The starting location in program memory of the image of initialized data. Initialized data includes .data, .got, .sdata, .lit8, and .lit4.
_ebase_address	The location of EBASE.
_end	The end of data allocation. Should be identical to _heap.
_gp	Points to the "middle" of the small variables region. By convention this is 0x8000 bytes from the first location used for small variables.
_heap	The starting location of the heap in DRM.
_ramfunc_begin	The starting location of the RAM functions. This should be located at a 2K boundary as it is used to initialize the BMXDKPBA register.
_ramfunc_end	The end location of the RAM functions.
_ramfunc_image_begin	The starting location in program memory of the image of RAM functions.
_ramfunc_length	The length of the .ramfunc section.
_stack	The starting location of the stack in DRM. Remember that the stack grows from the bottom of data memory so this symbol should point to the bottom of the section allocated for the stack.
_vector_spacing	The initialization value for the vector spacing field in the IntCtl register.

5.8   CONTENTS OF THE DEFAULT LINKER SCRIPT

The default linker script contains the following categories of information:

- Output Format and Entry Points
- Default Values for Minimum Stack and Heap Sizes
- Processor Definitions Include File

- Inclusion of Processor-Specific Object File(s)
- Base Exception Vector Address and Vector Spacing Symbols
- Memory Address Equates
- Memory Regions
- Configuration Words Input/Output Section Map
• Input/Output Section Map

> **Note:** All addresses specified in the linker scripts should be specified as virtual not physical addresses.

5.8.1   Output Format and Entry Points

The first several lines of the default linker script define the output format and the entry point for the application. Copies of the default linker scripts are provided in `C:\program files\...\MPLAB C32\pic32mx\lib\ldscripts`.

```
OUTPUT_FORMAT("elf32-tradlittlemips")
OUTPUT_ARCH(pic32mx)
ENTRY(_reset)
```

The `OUTPUT_FORMAT` line selects the object file format for the output file. The output object file format generated by the 32-bit language tools is a traditional, little-endian, MIPS, ELF32 format.

The `OUTPUT_ARCH` line selects the specific machine architecture for the output file. The output files generated by the 32-bit language tools contains information that identifies the file was generated for the PIC32MX architecture.

The `ENTRY` line selects the entry point of the application. This is the symbol identifying the location of the first instruction to execute. The 32-bit language tools begins execution at the instruction identified by the `_reset` label.

5.8.2   Default Values for Minimum Stack and Heap Sizes

The next section of the default linker script provides default values for the minimum stack and heap sizes.

```
/*
 * Provide for a minimum stack and heap size
 * - _min_stack_size - represents the minimum space that must
 *                     be made available for the stack.  Can
 *                     be overridden from the command line
 *                     using the linker's --defsym option.
 * - _min_heap_size  - represents the minimum space that must
 *                     be made available for the heap.  Can
 *                     be overridden from the command line
 *                     using the linker's --defsym option.
 */
EXTERN (_min_stack_size _min_heap_size)
PROVIDE(_min_stack_size = 0x400) ;
PROVIDE(_min_heap_size = 0) ;
```

The `EXTERN` line ensures that the rest of the linker script has access to the default values of `_min_stack_size` and `_min_heap_size` assuming that the user does not override these values using the linker's `--defsym` command line option.

The two `PROVIDE` lines ensure that a default value is provided for both `_min_stack_size` and `_min_heap_size`. The default value for the minimum stack size is `1024` bytes (`0x400`). The default value for the minimum heap size is `0` bytes.

5.8.3 Processor Definitions Include File

The next line in the default linker script pulls in information specific to the processor.

```
INCLUDE procdefs.ld
```

The file `procdefs.ld` is included in the linker script at this point. The file is searched for in the current directory and in any directory specified with the `-L` command line option. The compiler shell ensures that the correct directory is passed to the linker with the `-L` command line option based on the processor selected with the `-mprocessor` command line option.

The processor definitions linker script contains the following pieces of information:

- Inclusion of Processor-Specific Object File(s)
- Base Exception Vector Address and Vector Spacing Symbols
- Memory Address Equates
- Memory Regions
- Configuration Words Input/Output Section Map

5.8.3.1 INCLUSION OF PROCESSOR-SPECIFIC OBJECT FILE(S)

This section of the processor definitions linker script ensures that the processor-specific object file(s) get included in the link.

```
/**************************************************************
 * Processor-specific object file.  Contains SFR definitions.
 **************************************************************/
INPUT("processor.o")
```

The `INPUT` line specifies that `processor.o` should be included in the link as if this file were named on the command line. The linker attempts to find this file in the current directory. If it is not found, the linker searches through the library search paths (i.e., the paths specified with the `-L` command line option).

5.8.3.2 BASE EXCEPTION VECTOR ADDRESS AND VECTOR SPACING SYMBOLS

This section of the processor definitions linker script defines values for the base exception vector address and vector spacing.

```
/**************************************************************
 * For interrupt vector handling
 **************************************************************/
_vector_spacing= 0x00000001;
_ebase_address= 0x9FC01000;
```

The first line defines a value of `1` for `_vector_spacing`. The available memory for exceptions only supports a vector spacing of `1`. The second line defines the location of the base exception vector address (`EBASE`). This address is located in the KSEG0 boot segment.

5.8.3.3 MEMORY ADDRESS EQUATES

This section of the processor definitions linker script provides information about certain memory addresses required by the default linker script.

```
/**************************************************************
 * Memory Address Equates
 **************************************************************/
_RESET_ADDR= 0xBFC00000;
_BEV_EXCPT_ADDR= 0xBFC00380;
_DBG_EXCPT_ADDR= 0xBFC00480;
_DBG_CODE_ADDR= 0xBFC02000;
_GEN_EXCPT_ADDR= _ebase_address + 0x180;
```

The _RESET_ADDR defines the processor's Reset address. This is the virtual begin address of the IFM Boot section in Kernel mode.

The _BEV_EXCPT_ADDR defines the address that the processor jumps to when an exception is encountered and $Status_{BEV} = 1$.

The _DBG_EXCPT_ADDR defines the address that the processor jumps to when a debug exception is encountered.

The _DBG_CODE_ADDR defines the address that the start address of the debug executive.

The _GEN_EXCPT_ADDR defines the address that the processor jumps to when an exception is encountered and $Status_{BEV} = 0$.

5.8.3.4   MEMORY REGIONS

This section of the processor definitions linker script provides information about the memory regions that are available on the device.

```
/****************************************************************
 * Memory Regions
 *
 * Memory regions without attributes cannot be used for
 * orphaned sections.  Only sections specifically assigned to
 * these regions can be allocated into these regions.
 ****************************************************************/
MEMORY
{
  kseg0_program_mem (rx) : ORIGIN = 0x9D000000, LENGTH = 0x8000
  kseg0_boot_mem         : ORIGIN = 0x9FC00490, LENGTH = 0x970
  exception_mem          : ORIGIN = 0x9FC01000, LENGTH = 0x1000
  kseg1_boot_mem         : ORIGIN = 0xBFC00000, LENGTH = 0x490
  debug_exec_mem         : ORIGIN = 0xBFC02000, LENGTH = 0xFF0
  config3                : ORIGIN = 0xBFC02FF0, LENGTH = 0x4
  config2                : ORIGIN = 0xBFC02FF4, LENGTH = 0x4
  config1                : ORIGIN = 0xBFC02FF8, LENGTH = 0x4
  config0                : ORIGIN = 0xBFC02FFC, LENGTH = 0x4
  kseg1_data_mem (w!x)   : ORIGIN = 0xA0000000, LENGTH = 0x2000
  sfrs                   : ORIGIN = 0xBF800000, LENGTH = 0x10000
}
```

Eleven memory regions are defined with an associated start address and length:

1. Program memory region (kseg0_program_mem) for application code
2. Boot memory regions (kseg0_boot_mem and kseg1_boot_mem)
3. Exception memory region (exception_mem)
4. Debug executive memory region (debug_exec_mem)
5. Configuration memory regions (config3, config2, config1, and config0)
6. Data memory region (kseg1_data_mem)
7. SFR memory region (sfrs)

The default linker script uses these names to locate sections into the correct regions. Sections which are non-standard become orphaned sections. The attributes of the memory regions are used to locate these orphaned sections. The attributes (rx) specify that read-only sections or executable sections can be located into the program memory regions. Similarly, the attributes (w!x) specify that sections that are not read-only and not executable can be located in the data memory region. Since no attributes are specified for the boot memory region, the configuration memory regions, or the SFR memory region, only specified sections may be located in these regions

(i.e., orphaned sections may not be located in the boot memory regions, the exception memory region, the configuration memory regions, the debug executive memory region, or the SFR memory region).

5.8.3.5 CONFIGURATION WORDS INPUT/OUTPUT SECTION MAP

The last section in the processor definitions linker script is the input/output section map for Configuration Words. This section map is additive to the Input/Output Section Map found in the default linker script (see **Section 5.8.4 "Input/Output Section Map"**). It defines how input sections for Configuration Words are mapped to output sections for Configuration Words. Note that input sections are portions of an application that are defined in source code, while output sections are created by the linker. Generally, several input sections may be combined into a single output section. All output sections are specified within a `SECTIONS` command in the linker script.

For each Configuration Word that exists on the specific processor, a distinct output section named `.config_address` exists where address is the location of the Configuration Word in memory. Each of these sections contains the data created by the `#pragma config` directive (see **Section 4.8.1 "#pragma config"**) for that Configuration Word. Each section is assigned to their respective memory region (`config`*n*).

```
SECTIONS
{
  .config_BFC02FF0 : {
    *(.config_BFC02FF0)
  } > config3
  .config_BFC02FF4 : {
    *(.config_BFC02FF4)
  } > config2
  .config_BFC02FF8 : {
    *(.config_BFC02FF8)
  } > config1
  .config_BFC02FFC : {
    *(.config_BFC02FFC)
  } > config0
}
```

5.8.4 Input/Output Section Map

The last section in the default linker script is the input/output section map. The section map is the heart of the linker script. It defines how input sections are mapped to output sections. Note that input sections are portions of an application that are defined in source code, while output sections are created by the linker. Generally, several input sections may be combined into a single output section. All output sections are specified within a `SECTIONS` command in the linker script.

The following output sections may be created by the linker:

- .reset Section
- .bev_excpt Section
- .dbg_excpt Section
- .dbg_code Section
- .app_excpt Section
- .vector_0 .. .vector_63 Sections
- .start-up Section
- .text Section
- .rodata Sectionn

- .sdata2 Section
- .sbss2 Section
- .dbg_data Section
- .data Section
- .got Section
- .sdata Section
- .lit8 Section
- .lit4 Section
- .sbss Section
- .bss Section
- .heap Section
- .stack Section
- .ramfunc Section
- Stack Location
- Debug Sections

5.8.4.1   .RESET SECTION

This section contains the code that is executed when the processor performs a Reset. This section is located at the Reset address (`_RESET_ADDR`) as specified in the processor definitions linker script and is assigned to the boot memory region (`kseg1_boot_mem`).

```
.reset _RESET_ADDR :
{
  *(.reset)
} > kseg1_boot_mem
```

5.8.4.2   .BEV_EXCPT SECTION

This section contains the handler for exceptions that occur when $Status_{BEV} = 1$. This section is located at the `BEV` exception address (`_BEV_EXCPT_ADDR`) as specified in the processor definitions linker script and is assigned to the boot memory region (`kseg1_boot_mem`).

```
.bev_excpt _BEV_EXCPT_ADDR :
{
  *(.bev_handler)
} > kseg1_boot_mem
```

5.8.4.3   .DBG_EXCPT SECTION

This section reserves space for the debug exception vector. This section is only allocated if the symbol `_DEBUGGER` has been defined. (This symbol is defined if the `-mdebugger` command line option is specified to the shell.) This section is located at the debug exception address (`_DBG_EXCPT_ADDR`) as specified in the processor definitions linker script and is assigned to the boot memory region (`kseg1_boot_mem`). The section is marked as `NOLOAD` as it is only intended to ensure that application code cannot be placed at locations reserved for the debug executive.

```
.dbg_excpt _DBG_EXCPT_ADDR (NOLOAD) :
{
  . += (DEFINED (_DEBUGGER) ? 0x8 : 0x0);
} > kseg1_boot_mem
```

5.8.4.4   .DBG_CODE SECTION

This section reserves space for the debug exception handler. This section is only allocated if the symbol `_DEBUGGER` has been defined. (This symbol is defined if the `-mdebugger` command line option is specified to the shell.) This section is located at the debug code address (`_DBG_CODE_ADDR`) as specified in the processor definitions linker script and is assigned to the debug executive memory region (`debug_exec_mem`). The section is marked as `NOLOAD` as it is only intended to ensure that application code cannot be placed at locations reserved for the debug executive.

```
.dbg_code _DBG_CODE_ADDR (NOLOAD) :
{
  . += (DEFINED (_DEBUGGER) ? 0xFF0 : 0x0);
} > debug_exec_mem
```

5.8.4.5   .APP_EXCPT SECTION

This section contains the handler for exceptions that occur when $Status_{BEV} = 0$. This section is located at the general exception address (`_GEN_EXCPT_ADDR`) as specified in the processor definitions linker script and is assigned to the exception memory region (`exception_mem`).

```
.app_excpt _GEN_EXCPT_ADDR :
{
  *(.gen_handler)
} > exception_mem
```

5.8.4.6   .VECTOR_0 .. .VECTOR_63 SECTIONS

These sections contain the handler for each of the interrupt vectors. These sections are located at the correct vectored addresses using the formula:

```
_ebase_address + 0x200 + (_vector_spacing << 5) * n
```

where `n` is the respective vector number.

Each of the sections is followed by an assert that ensures the code located at the vector does not exceed the vector spacing specified.

```
.vector_n _ebase_address + 0x200 + (_vector_spacing << 5) * n :
  {
    *(.vector_n)
  } > exception_mem
ASSERT (SIZEOF(.vector_n) < (_vector_spacing << 5), "function at
exception vector n too large")
```

5.8.4.7   .START-UP SECTION

This section contains the C start-up code. This section is assigned to the KSEG0 boot memory region (`kseg0_boot_mem`).

```
.startup ORIGIN(kseg0_boot_mem) :
  {
    *(.startup)
  } > kseg0_boot_mem
```

5.8.4.8   .TEXT SECTION

This section collects executable code from all of the application's input files. This section is assigned to the program memory region (`kseg0_program_mem`) and has a fill value of `NOP (0)`. Symbols are defined to represent the begin (`_text_begin`) and end (`_text_end`) addresses of this section.

```
.text ORIGIN(kseg0_program_mem) :
{
  _text_begin = . ;
```

```
    *(.text .stub .text.* .gnu.linkonce.t.*)
    KEEP (*(.text.*personality*))
    *(.gnu.warning)
    *(.mips16.fn.*)
    *(.mips16.call.*)
    _text_end = . ;
} > kseg0_program_mem =0
```

5.8.4.9 .RODATA SECTION

This section collects the read-only sections from all of the application's input files. This section is assigned to the program memory region (`kseg0_program_mem`).

```
.rodata    :
{
  *(.rodata .rodata.* .gnu.linkonce.r.*)
  *(.rodata1)
} > kseg0_program_mem
```

5.8.4.10 .SDATA2 SECTION

This section collects the small initialized constant global and static data from all of the application's input files. Because of the constant nature of the data, this section is also a read-only section. This section is assigned to the program memory region (`kseg0_program_mem`).

```
/*
 * Small initialized constant global and static data can be
 * placed in the .sdata2 section.  This is different from
 * .sdata, which contains small initialized non-constant
 * global and static data.
 */
.sdata2    :
{
  *(.sdata2 .sdata2.* .gnu.linkonce.s2.*)
} > kseg0_program_mem
```

5.8.4.11 .SBSS2 SECTION

This section collects the small uninitialized constant global and static data from all of the application's input files. Because of the constant nature of the data, this section is also a read-only section. This section is assigned to the program memory region (`kseg0_program_mem`).

```
/*
 * Uninitialized constant global and static data (i.e.,
 * variables which will always be zero).  Again, this is
 * different from .sbss, which contains small non-initialized,
 * non-constant global and static data.
 */
.sbss2    :
{
  *(.sbss2 .sbss2.* .gnu.linkonce.sb2.*)
} > kseg0_program_mem
```

5.8.4.12 .DBG_DATA SECTION

This section reserves space for the data required by the debug exception handler. This section is only allocated if the symbol `_DEBUGGER` has been defined. (This symbol is defined if the `-mdebugger` command line option is specified to the shell.) This section is assigned to the data memory region (`kseg1_data_mem`). The section is marked as `NOLOAD` as it is only intended to ensure that application data cannot be placed at locations reserved for the debug executive.

```
.dbg_data (NOLOAD) :
{
  . += (DEFINED (_DEBUGGER) ? 0x200 : 0x0);
} > kseg1_data_mem
```

5.8.4.13 .DATA SECTION

This section collects the initialized data from all of the application's input files. This section is assigned to the data memory region (kseg1_data_mem) with a load address located in the program memory region (kseg0_program_mem). Symbols are defined to represent the virtual begin (_data_begin) and end (_data_end) addresses of this section, as well as the physical begin address of the data in program memory (_data_image_begin).

```
.data    :
{
  _data_begin = . ;
  *(.data .data.* .gnu.linkonce.d.*)
  KEEP (*(.gnu.linkonce.d.*personality*))
  *(.data1)
} > kseg1_data_mem AT> kseg0_program_mem
_data_image_begin = LOADADDR(.data) ;
```

5.8.4.14 .GOT SECTION

This section collects the global offset table from all of the application's input files. This section is assigned to the data memory region (kseg1_data_mem) with a load address located in the program memory region (kseg0_program_mem). A symbol is defined to represent the location of the Global Pointer (_gp).

```
_gp = ALIGN(16) + 0x7FF0 ;
.got    :
{
    *(.got.plt) *(.got)
} > kseg1_data_mem AT> kseg0_program_mem
```

5.8.4.15 .SDATA SECTION

This section collects the small initialized data from all of the application's input files. This section is assigned to the data memory region (kseg1_data_mem) with a load address located in the program memory region (kseg0_program_mem). Symbols are defined to represent the virtual begin (_sdata_begin) and end (_sdata_end) addresses of this section.

```
/*
 * We want the small data sections together, so
 * single-instruction offsets can access them all, and
 * initialized data all before uninitialized, so
 * we can shorten the on-disk segment size.
 */
.sdata    :
{
  _sdata_begin = . ;
  *(.sdata .sdata.* .gnu.linkonce.s.*)
  _sdata_end = . ;
} > kseg1_data_mem AT> kseg0_program_mem
```

5.8.4.16 .LIT8 SECTION

This section collects the 8-byte constants (usually floating-point) which the assembler decides to store in memory rather than in the instruction stream from all of the application's input files. This section is assigned to the data memory region (kseg1_data_mem) with a load address located in the program memory region (kseg0_program_mem).

```
.lit8           :
{
  *(.lit8)
} > kseg1_data_mem AT> kseg0_program_mem
```

5.8.4.17 .LIT4 SECTION

This section collects the 4-byte constants (usually floating-point) which the assembler decides to store in memory rather than in the instruction stream from all of the application's input files. This section is assigned to the data memory region (kseg1_data_mem) with a load address located in the program memory region (kseg0_program_mem). A symbol is defined to represent the virtual end address of the initialized data (_data_end).

```
.lit4           :
{
  *(.lit4)
} > kseg1_data_mem AT> kseg0_program_mem
_data_end = . ;
```

5.8.4.18 .SBSS SECTION

This section collects the small uninitialized data from all of the application's input files. This section is assigned to the data memory region (kseg1_data_mem). A symbol is defined to represent the virtual begin address of uninitialized data (_bss_begin). Symbols are also defined to represent the virtual begin (_sbss_begin) and end (_sbss_end) addresses of this section.

```
_bss_begin = . ;
.sbss    :
{
  _sbss_begin = . ;
  *(.dynsbss)
  *(.sbss .sbss.* .gnu.linkonce.sb.*)
  *(.scommon)
  _sbss_end = . ;
} > kseg1_data_mem
```

5.8.4.19 .BSS SECTION

This section collects the uninitialized data from all of the application's input files. This section is assigned to the data memory region (kseg1_data_mem). A symbol is defined to represent the virtual end address of uninitialized data (_bss_end). A symbol is also defined to represent the virtual end address of data memory (_end).

```
.bss    :
{
  *(.dynbss)
  *(.bss .bss.* .gnu.linkonce.b.*)
  *(COMMON)
  /*
   * Align here to ensure that the .bss section occupies
   * space up to _end.  Align after .bss to ensure correct
   * alignment even if the .bss section disappears because
   * there are no input sections.
```

```
  */
  . = ALIGN(32 / 8) ;
} > kseg1_data_mem
. = ALIGN(32 / 8) ;
_end = . ;
_bss_end = . ;
```

5.8.4.20   .HEAP SECTION

This section reserves space for the heap, which is required for dynamic memory allocation. A symbol is defined to represent the virtual address of the heap (_heap). The minimum amount of space reserved for the heap is determined by the symbol _min_heap_size.

```
/* Heap allocating takes a chunk of memory following BSS */
.heap ALIGN(4) :
{
  _heap = . ;
  . += _min_heap_size ;
} > kseg1_data_mem
```

5.8.4.21   .STACK SECTION

This section reserves space for the stack. The minimum amount of space reserved for the stack is determined by the symbol _min_stack_size.

```
/* Stack allocation follows the heap */
.stack ALIGN(4) :
{
  . += _min_stack_size ;
} > kseg1_data_mem
```

5.8.4.22   .RAMFUNC SECTION

This section collects the RAM functions from all of the application's input files. This section is assigned to the data memory region (kseg1_data_mem) with a load address located in the program memory region (kseg0_program_mem). Symbols are defined to represent the virtual begin (_ramfunc_begin) and end (_ramfunc_end) addresses of this section, as well as the physical begin address of the RAM functions in program memory (_ramfunc_image_begin) and a length of the RAM functions (_ramfunc_length). In addition, the addresses for the bus matrix registers are calculated (_bmxdkpba_address, _bmxdudba_address, and _bmxdupba_address).

```
/*
 * RAM functions go at the end of our stack and heap allocation.
 * Alignment of 2K required by the boundary register (BMXDKPBA).
 */
.ramfunc ALIGN(2K) :
{
  _ramfunc_begin = . ;
  *(.ramfunc  .ramfunc.*)
  . = ALIGN(4) ;
  _ramfunc_end = . ;
} > kseg1_data_mem AT> kseg0_program_mem
_ramfunc_image_begin = LOADADDR(.ramfunc) ;
_ramfunc_length = SIZEOF(.ramfunc) ;
_bmxdkpba_address = _ramfunc_begin - ORIGIN(kseg1_data_mem) ;
_bmxdudba_address = LENGTH(kseg1_data_mem);
_bmxdupba_address = LENGTH(kseg1_data_mem);
```

5.8.4.23 STACK LOCATION

A symbol is defined to represent the location of the Stack Pointer (`_stack`). This location is dependent on whether RAM functions exist in the application. If RAM functions exist, then the location of the Stack Pointer should include the gap between the stack section and the beginning of the `.ramfunc` section caused by the alignment of the `.ramfunc` section minus one word. If RAM functions do not exist, then the location of the Stack Pointer should be the end of the KSEG1 data memory.

```
/*
 * The actual top of stack should include the gap between
 * the stack section and the beginning of the .ramfunc
 * section caused by the alignment of the .ramfunc section
 * minus 1 word.  If RAM functions do not exist, then the top
 * of the stack should point to the end of the kseg1 data
 * memory.
 */
_stack = (_ramfunc_length > 0)
       ? _ramfunc_begin - 4
       : ORIGIN(kseg1_data_mem) + LENGTH(kseg1_data_mem) ;
ASSERT((_min_stack_size + _min_heap_size) <= (_stack - _heap),
   "Not enough space to allocate both stack and heap.  Reduce heap
and/or stack size.")
```

5.8.4.24 DEBUG SECTIONS

The debug sections contain debugging information. They are not loaded into program flash.

```
/* Stabs debugging sections.  */
.stab         0 : { *(.stab) }
.stabstr      0 : { *(.stabstr) }
.stab.excl    0 : { *(.stab.excl) }
.stab.exclstr 0 : { *(.stab.exclstr) }
.stab.index   0 : { *(.stab.index) }
.stab.indexstr 0 : { *(.stab.indexstr) }
.comment      0 : { *(.comment) }
/* DWARF debug sections.
   Symbols in the DWARF debugging sections are relative
   to the beginning of the section so we begin them at 0.  */
/* DWARF 1 */
.debug        0 : { *(.debug) }
.line         0 : { *(.line) }
/* GNU DWARF 1 extensions */
.debug_srcinfo  0 : { *(.debug_srcinfo) }
.debug_sfnames  0 : { *(.debug_sfnames) }
/* DWARF 1.1 and DWARF 2 */
.debug_aranges  0 : { *(.debug_aranges) }
.debug_pubnames 0 : { *(.debug_pubnames) }


/* DWARF 2 */
.debug_info     0 : { *(.debug_info .gnu.linkonce.wi.*) }
.debug_abbrev   0 : { *(.debug_abbrev) }
.debug_line     0 : { *(.debug_line) }
.debug_frame    0 : { *(.debug_frame) }
.debug_str      0 : { *(.debug_str) }
.debug_loc      0 : { *(.debug_loc) }
.debug_macinfo  0 : { *(.debug_macinfo) }
/* SGI/MIPS DWARF 2 extensions */
.debug_weaknames 0 : { *(.debug_weaknames) }
.debug_funcnames 0 : { *(.debug_funcnames) }
```

```
.debug_typenames 0 : { *(.debug_typenames) }
.debug_varnames  0 : { *(.debug_varnames) }
/DISCARD/ : { *(.note.GNU-stack) }
```

5.9   RAM FUNCTIONS

Functions may be located in RAM to improve performance. The `__ramfunc__` and `__longramfunc__` specifiers are used on a function declaration to specify that the function will be executed out of RAM.

Functions specified as a RAM function will be copied to RAM by the start-up code and all calls to those functions will reference the RAM location. Functions located in RAM will be in a different 512MB memory segment than functions located in program memory, so the `longcall` attribute should be applied to any RAM function which will be called from a function not in RAM. The `__longramfunc__` specifier will apply the `longcall` attribute as well as place the function in RAM[1].

```
/* function 'foo' will be placed in RAM */
void __ramfunc__ foo (void)
{
}

/* function 'bar' will be placed in RAM and will be invoked
   using the full 32 bit address */
void __longramfunc__ bar (void)
{
}
```

---

1. Specifying `__longramfunc__` is functionally equivalent to specifying both `__ramfunc__` and `__longcall__`.

NOTES:

Appendix A. Implementation Defined Behavior

A.1 INTRODUCTION

This chapter discusses the choices for implementation defined behavior in compiler.

A.2 HIGHLIGHTS

Items discussed in this chapter are:

- Overview
- Translation
- Environment
- Identifiers
- Characters
- Integers
- Floating-Point
- Arrays and Pointers
- Hints
- Structures, Unions, Enumerations, and Bit fields
- Qualifiers
- Declarators
- Statements
- Pre-Processing Directives
- Library Functions
- Architecture

A.3 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.

A.4 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.

A.5   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 execute.

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.

A.6 IDENTIFIERS

ISO Standard: "Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2)."

Implementation: No.

ISO Standard: "The number of significant initial characters in an identifier (5.2.4.1, 6.4.2)."

Implementation: All characters are significant.

A.7 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, signed char is functionally equivalent to plain char. The options `-funsigned-char` and `-fsigned-char` can be used to change the default.

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.

A.8 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.

A.9   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.

A.10 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.

A.11 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.

A.12 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, a plain `int` bit field is treated as a signed integer. This behavior can be altered by use of the `-funsigned-bitfields` command line option.

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.

A.13 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 */
```

A.14 DECLARATORS

ISO Standard:	"The maximum number of declarators that may modify an arithmetic, structure or union type (C90 6.5.4)."
Implementation:	No limit.

A.15 STATEMENTS

ISO Standard:	"The maximum number of case values in a switch statement (C90 6.6.4.2)."
Implementation:	No limit.

A.16 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/pic32mx/3.4.4/include`

`<install directory>/pic32mx/include`

ISO Standard: "How the named source file is searched for in an included " " delimited header(C90 6.8.2, C99 6.10.2)."

Implementation: The compiler first searches for the named file in the directory containing the including file, the directories specified by the `-iquote` command line option (if any), then the directories which are searched for a < > delimited header.

ISO Standard: "The method by which preprocessing tokens are combined into a header name (C90 6.8.2, C99 6.10.2)."

Implementation: All tokens, including whitespace, are considered part of the header file name. Macro expansion is not performed on tokens inside the delimiters.

ISO Standard: "The nesting limit for `#include` processing (C90 6.8.2, C99 6.10.2)."

Implementation: No limit.

ISO Standard: "The behavior on each recognized non-`STDC` #pragma directive (C90 6.8.6, C99 6.10.6)."

Implementation: See **Section 1.7 "Attributes and Pragmas"**.

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.

A.17 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 filename, then `unlink` is called to remove the old filename. Typically, `link` will fail if the new filename 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.

MPLAB® C Compiler for PIC32 MCUs User's Guide

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.

A.18 ARCHITECTURE

ISO Standard: "The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (C90 and C99 5.2.4.2, C99 7.18.2, 7.18.3)."

Implementation: See **Section 1.5.6 "`limits.h`"**.

ISO Standard: "The number, order, and encoding of bytes in any object (when not explicitly specified in the standard) (C99 6.2.6.1)."

Implementation: Little endian, populated from Least Significant Byte first. See **Section 1.5 "Data Storage"**.

ISO Standard: "The value of the result of the size of operator (C90 6.3.3.4, C99 6.5.3.4)."

Implementation: See **Section 1.5 "Data Storage"**.

NOTES:

Appendix B. Open Source Licensing

B.1 INTRODUCTION

This chapter gives a summary of the open source licenses used for portions of the compiler package.

B.2 GENERAL PUBLIC LICENSE

The executables for the compiler, assembler, linker, and associated binary utilities are covered under the GNU General Public License. See the file doc/COPYING.GPL in the product installation directory for the full text of the license.

B.3 BSD LICENSE

Portions of the standard library are distributed under the terms of the "BSD" license from the University of California:

Copyright © Regents of the University of California.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

This product includes software developed by the University of California, Berkeley and its contributors.

4. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

B.4   SUN MICROSYSTEMS

Portions of the standard library are copyright Sun Microsystems and are distributed under the permissions granted by the following terms:

Developed at SunPro, a Sun Microsystems, Inc. business. Permission to use, copy, modify, and distribute this software is freely granted, provided that this notice is preserved.

Index

Index

MPLAB® C Compiler for PIC32 MCUs User's Guide

Index

NOTES:

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Cleveland
Independence, OH
Tel: 216-447-0464
Fax: 216-447-0643

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4080

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-6578-300
Fax: 886-3-6578-370

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

03/26/09