MPLAB® C18 C COMPILER USER'S GUIDE

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

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

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

Trademarks

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

Microchip received ISO/TS-16949:2002 quality system certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona and Mountain View, California in October 2003. The Company's quality system processes and procedures are for its PICmicro® 8-bit MCUs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Table of Contents

Appendix E. Extended Mode

NOTES:

MPLAB® C18 C COMPILER USER'S GUIDE

Preface

NOTICE TO CUSTOMERS

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

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

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

INTRODUCTION

This document discusses the technical details of the MPLAB® C18 compiler. This document will explain all functionality of the MPLAB C18 compiler. It assumes that the programmer already:

- knows how to write C programs
- knows how to use the MPLAB Integrated Development Environment (IDE) to create and debug projects
- has read and understands the processor data sheet for which code is being written

DOCUMENT LAYOUT

This document layout is as follows:

- **Chapter 1: Introduction** – Provides an overview of the MPLAB C18 compiler and information on invoking the compiler.
- **Chapter 2: Language Specifics** – Discusses how the MPLAB C18 compiler differs from the ANSI standard.
- **Chapter 3: Run-time Model** – Discusses how the MPLAB C18 compiler utilizes the resources of the PIC18 PICmicro® microcontrollers.
- **Chapter 4: Optimizations** – Discusses the optimizations that are performed by the MPLAB C18 compiler.
- **Chapter 5: Sample Application** – Provides a sample application and describes the source code with references to the specific topics discussed in the User's Guide.

- **Appendix A: COFF File Format** – Provides details of the Microchip COFF format.
- **Appendix B: ANSI Implementation-defined Behavior** – Discusses MPLAB C18 implementation-defined behavior as required by the ANSI standard.
- **Appendix C: Command-line Summary** – Lists command-line options along with references to sections that discuss each of the command-line options.
- **Appendix D: MPLAB C18 Diagnostics** – Lists errors, warnings and messages.
- **Appendix E: Extended Mode** – Discusses differences between Non-extended and Extended modes.

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*
Courier font:		
Plain Courier	Sample source code	`#define START`
	Filenames	`autoexec.bat`
	File paths	`c:\mcc18\h`
	Keywords	`_asm, _endasm, static`
	Command-line options	`-Opa+, -Opa-`
Italic Courier	A variable argument	*`file`*`.o`, where *`file`* can be any valid filename
`0b`*`nnnn`*	A binary number where *n* is a binary digit	`0b00100`, `0b10`
`0x`*`nnnn`*	A hexadecimal number where *n* is a hexadecimal digit	`0xFFFF`, `0x007A`
Square brackets [ ]	Optional arguments	`mcc18 [options]` *`file`* `[options]`
Ellipses...	Replaces repeated text	`var_name [, var_name...]`

	Represents code supplied by user	`void main (void)`
		`{ ...`
		`}`

Icon:		
	Features supported only in the full version of the software.	 **1.2.5 Selecting the Mode**

PIC18 DEVELOPMENT REFERENCES

readme.c18

This file is included with the software and contains update information that may not be included in this document.

PIC18 Configuration Settings Addendum (DS51537)

Lists the Configuration Bit Settings for the Microchip PIC18 devices supported by the MPLAB C18 C compiler's `#pragma config` directive and the MPASM™ `CONFIG` directive.

MPLAB® C18 C Compiler Getting Started Guide (DS51295)

Describes how to install the MPLAB C18 compiler, how to write simple programs and how to use the MPLAB IDE with the compiler.

MPLAB® C18 C Compiler Libraries (DS51297)

Reference guide for MPLAB C18 libraries and precompiled object files. Lists all library functions provided with the MPLAB C18 C compiler with detailed descriptions of their use.

MPLAB® IDE V6.XX Quick Start Guide (DS51281)

Describes how to set up the MPLAB IDE software and use it to create projects and program devices.

MPASM™ User's Guide with MPLINK™ Linker and MPLIB™ Librarian (DS33014)

Describes how to use the Microchip PICmicro MCU assembler (MPASM), linker (MPLINK) and librarian (MPLIB).

PICmicro® 18C MCU Family Reference Manual (DS39500)

Focuses on the Enhanced MCU family of devices. The operation of the Enhanced MCU family architecture and peripheral modules is explained but does not cover the specifics of each device.

PIC18 Device Data Sheets and Application Notes

Data sheets describe the operation and electrical specifications of PIC18 devices. Application notes describe how to use PIC18 devices.

To obtain any of the above listed documents, visit the Microchip web site (www.microchip.com) to retrieve these documents in Adobe Acrobat (.pdf) format.

C REFERENCES

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.

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

Covers the C programming language in great detail. This book is an authoritative reference manual that provides a complete description of the C language, the run-time libraries and a style of C programming that emphasizes correctness, portability and maintainability.

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

Presents a concise exposition of C as defined by the ANSI standard. This book is an excellent reference for C programmers.

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

Another excellent reference for learning ANSI C, used in colleges and universities.

Peatman, John B. *Embedded Design with the PIC18F452 Microcontroller*, First Edition. Pearson Education, Inc., Upper Saddle River, New Jersey 07458.

Focuses on Microchip Technology's PIC18FXXX family and writing enhanced application code.

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

Although this book focuses on Motorola microcontrollers, the basic principles of programming with C for microcontrollers is useful.

OTHER REFERENCES

Standards Committee of the IEEE Computer Society - *IEEE Standard for Binary Floating-Point Arithmetic*. The Institute of Electrical and Electronics Engineers, Inc, 345 East 47th. Street, New York, New York, 10017.

This standard describes the floating point format used in MPLAB C18.

THE MICROCHIP WEB SITE

Microchip provides online support via our WWW 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 (FAQ), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

DEVELOPMENT SYSTEMS CUSTOMER CHANGE NOTIFICATION SERVICE

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

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

The Development Systems product group categories are:

- **Compilers** – The latest information on Microchip C compilers and other language tools. These include the MPLAB C17, MPLAB C18 and MPLAB C30 C compilers; MPASM™ and MPLAB ASM30 assemblers; MPLINK™ and MPLAB LINK30 object linkers; and MPLIB™ and MPLAB LIB30 object librarians.
- **Emulators** – The latest information on Microchip in-circuit emulators.This includes the MPLAB ICE 2000 and MPLAB ICE 4000.
- **In-Circuit Debuggers** – The latest information on the Microchip in-circuit debugger, MPLAB ICD 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 SIM and MPLAB SIM30 simulators, MPLAB IDE Project Manager and general editing and debugging features.
- **Programmers** – The latest information on Microchip programmers. These include the MPLAB PM3 and PRO MATE® II device programmers and the PICSTART® Plus development programmer.

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support
- Development Systems Information Line

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

In addition, there is a Development Systems Information Line which lists the latest versions of Microchip's development systems software products. This line also provides information on how customers can receive currently available upgrade kits.

The Development Systems Information Line numbers are:

1-800-755-2345 – United States and most of Canada

1-480-792-7302 – Other International Locations

Chapter 1. Introduction

1.1 OVERVIEW

The MPLAB C18 compiler is a free-standing, optimizing ANSI C compiler for the PIC18 PICmicro microcontrollers (MCU). The compiler deviates from the ANSI standard X3.159-1989 only where the standard conflicts with efficient PICmicro MCU support. The compiler is a 32-bit Windows console application and is fully compatible with Microchip's MPLAB IDE, allowing source-level debugging with the MPLAB ICE in-circuit emulator, the MPLAB ICD 2 in-circuit debugger or the MPLAB SIM simulator.

The MPLAB C18 compiler has the following features:

- ANSI '89 compatibility
- Integration with the MPLAB IDE for easy-to-use project management and source-level debugging
- Generation of relocatable object modules for enhanced code reuse
- Compatibility with object modules generated by the MPASM assembler, allowing complete freedom in mixing assembly and C programming in a single project
- Transparent read/write access to external memory
- Strong support for inline assembly when total control is absolutely necessary
- Efficient code generator engine with multi-level optimization
- Extensive library support, including PWM, SPI™, I^2C™, UART, USART, string manipulation and math libraries
- Full user-level control over data and code memory allocation

1.2 INVOKING THE COMPILER

The *MPLAB® C18 Getting Started Guide* (DS51295) describes how to use the compiler with the MPLAB IDE. The compiler can also be invoked from the command line. The command-line usage is:

```
mcc18 [options] file [options]
```

A single source file and any number of command-line options can be specified. The `--help` command-line option lists all command-line options accepted by the compiler. The `-verbose` command-line option causes the compiler to show a banner containing the version number and the total number of errors, warnings and messages upon completion.

1.2.1 Creating Output Files

By default, the compiler will generate an output object file named *file*.o, where *file* is the name of the source file specified on the command line minus the extension. The output object file name can be overridden with the -fo command-line option. For example:

```
mcc18 -fo bar.o foo.c
```

If the source file contains errors, then the compiler generates an error file named *file*.err, where *file* is the name of the source file specified on the command line minus the extension. The error file name can be overridden using the -fe command-line option. For example:

```
mcc18 -fe bar.err foo.c
```

1.2.2 Displaying Diagnostics

Diagnostics can be controlled using the -w and -nw command-line options. The -w command-line option sets the level of warning diagnostics (1, 2 or 3). Table 1-1 shows the level of warning diagnostics and the type of diagnostics that are shown. The -nw command-line option suppresses specific messages (**Appendix D. "MPLAB C18 Diagnostics"** or the --help-message-list command-line option lists all messages generated by the compiler). Help on all messages can be seen using the --help-message-all command-line option. For help on a specific diagnostic, the --help-message command-line option can be used. For example:

```
mcc18 --help-message=2068
```

displays the following results:

```
2068: obsolete use of implicit 'int' detected.

The ANSI standard allows a variable to be declared without a base type
being specified, e.g., "extern x;", in which case a base type of 'int'
is implied. This usage is deprecated by the standard as obsolete, and
therefore a diagnostic is issued to that effect.
```

TABLE 1-1: WARNING LEVELS

Warning Level	Diagnostics Shown
1	Errors (fatal and non-fatal)
2	Level 1 plus warnings
3	Level 2 plus messages

1.2.3 Defining Macros

The -D command-line option allows a macro to be defined. The -D command-line option can be specified in one of two ways: -D*name* or -D*name=value*. -D*name* defines the macro *name* with 1 as its definition. -D*name=value* defines the macro *name* with *value* as its definition. For example:

```
mcc18 -DMODE
```

defines the macro MODE to have a value of 1, whereas:

```
mcc18 -DMODE=2
```

defines the macro MODE to have a value of 2.

An instance of utilizing the -D command-line option is in conditional compilation of code. For example:

```
#if MODE == 1
   x = 5;
#elif MODE == 2
   x = 6;
#else
   x = 7;
#endif
```

1.2.4 Selecting the Processor

By default, MPLAB C18 compiles an application for a generic PIC18 PICmicro microcontroller. The object file can be limited to a specific processor with the -p*processor* command-line option, where *processor* specifies the particular processor to utilize. For example, to limit an object file for use with only the PIC18F452, the command-line option -p18f452 should be used. The command-line option -p18cxx explicitly specifies that the source is being compiled for a generic PIC18 microcontroller.

1.2.5 Selecting the Mode

The compiler can operate in one of two different modes: Extended[1] or Non-extended. When operating in the Extended mode, the compiler will utilize the extended instructions (i.e., ADDFSR, ADDULNK, CALLW, MOVSF, MOVSS, PUSHL, SUBFSR and SUBULNK) and the indexed with literal offset addressing, which generally requires fewer instructions for accessing stack-based variables (resulting in a smaller program memory image). When operating in Non-extended mode, the compiler will not utilize the extended instructions nor the indexed with literal offset addressing. The --extended and --no-extended command-line options tell the compiler the mode in which to operate. When the --extended command-line option is specified, the compiler expects that the processor selected with the -p option supports the extended instruction set or is being compiled for a generic PIC18 microcontroller (see **Section 1.2.4 "Selecting the Processor"**). The --no-extended command-line option can be utilized with any PIC18 microcontroller, including the generic microcontroller. If neither the --extended nor the --no-extended command-line options is specified on the command line, the compiler will operate in Non-extended mode, regardless of the processor selected. Table 1-2 outlines the mode in which the compiler will operate based on the command-line options specified.

1. When the time limit for the demo version expires, the compiler cannot operate in Extended mode.

TABLE 1-2: MODE SELECTION

	`-p` ***extended***	`-p` ***no-extended***	`-p18cxx`	**No Processor Specified**
`--extended`	Extended	Error	Extended	Extended
`--no-extended`	Non-extended	Non-extended	Non-extended	Non-extended
Not Specified	Non-extended	Non-extended	Non-extended	Non-extended

Note: If the compiler is invoked with `mcc18 --help`, the help displayed will be for the compiler operating in the Non-extended mode; however, not all of the command-line options are valid when the compiler is operating in the Extended mode. The command line `mcc18 --extended --help` should be utilized to see help for the compiler operating in the Extended mode.

Note: Other command-line options are discussed throughout the User's Guide, and a summary of all the command-line options can be found in **Appendix C. "Command-line Summary"**.

Chapter 2. Language Specifics

2.1 DATA TYPES AND LIMITS

2.1.1 Integer Types

The MPLAB C18 compiler supports the standard ANSI-defined integer types. The ranges of the standard integer types are documented in Table 2-1. In addition, MPLAB C18 supports a 24-bit integer type `short long int` (or `long short int`), in both a signed and unsigned variety. The ranges of this type are also documented in Table 2-1.

TABLE 2-1: INTEGER DATA TYPE SIZES AND LIMITS

Type	Size	Minimum	Maximum
`char`(1,2)	8 bits	-128	127
`signed char`	8 bits	-128	127
`unsigned char`	8 bits	0	255
`int`	16 bits	-32,768	32,767
`unsigned int`	16 bits	0	65,535
`short`	16 bits	-32,768	32,767
`unsigned short`	16 bits	0	65,535
`short long`	24 bits	-8,388,608	8,388,607
`unsigned short long`	24 bits	0	16,777,215
`long`	32 bits	-2,147,483,648	2,147,483,647
`unsigned long`	32 bits	0	4,294,967,295

Note 1: A plain *`char`* is signed by default.

2: A plain *`char`* may be unsigned by default via the `-k` command-line option.

2.1.2 Floating-point Types

32-bit floating-point types are native to MPLAB C18 using either the `double` or `float` data types. MPLAB C18 utilizes the IEEE-754 floating-point standard to represent floating-point types. The ranges of the floating-point type are documented in Table 2-2.

TABLE 2-2: FLOATING-POINT DATA TYPE SIZES AND LIMITS

Type	Size	Minimum Exponent	Maximum Exponent	Minimum Normalized	Maximum Normalized
`float`	32 bits	-126	128	$2^{-126} \approx$ 1.17549435e - 38	$2^{128} * (2\text{-}2^{-15}) \approx$ 6.80564693e + 38
`double`	32 bits	-126	128	$2^{-126} \approx$ 1.17549435e - 38	$2^{128} * (2\text{-}2^{-15}) \approx$ 6.80564693e + 38

2.2 DATA TYPE STORAGE - ENDIANNESS

Endianness refers to the ordering of bytes in a multi-byte value. MPLAB C18 stores data in little-endian format. Bytes at lower addresses have lower significance (the value is stored "little-end-first"). For example:

```
#pragma idata test=0x0200
long l=0xAABBCCDD;
```

results in a memory layout as follows:

Address	*0x0200*	*0x0201*	*0x0202*	*0x0203*
Content	0xDD	0xCC	0xBB	0xAA

2.3 STORAGE CLASSES

MPLAB C18 supports the ANSI standard storage classes (`auto`, `extern`, `register`, `static` and `typedef`).

2.3.1 Overlay

The MPLAB C18 compiler introduces a storage class of `overlay`. The `overlay` storage class applies only when the compiler is operating in Non-extended mode (see **Section 1.2.5 "Selecting the Mode"**). The `overlay` storage class may be applied to local variables (but not formal parameters, function definitions or global variables). The `overlay` storage class will allocate the associated symbols into a function-specific, static overlay section. Such a variable will be allocated statically, but initialized upon each function entry. For example, in:

```
void f (void)
{
  overlay int x = 5;
  x++;
}
```

`x` will be initialized to 5 upon each function entry, although its storage will be statically allocated. If no initializer is present, then its value upon function entry is undefined.

The MPLINK linker will attempt to overlay local storage specified as `overlay` from functions that are guaranteed not to be active simultaneously. For example, in:

```
int f (void)
{
  overlay int x = 1;
  return x;
}

int g (void)
{
  overlay int y = 2;
  return y;
}
```

If f and g will never be active at the same time, x and y become candidates for sharing the same memory location. However, in:

```
int f (void)
{
  overlay int x = 1;
  return x;
}

int g (void)
{
  overlay int y = 2;
  y = f ();
  return y;
}
```

since f and g may be simultaneously active, x and y will not be overlaid. The advantage of using overlay locals is that they are statically allocated, which means that, in general, fewer instructions are required to access them (resulting in a smaller program memory image). At the same time, the total data memory allocation required for these variables may be less than what would be required had they been declared as static due to the fact that some of the variables may be overlaid.

If the MPLINK linker detects a recursive function that contains a local variable of storage class overlay, it emits an error and aborts. If the MPLINK linker detects a call through a function pointer in any module and a local variable of storage class overlay in any (and not necessarily the same) module, it emits an error and aborts.

The default storage class for local variables is auto. This can be overridden explicitly with the static or overlay keywords or implicitly with either the -scs (static local variables) or -sco (overlay local variables) command-line option. For completeness, MPLAB C18 also supports the -sca command-line option. This option allows the storage class for local variables to be explicitly specified as auto.

2.3.2 static Function Arguments

Function parameters can have storage class auto or static. An auto parameter is placed on the software stack, enabling reentrancy. A static parameter is allocated globally, enabling direct access for generally smaller code. static parameters are valid only when the compiler is operating in Non-extended mode (see **Section 1.2.5 "Selecting the Mode"**).

The default storage class for function parameters is auto. This can be overridden explicitly with the static keyword or implicitly with the -scs command-line option. The -sco command-line option will also implicitly override function parameters' storage class with static.

2.4 STORAGE QUALIFIERS

In addition to the ANSI standard storage qualifiers (`const`, `volatile`), the MPLAB C18 compiler introduces storage qualifiers of `far`, `near`, `rom` and `ram`. Syntactically, these new qualifiers bind to identifiers just as the `const` and `volatile` qualifiers do in ANSI C. Table 2-3 shows the location of an object based on the storage qualifiers specified when it was defined. The default storage qualifiers for an object defined without explicit storage qualifiers are `far` and `ram`.

TABLE 2-3: LOCATION OF OBJECT BASED ON STORAGE QUALIFIERS

	rom	**ram**
`far`	Anywhere in program memory	Anywhere in data memory (default)
`near`	In program memory with address less than 64K	In access memory

2.4.1 `near/far` Data Memory Objects

The `far` qualifier is used to denote that a variable that is located in data memory lives in a memory bank and that a bank switching instruction is required prior to accessing this variable. The `near` qualifier is used to denote that a variable located in data memory lives in access RAM.

2.4.2 `near/far` Program Memory Objects

The `far` qualifier is used to denote that a variable that is located in program memory can be found anywhere in program memory, or, if a pointer, that it can access up to and beyond 64K of program memory space. The `near` qualifier is used to denote that a variable located in program memory is found at an address less than 64K, or, if a pointer, that it can access only up to 64K of program memory space.

2.4.3 `ram/rom` Qualifiers

Because the PICmicro microcontrollers use separate program memory and data memory address busses in their design, MPLAB C18 requires extensions to distinguish between data located in program memory and data located in data memory. The ANSI/ISO C standard allows for code and data to be in separate address spaces, but this is not sufficient to locate data in the code space as well. To this purpose, MPLAB C18 introduces the `rom` and `ram` qualifiers. The `rom` qualifier denotes that the object is located in program memory, whereas the `ram` qualifier denotes that the object is located in data memory.

Pointers can point to either data memory (`ram` pointers) or program memory (`rom` pointers). Pointers are assumed to be `ram` pointers unless declared as `rom`. The size of a pointer is dependent on the type of the pointer and is documented in Table 2-4.

Note: When writing to a `rom` variable, the compiler uses a `TBLWT` instruction; however, there may be additional application code that needs to be written based on the type of memory being utilized. See the data sheet for more information.

TABLE 2-4: POINTER SIZES

Pointer Type	Example	Size
Data memory pointer	`char * dmp;`	16 bits
Near program memory pointer	`rom near char * npmp;`	16 bits
Far program memory pointer	`rom far char * fpmp;`	24 bits

2.5 INCLUDE FILE SEARCH PATHS

2.5.1 System Header Files

Source files included with `#include <filename>` are searched for in the path specified in the `MCC_INCLUDE` environment variable and the directories specified via the `-I` command-line option. Both the `MCC_INCLUDE` environment variable and the `-I` values are a semi-colon delimited list of directories to search. If the included file exists in both a directory listed in the `MCC_INCLUDE` environment variable and a directory listed in a `-I` command-line option, the file will be included from the directory listed in the `-I` command-line option. This allows the `MCC_INCLUDE` environment variable to be overridden with a `-I` command-line option.

2.5.2 User Header Files

Source files included with `#include` "`filename`" are searched for in the directory containing the including file. If not found, the file is searched for as a system header file (see **Section 2.5.1 "System Header Files"**).

2.6 PREDEFINED MACRO NAMES

In addition to the standard predefined macro names, MPLAB C18 provides the following predefined macros:

`__18CXX` The constant 1, intended to indicate the MPLAB C18 compiler.

`__`*`PROCESSOR`* The constant 1 if compiled for the particular processor. For example, `__18C452` would be defined as the constant 1 if compiled with the `-p18c452` command-line option and `__18F258` would be defined as the constant 1 if compiled with the `-p18f258` command-line option.

`__SMALL__` The constant 1 if compiled with the `-ms` command-line option.

`__LARGE__` The constant 1 if compiled with the `-ml` command-line option.

`__TRADITIONAL18__` The constant 1 if the Non-extended mode is being used (see **Section 1.2.5 "Selecting the Mode"**).

`__EXTENDED18__` The constant 1 if the Extended mode is being used (see **Section 1.2.5 "Selecting the Mode"**).

2.7 ISO DIVERGENCES

2.7.1 Integer Promotions

ISO mandates that all arithmetic be performed at `int` precision or greater. By default, MPLAB C18 will perform arithmetic at the size of the largest operand, even if both operands are smaller than an `int`. The ISO mandated behavior can be instated via the `-Oi` command-line option.

For example:

```
unsigned char a, b;
unsigned i;

a = b = 0x80;
i = a + b; /* ISO requires that i == 0x100, but in C18 i == 0 */
```

Note that this divergence also applies to constant literals. The chosen type for constant literals is the first one from the appropriate group that can represent the value of the constant without overflow.

For example:

```
#define A 0x10 /* A will be considered a char unless -Oi
                  specified */
#define B 0x10 /* B will be considered a char unless -Oi
                  specified */
#define C (A) * (B)

unsigned i;
i = C; /* ISO requires that i == 0x100, but in C18 i == 0 */
```

2.7.2 Numeric Constants

MPLAB C18 supports the standard prefixes for specifying hexadecimal (`0x`) and octal (`0`) values and adds support for specifying binary values using the `0b` prefix. For example, the value two hundred thirty seven may be denoted as the binary constant `0b11101101`.

2.7.3 String Constants

The primary use of data located in program memory is for static strings. In keeping with this, MPLAB C18 automatically places all string constants in program memory. This type of a string constant is "array of `char` located in program memory", (`const rom char []`). The `.stringtable` section is a `romdata` (see **Section 2.9.1 "`#pragma sectiontype`"**) section that contains all constant strings. For example the string "`hello`" in the following would be located in the `.stringtable` section:

```
strcmppgm2ram (Foo, "hello");
```

Due to the fact that constant strings are kept in program memory, there are multiple versions of the standard functions that deal with strings. For example, the `strcpy` function has four variants, allowing the copying of a string to and from data and program memory:

```
/*
 * Copy string s2 in data memory to string s1 in data memory
 */
char *strcpy (auto char *s1, auto const char *s2);

/*
 * Copy string s2 in program memory to string s1 in data
 * memory
 */
char *strcpypgm2ram (auto char *s1, auto const rom char *s2);

/*
 * Copy string s2 in data memory to string s1 in program
 * memory
 */
rom char *strcpyram2pgm (auto rom char *s1, auto const char *s2);

/*
 * Copy string s2 in program memory to string s1 in program
 * memory
 */
rom char *strcpypgm2pgm (auto rom char *s1,
                         auto const rom char *s2);
```

When using MPLAB C18, a string table in program memory can be declared as:

```
rom const char table[][20] =   { "string 1", "string 2",
                                 "string 3", "string 4" };
rom const char *rom table2[] = { "string 1", "string 2",
                                 "string 3", "string 4" };
```

The declaration of `table` declares an array of four strings that are each 20 characters long, and so takes 80 bytes of program memory. `table2` is declared as an array of pointers to program memory. The `rom` qualifier after the `*` places the array of pointers in program memory as well. All of the strings in `table2` are 9 bytes long, and the array is four elements long, so `table2` takes (9*4+4*2) = 44 bytes of program memory. Accesses to `table2` may be less efficient than accesses to `table`, however, because of the additional level of indirection required by the pointer.

An important consequence of the separate address spaces for MPLAB C18 is that pointers to data in program memory and pointers to data in data memory are not compatible. Two pointer types are not compatible unless they point to objects of compatible types and the objects they point to are located in the same address space. For example, a pointer to a string in program memory and a pointer to a string in data memory are not compatible because they refer to different address spaces.

A function to copy a string from program to data memory could be written as follows:

```
void str2ram(static char *dest, static char rom *src)
{
  while ((*dest++ = *src++) != '\0')
    ;
}
```

The following code will send a string located in program memory to the USART on a PIC18C452 using the PICmicro MCU C libraries. The library function to send a string to the USART, `putsUSART(const char *str)`, takes a pointer to a string as its argument, but that string must be in data memory.

```
rom char mystring[] = "Send me to the USART";

void foo( void )
{
  char strbuffer[21];
  str2ram (strbuffer, mystring);
  putsUSART (strbuffer);
}
```

Alternatively, the library routine can be modified to read from a string located in program memory.

```
/*
 * The only changes required to the library routine are to
 * change the name so the new routine does not conflict with
 * the original routine and to add the rom qualifier to the
 * parameter.
 */
void putsUSART_rom( static const rom char *data )
{
  /* Send characters up to the null */
  do
  {
    while (BusyUSART())
      ;

    /* Write a byte to the USART */
    putcUSART (*data);
  } while (*data++);
}
```

2.7.4 `stdio.h` Functions

The output functions defined in `stdio.h` differ from the ANSI defined versions with regards to data in program memory, floating-point format support, and MPLAB C18 specific extensions.

The functions `puts` and `fputs` expect the output string to be stored in program memory. The functions `vsprintf`, `vprintf`, `sprintf`, `printf`, `fprintf` and `vfprintf` expect the format string to be stored in program memory.

The functions `vsprintf`, `vprintf`, `sprintf`, `printf`, `fprintf` and `vfprintf` do not support floating-point conversion specifiers.

The MPLAB C18 specific extensions for 24-bit integers and data in program memory are described in Section 4.7 of *MPLAB C18 C Compiler Libraries*.

2.8 LANGUAGE EXTENSIONS

2.8.1 Anonymous Structures

MPLAB C18 supports anonymous structures inside of unions. An anonymous structure has the form:

```
struct { member-list };
```

An anonymous structure defines an unnamed object. The names of the members of an anonymous structure must be distinct from other names in the scope in which the structure is declared. The members are used directly in that scope without the usual member access syntax.

For example:

```
union foo
{
  struct
  {
    int a;
    int b;
  };
  char c;
} bar;
char c;

...

bar.a = c; /* 'a' is a member of the anonymous structure
              located inside 'bar' */
```

A structure for which objects or pointers are declared is not an anonymous structure. For example:

```
union foo
{
  struct
  {
    int a;
    int b;
  } f, *ptr;
  char c;
} bar;
char c:

...

bar.a = c;      /* error */
bar.ptr->a = c; /* ok */
```

The assignment to `bar.a` is illegal since the member name is not associated with any particular object.

2.8.2 Inline Assembly

MPLAB C18 provides an internal assembler using a syntax similar to the MPASM assembler. The block of assembly code must begin with `_asm` and end with `_endasm`. The syntax within the block is:

```
[label:] [<instruction> [arg1[, arg2[, arg3]]]]
```

The internal assembler differs from the MPASM assembler as follows:

- No directive support
- Comments must be C or C++ notation
- Full text mnemonics must be used for table reads/writes. i.e.,
  - TBLRD
  - TBLRDPOSTDEC
  - TBLRDPOSTINC
  - TBLRDPREINC
  - TBLWT
  - TBLWTPOSTDEC
  - TBLWTPOSTINC
  - TBLWTPREINC
- No defaults for instruction operands – all operands must be fully specified
- Default radix is decimal
- Literals are specified using C radix notation, not MPASM assembler notation. For example, a hex number should be specified as `0x1234`, not `H'1234'`.
- Label must include colon
- Indexed addressing syntax (i.e., `[]`) is not supported – must specify literal and access bit (e.g., specify as `CLRF 2,0`, not `CLRF [2]`)

For example:

```
_asm
  /* User assembly code */
  MOVLW 10       // Move decimal 10 to count
  MOVWF count, 0

  /* Loop until count is 0 */
  start:
    DECFSZ count, 1, 0
    GOTO done
    BRA start
  done:
_endasm
```

It is generally recommended to limit the use of inline assembly to a minimum. Any functions containing inline assembly will not be optimized by the compiler. To write large fragments of assembly code, use the MPASM assembler and link the modules to the C modules using the MPLINK linker.

2.9 PRAGMAS

2.9.1 #pragma *sectiontype*

The section declaration pragmas change the current section into which MPLAB C18 will allocate information of the associated type.

A section is a portion of an application located at a specific address of memory. Sections can contain code or data. A section can be located in either program or data memory. There are two types of sections for each type of memory.

- program memory
  - `code` – contains executable instructions
  - `romdata` – contains variables and constants
- data memory
  - `udata` – contains statically allocated uninitialized user variables
  - `idata` – contains statically allocated initialized user variables

Sections are absolute, assigned or unassigned. An absolute section is one that is given an explicit address via the `=address` of the section declaration pragma. An assigned section is one that is ascribed to a specific section via the `SECTION` directive of the linker script. An unassigned section is one that is neither absolute nor assigned.

2.9.1.1 SYNTAX

section-directive:

```
    # pragma udata [attribute-list] [section-name [=address]]
  | # pragma idata [attribute-list] [section-name [=address]]
  | # pragma romdata [overlay] [section-name [=address]]
  | # pragma code [overlay] [section-name [=address]]
```

attribute-list:

```
   attribute
  | attribute-list attribute
```

attribute:

```
    access
  | overlay
```

section-name: C identifier

address: integer constant

2.9.1.2 SECTION CONTENTS

A `code` section contains executable content, located in program memory. A `romdata` section contains data allocated into program memory (normally variables declared with the `rom` qualifier). For additional information on `romdata` usage (e.g., for memory-mapped peripherals) see the MPLINK linker's portion of the *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014). A `udata` section contains uninitialized global data statically allocated into data memory. An `idata` section contains initialized global data statically allocated into data memory.

Table 2-5 shows which section each of the objects in the following example will be located in:

```
rom int ri;
rom char rc = 'A';

int ui;
char uc;

int ii = 0;
char ic = 'A';

void foobar (void)
{
  static rom int foobar_ri;
  static rom char foobar_rc = 'Z';
  ...
}
void foo (void)
{
  static int foo_ui;
  static char foo_uc;
  ...
}

void bar (void)
{
  static int bar_ii = 5;
  static char bar_ic = 'Z';
  ...
}
```

TABLE 2-5: OBJECTS' SECTION LOCATION

Object	Section Location
ri	romdata
rc	romdata
foobar_ri	romdata
foobar_rc	romdata
ui	udata
uc	udata
foo_ui	udata
foo_uc	udata
ii	idata
ic	idata
bar_ii	idata
bar_ic	idata
foo	code
bar	code
foobar	code

2.9.1.3 DEFAULT SECTIONS

A default section exists for each section type in MPLAB C18 (see Table 2-6).

TABLE 2-6: DEFAULT SECTION NAMES

Section Type	Default Name
code	.code_*filename*
romdata	.romdata_*filename*
udata	.udata_*filename*
idata	.idata_*filename*

Note: *filename* is the name of the object file being generated. For example, "mcc18 foo.c -fo=foo.o" will produce an object file with a default code section named ".code_foo.o".

Specifying a section name that has been previously declared causes MPLAB C18 to resume allocating data of the associated type into the specified section. The section attributes must match the previous declaration; otherwise, an error will occur (see **Appendix D. "MPLAB C18 Diagnostics"**).

A section pragma directive with no name resets the allocation of data of the associated type to the default section for the current module. For example:

```
/*
 * The following statement changes the current code
 * section to the absolute section high_vector
 */
#pragma code high_vector=0x08
...

/*
 * The following statement returns to the default code
 * section
 */
#pragma code
...
```

When the MPLAB C18 compiler begins compiling a source file, it has default data sections for both initialized and uninitialized data. These default sections are located in either access or non-access RAM depending on whether the compiler was invoked with a `-Oa+` option or not, respectively. The `-Oa+` command-line option applies only when operating in Non-extended mode (see **Section 1.2.5 "Selecting the Mode"**). When a `#pragma udata [access]` *`name`* directive is encountered in the source code, the current uninitialized data section becomes *`name`*, which is located in access or non-access RAM depending on whether the optional `access` attribute was specified. The same is true for the current initialized data section when a `#pragma idata [access]` *`name`* directive is encountered.

Objects are placed in the current initialized data section when an object definition with an explicit initializer is encountered. Objects without an explicit initializer in their definition are placed in the current uninitialized data section. For example, in the following code snippet, `i` would be located in the current initialized data section and `u` would be placed in the current uninitialized data section.

```
int i = 5;
int u;

void main(void)
{
   ...
}
```

If an object's definition has an explicit `far` qualifier (see **Section 2.4 "Storage Qualifiers"**), the object is located in non-access memory. Similarly, an explicit `near` qualifier (see **Section 2.4 "Storage Qualifiers"**) tells the compiler that the object is located in access memory. If an object's definition has neither the `near` or `far` qualifier, the compiler looks at whether the `-Oa+` option was specified on the command line.

2.9.1.4 RESERVED SECTION NAMES

Table 2-7 lists the section names reserved for use by the compiler.

TABLE 2-7: RESERVED SECTION NAMES

Section Name	Purpose
`_entry_scn`	Contains a jump to the startup code. Located at the `RESET` vector.
`_startup_scn`	Contains the startup code, which calls the application's `main()` function.
`_cinit_scn`	Contains the startup function that performs data initialization.
`.cinit`	Contains a copy of initialized data in program memory that is used by the startup code to perform the initialization.
`MATH_DATA`	Contains arguments, return values, and temporary locations used by the math library functions.
`.tmpdata`	Contains the compiler temporary variables for the non-interrupt service routine source.
`isr``_tmp`	Contains the compiler temporary variables for the interrupt service routine, *`isr`* (see **Section 2.9.2 "`#pragma interruptlow` `fname` / `#pragma interrupt` `fname`"**).
`.stringtable`	Contains all constant strings (see **Section 2.7.3 "String Constants"**).
`.code_`*`filename`*	Contains, by default, the executable content for the file, *`filename`*.

TABLE 2-7: RESERVED SECTION NAMES (CONTINUED)

Section Name	Purpose
.idata_*filename*	Contains, by default, the initialized data for the file, *filename*.
.udata_*filename*	Contains, by default, the uninitialized data for the file, *filename*.
.romdata_*filename*	Contains, by default, the data allocated in program memory for the file, *filename*.
.config_*address_filename*	Contains the configuration settings specified for the given *address* and *filename*.
.stack	Contains the software stack.
CTYPE	Contains the executable content for the character classification functions (see *MPLAB C18 C Compiler Libraries*).
D100TCYXCODE	Contains the library function Delay100TCYx (see *MPLAB C18 C Compiler Libraries*).
D10KTCYXCODE	Contains the library function Delay10KTCYx (see *MPLAB C18 C Compiler Libraries*).
D10TCYXCODE	Contains the library function Delay10TCYx (see *MPLAB C18 C Compiler Libraries*).
D1KTCYXCODE	Contains the library function Delay1KTCYx (see *MPLAB C18 C Compiler Libraries*).
DELAYDAT1	Contains uninitialized data used by some of the Delay functions (see *MPLAB C18 C Compiler Libraries*).
DELAYDAT2	Contains uninitialized data used by some of the Delay functions (see *MPLAB C18 C Compiler Libraries*).
PROG	Contains the executable content of the math library (see *MPLAB C18 C Compiler Libraries*).
SEED_DATA	Contains the initialized data used by rand and srand functions (see *MPLAB C18 C Compiler Libraries*).
SFR_BANKED*	Contains the SFRs located in banked RAM.
SFR_UNBANKED*	Contains the SFRs located in access RAM.
STDIO	Contains the executable content of the peripheral output routines for the standard library output functions.
STDLIB	Contains the executable content of the data conversion functions (see *MPLAB C18 C Compiler Libraries*).
STRING	Contains the memory and string manipulation functions (see *MPLAB C18 C Compiler Libraries*).
UARTCODE	Contains the executable content for the software UART functions (see *MPLAB C18 C Compiler Libraries*).
UARTDATA	Contains uninitialized data used by the software UART functions (see *MPLAB C18 C Compiler Libraries*).

Note: The * denotes a wildcard.

2.9.1.5 SECTION ATTRIBUTES

The `#pragma` *`sectiontype`* directive may optionally include two section attributes – `access` or `overlay`.

2.9.1.5.1 `access`

The `access` attribute tells the compiler to locate the specified section in an access region of data memory (see the device data sheets or the *PICmicro® 18C MCU Family Reference Manual* (DS39500) for more on access data memory).

Data sections with the `access` attribute will be placed into memory regions that are defined as `ACCESSBANK` in the linker script file. These regions are those accessed via the access bit of an instruction, i.e., no banking is required (see the device data sheet). Variables located in an access section must be declared with the `near` keyword. For example:

```
#pragma udata access my_access
/* all accesses to these will be unbanked */
near unsigned char av1, av2;
```

2.9.1.5.2 `overlay`

The `overlay` attribute permits other sections to be located at the same physical address. This can conserve memory by locating variables to the same location (as long as both are not active at the same time.) The `overlay` attribute can be used in conjunction with the `access` attribute.

In order to overlay two sections, four requirements must be met:

1. Each section must reside in a different source file.
2. Both sections must have the same name.
3. If the `access` attribute is specified with one section, it must be specified with the other.
4. If an absolute address is specified with one section, the same absolute address must be specified with the other.

Code sections that have the `overlay` attribute can be located at an address that overlaps other `overlay` code sections. For example:

file1.c:

```
#pragma code overlay my_overlay_scn=0x1000
void f (void)
{
 ...
}
```

file2.c:

```
#pragma code overlay my_overlay_scn=0x1000
void g (void)
{
 ...
}
```

Data sections that have the `overlay` attribute can be located at an address that overlaps other `overlay` data sections. This feature can be useful for allowing a single data range to be used for multiple variables that are never active simultaneously. For example:

file1.c:

```
#pragma udata overlay my_overlay_data=0x1fc
/* 2 bytes will be located at 0x1fc and 0x1fe */
int int_var1, int_var2;
```

file2.c:

```
#pragma udata overlay my_overlay_data=0x1fc
/* 4 bytes will be located at 0x1fc */
long long_var;
```

For more information on the handling of overlay sections see *MPASM™ User's Guide with MPLINK™ and MPLIB™* (DS33014).

2.9.1.6 LOCATING CODE

Following a `#pragma code` directive, all generated code will be assigned to the specified code section until another `#pragma code` directive is encountered. An absolute code section allows the location of code to a specific address. For example:

```
#pragma code my_code=0x2000
```

will locate the code section `my_code` at program memory address `0x2000`.

The linker will enforce that code sections be placed in program memory regions however, a code section can be located in a specified memory region. The `SECTION` directive of the linker script is used to assign a section to a specific memory region. The following linker script directive assigns code section `my_code1` to memory region `page1`:

```
SECTION NAME=my_code1 ROM=page1
```

2.9.1.7 LOCATING DATA

Data can be placed in either data or program memory with the MPLAB C18 compiler. Data that is placed in on-chip program memory can be read but not written without additional user-supplied code. Data placed in external program memory can generally be either read or written without additional user-supplied code.

For example, the following declares a section for statically allocated uninitialized data (`udata`) at absolute address `0x120`:

```
#pragma udata my_new_data_section=0x120
```

The `rom` keyword tells the compiler that a variable should be placed in program memory. The compiler will allocate this variable into the current `romdata` type section. For example:

```
#pragma romdata const_table
const rom char my_const_array[10] = {0, 1, 2, 3, 4, 5,
                                     6, 7, 8, 9};

/* Resume allocation of romdata into the default section */
#pragma romdata
```

The linker will enforce that `romdata` sections be placed in program memory regions and that `udata` and `idata` sections be placed in data memory regions however, a data section can also be located in a specified memory region. The `SECTION` directive of the linker script is used to assign a section to a specific memory region. The following assigns `udata` section `my_data` to memory region `gpr1`:

```
SECTION NAME=my_data RAM=gpr1
```

2.9.2 #pragma interruptlow *fname* / #pragma interrupt *fname*

The `interrupt` pragma declares a function to be a high-priority interrupt service routine (ISR); the `interruptlow` pragma declares a function to be a low-priority interrupt service routine.

An interrupt suspends the execution of a running application, saves the current context information and transfers control to an ISR so that the event may be processed. Upon completion of the ISR, previous context information is restored and normal execution of the application resumes. The minimal context saved and restored for an interrupt is `WREG`, `BSR` and `STATUS`. A high-priority interrupt uses the shadow registers to save and restore the minimal context, while a low-priority interrupt uses the software stack to save and restore the minimal context. As a consequence, a high-priority interrupt terminates with a fast “return from interrupt”, while a low-priority interrupt terminates with a normal “return from interrupt”. Two `MOVFF` instructions are required for each byte of context preserved via the software stack except for `WREG`, which requires a `MOVWF` instruction and a `MOVF` instruction; therefore, in order to preserve the minimal context, a low-priority interrupt has an additional 10-word overhead beyond the requirements of a high-priority interrupt.

Interrupt service routines use a temporary data section that is distinct from that used by normal C functions. Any temporary data required during the evaluation of expressions in the interrupt service routine is allocated in this section and is not overlaid with the temporary locations of other functions, including other interrupt functions. The interrupt pragmas allow the interrupt temporary data section to be named. If this section is not named, the compiler temporary variables are created in a `udata` section named *fname*`_tmp`. For example:

```
void foo(void);
...
#pragma interrupt foo
void foo(void)
{
  /* perform interrupt function here */
}
```

The compiler temporary variables for interrupt service routine `foo` will be placed in the `udata` section `foo_tmp`.

2.9.2.1 SYNTAX

interrupt-directive:

 # pragma interrupt *function-name* [*tmp-section-name*] [**save=***save-list*]
| **# pragma interruptlow** *function-name* [*tmp-section-name*] [**save=***save-list*]

save-list:

save-specifier
| *save-list, save-specifier*

save-specifier:

symbol-name
| **section("***section-name***")**

function-name: C identifier – names the C function serving as an ISR.

tmp-section-name: C identifier – names the section in which to allocate the ISR's temporary data

symbol-name: C identifier – names the variable that will be restored following interrupt processing

section-name: C identifier with the exception that the first character can be a dot (.) – names the section that will be restored following interrupt processing

2.9.2.2 INTERRUPT SERVICE ROUTINES

An MPLAB C18 ISR is like any other C function in that it can have local variables and access global variables; however, an ISR must be declared with no parameters and no return value since the ISR, in response to a hardware interrupt, is invoked asynchronously. Global variables that are accessed by both an ISR and mainline functions should be declared `volatile`.

ISR's should only be invoked through a hardware interrupt and not from other C functions. An ISR uses the return from interrupt (`RETFIE`) instruction to exit from the function rather than the normal `RETURN` instruction. Using a fast `RETFIE` instruction out of context can corrupt `WREG`, `BSR` and `STATUS`.

2.9.2.3 INTERRUPT VECTORS

MPLAB C18 does not automatically place an ISR at the interrupt vector. Commonly, a GOTO instruction is placed at the interrupt vector for transferring control to the ISR proper. For example:

```
#include <p18cxxx.h>

void low_isr(void);
void high_isr(void);

/*
 * For PIC18 devices the low interrupt vector is found at
 * 00000018h. The following code will branch to the
 * low_interrupt_service_routine function to handle
 * interrupts that occur at the low vector.
 */
#pragma code low_vector=0x18
void interrupt_at_low_vector(void)
{
  _asm GOTO low_isr _endasm
}
#pragma code /* return to the default code section */

#pragma interruptlow low_isr
void low_isr (void)
{
  /* ... */
}

/*
 * For PIC18 devices the high interrupt vector is found at
 * 00000008h. The following code will branch to the
 * high_interrupt_service_routine function to handle
 * interrupts that occur at the high vector.
 */
#pragma code high_vector=0x08
void interrupt_at_high_vector(void)
{
  _asm GOTO high_isr _endasm
}
#pragma code /* return to the default code section */

#pragma interrupt high_isr
void high_isr (void)
{
  /* ... */
}
```

For a complete example, see **Chapter 5. “Sample Application”**

2.9.2.4 ISR CONTEXT SAVING

MPLAB C18 will preserve a basic context by default (**Section 3.4 "Compiler Managed Resources"**), and the `save=` clause allows additional arbitrary symbols to be saved and restored by the function.

To save a user-defined global variable named `myint`, the following pragma directive would be used:

```
#pragma interrupt high_interrupt_service_routine save=myint
```

In addition to variables, entire data sections can also be named in the `save=` clause. For example, to save a user-defined section named `mydata`, the following pragma directive would be used:

```
#pragma interrupt high_interrupt_service_routine save=section("mydata")
```

If an interrupt service routine calls another function, the normal functions' temporary data section (which is named `.tmpdata`) should be saved using a `save=section(".tmpdata")` qualifier on the interrupt pragma directive. For example:

```
#pragma interrupt high_interrupt_service_routine save=section(".tmpdata")
```

If the ISR changes any file registers other than the basic context, then they should be named in the `save=` clause. The generated code should be examined to determine which file registers are used and need to be saved.

> **Note:** If an ISR calls a function that returns a value less than or equal to 32 bits in size, the locations associated with the return value (see **Section 3.2.3 "Return Values"**) should be specified in the `save=` list of the interrupt pragma.

If an interrupt service routine calls a function that returns 16-bit data, the `PROD` file register should be saved using a `save=PROD` qualifier on the interrupt pragma directive. For example:

```
#pragma interruptlow low_interrupt_service_routine save=PROD
```

If an interrupt service routine uses math library functions or calls a function that returns 24- or 32-bit data, the math data section (which is named `MATH_DATA`) should be saved using a `save=section("MATH_DATA")` qualifier on the interrupt pragma directive. For example:

```
#pragma interrupt high_interrupt_service_routine save=section("MATH_DATA")
```

All previous examples show a single value being saved. Multiple variables and sections may be saved using the same `save=` qualifier. If an interrupt service routine used the `PROD` file register, the `.tmpdata` section, the `myint` variable, and the `mydata` section, these should be saved using the `save=PROD`, `section (".tmpdata")`, `myint`, `section ("mydata")` qualifier on the interrupt pragma directive. For example:

```
#pragma interrupt isr save=PROD, section(".tmpdata"), myint, section("mydata")
```

2.9.2.5 LATENCY

The time between when an interrupt occurs and when the first ISR instruction is executed is the latency of the interrupt. The three elements that affect latency are:

1. **Processor servicing of interrupt**: The amount of time it takes the processor to recognize the interrupt and branch to the first address of the interrupt vector. To determine this value refer to the processor data sheet for the specific processor and interrupt source being used.
2. **Interrupt vector execution**: The amount of time it takes to execute the code at the interrupt vector that branches to the ISR.
3. **ISR prologue code**: The amount of time it takes MPLAB C18 to save the compiler managed resources and the data in the `save=` list.

2.9.2.6 NESTING INTERRUPTS

Low-priority interrupts may be nested since active registers are saved onto the software stack. Only a single instance of a high-priority interrupt service routine may be active at a time since these ISR's use the single-level hardware shadow registers.

If nesting of low-priority interrupts is desired, a statement to set the `GIEL` bit can be added near the beginning of the ISR. See the processor data sheet for details.

2.9.3 `#pragma varlocate` *bank variable-name*
`#pragma varlocate "`*section-name*`"` *variable-name*

The `varlocate` pragma tells the compiler where a variable will be located at link time, enabling the compiler to perform more efficient bank switching**.**

The `varlocate` specifications are not enforced by the compiler or linker. The sections that contain the variables should be assigned to the correct bank explicitly in the linker script or via absolute sections in the module(s) where they are defined.

2.9.3.1 SYNTAX

variable-locate-directive:

```
    # pragma varlocate bank variable-name[, variable-name...]
  | # pragma varlocate "section-name" variable-name[, variable-name...]
```

bank: integer constant

variable-name: C identifier

section-name: C identifier

2.9.3.2 EXAMPLE USING # pragma varlocate *bank variable-name*

In one file, c1 and c2 are explicitly assigned to bank 1.

```
#pragma udata bank1=0x100
signed char c1;
signed char c2;
```

In a second file, the compiler is told that both c1 and c2 are located in bank 1.

```
#pragma varlocate 1 c1
extern signed char c1;

#pragma varlocate 1 c2
extern signed char c2;

void main (void)
{
  c1 += 5;
  /* No MOVLB instruction needs to be generated here. */
  c2 += 5;
}
```

When c1 and c2 are used in the second file, the compiler knows that both variables are in the same bank and does not need to generate a second MOVLB instruction when using c2 immediately after c1.

2.9.3.3 EXAMPLE USING # pragma varlocate "*section-name*" *variable-name*

In one file, c3 and c4 are created in the udata section my_section.

```
#pragma udata my_section
signed char c3;
signed char c4;
#pragma udata
```

In a second file, the compiler is told that both c3 and c4 are located in the udata section my_section.

```
#pragma varlocate "my_section" c3, c4
extern signed char c3;
extern signed char c4;

void main (void)
{
  c3 += 5;
  /* No MOVLB instruction needs to be generated here. */
  c4 += 5;
}
```

When c3 and c4 are used in the second file, the compiler knows that both variables are in the same section and does not need to generate a second MOVLB instruction when using c4 immediately after c3.

2.9.4 #pragma config

The `#pragma config` directive specifies the processor-specific configuration settings (i.e., configuration bits) to be used by the application.

Configuration settings may be specified with multiple `#pragma config` directives. MPLAB C18 verifies that the configuration settings specified are valid for the processor for which it is compiling. If a given setting in the configuration byte has not been specified in any `#pragma config` directive, the bits associated with that setting will default to the unprogrammed value.

For each configuration byte for which a setting is specified with the `#pragma config` directive, the compiler generates an absolute `romdata` section named `.config_`*`address`*`_`*`filename`*, where *`address`* is the hexadecimal representation of the address of the configuration byte, and *`filename`* is the name of the object file being generated. For example, if a configuration setting was specified for the configuration byte located at address `0x300001` and the source file was compiled with the command-line option "`mcc18 foo.c -fo=foo.o`", a `romdata` section named `.config_300001_foo.o` would be created.

2.9.4.1 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 `--help-config` command-line option. Additionally, the available settings and associated values for each device are listed in the *PIC18 Configuration Settings Addendum* (DS51537).

2.9.4.2 EXAMPLE

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

- Enables the Watchdog Timer,
- Sets the Watchdog Postscaler to 1:128, and
- Selects the HS oscillator

```
#pragma config WDT = ON, WDTPS = 128
#pragma config OSC = HS
...
void main (void)
{
...
}
```

2.10 PROCESSOR-SPECIFIC HEADER FILES

The processor-specific header file is a C file that contains external declarations for the special function registers, which are defined in the register definitions file (see **Section 2.11 "Processor-specific Register Definitions Files"**). For example, in the PIC18C452 processor-specific header file, PORTA is declared as:

```
extern volatile near unsigned char PORTA;
```

and as:

```
extern volatile near union  {
  struct {
    unsigned RA0:1;
    unsigned RA1:1;
    unsigned RA2:1;
    unsigned RA3:1;
    unsigned RA4:1;
    unsigned RA5:1;
    unsigned RA6:1;
  } ;
  struct {
    unsigned AN0:1;
    unsigned AN1:1;
    unsigned AN2:1;
    unsigned AN3:1;
    unsigned T0CKI:1;
    unsigned SS:1;
    unsigned OSC2:1;
  } ;
  struct {
    unsigned :2;
    unsigned VREFM:1;
    unsigned VREFP:1;
    unsigned :1;
    unsigned AN4:1;
    unsigned CLKOUT:1;
  } ;
  struct {
    unsigned :5;
    unsigned LVDIN:1;
  } ;
} PORTAbits ;
```

The first declaration specifies that `PORTA` is a byte (`unsigned char`). The `extern` modifier is needed since the variables are declared in the register definitions file. The `volatile` modifier tells the compiler that it cannot assume that `PORTA` retains values assigned to it. The `near` modifier specifies that the port is located in access RAM.

The second declaration specifies that `PORTAbits` is a union of bit-addressable anonymous structures (see **Section 2.8.1 "Anonymous Structures"**). Since individual bits in a special function register may have more than one function (and hence more than one name), there are multiple structure definitions inside the union all referring to the same register. Respective bits in all structure definitions refer to the same bit in the register. Where a bit has only one function for its position, it is simply padded in other structure definitions. For example, bits 1 and 2 on `PORTA` are simply padded in the third and fourth structures because they only have two names, whereas, bit 6 has four names and is specified in each of the structures.

Any of the following statements can be written to use the PORTA special function register:

```
PORTA = 0x34;       /* Assigns the value 0x34 to the port */
PORTAbits.AN0 = 1; /* Sets the AN0 pin high */
PORTAbits.RA0 = 1; /* Sets the RA0 pin high, same as above
                      statement */
```

In addition to register declarations, the processor-specific header file defines inline assembly macros. These macros represent certain PICmicro MCU instructions that an application may need to execute from C code. Although, these instructions could be included as inline assembly instructions, as a convenience they are provided as C macros (see Table 2-8).

In order to use the processor-specific header file, the header file that pertains to the device being used should be included (e.g., if using a PIC18C452, `#include <p18c452.h>`) in the application source code. The processor-specific header files are located in the `c:\mcc18\h` directory, where `c:\mcc18` is the directory where the compiler is installed. Alternatively, `#include <p18cxxx.h>` will include the proper processor-specific header file based on the processor selected on the command line via the `-p` command-line option.

TABLE 2-8: C MACROS PROVIDED FOR PICmicro MCU INSTRUCTIONS

Instruction Macro(1)	Action
`Nop()`	Executes a no operation (`NOP`)
`ClrWdt()`	Clears the Watchdog Timer (`CLRWDT`)
`Sleep()`	Executes a `SLEEP` instruction
`Reset()`	Executes a device reset (`RESET`)
`Rlcf(`*var*`, `*dest*`, `*access*`)` (2,3)	Rotates *var* to the left through the carry bit
`Rlncf(`*var*`, `*dest*`, `*access*`)` (2,3)	Rotates *var* to the left without going through the carry bit
`Rrcf(`*var*`, `*dest*`, `*access*`)` (2,3)	Rotates *var* to the right through the carry bit
`Rrncf(`*var*`, `*dest*`, `*access*`)` (2,3)	Rotates *var* to the right without going through the carry bit
`Swapf(`*var*`, `*dest*`, `*access*`)` (2,3)	Swaps the upper and lower nibble of *var*

Note 1: Using any of these macros in a function affects the ability of the MPLAB C18 compiler to perform optimizations on that function.

2: *var* must be an 8-bit quantity (i.e., `char`) and not located on the stack.

3: If *dest* is `0`, the result is stored in `WREG`, and if *dest* is `1`, the result is stored in *var*. If *access* is `0`, the access bank will be selected, overriding the `BSR` value. If *access* is `1`, then the bank will be selected as per the `BSR` value.

2.11 PROCESSOR-SPECIFIC REGISTER DEFINITIONS FILES

The processor-specific register definitions file is an assembly file that contains definitions for all the special function registers on a given device. The processor-specific register definitions file, when compiled, will become an object file that will need to be linked with the application (e.g., `p18c452.asm` compiles to `p18c452.o`). This object file is contained in `p18xxxx.lib` (e.g., `p18c452.o` is contained in `p18c452.lib`).

The source code for the processor-specific register definitions files is found in both the `c:\mcc18\src\traditional\proc` and `c:\mcc18\src\extended\proc` directories. Compiled object code is found in the `c:\mcc18\lib` directory, where `c:\mcc18` is the directory where the compiler is installed.

For example, `PORTA` is defined in the PIC18C452 processor-specific register definitions file as:

```
SFR_UNBANKED0 UDATA_ACS H'f80'
PORTA
PORTAbits RES 1 ; 0xf80
```

The first line specifies the file register bank where `PORTA` is located and the starting address for that bank. `PORTA` has two labels, `PORTAbits` and `PORTA`, both referring to the same location (in this case `0xf80`).

Chapter 3. Run-time Model

This section discusses the run-time model or the set of assumptions that the MPLAB C18 compiler operates, including information about how the MPLAB C18 compiler uses the resources of the PIC18 PICmicro microcontrollers.

3.1 MEMORY MODELS

MPLAB C18 provides full library support for both a small and a large memory model (see Table 3-1). The small memory model is selected using the `-ms` command-line option and the large memory model using the `-ml` option. If neither is provided, the small memory model is used by default.

TABLE 3-1: MEMORY MODEL SUMMARY

Memory Model	Command-line Switch	Default ROM Range Qualifier	Size of Pointers to Program Space
small	`-ms`	`near`	16 bits
large	`-ml`	`far`	24 bits

The difference between the small and large models is the size of pointers that point to program memory. In the small memory model, both function and data pointers that point to program memory use 16 bits. This has the effect of restricting pointers to addressing only the first 64k of program memory in the small model. In the large memory model, 24 bits are used. Applications using more than 64k of program memory must use the large memory model.

The memory model setting can be overridden on a case-by-case basis by using the `near` or `far` qualifier when declaring a pointer into program space. Pointers to `near` memory use 16 bits as in the small memory model, and pointers to `far` memory use 24 bits as in the large memory model.

The following example creates a pointer to program memory that can address up to and beyond 64k of program memory space, even when the small memory model is being used[1]:

```
far rom *pgm_ptr;
```

The following example creates a function pointer that can address up to and beyond 64k of program memory space, even when the small memory model is being used[2]:

```
far rom void (*fp) (void);
```

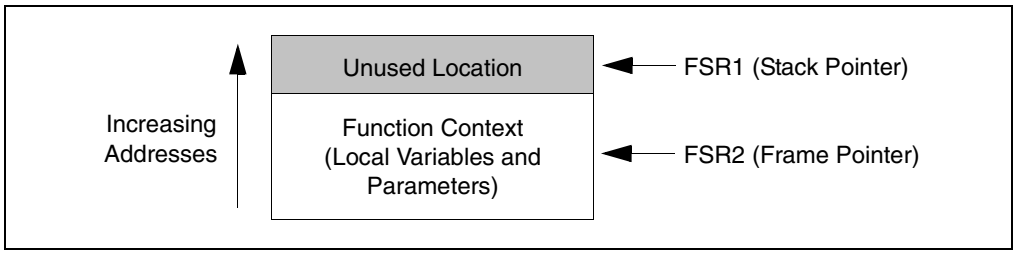
If the same memory model is not used for all files in a project, all global pointers to program memory should be declared with explicit `near` or `far` qualifiers so that they are accessed correctly in all modules. The pre-compiled libraries distributed with MPLAB C18 can be used with either the small or large memory models.

1. Following the use of a `far` data pointer in a small memory model program, the `TBLPTRU` byte must be cleared by the user. MPLAB C18 does not clear this byte.
2. Following the use of a `far` function pointer in a small memory model program, the `PCLATU` byte must be cleared by the user. MPLAB C18 does not clear this byte.

3.2 CALLING CONVENTIONS

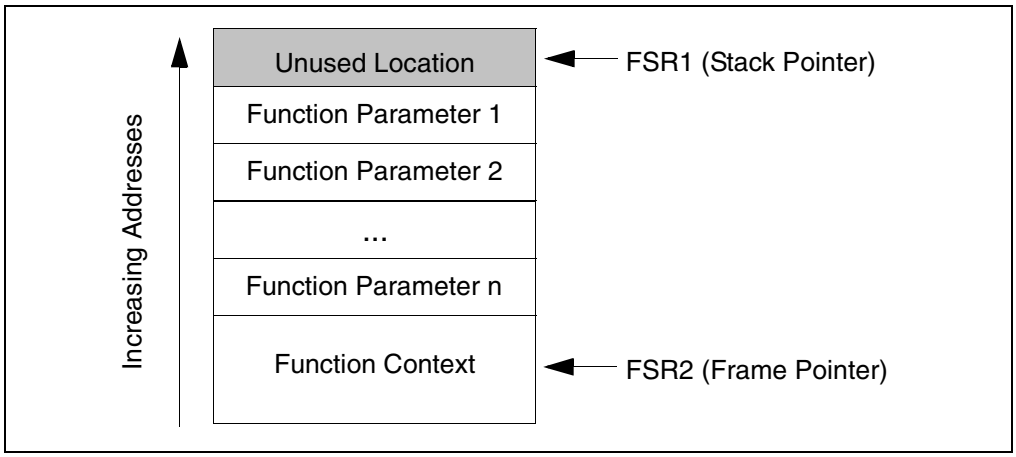
The MPLAB C18 software stack is an upward growing stack data structure on which the compiler places function arguments and local variables that have the storage class `auto`. The software stack is distinct from the hardware stack upon which the PICmicro microcontroller places function call return addresses. Figure 3-1 shows an example of the software stack.

FIGURE 3-1: EXAMPLE OF SOFTWARE STACK



The stack pointer (`FSR1`) always points to the next available stack location. MPLAB C18 uses `FSR2` as the frame pointer, providing quick access to local variables and parameters. When a function is invoked, its stack-based arguments are pushed onto the stack in right-to-left order and the function is called. The leftmost function argument is on the top of the software stack upon entry into the function. Figure 3-2 shows the software stack immediately prior to a function call.

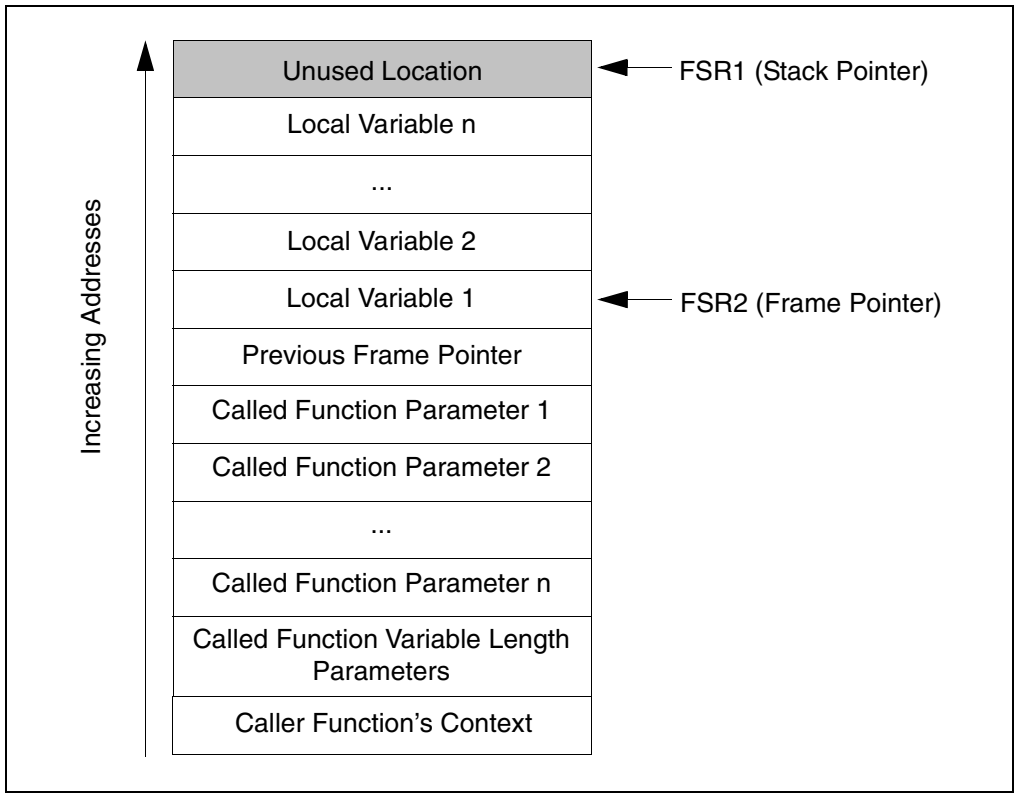
FIGURE 3-2: EXAMPLE OF SOFTWARE STACK IMMEDIATELY PRIOR TO FUNCTION CALL



3.2.1 Non-extended Mode Convention

For the Non-extended mode, the frame pointer references the location on the stack that separates the stack-based arguments from the stack-based local variables. Stack-based arguments are located at negative offsets from the frame pointer, and stack based local variables are located at positive offsets from the frame pointer. Immediately upon entry into a C function, the called function pushes the value of `FSR2` onto the stack and copies the value of `FSR1` into `FSR2`, thereby saving the context of the calling function and initializing the frame pointer of the current function. Then the total size of stack-based local variables for the function is added to the stack pointer, allocating stack space for those variables. References to stack-based local variables and stack-based arguments are resolved according to offsets from the frame pointer. Figure 3-3 shows a software stack following a call to a C function in Non-extended mode.

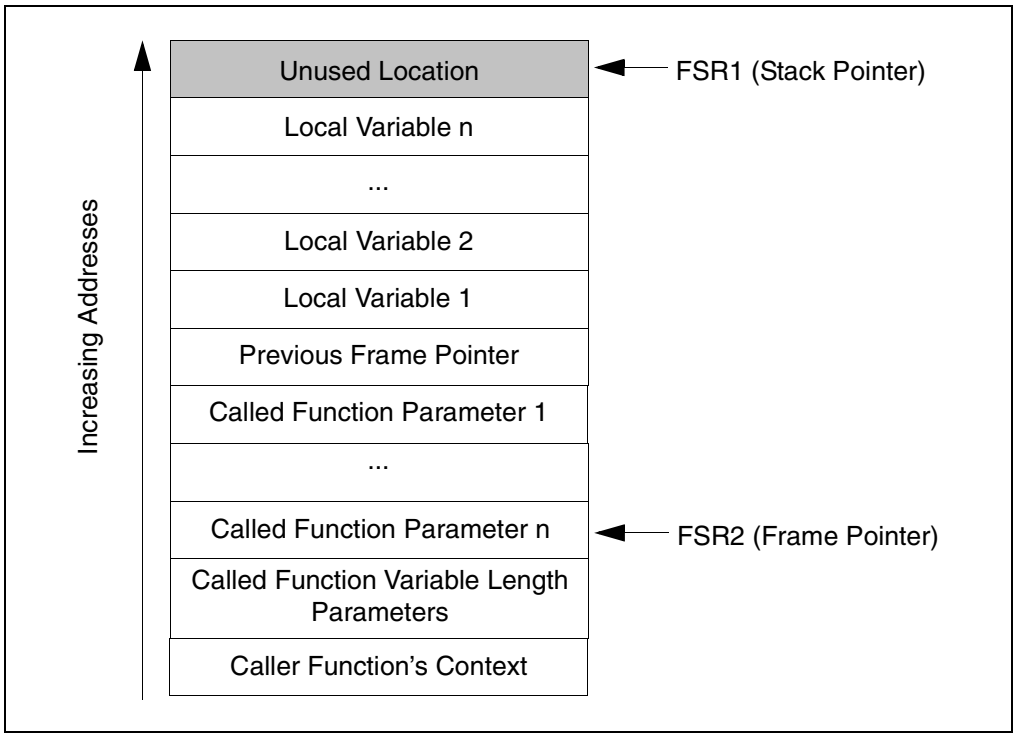
FIGURE 3-3: EXAMPLE OF SOFTWARE STACK FOLLOWING A C FUNCTION CALL IN NON-EXTENDED MODE



3.2.2 Extended Mode Convention

For the Extended mode, the frame pointer references the low byte of the rightmost named parameter of the function. Both local variables and parameters are located at a non-negative offset from the frame pointer, allowing the compiler to access them via indexed with literal offset addressing. Upon entry to the called function, the value of `FSR2` is saved to the stack, the value of `FSR1` is copied to `FSR2`, and the size of the named parameters plus the size of the saved frame pointer is subtracted from `FSR2`. This saves the calling function's frame pointer and initializes the current function's frame pointer. Then, the total size of the local variables for the function is added to `FSR1`, allocating space for those locals. Figure 3-4 shows a software stack following a call to a C function in Extended mode.

FIGURE 3-4: EXAMPLE OF SOFTWARE STACK IMMEDIATELY FOLLOWING A C FUNCTION CALL IN EXTENDED MODE



3.2.3 Return Values

The location of the return value is dependent on the size of the return value. Table 3-2 details the location of the return value based on its size.

TABLE 3-2: RETURN VALUES

Return Value Size	Return Value Location
8 bits	WREG
16 bits	PRODH:PRODL
24 bits	[Non-extended mode] (AARGB2+2):(AARGB2+1):AARGB2[1] [Extended mode] __RETVAL2:__RETVAL1:__RETVAL0[1]
32 bits	[Non-extended mode] (AARGB3+3):(AARGB3+2):(AARGB3+1):AARGB3[1] [Extended mode] __RETVAL3:__RETVAL2:__RETVAL1:__RETVAL0[1]
> 32 bits	On the stack, and FSR0 points to the return value

Note 1: Locations reserved for use by the compiler.

3.2.4 Managing the Software Stack

The stack is sized and placed via the linker script with the STACK directive. The STACK directive has two arguments: SIZE and RAM to control the allocated stack size and its location, respectively. For example, to allocate a 128 byte stack and place that stack in the memory region gpr3:

```
STACK SIZE=0x80 RAM=gpr3
```

MPLAB C18 supports stack sizes greater than 256 bytes. The default linker scripts allocate one memory region per bank of memory, so to allocate a stack larger than 256 bytes requires combining two or more memory regions, as the stack section cannot cross memory region boundaries. For example, the default linker script for the PIC18C452 contains the definitions:

```
DATABANK NAME=gpr4 START=0x400 END=0x4ff
DATABANK NAME=gpr5 START=0x500 END=0x5ff
...
STACK SIZE=0x100 RAM=gpr5
```

To allocate a 512 byte stack in banks 4 and 5, these definitions should be replaced with:

```
DATABANK NAME=stackregion START=0x400 END=0x5ff PROTECTED
STACK SIZE=0x200 RAM=stackregion
```

If a stack larger than 256 bytes is used, the -ls option must be given to the compiler. There is a slight performance penalty that is incurred when using a large stack, as both bytes of the frame pointer (FSR2L and FSR2H) must be incremented/decremented when doing a push/pop, rather than just the low-byte.

The size of the software stack required by an application varies with the complexity of the program. When nesting function calls, all auto parameters and variables of the calling function will remain on the stack. Therefore, the stack must be large enough to accommodate the requirements by all functions in a call tree.

MPLAB C18 supports parameters and local variables allocated either on the software stack or directly from global memory. The static keyword places a local variable or a function parameter in global memory instead of on the software stack.[1] In general, stack-based local variables and function parameters require more code to access than static local variables and function parameters (see **Section 2.3.2 "static Function Arguments"**). Functions that use stack-based variables are more flexible in that they can be reentrant and/or recursive.

3.2.5 Mixing C and Assembly

3.2.5.1 CALLING C FUNCTIONS FROM ASSEMBLY

When calling C functions from assembly:

- C functions are inherently global, unless defined as static.
- The C function name must be declared as an extern symbol in the assembly file.
- A CALL or an RCALL must be used to make the function call.

1. static parameters are valid only when the compiler is operating in Non-extended mode (see **Section 1.2.5 "Selecting the Mode"**).

3.2.5.1.1 `auto` Parameters

`auto` parameters are pushed onto the software stack from right to left. For multi-byte data, the low byte is pushed onto the software stack first.

EXAMPLE 3-1:

Given the following prototype for a C function:

```
char add (auto char x, auto char y);
```

to call the function `add` with values `x = 0x61` and `y = 0x65`, the value for `y` must be pushed onto the software stack followed by the value of `x`. The return value, since it is 8 bits, will be returned in `WREG` (see Table 3-2), i.e.,

```
            EXTERN add  ; defined in C module
...
MOVLW 0x65
MOVWF POSTINC1 ; y = 0x65 pushed onto stack
MOVLW 0x61
MOVWF POSTINC1 ; x = 0x61 pushed onto stack
CALL  add
MOVWF result   ; result is returned in WREG
...
```

EXAMPLE 3-2:

Given the following prototype for a C function:

```
int sub (auto int x, auto int y);
```

to call the function `sub` with values `x = 0x7861` and `y = 0x1265`, the value for `y` must be pushed onto the software stack followed by the value of `x`. The return value, since it is 16 bits, will be returned in `PRODH:PRODL` (see Table 3-2), i.e.,

```
              EXTERN sub  ; defined in C module
...
MOVLW 0x65
MOVWF POSTINC1
MOVLW 0x12
MOVWF POSTINC1        ; y = 0x1265 pushed onto stack
MOVLW 0x61
MOVWF POSTINC1
MOVLW 0x78
MOVWF POSTINC1        ; x = 0x7861 pushed onto stack
CALL  sub
MOVFF PRODL, result
MOVFF PRODH, result+1 ; result is returned in PRODH:PRODL
...
```

3.2.5.1.2 `static` Parameters

`static` parameters are allocated globally, enabling direct access. `static` parameters are valid only when the compiler is operating in Non-extended mode (see **Section 1.2.5 "Selecting the Mode"**). The naming convention for `static` parameters is *`__function_name:n`*, where *`function_name`* is replaced by the name of the function and *`n`* is the parameter position, with numbering starting from 0. For example, given the following prototype for a C function:

```
char add (static char x, static char y);
```

the value for `y` is accessed by using `__add:1`, and the value of `x` is accessed by using `__add:0`.

> **Note:** Since ':' is not a valid character in the MPASM assembler's labels, accessing `static` parameters in assembly functions is not supported.

3.2.5.2 CALLING ASSEMBLY FUNCTIONS FROM C

When calling assembly functions from C:

- The function label must be declared as `global` in the ASM module.
- The function must be declared as `extern` in the C module.
- If case sensitivity is disabled for the ASM module, the function must be declared as ALL CAPS in the C module.
- The function must maintain the MPLAB C18 compiler's run-time model (e.g., return values must be returned in the locations specified in Table 3-2).
- The function is called from C using standard C function notation.

EXAMPLE 3-3:

Given the following function written in assembly:

```
            UDATA_ACS
delay_temp  RES     1

            CODE
asm_delay
          SETF   delay_temp
not_done
          DECF   delay_temp
          BNZ    not_done
done
          RETURN

               GLOBAL asm_delay  ; export so linker can see it
               END
```

to call the function `asm_delay` from a C source file, an external prototype for the assembly function must be added, and the function called using standard C function notation:

```
/* asm_delay is found in an assembly file */
extern void asm_delay (void);

void main (void)
{
  asm_delay ();
}
```

EXAMPLE 3-4:

Given the following function written in assembly,

```
                INCLUDE "p18c452.inc"

                CODE
asm_timed_delay
not_done
           ; Figure 3-2 is what the stack looks like upon
           ; entry to this function.
           ;
           ; 'time' is passed on the stack and must be >= 0
           MOVLW  0xff
           DECF   PLUSW1, 0x1, 0x0
           BNZ    not_done
done
           RETURN
                ; export so linker can see it
                GLOBAL asm_timed_delay
                END
```

to call the function `asm_timed_delay` from a C source file, an external prototype for the assembly function must be added, and the function called using standard C function notation:

```
/* asm_timed_delay is found in an assembly file */
extern void asm_timed_delay (unsigned char);

void main (void)
{
  asm_timed_delay (0x80);
}
```

3.2.5.3 USING C VARIABLES IN ASSEMBLY

When using C variables in assembly:

- The C variable must have global scope in the C source file.
- The C variable must be declared as an `extern` symbol in the assembly file.

EXAMPLE 3-5:

Given the following written in C:

```
unsigned int c_variable;

void main (void)
{
  ...
}
```

to modify the variable `c_variable` from assembly, an external declaration must be added for the variable in the assembly source file:

```
     EXTERN c_variable  ; defined in C module
MYCODE CODE
asm_function
     MOVLW 0xff
     ; put 0xffff in the C declared variable
     MOVWF c_variable
     MOVWF c_variable+1
done
     RETURN

     ; export so linker can see it
     GLOBAL asm_function
     END
```

3.2.5.4 USING ASSEMBLY VARIABLES IN C

When using assembly variables in C:

- The variable must be declared as `global` in the ASM module.
- The variable must be declared as `extern` in the C module.
- If case sensitivity is disabled for the ASM module, the variable must be declared as ALL CAPS in the C module.

EXAMPLE 3-6:

Given the following written in assembly,

```
MYDATA UDATA
asm_variable RES   2 ; 2 byte variable

     ; export so linker can see it
     GLOBAL asm_variable
     END
```

to change the variable `asm_variable` from a C source file, an external declaration must be added for the variable in the C source file. The variable can be used as if it were a C variable:

```
extern unsigned int asm_variable;

void change_asm_variable (void)
{
  asm_variable = 0x1234;
}
```

3.3 STARTUP CODE

3.3.1 Default Behavior

The MPLAB C18 startup begins at the `reset` vector (address 0). The `reset` vector jumps to a function that initializes `FSR1` and `FSR2` to reference the software stack, optionally calls a function to initialize `idata` sections (data memory initialized data) from program memory, and loops on a call to the application's `main()` function.

Whether the startup code initializes `idata` sections is determined by which startup code module is linked with the application. The `c018i.o` and `c018i_e.o` modules perform the initialization, while the `c018.o` and `c018_e.o` modules do not. The default linker scripts provided by MPLAB C18 link with either the `c018i.o` or `c018i_e.o` module depending on whether Non-extended mode or Extended mode is being utilized, respectively.

The ANSI standard requires that all objects with static storage duration that are not initialized explicitly are set to zero. With both the `c018.o/c018_e.o` and `c018i.o/c018i_e.o` startup code modules, this requirement is not met. A third type of startup module, `c018iz.o` and `c018iz_e.o`, is provided to meet this requirement. If this startup code module is linked with the application, then, in addition to initializing `idata` sections, all objects with static storage duration that are not initialized explicitly are set to zero.

To perform initialization of data memory, the MPLINK linker creates a copy of initialized data memory in program memory that the startup code copies to data memory. The `.cinit` section is populated by the MPLINK linker to describe where the program memory images should be copied. Table 3-3 describes the format of the `.cinit` section.

TABLE 3-3: FORMAT OF `.cinit`

Field	Description	Size
`num_init`	Number of sections	16 bit
`from_addr_0`	Program memory start address of section `0`	32 bit
`to_addr_0`	Data memory start address of section `0`	32 bit
`size_0`	Number of data memory bytes to initialize for section `0`	32 bit
`...`	...	...
`from_addr_`*n*(1)	Program memory start address of section *n*(1)	32 bit
`to_addr_`*n*(1)	Data memory start address of section *n*(1)	32 bit
`size_`*n*(1)	Number of data memory bytes to initialize for section *n*(1)	32 bit

Note 1: *n* = `num_init - 1`

After the startup code sets up the stack and optionally copies initialized data, it calls the `main()` function of the C program. There are no arguments passed to `main()`. MPLAB C18 transfers control to `main()` via a looped call, i.e.:

```
loop:
  // Call the user's main routine
  main();
goto loop;
```

3.3.2 Customization

To execute application-specific code immediately after a device `reset` but before any other code generated by the compiler is executed, edit the desired startup file and add the code to the beginning of the `_entry()` function.

To customize the startup files if using Non-extended mode:

1. Go to the `c:\mcc18\src\traditional\startup` directory, where `c:\mcc18` is the directory where the compiler is installed.
2. Edit either `c018.c`, `c018i.c` or `c018iz.c` to add any customized startup code desired.
3. Compile the updated startup file to generate either `c018.o`, `c018i.o` or `c018iz.o`.
4. Copy the startup module to `c:\mcc18\lib`, where `c:\mcc18` is the directory where the compiler is installed.

To customize the startup files if using Extended mode:

1. Go to the `c:\mcc18\src\extended\startup` directory, where `c:\mcc18` is the directory where the compiler is installed.
2. Edit either `c018_e.c`, `c018i_e.c` or `c018iz_e.c` to add any customized startup code desired.
3. Compile the updated startup file to generate either `c018_e.o`, `c018i_e.o` or `c018iz_e.o`.
4. Copy the startup module to `c:\mcc18\lib`, where `C:\mcc18` is the directory where the compiler is installed.

3.4 COMPILER MANAGED RESOURCES

Certain special function registers and data sections of the PIC18 PICmicro microcontrollers are used by MPLAB C18 and are not available for general purpose user code. Table 3-4 indicates each of these resources, their primary use by the compiler, and whether the compiler automatically saves the resource when entering an ISR.

TABLE 3-4: COMPILER RESERVED RESOURCES

Compiler-managed Resource	Primary Use(s)	Automatically Saved
PC	Execution control	✓
WREG	Intermediate calculations	✓
STATUS	Calculation results	✓
BSR	Bank selection	✓
PROD	Multiplication results, return values, intermediate calculations	
section .tmpdata	Intermediate calculations	
FSR0	Pointers to RAM	✓
FSR1	Stack pointer	✓
FSR2	Frame pointer	✓
TBLPTR	Accessing values in program memory	
TABLAT	Accessing values in program memory	
PCLATH	Function pointer invocation	
PCLATU	Function pointer invocation	
section MATH_DATA	Arguments, return values and temporary locations for math library functions	

Note: Compiler temporary variables are placed in a udata section named .tmpdata. Interrupt service routines each create a separate section for temporary data storage (see **Section 2.9.2 "`#pragma interruptlow` fname / `#pragma interrupt` fname"**).

Chapter 4. Optimizations

The MPLAB C18 compiler is an optimizing compiler. It performs optimizations that are primarily intended to reduce code size. All of the optimizations that can be performed by the MPLAB C18 compiler are enabled by default, but can be completely disabled using the `-O-` command-line option. The MPLAB C18 compiler also allows optimizations to be enabled or disabled on a case-by-case basis. Table 4-1 outlines each of the optimizations that can be performed by the MPLAB C18 compiler, including the command-line option to enable or disable it, whether or not it affects debugging, and the section where it is discussed.

Note: Optimizations will not occur on any function containing inline assembly code.

TABLE 4-1: MPLAB C18 Optimizations

Optimization	To Enable	To Disable	Affects Debugging	Section
Duplicate String Merging	`-Om+`	`-Om-`		4.1
Branches	`-Ob+`	`-Ob-`		4.2
Banking	`-On+`	`-On-`		4.3
WREG Content Tracking	`-Ow+`	`-Ow-`		4.4
Code Straightening	`-Os+`	`-Os-`		4.5
Tail Merging	`-Ot+`	`-Ot-`	✓	4.6
Unreachable Code Removal	`-Ou+`	`-Ou-`	✓	4.7
Copy Propagation	`-Op+`	`-Op-`	✓	4.8
Redundant Store Removal	`-Or+`	`-Or-`	✓	4.9
Dead Code Removal	`-Od+`	`-Od-`	✓	4.10
Procedural Abstraction	`-Opa+`	`-Opa-`	✓	4.11

4.1 DUPLICATE STRING MERGING

`-Om+ / -Om-`

Duplicate string merging, when enabled, will take two or more identical literal strings and combine them into a single string table entry with a single instance of the raw data stored in program memory. For example, given the following, when duplicate string merging is enabled (`-Om+`), only a single instance of the data for the string "`foo`" would be stored in the output object file, and both `a` and `b` would reference this data.

```
const rom char *a = "foo";
const rom char *b = "foo";
```

The `-Om-` command-line option disables duplicate string merging.

Duplicate string merging should not affect the ability to debug source code.

4.2 BRANCHES

-Ob+ / -Ob-

The following branch optimizations are performed by the MPLAB C18 compiler when the -Ob+ command-line option is specified:

1. A branch (conditional or unconditional) to an unconditional branch can be modified to target the latter's target instead.
2. An unconditional branch to a RETURN, ADDULNK or SUBULNK instruction can be replaced by a RETURN, ADDULNK or SUBULNK instruction, respectively.
3. A branch (conditional or unconditional) to the instruction immediately following the branch can be removed.
4. A conditional branch to a conditional branch can be modified to target the latter's target if both branches branch on the same condition.
5. A conditional branch immediately followed by an unconditional branch to the same destination can be removed (i.e., the unconditional branch is sufficient).

The -Ob- command-line option disables branch optimizations.

Some of the branch optimizations save program space, while others may expose unreachable code, which can be removed by Unreachable Code Removal (see **Section 4.7 "Unreachable Code Removal"**). Branch optimization should not affect the ability to debug source code.

4.3 BANKING

-On+ / -On-

Banking optimization removes MOVLB instruction in instances where it can be determined that the Bank Select register already contains the correct value. For example, given the following C source code fragment:

```
unsigned char a, b;
a = 5;
b = 5;
```

If compiled with banking optimization disabled (-On-), MPLAB C18 will load the Bank register prior to each assignment:

```
0x000000 MOVLB a
0x000002 MOVLW 0x5
0x000004 MOVWF a,0x1
0x000006 MOVLB b
0x000008 MOVWF b,0x1
```

When this same code is compiled with banking optimization enabled (-On+), MPLAB C18 may be able to eliminate the second MOVLB instruction by determining that the value of the Bank register will not change:

```
0x000000 MOVLB a
0x000002 MOVLW 0x5
0x000004 MOVWF a,0x1
0x000006 MOVWF b,0x1
```

The banking optimization should not affect the ability to debug source code.

4.4 WREG CONTENT TRACKING

-Ow+ / -Ow-

WREG content tracking removes MOVLW instructions in instances where it can be determined that the Working register already contains the correct value. For example, given the following C source code fragment:

```
unsigned char a, b;
a = 5;
b = 5;
```

If compiled with WREG content tracking disabled (-Ow-), MPLAB C18 will load a value of 5 into the Working register prior to each assignment:

```
0x000000 MOVLW 0x5
0x000002 MOVWF a,0x1
0x000004 MOVLW 0x5
0x000006 MOVWF b,0x1
```

When this same code is compiled with WREG tracking enabled (-Ow+), MPLAB C18 may be able to eliminate the second MOVLW instruction by determining that the value of WREG must already be 5 at this point:

```
0x000000 MOVLW 0x5
0x000002 MOVWF a,0x1
0x000004 MOVWF b,0x1
```

WREG content tracking should not affect the ability to debug source code.

4.5 CODE STRAIGHTENING

-Os+ / -Os-

Code straightening attempts to reorder code sequences so that they appear in the order in which they will be executed. This can move or remove branching instructions so that code may be smaller and more efficient. An example where this may occur in C is:

```
first:
  sub1();
  goto second;
third:
  sub3();
  goto fourth;
second:
  sub2();
  goto third;
fourth:
  sub4();
```

In this example, the function calls will occur in numerical order, namely: sub1, sub2, sub3 and then sub4. With code straightening disabled (-Os-), the original flow of the code is mirrored in the generated assembly code:

```
0x000000 first  CALL sub1,0x0
0x000002
0x000004        BRA second
0x000006 third  CALL sub3,0x0
0x000008
0x00000a        BRA fourth
0x00000c second CALL sub2,0x0
0x00000e
0x000010        BRA third
0x000012 fourth CALL sub4,0x0
0x000014
```

With code straightening enabled (`-Os+`), the code is reordered sequentially, removing the branching instructions:

```
0x000000 first  CALL sub1,0x0
0x000002
0x000004 second CALL sub2,0x0
0x000006
0x000008 third  CALL sub3,0x0
0x00000a
0x00000c fourth CALL sub4,0x0
0x00000e
```

Code straightening should not affect the ability to debug source code.

4.6 TAIL MERGING

`-Ot+ / -Ot-`

Tail merging attempts to combine multiple sequences of identical instructions into a single sequence. For example, given the following C source code fragment:

```
if ( user_value )
  PORTB = 0x55;
else
  PORTB = 0x80
```

When compiled with tail merging disabled (`-Ot-`), a `MOVWF PORTB,0x0` is generated in both cases of the if statement:

```
0x000000 MOVF user_value,0x0,0x0
0x000002 BZ 0xa
0x000004 MOVLW 0x55
0x000006 MOVWF PORTB,0x0
0x000008 BRA 0xe
0x00000a MOVLW 0x80
0x00000c MOVWF PORTB,0x0
0x00000e RETURN 0x0
```

However, when compiled with tail merging enabled (`-Ot+`), only a single `MOVWF PORTB,0x0` is generated and is used by both the if and else portions of the code:

```
0x000000 MOVF user_value,0x0,0x0
0x000002 BZ 0x8
0x000004 MOVLW 0x55
0x000006 BRA 0xa
0x000008 MOVLW 0x80
0x00000a MOVWF PORTB,0x0
0x00000c RETURN 0x0
```

When debugging source code compiled with this optimization enabled, the incorrect source line may be highlighted because two or more source lines may share a single sequence of assembly code, making it difficult for the debugger to identify which source line is being executed.

4.7 UNREACHABLE CODE REMOVAL

-Ou+ / -Ou-

Unreachable code will attempt to remove any code that can be provably demonstrated to not execute during normal program flow. An example where this may occur in C is:

```
if (1)
{
  x = 5;
}
else
{
  x = 6;
}
```

In this code it is obvious that the `else` portion of this code snippet can never be reached. With unreachable code disabled (`-Ou-`), the generated assembly code will include the instructions necessary to move 6 to `x` and the instruction to branch around these instructions:

```
0x000000 MOVLB x
0x000002 MOVLW 0x5
0x000004 BRA 0xa
0x000006 MOVLB x
0x000008 MOVLW 0x6
0x00000a MOVWF x,0x1
```

With unreachable code enabled (`-Ou+`), the generated assembly code will not include the instructions for the else:

```
0x000000 MOVLB x
0x000002 MOVLW 0x5
0x000004 MOVWF x,0x1
```

The unreachable code optimization may affect the ability to set breakpoints on certain lines of C source code.

4.8 COPY PROPAGATION

-Op+ / -Op-

Copy propagation is a transformation that, given an assignment $x \leftarrow y$ for some variables *x* and *y*, replaces later uses of *x* with uses of *y*, as long as intervening instructions have not changed the value of either *x* or *y*. This optimization by itself does not save any instructions, but enables dead code removal (see **Section 4.10 "Dead Code Removal"**). An example where this may occur in C is:

```
char c;
void foo (char a)
{
  char b;
  b = a;
  c = b;
}
```

With copy propagation disabled (`-Op-`), the original code is mirrored in the generated assembly code:

```
0x000000 foo    MOVFF a,b
0x000002
0x000004        MOVFF b,c
0x000006
0x000008        RETURN 0x0
```

With copy propagation enabled (`-Op+`), instead of `b` being moved to `c` for the second instruction, `a` is moved to `c`:

```
0x000000 foo    MOVFF a,b
0x000002
0x000004        MOVFF a,c
0x000006
0x000008        RETURN 0x0
```

Dead code removal would then delete the useless assignment of `a` to `b` (see **Section 4.10 "Dead Code Removal"**).

Copy propagation may affect the ability to debug source code.

4.9 REDUNDANT STORE REMOVAL `-Or+ / -Or-`

When assignment of the form $x \leftarrow y$ appears multiple times in an instruction sequence and the intervening code has not changed the value of *x* or *y*, the second assignment may be removed. This is a special case of common subexpression elimination. An example where this may occur in C is:

```
char c;
void foo (char a)
{
  c = a;
  c = a;
}
```

With redundant store removal disabled (`-Or-`), the original code is mirrored in the generated assembly code:

```
0x000000 foo    MOVFF a,c
0x000002
0x000004        MOVFF a,c
0x000006
0x000008        RETURN 0x0
```

With redundant store removal enabled (`-Or+`), the second assignment of `c` to `a` is not required:

```
0x000000 foo    MOVFF a,c
0x000002
0x000004        RETURN 0x0
```

Redundant store removal may affect the ability to set breakpoints on certain lines of C source code.

4.10 DEAD CODE REMOVAL

-Od+ / -Od-

Values computed in a function which are not used on any path to the function's exit are considered dead. Instructions which compute only dead values are themselves considered dead. Values stored to locations visible outside the scope of the function are considered used (and therefore not dead), since it is not determinable whether the value is used or not. Using the same example as that shown in **Section 4.8 "Copy Propagation"**:

```
char c;
void foo (char a)
{
    char b;
    b = a;
    c = b;
}
```

With copy propagation enabled (-Op+) and dead code removal disabled (-Od-), the generated assembly code is that shown in **Section 4.8 "Copy Propagation"**:

```
0x000000 foo    MOVFF a,b
0x000002
0x000004        MOVFF a,c
0x000006
0x000008        RETURN 0x0
```

With copy propagation enabled (-Op+) and dead code removal enabled (-Od+), instead of b being moved to c for the second instruction, a is moved to c thus making the assignment to b dead and able to be removed:

```
0x000000 foo    MOVFF a,c
0x000002
0x000004        RETURN 0x0
```

The dead code removal optimization may affect the ability to set breakpoints on certain lines of C source code.

4.11 PROCEDURAL ABSTRACTION

-Opa+ / -Opa-

MPLAB C18, like most compilers, frequently generates code sequences that appear multiple times in a single object file. This optimization reduces the size of the generated code by creating a procedure containing the repeated code and replacing the copies with a call to the procedure. Procedural abstraction is performed across all functions in a given code section.[1]

Note:	Procedural abstraction generates a savings in program space at the potential expense of execution time.

For example, given the following C source code fragment:

```
distance -= time * speed;
position += time * speed;
```

1. When the time limit for the demo version expires, procedural abstraction will not be performed.

When compiled with procedural abstraction disabled (`-Opa-`), the code sequence generated for `time * speed` is generated for each instruction listed above. It is shown in **bold** below.

```
0x000000 main       MOVLB time
0x000002            MOVF time,0x0,0x1
0x000004            MULWF speed,0x1
0x000006            MOVF PRODL,0x0,0x0
0x000008            MOVWF PRODL,0x0
0x00000a            CLRF PRODL+1,0x0
0x00000c            MOVF WREG,0x0,0x0
0x00000e            SUBWF distance,0x1,0x1
0x000010            MOVF PRODL+1,0x0,0x0
0x000012            SUBWFB distance+1,0x1,0x1
0x000014            MOVF time,0x0,0x1
0x000016            MULWF speed,0x1
0x000018            MOVF PRODL,0x0,0x0
0x00001a            MOVWF PRODL,0x0
0x00001c            CLRF PRODL+1,0x0
0x00001e            MOVF WREG,0x0,0x0
0x000020            ADDWF position,0x1,0x1
0x000022            MOVF PRODL+1,0x0,0x0
0x000024            ADDWFC position+1,0x1,0x1
0x000026            RETURN 0x0
```

Whereas, when compiled with procedural abstraction enabled (`-Opa+`), these two code sequences are abstracted into a procedure and the repeated code is replaced by a call to the procedure.

```
0x000000 main       MOVLB time
0x000002            CALL _pa_0,0x0
0x000004
0x000006            SUBWF distance,0x1,0x1
0x000008            MOVF PRODL+1,0x0,0x0
0x00000a            SUBWFB distance+1,0x1,0x1
0x00000c            CALL _pa_0,0x0
0x00000e
0x000010            ADDWF position,0x1,0x1
0x000012            MOVF PRODL+1,0x0,0x0
0x000014            ADDWFC position+1,0x1,0x1
0x000016            RETURN 0x0
0x000018 __pa_0     MOVF time,0x0,0x1
0x00001a            MULWF speed,0x1
0x00001c            MOVF PRODL,0x0,0x0
0x00001e            MOVWF PRODL,0x0
0x000020            CLRF PRODL+1,0x0
0x000022            MOVF WREG,0x0,0x0
0x000024            RETURN 0x0
```

Not all matches are able to be abstracted in a single pass of procedural abstraction. Procedural abstraction is performed until no more abstractions occur or a maximum of four passes. The number of passes can be controlled via the `-pa=`*n* command-line option. Procedural abstraction can potentially add an additional 2^n - 1 levels of function calls, where *n* is the total number of passes. If the hardware stack is a limited resource in an application, the `-pa=`*n* command-line option can be used to adjust the number of times procedural abstraction is performed.

When debugging source code compiled with this optimization enabled, the incorrect source line may be highlighted because two or more source lines may share a single sequence of assembly code, making it difficult for the debugger to identify which source line is being executed.

Chapter 5. Sample Application

The following sample application will flash LEDs connected to `PORTB` of a PIC18C452 microcontroller. The command line used to build this application is:

```
mcc18 -p 18c452 -I c:\mcc18\h leds.c
```

where `c:\mcc18` is the directory in which the compiler is installed. This sample application was designed for use with a PICDEM™ 2 demo board. This sample covers the following items:

1. Interrupt handling (`#pragma interruptlow`, interrupt vectors, interrupt service routines and context saving)
2. System header files
3. Processor-specific header files
4. `#pragma` *`sectiontype`*
5. Inline assembly

```
/*  1 */ #include <p18cxxx.h>
/*  2 */ #include <timers.h>
/*  3 */
/*  4 */ #define NUMBER_OF_LEDS 8
/*  5 */
/*  6 */ void timer_isr (void);
/*  7 */
/*  8 */ static unsigned char s_count = 0;
/*  9 */
/* 10 */ #pragma code low_vector=0x18
/* 11 */ void low_interrupt (void)
/* 12 */ {
/* 13 */   _asm GOTO timer_isr _endasm
/* 14 */ }
/* 15 */
/* 16 */ #pragma code
/* 17 */
/* 18 */ #pragma interruptlow timer_isr save=PROD
/* 19 */ void
/* 20 */ timer_isr (void)
/* 21 */ {
/* 22 */   static unsigned char led_display = 0;
/* 23 */
/* 24 */   INTCONbits.TMR0IF = 0;
/* 25 */
/* 26 */   s_count = s_count % (NUMBER_OF_LEDS + 1);
/* 27 */
/* 28 */   led_display = (1 << s_count++) - 1;
/* 29 */
/* 30 */   PORTB = led_display;
/* 31 */ }
/* 32 */
/* 33 */ void
/* 34 */ main (void)
/* 35 */ {
/* 36 */   TRISB = 0;
/* 37 */   PORTB = 0;
/* 38 */
/* 39 */   OpenTimer0 (TIMER_INT_ON & T0_SOURCE_INT & T0_16BIT);
/* 40 */   INTCONbits.GIE = 1;
/* 41 */
/* 42 */   while (1)
/* 43 */     {
/* 44 */     }
/* 45 */ }
```

Line 1: This line includes the generic processor header file. The correct processor is selected via the `-p` command-line option. (See **Section 2.5.1 "System Header Files"** and **Section 2.10 "Processor-specific Header Files"**)

Line 10: For PIC18 devices, the low interrupt vector is found at 000000018h. This line of code changes the default code section to the absolute code section named `low_vector` located at address `0x18`. (See **Section 2.9.1 "`#pragma sectiontype`"** and **Section 2.9.2.3 "Interrupt Vectors"**)

Line 13: This line contains inline assembly that will jump to the ISR. (See **Section 2.8.2 "Inline Assembly"** and **Section 2.9.2.3 "Interrupt Vectors"**)

Line 16: This line returns the compiler to the default code section. (See **Section 2.9.1 "`#pragma sectiontype`"** and Table 2-6)

Line 18: This line specifies the function `timer_isr` as a low-priority interrupt service routine. This is required in order for the compiler to generate a `RETFIE` instruction instead of a `RETURN` instruction for the `timer_isr` function. In addition, it ensures that `PROD` special function register will be saved. (**Section 2.9.2 "`#pragma interruptlow fname` / `#pragma interrupt fname`"** and **Section 2.9.2.4 "ISR Context Saving"**)

Line 19-20: These lines define the `timer_isr` function. Notice that it does not take any parameters, and does not return anything (as required by ISRs). (See **Section 2.9.2.2 "Interrupt Service Routines"**)

Line 24: This line clears the `TMR0` interrupt flag to stop the program from processing the same interrupt multiple times. (See **Section 2.10 "Processor-specific Header Files"**)

Line 30: This line demonstrates how to modify the special function register `PORTB` in C. (See **Section 2.10 "Processor-specific Header Files"**)

Line 36-37: These lines initialize the special function registers `TRISB` and `PORTB`. (See **Section 2.10 "Processor-specific Header Files"**)

Line 39: This line enables the `TMR0` interrupt, setting up the timer as an internal 16-bit clock.

Line 40: This line enables global interrupts. (See **Section 2.10 "Processor-specific Header Files"**)

NOTES:

Appendix A. COFF File Format

The Microchip COFF specification is based upon the UNIX® System V COFF format, as described in *Understanding and Using COFF*, Gintaras R. Gircys © 1988, O'Reilly and Associates, Inc. Special mention is made where the Microchip format differs from that described there.

A.1 `struct filehdr` - FILE HEADER

The `filehdr` structure holds information regarding the file. It is the first entry in a COFF file. It is used to denote where the optional file header, symbol table and section headers begin.

```
typedef struct filehdr
{
   unsigned short f_magic;
   unsigned short f_nscns;
   unsigned long f_timdat;
   unsigned long f_symptr;
   unsigned long f_nsyms;
   unsigned short f_opthdr;
   unsigned short f_flags;
} filehdr_t;
```

A.1.1 `unsigned short f_magic`

The magic number is used to identify the implementation of COFF that the file follows. For Microchip PICmicro MCU COFF files, this number is `0x1234`.

A.1.2 `unsigned short f_nscns`

The number of sections in the COFF file.

A.1.3 `unsigned long f_timdat`

The time and date stamp when the COFF file was created (this value is a count of the number of seconds since midnight January 1, 1970).

A.1.4 `unsigned long f_symptr`

A pointer to the symbol table.

A.1.5 `unsigned long f_nsyms`

The number of entries in the symbol table.

A.1.6 `unsigned short f_opthdr`

The size of the optional header record.

A.1.7 unsigned short f_flags

Information on what is contained in the COFF file. Table A-1 shows the different file header flags, along with a description and respective values.

TABLE A-1: FILE HEADER FLAGS

Flag	Description	Value
F_RELFLG	Relocation information has been stripped from the COFF file.	0x0001
F_EXEC	The file is executable, and has no unresolved external symbols.	0x0002
F_LNNO	Line number information has been stripped from the COFF file.	0x0004
L_SYMS	Local symbols have been stripped from the COFF file.	0x0080
F_EXTENDED18	The COFF file was produced utilizing the Extended mode.	0x4000
F_GENERIC	The COFF file is processor independent.	0x8000

A.2 struct opthdr - OPTIONAL FILE HEADER

The opthdr structure contains implementation dependent file level information. For PICmicro MCU COFF files, it is used to specify the name of the target processor, version of the compiler/assembler and to define relocation types.

Note that the layout of this header is specific to the implementation (i.e., the Microchip optional header is not the same format as the System V optional header).

```
typedef struct opthdr
{
   unsigned short magic;
   unsigned short vstamp;
   unsigned long proc_type;
   unsigned long rom_width_bits;
   unsigned long ram_width_bits;
} opthdr_t;
```

A.2.1 unsigned short magic

The magic number can be used to determine the appropriate layout.

A.2.2 unsigned short vstamp

Version stamp.

A.2.3 unsigned long proc_type

Target processor type. Table A-2 shows the processor type along with the associated value stored in this field.

TABLE A-2: PROCESSOR TYPE

Processor	Value
PIC18C242	0x8242
PIC18C252	0x8252
PIC18C442	0x8442
PIC18C452(1)	0x8452
PIC18C601	0x8601
PIC18C658	0x8658
PIC18C801	0x8801
PIC18C858	0x8858
PIC18F1220	0xA122
PIC18F1320	0xA132
PIC18F2220	0xA222
PIC18F2320	0xA232
PIC18F2331	0x2331
PIC18F2410	0x2410
PIC18F242	0x242F
PIC18F2420	0x2420
PIC18F2431	0x2431
PIC18F2439	0x2439
PIC18F2455	0x2455
PIC18F248	0x8248
PIC18F2480	0x2480
PIC18F2510	0x2510
PIC18F2515	0x2515
PIC18F252	0x252F
PIC18F2520	0x2520
PIC18F2525	0x2525
PIC18F2539	0x2539
PIC18F2550	0x2550
PIC18F258	0x8258
PIC18F2580	0x2580
PIC18F2585	0x2585
PIC18F2610	0x2610
PIC18F2620	0x2620
PIC18F2680	0x2680
PIC18F4220	0xA422
PIC18F4320	0xA432
PIC18F4331	0x4331
PIC18F4410	0x4410
PIC18F442	0x442F
PIC18F4420	0x4420
PIC18F4431	0x4431

Processor	Value
PIC18F4439	0x4439
PIC18F4455	0x4455
PIC18F448	0x8448
PIC18F4480	0x4480
PIC18F4510	0x4510
PIC18F4515	0x4515
PIC18F452	0x452F
PIC18F4520	0x4520
PIC18F4525	0x4525
PIC18F4539	0x4539
PIC18F4550	0x4550
PIC18F458	0x8458
PIC18F4580	0x4580
PIC18F4585	0x4585
PIC18F4610	0x4610
PIC18F4620(2)	0x4620
PIC18F4680	0x4680
PIC18F6310	0x6310
PIC18F6390	0x6390
PIC18F6410	0x6410
PIC18F6490	0x6490
PIC18F64J15	0xB415
PIC18F6520	0xA652
PIC18F6525	0x6525
PIC18F6585	0x6585
PIC18F65J10	0xB510
PIC18F65J15	0xB515
PIC18F6620	0xA662
PIC18F6621	0xA621
PIC18F6627	0x6627
PIC18F6680	0x6680
PIC18F66J10	0xB610
PIC18F66J15	0xB615
PIC18F6720	0xA672
PIC18F6722	0x6722
PIC18F67J10	0xB710
PIC18F8310	0x8310
PIC18F8390	0x8390
PIC18F8410	0x8410
PIC18F8490	0x8490
PIC18F84J15	0xC415

TABLE A-2: PROCESSOR TYPE (CONTINUED)

Processor	Value	Processor	Value
PIC18F8520	0xA852	PIC18F8627	0x8625
PIC18F8525	0x8525	PIC18F8680	0x8680
PIC18F8585	0x8585	PIC18F86J10	0xC610
PIC18F85J10	0xC510	PIC18F86J15	0xC615
PIC18F85J15	0xC515	PIC18F8720	0xA872
PIC18F8620	0xA862	PIC18F8722	0x8721
PIC18F8621	0x8621	PIC18F87J10	0xC710

Note 1: This is the processor utilized when compiling for the generic processor when the compiler is operating in Non-extended mode.

2: This is the processor utilized when compiling for the generic processor when the compiler is operating in Extended mode.

A.2.4 `unsigned long rom_width_bits`

Width of program memory in bits.

A.2.5 `unsigned long ram_width_bits`

Width of data memory in bits.

A.3 `struct scnhdr` - SECTION HEADER

The `scnhdr` structure contains information related to an individual section. The PICmicro MCU COFF files make a slight departure from the normal COFF definition of the section name. Since the PICmicro MCU COFF section names may be longer than eight characters, the PICmicro MCU COFF files allow a string table entry for long names.

```
typedef struct scnhdr
{
   union
   {
     char _s_name[8] /* section name is a string */
     struct
       {
         unsigned long _s_zeroes
         unsigned long _s_offset
       }_s_s;
   }_s;

   unsigned long s_paddr;
   unsigned long s_vaddr;
   unsigned long s_size;
   unsigned long s_scnptr;
   unsigned long s_relptr;
   unsigned long s_lnnoptr;
   unsigned short s_nreloc;
   unsigned short s_nlnno;
   unsigned long s_flags;
} scnhdr_t;
```

A.3.1 union _s

A string or a reference into the string table. Strings of fewer than eight characters are stored directly, and all others are stored in the string table. If the first four characters of the string are 0, then the last four bytes are assumed to be an offset into the string table. This is a bit nasty as it is not strictly conforming to the ANSI specification (i.e., type munging is undefined behavior by the standard), but it is effective and it maintains binary compatibility with the System V layout, which other options would not do. This implementation has the advantage of mirroring the standard System V structure used for long symbol names.

A.3.1.1 `char s_name[8]`

In-place section name. If the section name is fewer than eight characters long, then the section name is stored in place.

A.3.1.2 `struct _s_s`

Section name is stored in the string table. If the first four characters of the section name are zero, then the last four form an offset into the string table to find the name of the section.

A.3.1.2.1 `unsigned long _s_zeroes`

First four characters of the section name are zero.

A.3.1.2.2 `unsigned long _s_offset`

Offset of section name in the string table.

A.3.1.3 `unsigned long s_paddr`

Physical address of the section.

A.3.1.4 `unsigned long s_vaddr`

Virtual address of the section. Always contains the same value as `s_paddr`.

A.3.2 unsigned long s_size

Size of this section.

A.3.3 unsigned long s_scnptr

Pointer to the raw data in the COFF file for this section.

A.3.4 unsigned long s_relptr

Pointer to the relocation information in the COFF file for this section.

A.3.5 unsigned long s_lnnoptr

Pointer to the line number information in the COFF file for this section.

A.3.6 unsigned short s_nreloc

The number of relocation entries for this section.

A.3.7 unsigned short s_nlnno

The number of line number entries for this section.

A.3.8 unsigned long s_flags

Section type and content flags. The flags which define the section type and the section qualifiers are stored as bit fields in the s_flags field. Masks are defined for the bit fields to ease access. Table A-3 shows the different section header flags, along with a description and respective values.

TABLE A-3: SECTION HEADER FLAGS

Flag	Description	Value
STYP_TEXT	Section contains executable code.	0x00020
STYP_DATA	Section contains initialized data.	0x00040
STYP_BSS	Section contains uninitialized data.	0x00080
STYP_DATA_ROM	Section contains initialized data for program memory.	0x00100
STYP_ABS	Section is absolute.	0x01000
STYP_SHARED	Section is shared across banks.	0x02000
STYP_OVERLAY	Section is overlaid with other sections of the same name from different object modules.	0x04000
STYP_ACCESS	Section is available using access bit.	0x08000
STYP_ACTREC	Section contains the overlay activation record for a function.	0x10000

A.4 struct reloc - RELOCATION ENTRY

Any instruction that accesses a relocatable identifier (variable, function, etc.) must have a relocation entry. This differs from the System V relocation data, where the offset is stored in the location being relocated to, in that the offset to add to the base address of the symbol is stored in the relocation entry. This is necessary because Microchip relocations are not restricted to just filling in an address+offset value into the data stream, but also do simple code modifications. It is much more straightforward to store the offset here, at the cost of a slightly increased file size.

```
typedef struct reloc
{
   unsigned long r_vaddr;
   unsigned long r_symndx;
   short r_offset;
   unsigned short r_type;
} reloc_t;
```

A.4.1 unsigned long r_vaddr

Address of reference (byte offset relative to start of raw data).

A.4.2 unsigned long r_symndx

Index into symbol table.

A.4.3 short r_offset

Signed offset to be added to the address of symbol r_symndx.

A.4.4 unsigned short r_type

Relocation type, implementation defined values. Table A-4 lists the relocation types, along with a description and respective values.

TABLE A-4: RELOCATION TYPES

Type	Description	Value
RELOCT_CALL	CALL instruction (first word only on PIC18)	1
RELOCT_GOTO	GOTO instruction (first word only on PIC18)	2
RELOCT_HIGH	Second 8 bits of an address	3
RELOCT_LOW	Low order 8 bits of an address	4
RELOCT_P	5 bits of address for the P operand of a PIC17 MOVFP or MOVPF instruction	5
RELOCT_BANKSEL	Generate the appropriate instruction to bank switch for a symbol	6
RELOCT_PAGESEL	Generate the appropriate instruction to page switch for a symbol	7
RELOCT_ALL	16 bits of an address	8
RELOCT_IBANKSEL	Generate indirect bank selecting instructions	9
RELOCT_F	8 bits of address for the F operand of a PIC17 MOVFP or MOVPF instruction	10
RELOCT_TRIS	File register address for TRIS instruction	11
RELOCT_MOVLR	MOVLR bank PIC17 banking instruction	12
RELOCT_MOVLB	MOVLB PIC17 and PIC18 banking instruction	13
RELOCT_GOTO2	Second word of an PIC18 GOTO instruction	14
RELOCT_CALL2	Second word of an PIC18 CALL instruction	14
RELOCT_FF1	Source register of the PIC18 MOVFF instruction	15
RELOCT_FF2	Destination register of the PIC18 MOVFF instruction	16
RELOCT_SF2	Destination register of the PIC18 MOVSF instruction	16
RELOCT_LFSR1	First word of the PIC18 LFSR instruction	17
RELOCT_LFSR2	Second word of the PIC18 LFSR instruction	18
RELOCT_BRA	PIC18 BRA instruction	19
RELOCT_RCALL	PIC18 RCALL instruction	19
RELOCT_CONDBRA	PIC18 relative conditional branch instructions	20
RELOCT_UPPER	Highest order 8 bits of a 24-bit address	21
RELOCT_ACCESS	PIC18 access bit	22
RELOCT_PAGESEL_WREG	Selecting the correct page using WREG as scratch	23
RELOCT_PAGESEL_BITS	Selecting the correct page using bit set/clear instructions	24
RELOCT_SCNSZ_LOW RELOCT_SCNSZ_HIGH RELOCT_SCNSZ_UPPER	Size of a section	25 26 27
RELOCT_SCNEND_LOW RELOCT_SCNEND_HIGH RELOCT_SCNEND_UPPER	Address of the end of a section	28 29 30
RELOCT_SCNEND_LFSR1 RELOCT_SCNEND_LFSR2	Address of the end of a section on LFSR	31 32
RELOCT_TRIS_4BIT	File register address for 4-bit TRIS instruction	33

A.5 `struct syment` - SYMBOL TABLE ENTRY

Symbols are created for all identifiers, as well as sections, function begins, function ends, block begins and block ends.

```
#define SYMNMLEN 8
struct syment
{
    union
    {
      char _n_name[SYMNMLEN];
      struct
      {
        unsigned long _n_zeroes;
        unsigned long _n_offset;
      } _n_n;
      char *_n_nptr[2];
    } _n;

    unsigned long n_value;
    short n_scnum;
    unsigned short n_type;
    char n_sclass;
    unsigned char n_numaux;
}
```

A.5.1 `union _n`

The symbol name may be stored directly as a string, or it may be a reference to the string table. Symbol names of fewer than eight characters are stored here, with all others being stored in the string table. It is from this structure that the inspiration comes for extending the section data structures to allow for section names to be stored in the symbol table.

A.5.1.1 `char _n_name [SYMNMLEN]`

In-place symbol name, if fewer than eight characters long.

A.5.1.2 `struct _n_n`

Symbol name is located in string table. If the first four characters of the symbol name are zero, then the last four form an offset into the string table to find the name of the symbol.

A.5.1.2.1 `unsigned long _n_zeros`

First four characters of the symbol name are zero.

A.5.1.2.2 `unsigned long _n_offset`

Offset of symbol name in the string table.

A.5.1.3 `char *_n_nptr`

Allows for overlaying.

A.5.2 `unsigned long n_value`

Value of symbol. Typically, this is the address of the symbol within the section in which it resides. For link-time constants (e.g., the Microchip symbol `_stksize`), the value is a literal value and not an address. To the linker, there is typically no difference. The distinction is only in the usage in the application code.

A.5.3 short n_scnum

References the section number where this symbol is located.

A.5.4 unsigned short n_type

Base type and derived type.

A.5.4.1 SYMBOL TYPES

Table A-5 lists the base types, along with a description and respective values.

TABLE A-5: BASE SYMBOL TYPES

Type	Description	Value
T_NULL	null	0
T_VOID	void	1
T_CHAR	character	2
T_SHORT	short integer	3
T_INT	integer	4
T_LONG	long integer	5
T_FLOAT	floating point	6
T_DOUBLE	double length floating point	7
T_STRUCT	structure	8
T_UNION	union	9
T_ENUM	enumeration	10
T_MOE	member of enumeration	11
T_UCHAR	unsigned character	12
T_USHORT	unsigned short	13
T_UINT	unsigned integer	14
T_ULONG	unsigned long	15

A.5.4.2 DERIVED TYPES

Pointers, arrays, and functions are handled via derived types. Table A-6 lists the derived types, along with a description and respective values.

TABLE A-6: DERIVED TYPES

Derived Type	Description	Value
DT_NON	no derived type	0
DT_PTR	pointer	1
DT_FCN	function	2
DT_ARY	array	3

A.5.5 char n_sclass

Storage class of the symbol. Table A-7 lists the storage classes, along with a description and respective values.

TABLE A-7: STORAGE CLASSES

Storage Class	Description	Value
C_EFCN	Physical end of function	0xFF
C_NULL	Null	0
C_AUTO	Automatic variable	1
C_EXT	External symbol	2
C_STAT	Static	3
C_REG	Register variable	4
C_EXTDEF	External definition	5
C_LABEL	Label	6
C_ULABEL	Undefined label	7
C_MOS	Member of structure	8
C_ARG	Function argument	9
C_STRTAG	Structure tag	10
C_MOU	Member of union	11
C_UNTAG	Union tag	12
C_TPDEF	Type definition	13
C_USTATIC	Undefined static	14
C_ENTAG	Enumeration tag	14
C_MOE	Member of enumeration	16
C_REGPARM	Register parameter	17
C_FIELD	Bit field	18
C_AUTOARG	Automatic argument	19
C_LASTENT	Dummy entry (end of block)	20
C_BLOCK	"bb" or "eb"	100
C_FCN	"bf" or "ef"	101
C_EOS	End of structure	102
C_FILE	File name	103
C_LINE	Line number reformatted as symbol table entry	104
C_ALIAS	Duplicate tag	105
C_HIDDEN	External symbol in dmert public library	106
C_EOF	End of file	107
C_LIST	Absolute listing on or off	108
C_SECTION	Section	109

A.5.6 unsigned char n_numaux

The number of auxiliary entries for this symbol.

A.6 `struct coff_lineno` - LINE NUMBER ENTRY

Any executable source line of code gets a `coff_lineno` entry in the line number table associated with its section. For a PICmicro MCU COFF file, this means that every instruction may have a `coff_lineno` entry since the debug information is often for debugging through the absolute listing file. Readers of this information should note that the COFF file is not required to have an entry for every instruction, though it typically does. This information is significantly different from the System V format.

```
struct coff_lineno
{
   unsigned long l_srcndx;
   unsigned short l_lnno;
   unsigned long l_paddr;
   unsigned short l_flags;
   unsigned long l_fcnndx;
} coff_lineno_t;
```

A.6.1 `unsigned long l_srcndx`

Symbol table index of associated source file.

A.6.2 `unsigned short l_lnno`

Line number.

A.6.3 `unsigned long l_paddr`

Address of code for this line number entry.

A.6.4 `unsigned short l_flags`

Bit flags for the line number entry. Table A-8 lists the bit flags, along with a description and respective values.

TABLE A-8: LINE NUMBER ENTRY FLAGS

Flag	Description	Value
LINENO_HASFCN	Set if `l_fcndx` is valid	0x01

A.6.5 `unsigned long l_fcnndx`

Symbol table index of associated function (if there is one).

A.7 `struct aux_file` - AUXILIARY SYMBOL TABLE ENTRY FOR A SOURCE FILE

```
typedef struct aux_file
{
   unsigned long x_offset;
   unsigned long x_incline;
   unsigned char x_flags;
   char _unused[9];
} aux_file_t;
```

A.7.1 `unsigned long x_offset`

String table offset for filename.

A.7.2 `unsigned long x_incline`

Line number at which this file was included. If 0, file was not included.

A.7.3 unsigned char x_flags

Bit flags for the .file entry. Table A-9 lists the bit flags, along with a description and respective values.

TABLE A-9: .file ENTRY FLAGS

Flag	Description	Value
X_FILE_DEBUG_ONLY	This .file entry was included for debugging purposes only.	0x01

A.8 struct aux_scn - AUXILIARY SYMBOL TABLE ENTRY FOR A SECTION

```
typedef struct aux_scn
{
   unsigned long x_scnlen;
   unsigned short x_nreloc;
   unsigned short x_nlinno;
   char _unused[10];
} aux_scn_t;
```

A.8.1 unsigned long x_scnlen

Section length.

A.8.2 unsigned short x_nreloc

Number of relocation entries.

A.8.3 unsigned short x_nlinno

Number of line numbers.

A.9 struct aux_tag - AUXILIARY SYMBOL TABLE ENTRY FOR A struct/union/enum TAGNAME

```
typedef struct aux_tag
{
   char _unused[6];
   unsigned short x_size;
   char _unused2[4];
   unsigned long x_endndx;
   char _unused3[2];
} aux_tag_t;
```

A.9.1 unsigned short x_size

Size of structure, union or enumeration.

A.9.2 unsigned long x_endndx

Symbol index of next entry beyond this structure, union or enumerated tag.

A.10 `struct aux_eos` - AUXILIARY SYMBOL TABLE ENTRY FOR AN END OF `struct/union/enum`

```
typedef struct aux_eos
{
   unsigned long x_tagndx;
   char _unused[2];
   unsigned short x_size;
   char _unused2[10];
} aux_eos_t;
```

A.10.1 `unsigned long x_tagndx`

Symbol index of a structure, union or enumerated tag.

A.10.2 `unsigned short x_size`

Size of a structure, union or enumeration.

A.11 `struct aux_fcn` - AUXILIARY SYMBOL TABLE ENTRY FOR A FUNCTION NAME

```
typedef struct aux_fcn
{
   unsigned long x_tagndx;
   unsigned long x_size;
   unsigned long x_lnnoptr;
   unsigned long x_endndx;
   short x_actscnum;
} aux_fcn_t;
```

A.11.1 `unsigned long x_tagndx`

The symbol table index of the structure or union tagname associated with the return value type, if the return value base type is structure or union.

A.11.2 `unsigned long x_lnnoptr`

File pointer to line numbers for this function.

A.11.3 `unsigned long x_endndx`

Symbol index of next entry beyond this function.

A.11.4 `short x_actscnum`

Section number of the static activation record data.

A.12 `struct aux_fcn_calls` - AUXILIARY SYMBOL TABLE ENTRY FOR FUNCTION CALL REFERENCES

```
typedef struct aux_fcn_calls
{
   unsigned long x_calleendx;
   unsigned long x_is_interrupt;
   char _unused[10];
} aux_fcn_calls_t;
```

A.12.1 `unsigned long x_calleendx`

Symbol index of the called function. If call of a higher order function, set to `AUX_FCN_CALLS_HIGHERORDER`.

```
#define AUX_FCN_CALLS_HIGHERORDER ((unsigned long)-1)
```

A.12.2 `unsigned long x_is_interrupt`

Specifies whether the function is an interrupt, and if so, the priority of the interrupt.

0: not an interrupt

1: low priority

2: high priority

A.13 `struct aux_arr` - AUXILIARY SYMBOL TABLE ENTRY FOR AN ARRAY

```
#define X_DIMNUM 4
typedef struct aux_arr
{
   unsigned long x_tagndx;
   unsigned short x_lnno;
   unsigned short x_size;
   unsigned short x_dimen[X_DIMNUM];
} aux_arr_t;
```

A.13.1 `unsigned long x_tagndx`

The symbol table index of the structure or union tagname associated with the array element type, if the base type is structure or union.

A.13.2 `unsigned short x_size`

Size of array.

A.13.3 `unsigned short x_dimen[X_DIMNUM]`

Size of first four dimensions.

A.14 `struct aux_eobf` - AUXILIARY SYMBOL TABLE ENTRY FOR THE END OF A BLOCK OR FUNCTION

```
typedef struct aux_eobf
{
   char _unused[4];
   unsigned short x_lnno;
   char _unused2[12];
} aux_eobf_t;
```

A.14.1 `unsigned short x_lnno`

C source line number of the end, relative to start of block/function.

A.15 `struct aux_bobf` - AUXILIARY SYMBOL TABLE ENTRY FOR THE BEGINNING OF A BLOCK OR FUNCTION

```
typedef struct aux_bobf
{
   char _unused[4];
   unsigned short x_lnno;
   char _unused2[6];
   unsigned long x_endndx;
   char _unused3[2];
} aux_bobf_t;
```

A.15.1 `unsigned short x_lnno`

C source line number of the beginning, relative to start enclosing scope.

A.15.2 `unsigned long x_endndx`

Symbol index of next entry past this block/function.

A.16 `struct aux_var` - AUXILIARY SYMBOL TABLE ENTRY FOR A VARIABLE OF TYPE `struct/union/enum`

```
typedef struct aux_var
{
   unsigned long x_tagndx;
   char _unused[2];
   unsigned short x_size;
   char _unused2[10];
} aux_var_t;
```

A.16.1 `unsigned long x_tagndx`

Symbol index of a structure, union or enumerated tag.

A.16.2 `unsigned short x_size`

Size of the structure, union or enumeration.

A.17 `struct aux_field` - AUXILIARY ENTRY FOR A BIT FIELD

```
typedef struct aux_field
{
   char _unused[6];
   unsigned short x_size;
   char _unused2[10];
} aux_field_t;
```

A.17.1 `unsigned short x_size`

The size of the bit field, in bits.

Appendix B. ANSI Implementation-defined Behavior

B.1 INTRODUCTION

This section discusses MPLAB C18 implementation-defined behavior. The ISO standard for C requires that vendors document the specifics of "implementation-defined" features of the language.

Note: The section numbers in parenthesis, e.g., (6.1.2), refer to the ANSI C standard X3.159-1989.

Implementation-defined behavior for the following sections is covered in section G.3 of the ANSI C Standard.

B.2 IDENTIFIERS

ANSI C Standard: "The number of significant initial characters (beyond 31) in an identifier without external linkage (6.1.2)."

"The number of significant initial characters (beyond 6) in an identifier with external linkage (6.1.2)."

"Whether case distinctions are significant in an identifier with external linkage (6.1.2)."

Implementation: All MPLAB C18 identifiers have at least 31 significant characters. Case distinctions are significant in an identifier with external linkage.

B.3 CHARACTERS

ANSI C Standard: "The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (6.1.3.4)."

Implementation: The value of the integer character constant is the 8-bit value of the first character. Wide characters are not supported.

ANSI C Standard: "Whether a 'plain' `char` has the same range of values as `signed char` or `unsigned char` (6.2.1.1)."

Implementation: A plain `char` has the same range of values as a `signed char`. For MPLAB C18, this may be changed to `unsigned char` via a command line switch (`-k`).

B.4 INTEGERS

ANSI C Standard: "A `char`, a `short int` or an `int` bit field, or their signed or unsigned varieties, or an enumeration type, may be used in an expression wherever an `int` or `unsigned int` may be used. If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. These are called the *integral promotions*. All other arithmetic types are unchanged by the integral promotions.

"The integral promotions preserve value including sign. (6.2.1.1)."

Implementation: MPLAB C18 does not enforce this by default. The `-Oi` option can be used to require the compiler to enforce the ANSI defined behavior. See **Section 2.7.1 "Integer Promotions"**.

ANSI C Standard: "The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (6.2.1.2)."

Implementation: When converting from a larger integer type to a smaller integer type, the high order bits of the value are discarded and the remaining bits are interpreted according to the type of the smaller integer type. When converting from an unsigned integer to a signed integer of equal size, the bits of the unsigned integer are simply reinterpreted according to the rules for a signed integer of that size.

ANSI C Standard: "The results of bitwise operations on signed integers (6.3)."

Implementation: The bitwise operators are applied to the signed integer as if it were an unsigned integer of the same type (i.e., the sign bit is treated as any other bit).

ANSI C Standard: "The sign of the remainder on integer division (6.3.5)."

Implementation: The remainder has the same sign as the quotient.

ANSI C Standard: "The result of a right shift of a negative-valued signed integral type (6.3.7)."

Implementation: The value is shifted as if it were an unsigned integral type of the same size (i.e., the sign bit is not propagated).

B.5 FLOATING-POINT

ANSI C Standard: "The representations and sets of values of the various types of floating-point numbers (6.1.2.5)."

"The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value (6.2.1.3)."

"The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number (6.2.1.4)."

Implementation: See **Section 2.1.2 "Floating-point Types"**.

The rounding to the nearest method is used.

B.6 ARRAYS AND POINTERS

ANSI C Standard: "The type of integer required to hold the maximum size of an array — that is, the type of the `sizeof` operator, `size_t` (6.3.3.4, 7.1.1)."

Implementation: `size_t` is defined as an `unsigned short long int`.

ANSI C Standard: "The result of casting a pointer to an integer or vice versa (6.3.4)."

Implementation: The integer will contain the binary value used to represent the pointer. If the pointer is larger than the integer, the representation will be truncated to fit in the integer.

ANSI C Standard: "The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t` (6.3.6, 7.1.1)."

Implementation: `ptrdiff_t` is defined as an `unsigned long short`.

B.7 REGISTERS

ANSI C Standard: "The extent to which objects can actually be placed in registers by use of the `register` storage-class specifier (6.5.1)."

Implementation: The `register` storage-class specifier is ignored.

B.8 STRUCTURES AND UNIONS

ANSI C Standard: "A member of a union object is accessed using a member of a different type (6.3.2.3)."

Implementation: The value of the member is the bits residing at the location for the member interpreted as the type of the member being accessed.

ANSI C Standard: "The padding and alignment of members of structures (6.5.2.1)."

Implementation: Members of structures and unions are aligned on byte boundaries.

B.9 BIT FIELDS

ANSI C Standard: "Whether a 'plain' `int` bit field is treated as a `signed int` or as an `unsigned int` bit field (6.5.2.1)."

Implementation: A "plain" `int` bit field is treated as a `signed int` bit field.

ANSI C Standard: "The order of allocation of bit fields within a unit (6.5.2.1)."

Implementation: Bit fields are allocated from least significant bit to most significant bit in order of occurrence.

ANSI C Standard: "Whether a bit field can straddle a storage-unit boundary (3.5.2.1)."

Implementation: A bit field cannot straddle a storage unit boundary.

B.10 ENUMERATIONS

ANSI C Standard: "The integer type chosen to represent the values of an enumeration type (6.5.2.2)."

Implementation: The smallest type capable of representing all values in the enumeration type.

B.11 SWITCH STATEMENT

ANSI C Standard: "The maximum number of `case` values in a `switch` statement (6.6.4.2)."

Implementation: The maximum number of values is limited only by target memory.

B.12 PREPROCESSING DIRECTIVES

ANSI C Standard: "The method for locating includable source files (6.8.2)."

Implementation: See **Section 2.5.1 "System Header Files"**.

ANSI C Standard: "The support for quoted names for includable source files (6.8.2)."

Implementation: See **Section 2.5.2 "User Header Files"**.

ANSI C Standard: "The behavior on each recognized `#pragma` directive (6.8.6)."

Implementation: See **Section 2.9 "Pragmas"**.

Appendix C. Command-line Summary

Usage: `mcc18 [options] file [options]`

TABLE C-1: COMMAND-LINE SUMMARY

Option	Description	Reference
`-?, --help`	Displays the help screen	1.2
`-I=<path>`	Add 'path' to include path	2.5.1, 2.5.2
`-fo=<name>`	Object file name	1.2.1
`-fe=<name>`	Error file name	1.2.1
`-k`	Set plain char type to unsigned char	2.1
`-ls`	Large stack (can span multiple banks)	3.2.4
`-ms`	Set compiler memory model to small model (default)	2.6, 3.1
`-ml`	Set compiler memory model to large model	2.6, 3.1
`-O, -O+`	Enable all optimizations (default)	4
`-O-`	Disable all optimizations	4
`-Od+`	Enable dead code removal (default)	4.10
`-Od-`	Disable dead code removal	4.10
`-Oi+`	Enable integer promotion	2.7.1
`-Oi-`	Disable integer promotion (default)	2.7.1
`-Om+`	Enable duplicate string merging (default)	4.1
`-Om-`	Disable duplicate string merging	4.1
`-On+`	Enable banking optimizer (default)	4.3
`-On-`	Disable banking optimizer	4.3
`-Op+`	Enable copy propagation (default)	4.8, 4.10
`-Op-`	Disable copy propagation	4.8, 4.10
`-Or+`	Enable redundant store elimination (default)	4.9
`-Or-`	Disable redundant store elimination	4.9
`-Ou+`	Enable unreachable code removal (default)	4.7
`-Ou-`	Disable unreachable code removal	4.7
`-Os+`	Enable code straightening (default)	4.5
`-Os-`	Disable code straightening	4.5
`-Ot+`	Enable tail merging (default)	4.6
`-Ot-`	Disable tail merging	4.6
`-Ob+`	Enable branch optimizations (default)	4.2
`-Ob-`	Disable branch optimizations	4.2
`-sca`	Enable default auto locals (default). Valid for Non-extended mode only.	2.3
`-scs`	Enable default static locals. Valid for Non-extended mode only.	2.3
`-sco`	Enable default overlay locals (statically allocate activation records). Valid for Non-extended mode only.	2.3

TABLE C-1: COMMAND-LINE SUMMARY (CONTINUED)

Option	Description	Reference
-Oa+	Enable default data in access memory. Valid for Non-extended mode only.	2.9.1.3
-Oa-	Disable default data in access memory (default). Valid for Non-extended mode only.	2.9.1.3
-Ow+	Enable WREG tracking (default)	4.4
-Ow-	Disable WREG tracking	4.4
-Opa+	Enable procedural abstraction (default)	4.11
-Opa-	Disable procedural abstraction	4.11
-pa=<repeat count>	Set procedural abstraction repeat count (default = 4)	4.11
-p=<processor>	Set processor (default is generic)	1.2.4, 2.6, 2.10
-D<macro>[=text]	Define a macro	1.2.3
-w={1|2|3}	Set warning level (default = 2)	1.2.2
-nw=<n>	Suppress message <n>	1.2.2
-verbose	Operate verbosely (show banner and other information)	1.2
--extended	Generate Extended mode code.	1.2.5
--no-extended	Generate Non-extended mode code.	1.2.5
--help-message-list	Display a list of all diagnostic messages	1.2.2
--help-message-all	Display help for all diagnostic messages	1.2.2
--help-message=<n>	Display help on diagnostic number <n>	1.2.2
--help-config	Display help on device-specific configuration settings	2.9.4

Appendix D. MPLAB C18 Diagnostics

This appendix lists errors, warnings, and messages generated by the MPLAB C18 compiler.

D.1 ERRORS

1000: %s

1002: syntax error, '%s' expected

The syntax of the pre-processor construct was expecting the specified token. Common causes include typographical errors, missing required operands to the directive, and mis-matched parenthesis.

1013: error in pragma directive

MPLAB C18 was expecting the pragma being parsed to be complete, but did not see a new line. This would be caused by extra text following the pragma.

1014: redundant attribute specifier declaring section '%s'

The #pragma sectiontype directive specifies the overlay or the access attribute multiple times.

1016: integer constant expected for #line directive

The line number operand of the #line preprocessor directive must be an integer constant.

1017: symbol name expected in 'interrupt' pragma

The 'save=' clause expects a comma-delimited list of statically allocated in-scope symbol names which are to be saved and restored by the interrupt function being specified. Common causes include specifying a symbol which is not currently in scope, not including a header file which declares the symbol being referenced, and typographical errors in the symbol name.

1018: function name expected in 'interrupt' pragma

The name of a function to be declared as an interrupt is expected as the first parameter to the 'interrupt' pragma. The function symbol must be currently in scope and must take no parameters and return no value. Common causes include a missing prototype for the function being declared as an interrupt and typographical errors.

1019: '%s' is a compiler managed resource - it should not appear in a save= list

The symbol named is not valid in a save= clause of an interrupt declaration. There are some locations which if saved/restored via a save= will produce aberrant code. These locations do not need additional context save and can be safely removed from the save= clause to correct the error.

1020: unexpected input following '%s'

Extra information exists on the given preprocessor construct.

1021: unterminated comment

A C-style comment (i.e., /*) was not terminated. The line number of the error message shows where the comment begins.

1022: end of file in argument '%s' for macro '%s'

The end of file was found while processing the specified argument in the specified macro. Most likely cause is a missing parenthesis.

1023: end of file in valist argument for macro '%s'

The end of file was found while processing the variable arguments in the specified macro. Most likely cause is a missing parenthesis.

1024: macro '%s' expects %d arguments, but only %d found

The specified macro expects a different number of arguments than specified. To use a macro, the number of arguments passed must match exactly the number of arguments defined for that macro.

1025: missing '%c' in header name

The end of the file was found while processing the header file name of a #include statement. The cause is a missing terminator for the #include directive on the line specified.

1026: malformed #include directive

Either a '"' or a '<' was expected after the #include, but something else was found. Most likely caused by a mis-typed directive.

1027: unable to locate '%s'

The specified header file could not be found in the include file search paths (either the system header files or the user header files). Make sure that the appropriate -I command-line options have been specified. Other causes include a mis-typed header file or insufficient access rights.

1028: %s without matching #if

The specified preprocessor directive was found without a matching #if. Most likely caused by a mismatch in nesting or possibly a misspelling.

1029: malformed expression in '%s'

The expression for the specified preprocessor directive is incorrect. Most likely caused by a mismatched parenthesis or a misspelling.

1030: identifier expected in %s

An identifier was expected in the specified preprocessor directive, but a C identifier was not found. Most likely cause is a mis-typed identifier.

1031: '%c' expected in 'defined'

The 'defined' preprocessor directive expects to be followed by either parentheses or an identifier. Most likely cause is a missing parenthesis or a mis-typed identifier.

1032: ')' expected in expansion of macro '%s'

A closing parenthesis was expected when expanding the specified macro. Most likely cause is a missing parenthesis.

1033: preprocessor can only input one file at a time

The preprocessor can only handle one source file as input. Most likely caused by an error in the compiler executable invoking the preprocessor. If invoking the preprocessor separately, correct the command line.

1034: previous definition of macro '%s' does not agree

According to the ANSI standard, an identifier currently defined as an object-like macro shall not be redefined by another #define preprocessing directive unless the second definition is an object-like macro definition and the two replacement lists are identical. Likewise, an identifier currently defined as a function-like macro shall not be redefined by another #define preprocessing directive unless the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical.

1035: expecting macro name, received '%s' instead

An identifier was expected, but a C identifier was not found. Most likely cause is a mis-typed identifier.

1036: syntax error in macro argument list, expecting ')'

Immediately after the variable argument list (...), a closing parenthesis is expected.

1037: duplicate parameter name '%s' in macro '%s'

A macro's parameter names must be unique.

1038: syntax error in macro argument list

Either a comma was expected in the argument list and not found, or if variable argument list (...) was specified, a closing parenthesis was expected and not found.

1039: illegal character in macro name '%c'

Whitespace or a begin parenthesis is expected after the macro name. Most likely cause is a mis-typed macro name.

1040: # or ## operator found in simple macro %s

The stringization (#) and concatenation (##) preprocessor operators can only be used with an argument of a function-like macro.

1041: # operator requires a parameter name as operand

The stringization preprocessor operator (#) requires a parameter name as the operand, but a C identifier was not found. Most likely cause is a mis-typed identifier.

1042: filename for %s directive exceeds maximum filename length

The name of the file specified in the specified preprocessor directive exceeds the maximum filename length of MAX_FILENAME_PATH_LEN.

1050: section address permitted only at definition

The absolute address in the location clause of the #pragma sectiontypedirective may only be specified in the first pragma defining this section.

1052: section overlay attribute does not match definition

MPLAB C18 requires that a previously declared section's attribute must match those which are being specified in the current #pragma sectiontype directive.

1053: section share attribute does not match definition

MPLAB C18 requires that a previously declared section's attribute must match those which are being specified in the current #pragma sectiontype directive.

1054: section type does not match definition

MPLAB C18 has previously seen this section name, but it was of a different type (i.e., code, idata, udata, romdata).

1055: section access attribute does not match definition

MPLAB C18 requires that a previously declared section's attribute must match those which are being specified in the current #pragma sectiontype directive.

1070: too many line numbers in section '%s'

The COFF file format only allows (32767 * 2 + 1) lines in a single section. Reduce the number of lines in your source file.

1071: too many relocations in section '%s'

The COFF file format only allows (32767 * 2 + 1) relocations in a single section. Reduce the number of variable references in your source file.

1072: too many function calls for ISR '%s'

An ISR may only call 253 distinct functions. The output object file format (COFF) limits the number of auxiliary entries to 255. An ISR requires two auxiliary entries and a distinct auxiliary entry is required for each call to a distinct function.

1073: too many function calls for '%s'

A non-interrupt function may only call 254 distinct functions. The output object file format (COFF) limits the number of auxiliary entries to 255. A non-interrupt function requires one auxiliary entry and a distinct auxiliary entry is required for each call to a distinct function.

1099: %s

source code '#error' directive message

1100: syntax error

Invalid function type definition.

1101: lvalue required

An expression which designates an object is required. Common causes include missing parentheses and a missing '*' operator.

1102: cannot assign to 'const' modified object

An object qualified with 'const' is declared to be read-only data and modifications to it are therefore not allowed.

1103: unknown escape sequence '%s'

The specified escape sequence is not known to the compiler. Check the ANSI standard for a list of valid character escape sequences.

1104: division by zero in constant expression

The compiler cannot process a constant expression which contains a divide by (or modulus by) zero.

1105: symbol '%s' has not been defined

A symbol has been referenced before it has been defined. Common causes include a misspelled symbol name, a missing header file that declares the symbol, and a reference to a symbol valid only in an inner scope.

1106: '%s' is not a function

A symbol must be a function name in order to be declared as an interrupt function.

1107: interrupt functions must not take parameters

When the processor vectors to an interrupt routine, no parameters are passed, so a function declared as an interrupt function should not expect parameters.

1108: interrupt functions must not return a value

Since interrupts are invoked asynchronously by the processor, there will not be a calling routine to which a value can be returned.

1109: type mismatch in redeclaration of '%s'

The type of the symbol declared is not compatible with the type of a previous declaration of the same symbol. Common causes include missing qualifiers or misplaced qualifiers.

1111: undefined label '%s' in '%s'

The label has been referenced via a 'goto' statement, but has not been defined in the function. Common causes include a misspelled label identifier and a reference to an out of scope label, (i.e., a label defined in another function).

1112: integer type expected in switch control expression

The control expression for a switch statement must be an integer type. Common causes include a missing '*' operator and a missing '[]' operator.

1113: integer constant expected for case label value

The value for a case label must be an integer constant.

1114: case label outside switch statement detected

A 'case' label is only valid inside the body of a switch statement. Common causes include a misplaced '}'.

1159: default label outside switch statement detected

A 'default' label is only valid inside the body of a switch statement. Common causes include a misplaced '}'.

1115: multiple default labels in switch statement

A switch statement can only have a single 'default' label. Common causes include a missing '}' to close an inner switch.

1116: type mismatch in return statement

The type of the return value is not compatible with the declared return type of the function. Common causes include a missing '*' or '[]' operator.

1117: scalar type expected in 'if' statement

An 'if' statement control expression must be of scalar type, (i.e., an integer or a pointer).

1118: scalar type expected in 'while' statement

A 'while' statement control expression must be of scalar type, (i.e., an integer or a pointer).

1119: scalar type expected in 'do..while' statement

A 'do..while' statement control expression must be of scalar type, (i.e., an integer or a pointer).

1120: scalar type expected in 'for' statement

A 'for' statement control expression must be of scalar type, (i.e., an integer or a pointer).

1121: scalar type expected in '?:' expression

A '?:' operator control expression must be of scalar type, (i.e., an integer or a pointer).

1122: scalar operand expected for '!' operator

The '!' operator requires that its operand be of scalar type.

1123: scalar operands expected for '||' operator

The logical OR operator, '||', requires scalar operands.

1124: scalar operands expected for '&&' operator

The logical AND operator, '&&', requires scalar operands.

1125: 'break' must appear in a loop or switch statement

A 'break' statement must be inside a 'while', 'do', 'for', or 'switch' statement. Common causes include a misplaced '}'.

1126: 'continue' must appear in a loop statement

A 'continue' statement must be inside a 'while', 'do', 'for', or 'switch' statement.

1127: operand type mismatch in '?:' operator

The types of the result operands of the '?:' operator must be either both scalar types or compatible types.

1128: compatible scalar operands required for comparison

A comparison operator must have operands of compatible scalar types.

1129: [] operator requires a pointer and an integer as operands

The array access operator, '[]', requires that one operand be a pointer and the other be an integer, that is, for 'x[y]' the expression '*(x+y)' must be valid. 'x[y]' is functionally equivalent to '*(x+y)'.

1130: pointer operand required for '*' operator

The '*' dereference operator requires a pointer to a non-void object as its operand

1131: type mismatch in assignment

The assignment operators require that the result of the right hand expression be of compatible type with the type of the result of the left hand expression. Common causes include a missing '*' or '[]' operator.

1132: integer type expected for right hand operand of '-=' operator

The '-=' operator requires that the right hand side be of integer type when the left hand side is of pointer type. Common causes include a missing '*' or '[]' operator.

1133: type mismatch in '-=' operator

The types of the operands of the '-=' operator must be such that for 'x-=y' the expression 'x=x-y' is valid.

1134: arithmetic operands required for multiplication operator

The '*' and '*=' multiplication operators require that their operands be of arithmetic type. Common causes include a missing '*' dereference operator or a missing '[]' index operator.

1135: integer operands required for modulus operator

The '%' and '%=' modulus operators require that their operands be of integer type. Common causes include a missing '*' dereference operator or a missing '[]' index operator.

1136: integer operands required for shift operator

The bitwise shift operators require that their operands be of integer type. Common causes include a missing '*' dereference operator or a missing '[]' index operator.

1137: integer types required for bitwise AND operator

The '&' and '&=' operators require that both operands be of integer type. Common causes include a missing '*' or '[]' operator.

1138: integer types required for bitwise OR operator

The '|' and '|=' operators require that both operands be of integer type. Common causes include a missing '*' or '[]' operator.

1139: integer types required for bitwise XOR operator

The '^' and '^=' operators require that both operands be of integer type. Common causes include a missing '*' or '[]' operator.

1140: integer type required for bitwise NOT operator

The '~' operator requires that the operand be of integer type. Common causes include a missing '*' or '[]' operator.

1141: integer type expected for pointer addition

The addition operator requires that when one operand is of pointer type, the other must be of integer type. Common causes include a missing '*' or '[]' operator.

1142: type mismatch in '+' operator

The types of the operands of the '+' operator must be such that one operand is of pointer type and the other is of integer type or both operands are of arithmetic type.

1143: pointer difference requires pointers to compatible types

When calculating the difference between two pointers, the pointers must point to objects of compatible type. Common causes include missing parentheses and a missing '[]' operator.

1144: integer type required for pointer subtraction

When the left hand operand of the subtraction operator is of pointer type, the right hand operand must be of integer type. Common causes include a missing '*' or '[]' operator.

1145: arithmetic type expected for subtraction operator

When the left hand operand is not of pointer type, the subtraction operator requires that both operands be of arithmetic type.

1146: type mismatch in argument %d

The type of an argument to a function call must be compatible with the declared type of the corresponding parameter.

1147: scalar type expected for increment operator

The increment operators require that the operand be a modifiable lvalue of scalar type.

1148: scalar type expected for decrement operator

The decrement operators require that the operand be a modifiable lvalue of scalar type.

1149: arithmetic type expected for unary plus

The unary plus operator requires that its operand be of arithmetic type.

1150: arithmetic type expected for unary minus

The unary minus operator requires that its operand be of arithmetic type.

1151: struct or union object designator expected

The member access operators, '.' and '->' require operands of struct/union and pointer to struct/union, respectively.

1152: scalar or void type expected for cast

An explicit cast requires that the type of the operand be of scalar type and the type being cast to be scalar type or void type.

1153: cannot assign array type objects

An object of array type may not be directly assigned. Assignment is allowed only to array elements.

1154: parameter %d in '%s' must have a name

Parameters in a function definition must have an identifier declarator to name them. The naming declarator is not required in prototypes, but is in a definition.

1155: 'overlay' symbol '%s' not in function scope

Variables may only be overlay within the scope of a function.

1156: member '%s' declared as having function type

Structure and union members cannot be of function type. Likely cause is an incorrectly declared function pointer.

1157: function 'main' should be declared as 'void main (void)'

The MPLAB C18 startup code will invoke function 'main' with no parameters and expects no return value. 'main' should always be declared to take no parameters and to not return a value.

1158: arithmetic operands required for division operator

The '/' and '/=' division operators require that their operands be of arithmetic type. Common causes include a missing '*' dereference operator or a missing '[]' index operator.

1160: conflicting storage classes specified

A declaration may only specify a single storage class.

1161: conflicting base types specified

A declaration may only specify a single base type (void, int, float, et.al.). Multiple instances of the same base type is also an error (e.g.,int int x;).

1162: both 'signed' and 'unsigned' specified

A type may include only one of 'signed' and 'unsigned'.

1163: function must be located in program memory

All functions must be located in program memory, as data memory is not executable.

1165: reference to incomplete tag '%s'

A forward reference struct or union tag cannot be referenced directly in a declaration. Only pointers to a forward referenced tag may be declared.

1166: invalid type specification

The type specification is not valid. Common causes include typographic errors or misuse of a typedef type. (e.g., "int enum myEnum xyz;" has an invalid type specification.)

1168: reference to undefined enumeration tag '%s'

An enumeration tag must be defined prior to any declarations which reference it. Unlike structure and union tags, forward references to enumeration tags are not allowed.

1169: anonymous members allowed in unions only

An anonymous structure member may be declared only as a member of a union.

1170: non-integral type bitfield detected

The type of a bitfield member of a structure must be an integral type.

1171: bitfield width greater than 8 detected

A bitfield must fit within a single storage unit, which for MPLAB C18 is a byte. Thus, a bitfield must contain 8 or fewer bits.

1172: enumeration value of '%s' does not match previous

When the same enumeration constant name is used in multiple enumeration tags, the value of the enumeration constant must be the same in each enumeration.

1173: cannot locate a parameter in program memory, '%s'

Since all parameters are located on the stack, it is not possible to locate a parameter in program memory. Common causes include a mis-typed pointer to program memory declaration.

1174: local '%s' in program memory can not be 'auto'

A local variable which is located in program memory must be declared as static or extern, as 'auto' local variables must be located on the stack.

1175: static parameter detected in function pointer '%s'

Function pointers require parameters be passed via the stack. When compiling with static locals enabled, declare parameters for function pointers and for functions whose addresses are assigned to function pointers explicitly to 'auto'.

1200: cannot reference the address of a bitfield

The address of a bitfield member of a structure cannot be referenced directly.

1201: cannot dereference a pointer to 'void' type

The '*' dereference operator requires a pointer to a non-void object as its operand.

1202: call of non-function

The operand of the '()' function call post-fix operator must be of type 'pointer to function.' Most commonly, this is a function identifier. Common causes include missing scope parentheses.

1203: too few arguments in function call

To call a function, the number of arguments passed must match exactly the number of parameters declared for the function.

1204: too many arguments in function call

To call a function, the number of arguments passed must match exactly the number of parameters declared for the function.

1205: unknown member '%s' in '%s'

The structure or union tag does not have a member of the name requested. Common causes include a misspelled member name and a missing member access operator for a nested structure.

1206: unknown member '%s'

The structure or union type does not have a member of the name requested. Common causes include a misspelled member name and a missing member access operator for a nested structure.

1207: tag '%s' is incomplete

An incomplete struct or union tag cannot be referenced by the member access operators. Common causes include a misspelled structure tag name in the symbol definition.

1208: "#pragma interrupt" detected inside function body

The 'interrupt' pragma is only available at file level scope.

1210: unknown symbol '%s' in interrupt save list

The 'interrupt' pragma requires that symbols listed in the 'save' list must be declared and in scope

1211: missing definition for interrupt function '%s'

The function was declared as an interrupt, but was never defined. The function definition of an interrupt function must be in the same module as the pragma declaring the function as an interrupt.

1212: static function '%s' referenced but not defined

The function has been declared as static and has been referenced elsewhere in the module, but there is no definition for the function present. Common causes include a misspelled function name in the function definition.

1213: initializer list expected

The symbol being initialized requires a brace-enclosed initializer list, but a single value initializer was found.

1214: constant expression expected in initializer

The initializer value for a statically allocated symbol must be a constant expression.

1216: string initializer used for non-character array object

A string literal initializer is only valid for initializing objects of type 'array of char' or type 'pointer to char' (either can be unsigned char as well).

1218: extraneous initializer values

The count of initializer values does not agree with the number of expected values based on the type of the object being initialized. There are too many values in the initializer list.

1219: integer constant expected

A constant expression of integral type was expected, but an expression of non-integral type or a non-constant expression was found.

1220: initializer detected in typedef declaration of '%s'

A typedef declaration cannot include initializers.

1221: empty initializer list detected

An initializer list cannot be empty. There must be one or more initializer values between the braces.

1222: "#pragma config" detected inside function body

The 'config' pragma is only available at file level scope.

1223: configuration setting '%s' has already been specified

The specified configuration setting has been specified either in a different #pragma config or previously in this #pragma config.

1224: configuration setting '%s' not recognized

The specified configuration setting is not recognized for the selected device. Make sure that the setting specified is all uppercase and spelled correctly. Use --help-config for information on the configuration settings available for the selected device.

1225: configuration value '%s' not recognized for configuration setting '%s'

The specified configuration value is not recognized for the selected device and configuration setting. Make sure that the value specified is all uppercase and spelled correctly. Use --help-config for information on the configuration settings and values available for the selected device.

1226: cannot specify both #pragma config and _CONFIG_DECL macro

Configuration settings can only be specified using either the #pragma config directive or the _CONFIG_DECL macro, preferably #pragma config.

1227: cannot specify #pragma config directive when compiling for generic device

The #pragma config directive is a processor-specific directive and requires that a specific processor be specified on the command line using the -p option.

1250: '%s' operand %s must be a literal

The specified operand for the opcode must be a literal value, not a symbol reference.

1251: '%s' operand count mismatch

The number of operands found for the specified opcode does not match the number of operands expected. Unlike the MPASM assembler, the MPLAB C1X in-line assembler expects all operands to be explicitly specified. There are no default values for operands such as the access bit or destination bit.

1252: invalid opcode '%s' detected for processor '%s'

The opcode specified is not valid for the target processor. Common causes include porting in-line assembly code from a processor with a different instruction set (e.g., PIC17CXX to PIC18CXX) and typographical errors in the spelling of the opcode.

1253: constant operand expected

Operands to in-line assembly opcodes must resolve to a constant expression, where a constant expression is defined as a literal constant or a statically allocated symbol reference optionally plus or minus an integer constant. Common causes include the use of a dynamically allocated symbol ('auto' local variables and parameters) as the operand to an in-line assembly opcode.

1300: stack frame too large

The size of the stack frame has exceeded the maximum addressable size. Commonly caused by too many local variables allocated as 'auto' storage class in a single function.

1301: parameter frame too large

The size of the parameter frame has exceeded the maximum addressable size. Commonly caused by too many parameters being passed to a single function.

1302: old style function declarations not supported

MPLAB C18 does not currently support the old K&R style function definitions. The in-line parameter type declarations recommended by the ANSI standard should be used instead.

1303: 'near' symbol defined in non-access qualified section

Statically allocated variables allocated into a non-access qualified section cannot be accessed via the access bit, and therefore defining them with the 'near' range qualifier would result in incorrect access to the location.

1304: illegal use of obsolete 'overlay' storage class for symbol '%s'

The overlay storage class is not supported in Extended mode. Also note that in Non-extended mode, the overlay storage class is valid only for local variables.

1500: unable to open file '%s'

The compiler was unable to open the named file. Common causes include misspelled filename and insufficient access rights.

1504: redefinition of '%s'

The same function name may not have multiple definitions.

1505: redeclaration of '%s'

The same variable name may not have multiple defining declarations.

1506: function '%s' cannot have 'overlay' storage class specifier

The 'overlay' storage class specifier may not be used with functions.

1507: variable '%s' of 'overlay' storage class cannot have 'near' qualifier

The compiler does not currently support variables of 'overlay' storage class in access ram.

1508: inconsistent linkage for %s

The identifier has been given both internal and external linkage.

1509: %s cannot have 'extern' storage class

The 'extern' storage class specifier may not be used with parameters.

1510: %s cannot have 'extern' storage class, block scope, and an initializer

The compiler does not support explicit initialization of block scope objects with 'extern' storage class.

1511: ran out of internal memory for temps

The compiler cannot support the allocation of any more temporary variables.

1512: redefinition of label '%s'

The same label may not have multiple definitions in the same function.

1513: redefinition of member '%s'

A structure or union may only have a single member with a given name.

1514: cast of a pointer to floating point is undefined

The requested cast is illegal. This error may be caused by omitting an array subscript on assignment.

1515: redefinition of case value %ld

A switch statement may only have a single case statement for a given value.

1516: array size must be greater than zero

The constant value given for the array size must be greater than zero.

D.2 WARNINGS

2001: non-near symbol '%s' declared in access section '%s'

Statically allocated variables declared into an access qualified section will always be placed by the linker into access data memory, and can therefore always be qualified with the 'near' range qualifier. Not specifying the 'near' range qualifier will not cause incorrect code, but may result in extraneous bank select instructions.

2002: unknown pragma '%s'

The compiler has encountered a pragma directive which is not recognized. As per ANSI/ISO requirements, the pragma is ignored. Common causes include misspelled pragma names.

2003: _CONFIG_DECL macro has been deprecated; please utilize #pragma config

The _CONFIG_DECL macro is considered obsolescent and is in the process of being phased out. It is being replaced with the #pragma config directive.

2025: default overlay locals is unsupported in Extended mode, -sco ignored

The overlay storage class is not supported in Extended mode.

2026: default static locals is unsupported in Extended mode, -scs ignored

The default storage class of static is not supported in Extended mode.

2027: default auto locals is redundant in Extended mode, -sca ignored

The default storage class for locals is always auto in Extended mode.

2028: default static locals is unsupported in Extended mode, -Ol ignored

The default storage class of static is not supported in Extended mode.

2029: default access RAM is unsupported in Extended mode, -Oa ignored

The default storage range of near is not supported in Extended mode.

2052: unexpected return value

A return of a value statement has been detected in a function declared to return no value. The return value will be ignored.

2053: return value expected

A return with no value has been detected in a function declared to return a value. The return value will be undefined.

2054: suspicious pointer conversion

A pointer has been used as an integer or an integer has been used as a pointer without an explicit cast.

2055: expression is always false

The control expression of a conditional statement evaluates to a constant false value.

2056: expression is always true

The control expression of a conditional statement evaluates to a constant true value.

2058: call of function without prototype

A function call has been made without an in-scope function prototype for the function being called. This can be un-safe, as no type-checking for the function arguments can be performed.

2059: unary minus of unsigned value

The unary minus operator is normally only applied to signed values.

2060: shift expression has no effect

Shifting a value by zero bits has no effect on the value of the expression.

2062: '->' operator expected, not '.'

A struct/union member access via a pointer to struct/union has been performed using the '.' operator.

2063: '.' operator expected, not '->'

A direct struct/union member access has been performed using the '->' operator.

2064: static function '%s' not defined

The function has been declared as static, but there is no definition for the function present. Common causes include a misspelled function name in the function definition.

2065: static function '%s' never referenced

The static function has been defined, but has not been referenced.

2066: type qualifier mismatch in assignment

Pointer assignment where the source and destination pointers point to objects of compatible type, but the source pointer points to an object which is 'const' or 'volatile' qualified and the destination pointer does not.

2068: obsolete use of implicit 'int' detected

The ANSI standard allows a variable to be declared without a base type being specified, e.g., "extern x;", in which case a base type of 'int' is implied. This usage is deprecated by the standard as obsolete, and therefore a diagnostic is issued to that effect.

2069: enumeration value exceeds maximum range

An enumeration value has been declared which is not expressible in a 'signed long' format and the enumeration tag has negative enumeration values. An 'unsigned long' representation will be used for the enumeration, but relative comparisons of those enumeration constants which have negative representations may not behave as expected.

2071: %s cannot have 'overlay' storage class; replacing with 'static'

Parameters with 'overlay' storage class are not permitted at this time. When the default local storage class is 'overlay', the 'static' storage class will be assigned to parameters.

2072: invalid storage class specifier for %s; ignoring

The storage class specifier used is not permitted for this declaration.

2073: null-terminated initializer string too long

The null-terminated initializer string cannot fit in the array object.

2100: obsolete use of 'overlay' for symbol '%s', processing as 'auto'

The overlay storage class is not supported in Extended mode. The declaration will be processed as if storage class 'auto' had been specified instead.

2101: obsolete use of 'static' storage for parameter '%s', treating as 'auto'

When compiling in Extended mode, MPLAB C18 requires all function parameters to be of automatic storage class. See the MPLAB C18 User's Guide for more information.

2102: near range specifier ignored for 'auto' variable '%s'

Automatic storage class variables are located on the stack, and so the 'near' range qualifier, to place the variable in access memory, does not apply.

D.3 MESSAGES

3000: test of floating point for equality detected

Testing two floating point values for equality will not always yield the desired results, as two expressions which are mathematically equivalent may evaluate to slightly different values when computed due to rounding error.

3002: comparison of a signed integer to an unsigned integer detected

Comparing a signed integer value to an unsigned integer value may yield unexpected results when the signed value is negative. To compare an unsigned integer to the binary equivalent representation of the signed value, the signed value should first be explicitly cast to the unsigned type of the same size.

NOTES:

Appendix E. Extended Mode

This appendix details the differences between the Non-extended and Extended modes. The differences include:

- Source Code Compatibility
  - Stack Frame Size
  - `static` Parameters
  - `overlay` Keyword
  - Inline Assembly
  - Predefined Macros
- Command-line Option Differences
- COFF File Differences

E.1 SOURCE CODE COMPATIBILITY

E.1.1 Stack Frame Size

When the compiler is operating in Extended mode, the total stack frame size (local variables, parameters and frame pointer preservation) is limited to 96 bytes per function. In Non-extended mode, for each function 120 bytes are available for locals and an additional 120 bytes are available for parameters.

E.1.2 `static` Parameters

`static` parameters are not supported when the compiler is operating in the Extended mode. A warning diagnostic will be issued when the compiler is operating in the Extended mode and a `static` parameter is seen. In addition, the compiler will act as if the code explicitly specified an `auto` parameter. The parameter will now be stored on the stack instead of being allocated globally. Since the total size of stack frame is limited to 96 bytes per function, the application may result in a "stack frame too large" diagnostic being issued that does not occur when the compiler is operating in Non-extended mode. To resolve this, the function will need to be modified to take fewer parameters.

E.1.3 `overlay` Keyword

The `overlay` keyword is not supported when the compiler is operating in the Extended mode. A warning diagnostic will be issued when the compiler is operating in the Extended mode and the `overlay` keyword is seen. In addition, the compiler will act as if the code explicitly specified the `auto` keyword. Similar to `static` parameters, the `overlay` local variable will now be stored on the stack instead of being allocated globally. Since the total size of the stack frame is limited to 96 bytes per function, the application may result in a "stack frame too large" diagnostic being issued that does not occur when the compiler is operating in the Non-extended mode. To resolve this, the function will need to be modified to contain fewer `auto` local variables. One way to do this is to change the `overlay` variables to `static`.

Note: Overlay sections (`#pragma overlay`) are supported by the compiler regardless of the mode in which it is operating.

E.1.4 Inline Assembly

When operating in Extended mode, the compiler will accept the extended instructions in inline assembly – ADDFSR, ADDULNK, CALLW, MOVSF, MOVSS, PUSHL, SUBFSR and SUBULNK; however, when operating in Non-extended mode, the compiler will issue an error when it encounters an extended instruction in inline assembly.

In addition, when operating in Extended mode, the compiler will not recognize the bracketed syntax used by the MPASM assembler for indicating the indexed with literal offset addressing (e.g., CLRF [2]). Instead, the compiler will recognize the indexed with literal offset addressing in inline assembly when the f operand is less than or equal to 0x5F and the access bit operand (a) is set to zero (e.g., CLRF 2, 0). This same instruction will be interpreted as referencing access RAM when the compiler is operating in Non-extended mode.

E.1.5 Predefined Macros

The predefined macros can be utilized in source code to make the source code compatible regardless of the mode in which the compiler is operating. The __EXTENDED18__ predefined macro will be the constant 1 when compiling for Extended mode; whereas, the __TRADITIONAL18__ predefined macro will be the constant 1 when compiling for Non-extended mode.

Here are some examples of specific instances where this may be useful:

1. Using the predefined macros to use static parameters in Non-extended mode and auto parameters in Extended mode:

```
#ifdef __EXTENDED18__
    #define SCLASS auto
#else
    #define SCLASS static
#endif

void foo (SCLASS int bar);
```

2. Using the predefined macros to utilize the overlay keyword in Non-extended mode and the auto keyword in Extended mode:

```
#ifdef __EXTENDED18__
    #define SCLASS auto
#else
    #define SCLASS overlay
#endif

void foo (void)
{
    SCLASS int bar;

    ...
}
```

3. Using the predefined macros to use only Non-extended mode instructions in inline assembly in Non-extended mode and to use Extended mode instructions in inline assembly in Extended mode:

```
_asm
#ifdef __EXTENDED18__
    PUSHL 5
#else
    MOVLW 5
    MOVWF POSTINC1, 0
#endif
    ...
    MOVF POSTDEC1, 1, 0
_endasm
```

E.2 COMMAND-LINE OPTION DIFFERENCES

The following command-line options are not supported when the compiler is operating in the Extended mode:

- Default Local Storage Class (`-scs/-sco/-sca`)
  When operating in the Extended mode, the compiler only supports default `auto` locals.
- Default Data in Access Memory (`-Oa+/-Oa-`)
  Since the amount of access RAM on an Extended mode device is limited, the compiler does not support data being placed in access RAM by default when operating in the Extended mode.

E.3 COFF FILE DIFFERENCES

E.3.1 Generic Processor

The processor type (`proc_type`) specified in the COFF file's optional file header when compiling for the generic processor (`-p18cxx`) will be set to PIC18F4620 when the compiler is operating in the Extended mode and will be set to PIC18C452 when the compiler is operating in the Non-extended mode.

E.3.2 File Header's `f_flags` Field

When operating in Extended mode, the COFF file that is generated will have the `F_EXTENDED18` bit of the file header's `f_flags` set. This bit is not set when the compiler is operating in the Non-extended mode.

NOTES:

Glossary

A

Absolute Section

A section with a fixed address that cannot be changed by the linker.

Access Memory

Special general purpose registers on the PIC18 PICmicro microcontrollers that allow access regardless of the setting of the Bank Select Register (`BSR`).

Address

The code that identifies where a piece of information is stored in memory.

Anonymous Structure

An unnamed object.

ANSI

American National Standards Institute

Assembler

A language tool that translates assembly source code into machine code.

Assembly

A symbolic language that describes the binary machine code in a readable form.

Assigned Section

A section that has been assigned to a target memory block in the linker command file.

Asynchronously

Multiple events that do not occur at the same time. This is generally used to refer to interrupts that may occur at any time during processor execution.

B

Binary

The base two numbering system that uses the digits 0-1. The rightmost digit counts ones, the next counts multiples of 2, then $2^2 = 4$, etc.

C

Central Processing Unit

The part of a device that is responsible for fetching the correct instruction for execution, decoding that instruction, and then executing that instruction. When necessary, it works in conjunction with the Arithmetic Logic Unit (ALU) to complete the execution of the instruction. It controls the program memory address bus, the data memory address bus and accesses to the stack.

Compiler

A program that translates a source file written in a high-level language into machine code.

Conditional Compilation

The act of compiling a program fragment only if a certain constant expression, specified by a preprocessor directive, is true.

CPU

Central Processing Unit

E

Endianness

The ordering of bytes in a multi-byte object.

Error File

A file containing the diagnostics generated by the MPLAB C18 compiler.

Extended Mode

In Extended mode, the compiler will utilize the extended instructions (i.e., `ADDFSR`, `ADDULNK`, `CALLW`, `MOVSF`, `MOVSS`, `PUSHL`, `SUBFSR` and `SUBULNK`) and the indexed with literal offset addressing.

F

Fatal Error

An error that will halt compilation immediately. No further messages will be produced.

Frame Pointer

A pointer that references the location on the stack that separates the stack-based arguments from the stack-based local variables.

Free-standing

An implementation that accepts any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (ANSI '89 standard clause 7) is confined to the contents of the standard headers `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>` and `<stdint.h>`.

H

Hexadecimal

The base 16 numbering system that uses the digits 0-9 plus the letters A-F (or a-f). The digits A-F represent decimal values of 10 to 15. The rightmost digit counts ones, the next counts multiples of 16, then 16^2 = 256, etc.

High-level Language

A language for writing programs that is further removed from the processor than assembly.

I

ICD

In-Circuit Debugger

ICE

In-Circuit Emulator

IDE

Integrated Development Environment

IEEE

Institute of Electrical and Electronics Engineers

Interrupt

A signal to the CPU that suspends the execution of a running application and transfers control to an ISR so that the event may be processed. Upon completion of the ISR, normal execution of the application resumes.

Interrupt Service Routine

A function that handles an interrupt.

ISO

International Organization for Standardization

ISR

Interrupt Service Routine

L

Latency

The time between when an event occurs and the response to it.

Librarian

A program that creates and manipulates libraries.

Library

A collection of relocatable object modules.

Linker

A program that combines object files and libraries to create executable code.

Little Endian

Within a given object, the least significant byte is stored at lower addresses.

M

Memory Model

A description that specifies the size of pointers that point to program memory.

Microcontroller

A highly integrated chip that contains a CPU, RAM, some form of ROM, I/O ports and timers.

MPASM Assembler

Microchip Technology's relocatable macro assembler for PICmicro microcontroller families.

MPLIB Object Librarian

Microchip Technology's librarian for PICmicro microcontroller families.

MPLINK Object Linker

Microchip Technology's linker for PICmicro microcontroller families.

N

Non-extended Mode

In Non-extended mode, the compiler will not utilize the extended instructions nor the indexed with literal offset addressing.

O

Object File

A file containing object code. It may be immediately executable or it may require linking with other object code files, (e.g. libraries), to produce a complete executable program.

Object Code

The machine code generated by an assembler or compiler.

Octal

The base 8 number system that only uses the digits 0-7. The rightmost digit counts ones, the next digit counts multiples of 8, then $8^2 = 64$, etc.

P

Pragma

A directive that has meaning to a specific compiler.

R

RAM

Random Access Memory

Random Access Memory

A memory device in which information can be accessed in any order.

Read Only Memory

Memory hardware that allows fast access to permanently stored data but prevents addition to or modification of the data.

ROM

Read Only Memory

Recursive

Self-referential (e.g., a function that calls itself).

Reentrant

A function that may have multiple, simultaneously active instances. This may happen due to either direct or indirect recursion or through execution during interrupt processing.

Relocatable

An object whose address has not been assigned to a fixed memory location.

Run-time Model

Set of assumptions under which the compiler operates.

S

Section

A portion of an application located at a specific address of memory.

Section Attribute

A characteristic ascribed to a section (e.g., an `access` section).

Special Function Register

Registers that control I/O processor functions, I/O status, timers or other modes or peripherals.

Storage Class

Determines the lifetime of the memory associated with the identified object.

Storage Qualifier

Indicates special properties of the objects being declared (e.g., `const`).

V

Vector

The memory locations that an application will jump to when either a reset or interrupt occurs.

NOTES:

Index

WORLDWIDE SALES AND SERVICE

AMERICAS

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

Atlanta
Alpharetta, GA
Tel: 770-640-0034
Fax: 770-640-0307

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

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

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

Detroit
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

San Jose
Mountain View, CA
Tel: 650-215-1444
Fax: 650-961-0286

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

ASIA/PACIFIC

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: 86-28-8676-6200
Fax: 86-28-8676-6599

China - Fuzhou
Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

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

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 - Shunde
Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

China - Qingdao
Tel: 86-532-502-7355
Fax: 86-532-502-7205

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-2229-0061
Fax: 91-80-2229-0062

India - New Delhi
Tel: 91-11-5160-8631
Fax: 91-11-5160-8632

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

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

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

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

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

Taiwan - Hsinchu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

EUROPE

Austria - Weis
Tel: 43-7242-2244-399
Fax: 43-7242-2244-393

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

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

Germany - Ismaning
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

England - Berkshire
Tel: 44-118-921-5869
Fax: 44-118-921-5820

03/01/05