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

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

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

Contact us

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

MPLAB® C Compiler for PIC24 MCUs and dsPIC® DSCs User's Guide

ISBN: 978-1-61341-294-7

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2009

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, 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

Preface

NOTICE TO CUSTOMERS

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

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

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

INTRODUCTION

This chapter contains general information that will be useful to know before using the MPLAB C Compiler for PIC24 MCUs and dsPIC® DSCs. Items discussed include:

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

DOCUMENT LAYOUT

This document describes how to use GNU language tools to write code for 16-bit applications. The document layout is as follows:

- **Chapter 1: Compiler Overview** – describes the compiler, development tools and feature set.
- **Chapter 2: Differences between 16-Bit Device C and ANSI C** – describes the differences between the C language supported by the compiler syntax and the standard ANSI-89 C.
- **Chapter 3: Using the Compiler on the Command Line** – describes how to use the compiler from the command line.
- **Chapter 4: Run Time Environment** – describes the compiler run-time model, including information on sections, initialization, memory models, the software stack and much more.
- **Chapter 5: Data Types** – describes the compiler integer, floating point and pointer data types.
- **Chapter 6: Additional C Pointers** – describes additional C pointers available.
- **Chapter 7: Device Support Files** – describes the compiler header and register definition files, as well as how to use with SFRs.
- **Chapter 8: Interrupts** – describes how to use interrupts.
- **Chapter 9: Mixing Assembly Language and C Modules** – provides guidelines to using the compiler with 16-bit assembly language modules.
- **Appendix A: Implementation-Defined Behavior** – details compiler-specific parameters described as implementation-defined in the ANSI standard.
- **Appendix B: Built-in Functions –** lists the built-in functions of the C compiler.
- **Appendix C: Diagnostics** – lists error and warning messages generated by the compiler.
- **Appendix D: MPLAB C Compiler for PIC18 MCUs vs. 16-Bit Devices** – highlights the differences between the PIC18 MCU C compiler and the 16-bit C compiler.
- **Appendix E: Deprecated Features –** details features that are considered obsolete.
- **Appendix F: ASCII Character Set** – contains the ASCII character set.
- **Appendix G: GNU Free Documentation License** – usage license for the Free Software Foundation.

CONVENTIONS USED IN THIS GUIDE

The following conventions may appear in this documentation:

DOCUMENTATION CONVENTIONS

Description	Represents	Examples
Arial font:		
Italic characters	Referenced books	*MPLAB® IDE User's Guide*
	Emphasized text	...is the *only* compiler...
Initial caps	A window	the Output window
	A dialog	the Settings dialog
	A menu selection	select Enable Programmer
Quotes	A field name in a window or dialog	"Save project before build"
Underlined, italic text with right angle bracket	A menu path	<u>*File>Save*</u>
Bold characters	A dialog button	Click **OK**
	A tab	Click the **Power** tab
Text in angle brackets < >	A key on the keyboard	Press <Enter>, <F1>
Courier font:		
Plain Courier	Sample source code	`#define START`
	Filenames	`autoexec.bat`
	File paths	`c:\mcc18\h`
	Keywords	`_asm, _endasm, static`
	Command-line options	`-Opa+, -Opa-`
	Bit values	`0, 1`
	Constants	`0xFF, 'A'`
Italic Courier	A variable argument	*`file`*`.o`, where *`file`* can be any valid filename
Square brackets [ ]	Optional arguments	`mpasmwin [options]` *`file`* `[options]`
Curly brackets and pipe character: { | }	Choice of mutually exclusive arguments; an OR selection	`errorlevel {0|1}`
Ellipses...	Replaces repeated text	`var_name [, var_name...]`
	Represents code supplied by user	`void main (void)` `{ ...` `}`
Sidebar Text		
STD	Standard edition only. This feature supported only in the standard edition of the software, i.e., not supported in standard evaluation (after 60 days) or lite editions.	`-mpa` option
DD	Device Dependent. This feature is not supported on all devices. Devices supported will be listed in the title or text.	`xmemory` attribute

RECOMMENDED READING

This documentation describes how to use the MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs. Other useful documents are listed below. The following Microchip documents are available and recommended as supplemental reference resources.

Readme Files

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

16-Bit Language Tools Getting Started (DS70094)

A guide to installing and working with the Microchip language tools for 16-bit devices. Examples using the 16-bit simulator SIM30 (a component of MPLAB SIM) are provided.

MPLAB® Assembler, Linker and Utilities for PIC24 MCUs and dsPIC® DSCs User's Guide (DS51317)

A guide to using the 16-bit assembler, object linker, object archiver/librarian and various utilities.

16-Bit Language Tools Libraries (DS51456)

A descriptive listing of libraries available for Microchip 16-bit devices. This includes standard (including math) libraries and C compiler built-in functions. DSP and 16-bit peripheral libraries are described in Readme files provided with each peripheral library type.

Device-Specific Documentation

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

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

C Standards Information

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

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

C Reference Manuals

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

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

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

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

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

THE MICROCHIP WEB SITE

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

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

DEVELOPMENT SYSTEMS CUSTOMER CHANGE NOTIFICATION SERVICE

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

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

The Development Systems product group categories are:

- **Compilers** – The latest information on Microchip C compilers, assemblers, linkers and other language tools. These include all MPLAB C compilers; all MPLAB assemblers (including MPASM™ assembler); all MPLAB linkers (including MPLINK™ object linker); and all MPLAB librarians (including MPLIB™ object librarian).
- **Emulators** – The latest information on Microchip in-circuit emulators. These include the MPLAB REAL ICE™ and MPLAB ICE 2000 in-circuit emulators
- **In-Circuit Debuggers** – The latest information on Microchip in-circuit debuggers. These include the MPLAB ICD 2 and 3 in-circuit debuggers and PICkit™ 2 and 3 debug express.
- **MPLAB® IDE** – The latest information on Microchip MPLAB IDE, the Windows® Integrated Development Environment for development systems tools. This list is focused on the MPLAB IDE, MPLAB IDE Project Manager, MPLAB Editor and MPLAB SIM simulator, as well as general editing and debugging features.
- **Programmers** – The latest information on Microchip programmers. These include the device (production) programmers MPLAB REAL ICE in-circuit emulator, MPLAB ICD 3 in-circuit debugger, MPLAB PM3, and PRO MATE II and development (nonproduction) programmers MPLAB ICD 2 in-circuit debugger, PICSTART® Plus and PICkit 1, 2 and 3.

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

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

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

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

Chapter 1. Compiler Overview

1.1 INTRODUCTION

The dsPIC® family of Digital Signal Controllers (dsPIC30F and dsPIC33F DSCs) combines the high performance required in DSP applications with standard microcontroller features needed for embedded applications. PIC24 MCUs are identical to the dsPIC DSCs with the exception that they do not have the digital signal controller module or that subset of instructions. They are a subset and are high-performance microcontrollers intended for applications that do not require the power of the DSC capabilities.

All of these devices are fully supported by a complete set of software development tools, including an optimizing C compiler, an assembler, a linker and an archiver/ librarian.

This chapter provides an overview of these tools and introduces the features of the optimizing C compiler, including how it works with the assembler and linker. The assembler and linker are discussed in detail in the "*MPLAB® Assembler, Linker and Utilities for PIC24 MCUs and dsPIC® DSCs User's Guide"* (DS51317).

1.2 HIGHLIGHTS

Items discussed in this chapter are:

- Compiler Description and Documentation
- Compiler and Other Development Tools
- Compiler Feature Set

1.3 COMPILER DESCRIPTION AND DOCUMENTATION

There are three Microchip compilers that support various Microchip 16-bit devices. Also, each one of these compilers comes in different editions, which support different levels of optimization.

MPLAB® C Compiler for		Device Support	Edition Support
1	PIC24 MCUs and dsPIC® DSCs	All 16-bit devices	Std, Std Eval
2	dsPIC DSCs	dsPIC30F/33F DSCs	Std, Std Eval, Lite
3	PIC24 MCUs	PIC24F/H MCUs	Std, Std Eval, Lite

Each compiler is an ANSI x3.159-1989-compliant, optimizing C compiler. Each compiler is a Windows® console application that provides a platform for developing C code. Each compiler is a port of the GCC compiler from the Free Software Foundation.

The first and second compilers include language extensions for dsPIC DSC embedded-control applications.

1.3.1 Compiler Editions

Each of the three compilers in **Section 1.3 “Compiler Description and Documentation”** come in one or more of the following editions:

- Standard (Purchased Compiler) – All optimization levels enabled.
- Standard Evaluation (Free) – All optimization levels enabled for 60 days, but then reverts to optimization level 1 only.
- Lite (Free) – Optimization level 1 only.

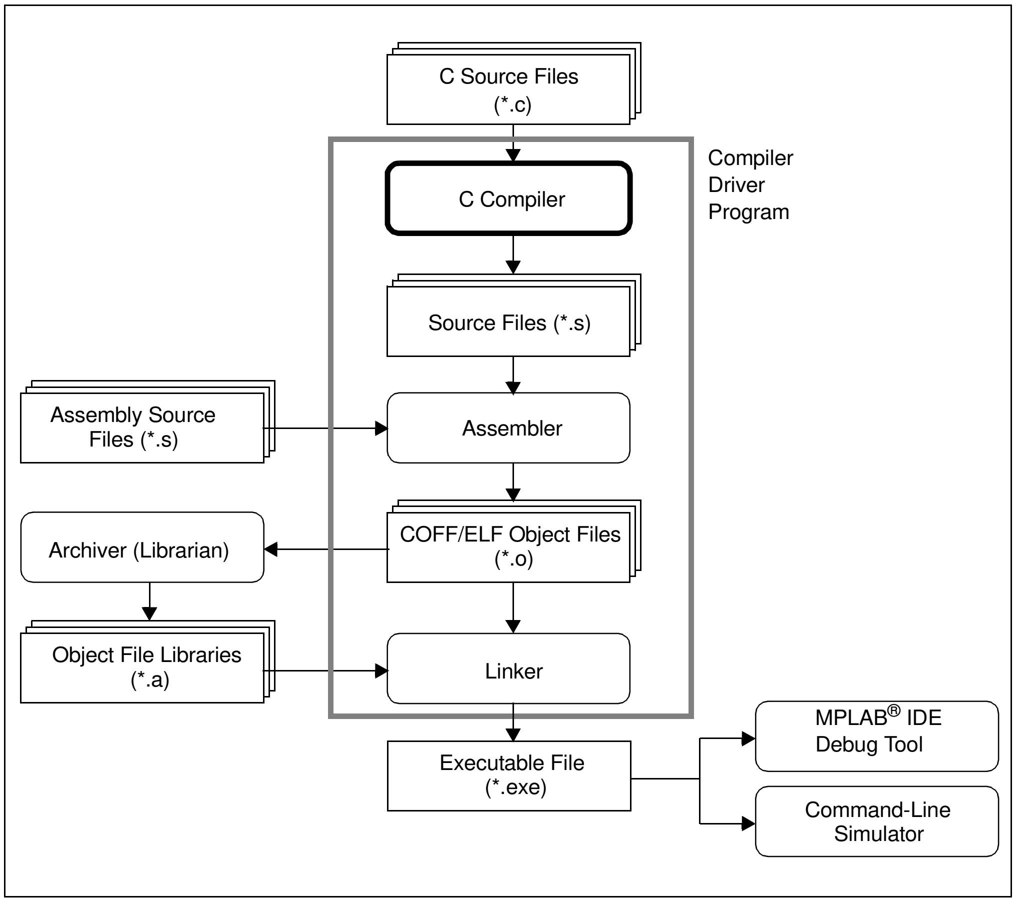
1.3.2 Compiler Documented in this Manual

This manual describes the standard edition of the Standard (purchased) compiler, since the Standard Evaluation and Lite compilers are subsets of the first. Features that are unique to specific devices, and therefore specific compilers, are noted with “DD” text the column (see the **Preface**) and text identifying the devices to which the information applies.

1.4 COMPILER AND OTHER DEVELOPMENT TOOLS

The MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs compiles C source files, producing assembly language files. These compiler-generated files are assembled and linked with other object files and libraries to produce the final application program in executable COFF or ELF file format. The COFF or ELF file can be loaded into the MPLAB IDE, where it can be tested and debugged, or the conversion utility can be used to convert the COFF or ELF file to Intel® hex format, suitable for loading into the command-line simulator or a device programmer. See Figure 1-1 for an overview of the software development data flow.

FIGURE 1-1: SOFTWARE DEVELOPMENT TOOLS DATA FLOW



1.5 COMPILER FEATURE SET

The compiler is a full-featured, optimizing compiler that translates standard ANSI C programs into 16-bit device assembly language source. The compiler also supports many command-line options and language extensions that allow full access to the 16-bit device hardware capabilities, and affords fine control of the compiler code generator. This section describes key features of the compiler.

1.5.1 ANSI C Standard

The compiler is a fully validated compiler that conforms to the ANSI C standard as defined by the ANSI specification and described in Kernighan and Ritchie's *The C Programming Language* (second edition). The ANSI standard includes extensions to the original C definition that are now standard features of the language. These extensions enhance portability and offer increased capability.

1.5.2 Optimization

The compiler uses a set of sophisticated optimization passes that employ many advanced techniques for generating efficient, compact code from C source. The optimization passes include high-level optimizations that are applicable to any C code, as well as 16-bit device-specific optimizations that take advantage of the particular features of the device architecture.

1.5.3 ANSI Standard Library Support

The compiler is distributed with a complete ANSI C standard library. All library functions have been validated, and conform to the ANSI C library standard. The library includes functions for string manipulation, dynamic memory allocation, data conversion, timekeeping and math functions (trigonometric, exponential and hyperbolic). The standard I/O functions for file handling are also included, and, as distributed, they support full access to the host file system using the command-line simulator. The fully functional source code for the low-level file I/O functions is provided in the compiler distribution, and may be used as a starting point for applications that require this capability.

1.5.4 Flexible Memory Models

The compiler supports both large and small code and data models. The small code model takes advantage of more efficient forms of call and branch instructions, while the small data model supports the use of compact instructions for accessing data in SFR space.

The compiler supports two models for accessing constant data. The "constants in data" model uses data memory, which is initialized by the run-time library. The "constants in code" model uses program memory, which is accessed through the Program Space Visibility (PSV) window.

1.5.5 Compiler Driver

The compiler includes a powerful command-line driver program. Using the driver program, application programs can be compiled, assembled and linked in a single step (see Figure 1-1).

Chapter 2. Differences Between 16-Bit Device C and ANSI C

2.1 INTRODUCTION

This section discusses the differences between the C language supported by MPLAB C Compiler for PIC24 MCUs and dsPIC® DSCs (formerly MPLAB C30) syntax and the 1989 standard ANSI C.

2.2 HIGHLIGHTS

Items discussed in this chapter are:

- Keyword Differences
- Statement Differences
- Expression Differences

2.3 KEYWORD DIFFERENCES

This section describes the keyword differences between plain ANSI C and the C accepted by the 16-bit device compiler. The new keywords are part of the base GCC implementation, and the discussion in this section is based on the standard GCC documentation, tailored for the specific syntax and semantics of the 16-bit compiler port of GCC.

- Specifying Attributes of Variables
- Specifying Attributes of Functions
- Inline Functions
- Variables in Specified Registers
- Complex Numbers
- Double-Word Integers
- Referring to a Type with `typeof`

2.3.1 Specifying Attributes of Variables

The compiler keyword `__attribute__` allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. The following attributes are currently supported for variables:

- `address (addr)`
- `aligned (alignment)`
- `boot`
- `deprecated`
- `eds`
- `fillupper`
- `far`
- `mode (mode)`
- `near`
- `noload`
- `page`
- `packed`
- `persistent`
- `reverse (alignment)`
- `section ("section-name")`
- `secure`
- `sfr (address)`
- `space (space)`
- `transparent_union`
- `unordered`
- `unused`
- `weak`

You may also specify attributes with __ (double underscore) preceding and following each keyword (e.g., `__aligned__` instead of `aligned`). This allows you to use them in header files without being concerned about a possible macro of the same name.

To specify multiple attributes, separate them by commas within the double parentheses, for example:

```
 __attribute__ ((aligned (16), packed)).
```

Note: It is important to use variable attributes consistently throughout a project. For example, if a variable is defined in file A with the `far` attribute, and declared `extern` in file B without `far`, then a link error may result.

`address (`*`addr`*`)`

The `address` attribute specifies an absolute address for the variable. This attribute can be used in conjunction with a `section` attribute. This can be used to start a group of variables at a specific address:

```
int foo __attribute__((section("mysection"),address(0x900)));
int bar __attribute__((section("mysection")));
int baz __attribute__((section("mysection")));
```

A variable with the `address` attribute cannot be placed into the `auto_psv` space (see the `space()` attribute or the `-mconst-in-code` option); attempts to do so will cause a warning and the compiler will place the variable into the PSV space. If the variable is to be placed into a PSV section, the address should be a program memory address.

```
int var __attribute__ ((address(0x800)));
```

aligned (*alignment*)

This attribute specifies a minimum alignment for the variable, measured in bytes. The alignment must be a power of two. For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable `x` on a 16-byte boundary. On the dsPIC DSC device, this could be used in conjunction with an `asm` expression to access DSP instructions and addressing modes that require aligned operands.

As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable to the maximum useful alignment for the dsPIC DSC device. For example, you could write:

```
short array[3] __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an aligned attribute specification, the compiler automatically sets the alignment for the declared variable to the largest alignment for any data type on the target machine – which in the case of the dsPIC DSC device is two bytes (one word).

The `aligned` attribute can only increase the alignment; but you can decrease it by specifying `packed` (see below). The `aligned` attribute conflicts with the `reverse` attribute. It is an error condition to specify both.

The `aligned` attribute can be combined with the `section` attribute. This will allow the alignment to take place in a named section. By default, when no section is specified, the compiler will generate a unique section for the variable. This will provide the linker with the best opportunity for satisfying the alignment restriction without using internal padding that may happen if other definitions appear within the same aligned section.

boot

This attribute can be used to define protected variables in Boot Segment (BS) RAM:

```
int __attribute__((boot)) boot_dat[16];
```

Variables defined in BS RAM will not be initialized on startup. Therefore all variables in BS RAM must be initialized using inline code. A diagnostic will be reported if initial values are specified on a `boot` variable.

An example of initialization is as follows:

```
int  __attribute__((boot)) time = 0; /* not supported */
int  __attribute__((boot)) time2;
void __attribute__((boot)) foo()
{
  time2 = 55; /* initial value must be assigned explicitly */
}
```

`deprecated`

The `deprecated` attribute causes the declaration to which it is attached to be specially recognized by the compiler. When a `deprecated` function or variable is used, the compiler will emit a warning.

A `deprecated` definition is still defined and, therefore, present in any object file. For example, compiling the following file:

```
int __attribute__((__deprecated__)) i;
int main() {
  return i;
}
```

will produce the warning:

```
deprecated.c:4: warning: `i' is deprecated (declared
 at deprecated.c:1)
```

`i` is still defined in the resulting object file in the normal way.

`eds`

In the attribute context the `eds`, for extended data space, attribute indicates to the compiler that the variable will may be allocated anywhere within data memory. Variables with this attribute will likely also need the `__eds__` type qualifier (see **Chapter 6. "Additional C Pointer Types"**) in order for the compiler to properly generate the correct access sequence. Note that the `__eds__` qualifier and the `eds` attribute are closely related, but not identical. On some devices, `eds` may need to be specified when allocating variables into certain memory spaces such as `space(ymemory)` or `space(dma)` as this memory may only exist in the extended data space.

`fillupper`

This attribute can be used to specify the upper byte of a variable stored into a `space(prog)` section.

For example:

```
int foo[26] __attribute__((space(prog),fillupper(0x23))) = { 0xDEAD };
```

will fill the upper bytes of array `foo` with 0x23, instead of 0x00. `foo[0]` will still be initialized to 0xDEAD.

The command line option `-mfillupper=0x23` will perform the same function.

`far`

The `far` attribute tells the compiler that the variable will not necessarily be allocated in near (first 8 KB) data space, (i.e., the variable can be located anywhere in data memory between 0x0000 and 0x7FFF).

`mode (`*`mode`*`)`

This attribute specifies the data type for the declaration as whichever type corresponds to the mode *`mode`*. This in effect lets you request an integer or floating point type according to its width. Valid values for *`mode`* are as follows:

Mode	Width	Compiler Type
QI	8 bits	`char`
HI	16 bits	`int`
SI	32 bits	`long`
DI	64 bits	`long long`
SF	32 bits	`float`
DF	64 bits	`long double`

This attribute is useful for writing code that is portable across all supported compiler targets. For example, the following function adds two 32-bit signed integers and returns a 32-bit signed integer result:

```
typedef int __attribute__((__mode__(SI))) int32;
int32
add32(int32 a, int32 b)
  {
        return(a+b);
  }
```

You may also specify a mode of `byte` or `__byte__` to indicate the mode corresponding to a one-byte integer, `word` or `__word__` for the mode of a one-word integer, and `pointer` or `__pointer__` for the mode used to represent pointers.

`near`

The `near` attribute tells the compiler that the variable is allocated in near data space (the first 8 KB of data memory). Such variables can sometimes be accessed more efficiently than variables not allocated (or not known to be allocated) in near data space.

```
int num __attribute__ ((near));
```

`noload`

The `noload` attribute indicates that space should be allocated for the variable, but that initial values should not be loaded. This attribute could be useful if an application is designed to load a variable into memory at run time, such as from a serial EEPROM.

```
int table1[50] __attribute__ ((noload)) = { 0 };
```

`page`

The `page` attribute places variable definitions into a specific page of memory. The page size depends on the type of memory selected by a `space` attribute. Objects residing in RAM will be constrained to a 32K page while objects residing in Flash will be constrained to a 64K page (upper byte not included).

```
unsigned int var[10] __attribute__ ((space(auto_psv)));
```

The `space(auto_psv)` or `space(psv)` attribute will use a single memory page by default.

```
__eds__ unsigned int var[10] __attribute__ ((space(eds), page));
```

When dealing with `space(eds)`, please refer to **Chapter 6. “Additional C Pointer Types”** for more information.

`packed`

The `packed` attribute specifies that a structure member should have the smallest possible alignment unless you specify a larger value with the `aligned` attribute.

Here is a structure in which the member `x` is packed, so that it immediately follows `a`, with no padding for alignment:

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

Note: The device architecture requires that words be aligned on even byte boundaries, so care must be taken when using the packed attribute to avoid run-time addressing errors.

persistent

The `persistent` attribute specifies that the variable should not be initialized or cleared at startup. A variable with the persistent attribute could be used to store state information that will remain valid after a device reset.

```
int last_mode __attribute__ ((persistent));
```

Persistent data is not normally initialized by the C run-time. However, from a cold-restart, persistent data may not have any meaningful value. This code example shows how to safely initialize such data:

```
#include "p24Fxxxx.h"

int last_mode __attribute__((persistent));

int main()
{
    if ((RCONbits.POR == 0) &&
        (RCONbits.BOR == 0)) {
      /* last_mode is valid */
    } else {
      /* initialize persistent data */
      last_mode = 0;
    }
}
```

reverse (*alignment*)

The `reverse` attribute specifies a minimum alignment for the ending address of a variable, plus one. The alignment is specified in bytes and must be a power of two. Reverse-aligned variables can be used for decrementing modulo buffers in dsPIC DSC assembly language. This attribute could be useful if an application defines variables in C that will be accessed from assembly language.

```
int buf1[128] __attribute__ ((reverse(256)));
```

The `reverse` attribute conflicts with the `aligned` and `section` attributes. An attempt to name a section for a reverse-aligned variable will be ignored with a warning. It is an error condition to specify both `reverse` and `aligned` for the same variable. A variable with the `reverse` attribute cannot be placed into the `auto_psv` space (see the `space()` attribute or the `-mconst-in-code` option); attempts to do so will cause a warning and the compiler will place the variable into the PSV space.

section (*"section-name"*)

By default, the compiler places the objects it generates in sections such as `.data` and `.bss`. The `section` attribute allows you to override this behavior by specifying that a variable (or function) lives in a particular section.

```
struct a { int i[32]; };
struct a buf __attribute__((section("userdata"))) = {{0}};
```

secure

This attribute can be used to define protected variables in Secure Segment (SS) RAM:

```
int __attribute__((secure)) secure_dat[16];
```

Variables defined in SS RAM will not be initialized on startup. Therefore all variables in SS RAM must be initialized using inline code. A diagnostic will be reported if initial values are specified on a `secure` variable.

String literals can be assigned to secure variables using inline code, but they require extra processing by the compiler. For example:

```
char  *msg __attribute__((secure)) = "Hello!\n"; /* not supported */
char *msg2 __attribute__((secure));
void __attribute__((secure)) foo2()
{
  *msg2 = "Goodbye..\n"; /* value assigned explicitly */
}
```

In this case, storage must be allocated for the string literal in a memory space which is accessible to the enclosing secure function. The compiler will allocate the string in a psv constant section designated for the secure segment.

`sfr (`*`address`*`)`

The `sfr` attribute tells the compiler that the variable is an SFR and also specifies the run-time address of the variable, using the *`address`* parameter.

```
extern volatile int __attribute__ ((sfr(0x200)))u1mod;
```

The use of the extern specifier is required in order to not produce an error.

Note: By convention, the `sfr` attribute is used only in processor header files. To define a general user variable at a specific address use the `address` attribute in conjunction with `near` or `far` to specify the correct addressing mode.

`space (`*`space`*`)`

Normally, the compiler allocates variables in general data space. The `space` attribute can be used to direct the compiler to allocate a variable in specific memory spaces. Memory spaces are discussed further in **Section 4.5 "Memory Spaces"**. The following arguments to the space attribute are accepted:

`data`

Allocate the variable in general data space. Variables in general data space can be accessed using ordinary C statements. This is the default allocation.

`eds`

Allocate the variable in the extended data space. For devices that do not have extended data space, this is equivalent to `space(data)`. Variables in `space(eds)` will generally require special handling to access. Refer to **Chapter 6. "Additional C Pointer Types"** for more information.
`space(eds)` has been deprecated in favour of the `eds` attribute.

DD **`xmemory`** - dsPIC30F/33F DSCs only

Allocate the variable in X data space. Variables in X data space can be accessed using ordinary C statements. An example of `xmemory` space allocation is:

```
int x[32] __attribute__ ((space(xmemory)));
```

DD **`ymemory`** - dsPIC30F/33F DSCs only

Allocate the variable in Y data space. Variables in Y data space can be accessed using ordinary C statements. An example of `ymemory` space allocation is:

```
int y[32] __attribute__ ((space(ymemory)));
```

`prog`

Allocate the variable in program space, in a section designated for executable code. Variables in program space can not be accessed using ordinary C statements. They must be explicitly accessed by the programmer, usually using table-access inline assembly instructions, or using the program space visibility window.

`auto_psv`

Allocate the variable in program space, in a compiler-managed section designated for automatic program space visibility window access. Variables in `auto_psv` space can be read (but not written) using ordinary C statements, and are subject to a maximum of 32K total space allocated. When specifying `space(auto_psv)`, it is not possible to assign a section name using the `section` attribute; any section name will be ignored with a warning. A variable in the `auto_psv` space cannot be placed at a specific address or given a reverse alignment.

Note: Variables placed in the `auto_psv` section are not loaded into data memory at startup. This attribute may be useful for reducing RAM usage.

DD **`dma`** - PIC24H MCUs, dsPIC33F DSCs only

Allocate the variable in DMA memory. Variables in DMA memory can be accessed using ordinary C statements and by the DMA peripheral. `__builtin_dmaoffset()` (see **Appendix B. "Built-in Functions"**) can be used to find the correct offset for configuring the DMA peripheral.

```
#include <p24Hxxxx.h>
unsigned int BufferA[8] __attribute__((space(dma)));
unsigned int BufferB[8] __attribute__((space(dma)));

int main()
{
  DMA1STA = __builtin_dmaoffset(BufferA);
  DMA1STB = __builtin_dmaoffset(BufferB);
  /* ... */
}
```

`psv`

Allocate the variable in program space, in a section designated for program space visibility window access. The linker will locate the section so that the entire variable can be accessed using a single setting of the PSVPAG register. Variables in PSV space are not managed by the compiler and can not be accessed using ordinary C statements. They must be explicitly accessed by the programmer, usually using table-access inline assembly instructions, or using the program space visibility window.

DD **`eedata`** - PIC24F, dsPIC30F/33F DSCs only

Allocate the variable in EEData space. Variables in EEData space can not be accessed using ordinary C statements. They must be explicitly accessed by the programmer, usually using table-access inline assembly instructions, or using the program space visibility window.

`pmp`

Allocate the variable in off chip memory associated with the PMP peripheral. For complete details please see **Section 6.3 "PMP Pointers"**.

external

Allocate the variable in a user defined memory space. For complete details please see **Section 6.4 "External Pointers"**.

transparent_union

This attribute, attached to a function parameter which is a `union`, means that the corresponding argument may have the type of any union member, but the argument is passed as if its type were that of the first union member. The argument is passed to the function using the calling conventions of the first member of the transparent union, not the calling conventions of the union itself. All members of the union must have the same machine representation; this is necessary for this argument passing to work properly.

unordered

The `unordered` attribute indicates that the placement of this variable may move relative to other variables within the current C source file.

```
const int __attribute__ ((unordered)) i;
```

unused

This attribute, attached to a variable, means that the variable is meant to be possibly unused. The compiler will not produce an unused variable warning for this variable.

weak

The `weak` attribute causes the declaration to be emitted as a weak symbol. A weak symbol may be superseded by a global definition. When `weak` is applied to a reference to an external symbol, the symbol is not required for linking. For example:

```
extern int __attribute__((__weak__)) s;
int foo() {
  if (&s) return s;
  return 0; /* possibly some other value */
}
```

In the above program, if `s` is not defined by some other module, the program will still link but `s` will not be given an address. The conditional verifies that `s` has been defined (and returns its value if it has). Otherwise '`0`' is returned. There are many uses for this feature, mostly to provide generic code that can link with an optional library.

The `weak` attribute may be applied to functions as well as variables:

```
extern int __attribute__((__weak__)) compress_data(void *buf);
int process(void *buf) {
  if (compress_data) {
    if (compress_data(buf) == -1) /* error */
  }
  /* process buf */
}
```

In the above code, the function `compress_data` will be used only if it is linked in from some other module. Deciding whether or not to use the feature becomes a link-time decision, not a compile time decision.