MPLAB® XC16 C 编译器
用户指南

© 2015 Microchip Technology Inc.

请注意以下有关 Microchip 器件代码保护功能的要点：

- Microchip 的产品均达到 Microchip 数据手册中所述的技术指标。
- Microchip 确信：在正常使用的情况下， Microchip 系列产品是当今市场上同类产品中最安全的产品之一。
- 目前，仍存在着恶意、甚至是非法破坏代码保护功能的行为。就我们所知，所有这些行为都不是以 Microchip 数据手册中规定的操作规范来使用 Microchip 产品的。这样做的人极可能侵犯了知识产权。
- Microchip 愿与那些注重代码完整性的客户合作。
- Microchip 或任何其他半导体厂商均无法保证其代码的安全性。代码保护并不意味着我们保证产品是 “牢不可破”的。

代码保护功能处于持续发展中。 Microchip 承诺将不断改进产品的代码保护功能。任何试图破坏 Microchip 代码保护功能的行为均可视为违反了《数字器件千年版权法案 （Digital Millennium Copyright Act）》。如果这种行为导致他人在未经授权的情况下，能访问您的软件或其他受版权保护的成果，您有权依据该法案提起诉讼，从而制止这种行为。

提供本文档的中文版本仅为了便于理解。请勿忽视文档中包含的英文部分，因为其中提供了有关 Microchip 产品性能和使用情况的有用信息。Microchip Technology Inc. 及其分公司和相关公司、各级主管与员工及事务代理机构对译文中可能存在的任何差错不承担任何责任。建议参考 Microchip Technology Inc. 的英文原版文档。

本出版物中所述的器件应用信息及其他类似内容仅为您提供便利，它们可能由更新之信息所替代。确保应用符合技术规范，是您自身应负的责任。**Microchip 对这些信息不作任何明示或暗示、书面或口头、法定或其他形式的声明或担保，包括但不限于针对其使用情况、质量、性能、适销性或特定用途的适用性的声明或担保。** Microchip 对因这些信息及使用这些信息而引起的后果不承担任何责任。如果将 Microchip 器件用于生命维持和 / 或生命安全应用，一切风险由买方自负。买方同意在由此引发任何一切伤害、索赔、诉讼或费用时，会维护和保障 Microchip 免于承担法律责任，并加以赔偿。除非另外声明，在 Microchip 知识产权保护下，不得暗中或以其他方式转让任何许可证。

商标

Microchip 的名称和徽标组合、 Microchip 徽标、 dsPIC、 FlashFlex、 flexPWR、 JukeBlox、 KEELOQ、 KEELOQ 徽标、 Kleer、 LANCheck、 MediaLB、 MOST、 MOST 徽标、 MPLAB、 OptoLyzer、 PIC、 PICSTART、 PIC32 徽标、 RightTouch、 SpyNIC、 SST、 SST 徽标、 SuperFlash 及 UNI/O 均为 Microchip Technology Inc. 在美国和其他国家或地区的注册商标。

The Embedded Control Solutions Company 和 mTouch 为 Microchip Technology Inc. 在美国的注册商标。

Analog-for-the-Digital Age、 BodyCom、 chipKIT、 chipKIT 徽标、 CodeGuard、 dsPICDEM、 dsPICDEM.net、 ECAN、 In-Circuit Serial Programming、 ICSP、 Inter-Chip Connectivity、 KleerNet、 KleerNet 徽标、 MiWi、 motorBench、 MPASM、 MPF、 MPLAB Certified 徽标、 MPLIB、 MPLINK、 MultiTRAK、 NetDetach、 Omniscient Code Generation、 PICDEM、 PICDEM.net、 PICkit、 PICtail、 RightTouch 徽标、 REAL ICE、 SQI、 Serial Quad I/O、 Total Endurance、 TSHARC、 USBCheck、 VariSense、 ViewSpan、 WiperLock、 Wireless DNA 和 ZENA 均为 Microchip Technology Inc. 在美国和其他国家或地区的商标。

SQTP 为 Microchip Technology Inc. 在美国的服务标记。

Silicon Storage Technology 为 Microchip Technology Inc. 在除美国外的国家或地区的注册商标。

GestIC 为 Microchip Technology Inc. 的子公司 Microchip Technology Germany II GmbH & Co. & KG 在除美国外的国家或地区的注册商标。

在此提及的所有其他商标均为各持有公司所有。

© 2015, Microchip Technology Inc. 版权所有。

ISBN：978-1-63277-453-8

QUALITY MANAGEMENT SYSTEM CERTIFIED BY DNV

ISO/TS 16949

Microchip 位于美国亚利桑那州 Chandler 和 Tempe 与位于俄勒冈州 Gresham 的全球总部、设计和晶圆生产厂及位于美国加利福尼亚州和印度的设计中心均通过了 ISO/TS-16949:2009 认证。 Microchip 的 PIC® MCU 与 dsPIC® DSC、KEELOQ® 跳码器件、串行 EEPROM、单片机外设、非易失性存储器和模拟产品严格遵守公司的质量体系流程。此外， Microchip 在开发系统的设计和生产方面的质量体系也已通过了 ISO 9001:2000 认证。

© 2015 Microchip Technology Inc.

目录

前言 9

第 1 章 编译器概述

1.1 简介 13
1.2 器件说明 13
1.3 编译器说明和文档 13
1.4 编译器和其他开发工具 15

第 2 章 通用 C 接口

2.1 简介 17
2.2 背景知识——对于可移植代码的追求 17
2.3 使用 CCI 20
2.4 ANSI 标准细化 21
2.5 ANSI 标准扩展 29
2.6 编译器特性 43

第 3 章 操作指南

3.1 简介 45
3.2 安装和激活编译器 45
3.3 调用编译器 47
3.4 编写源代码 49
3.5 让应用程序执行所需的操作 60
3.6 了解编译过程 62
3.7 修复无法工作的代码 67

第 4 章 XC16 工具链和 MPLAB X IDE

4.1 简介 69
4.2 MPLAB X IDE 和工具安装 69
4.3 MPLAB X IDE 设置 70
4.4 MPLAB X IDE 项目 71
4.5 项目设置 73
4.6 项目示例 85

第 5 章 编译器命令行驱动程序

5.1 简介 89
5.2 调用编译器 90
5.3 编译序列 93
5.4 运行时文件 98
5.5 编译器输出 99
5.6 编译器消息 100
5.7 驱动程序选项说明 101
5.8 MPLAB X IDE 工具链或 MPLAB IDE 工具套件等效选项 124

第 6 章 与器件相关的特性
6.1 简介 125
6.2 器件支持 125
6.3 器件头文件 125
6.4 堆栈 127
6.5 配置位访问 128
6.6 使用 SFR 129
6.7 位反转寻址和模寻址 131
第 7 章 MPLAB XC16 和 ANSI C 之间的差别
7.1 不符合 ANSI C 标准的方面 133
7.2 ANSI C 标准的扩展 133
7.3 实现定义的行为 133
第 8 章 支持的数据类型和变量
8.1 简介 135
8.2 标识符 135
8.3 整型数据类型 136
8.4 浮点型数据类型 138
8.5 定点型数据类型 139
8.6 结构和联合 140
8.7 指针类型 142
8.8 复数数据类型 144
8.9 字面常量类型和格式 145
8.10 标准类型限定符 147
8.11 特定于编译器的类型限定符 148
8.12 变量属性 151
第 9 章 定点算术支持
9.1 简介 161
9.2 使能定点算术支持 161
9.3 数据类型 162
9.4 舍入 163
9.5 被零除 163
9.6 外部定义 163
9.7 混合使用 C 代码和汇编代码 164
第 10 章 存储器分配和访问
10.1 简介 165
10.2 地址空间 166
10.3 数据存储空间中的变量 167
10.4 程序空间中的变量 174
10.5 并行主端口访问 179
10.6 外部存储器访问 181
10.7 扩展数据空间访问 185
10.8 dataflash 访问 186
10.9 双分区存储器访问 186

© 2015 Microchip Technology Inc.

10.10 压缩存储在闪存中的数据 186
10.11 变量到寄存器的分配 188
10.12 EEPROM 中的变量 188
10.13 动态存储器分配 190
10.14 存储模型 191

第 11 章 操作符和语句

11.1 简介 193
11.2 内置函数 193
11.3 整型提升 193

第 12 章 寄存器使用

12.1 简介 195
12.2 寄存器变量 195
12.3 更改寄存器内容 196

第 13 章 函数

13.1 简介 197
13.2 编写函数 197
13.3 函数长度限制 206
13.4 函数代码的分配 206
13.5 更改默认的函数分配 206
13.6 内联函数 207
13.7 存储模型 208
13.8 函数调用约定 210

第 14 章 中断

14.1 简介 213
14.2 中断操作 214
14.3 编写中断服务程序 215
14.4 指定中断向量 217
14.5 中断服务程序现场保存 219
14.6 嵌套中断 219
14.7 允许 / 禁止中断 220
14.8 ISR 注意事项 221

第 15 章 main、运行时启动和复位

15.1 简介 227
15.2 main 函数 227
15.3 运行时启动和初始化 227

第 16 章 混合使用 C 代码和汇编代码

16.1 简介 229
16.2 混合使用汇编语言与 C 变量及函数 229
16.3 使用行内汇编语言 232
16.4 预定义的汇编宏 237

第 17 章 库程序

17.1 简介 239

第 18 章 优化
18.1 简介 241
18.2 设置优化级别 241
18.3 优化功能汇总 242
第 19 章 预处理
19.1 简介 243
19.2 C 语言注释 243
19.3 预处理伪指令 244
19.4 预定义的宏名称 245
19.5 pragma 伪指令与属性 247
第 20 章 链接程序
20.1 简介 249
20.2 默认存储空间 249
20.3 替换库符号 251
20.4 链接器定义的符号 251
20.5 默认链接描述文件 252
附录 A 实现定义的行为
A.1 简介 253
A.2 翻译 254
A.3 环境 254
A.4 标识符 255
A.5 字符 255
A.6 整型 256
A.7 浮点型 257
A.8 数组和指针 257
A.9 寄存器 258
A.10 结构、联合、枚举和位域 258
A.11 限定符 258
A.12 声明符 258
A.13 语句 259
A.14 预处理伪指令 259
A.15 库函数 260
A.16 信号 261
A.17 流和文件 262
A.18 临时文件 263
A.19 errno 263
A.20 存储器 263
A.21 abort 263
A.22 exit 263
A.23 getenv 263
A.24 system 263
A.25 strerror 264

© 2015 Microchip Technology Inc.

附录 B 嵌入式编译器兼容性模式
B.1 简介 265
B.2 在兼容性模式下编译 266
B.3 语法兼容性 266
B.4 数据类型 267
B.5 操作符 267
B.6 扩展关键字 268
B.7 固有函数 269
B.8 pragma 伪指令 269
附录 C 诊断
C.1 简介 271
C.2 错误 271
C.3 警告 290
附录 D GNU 自由文档许可证
D.1 前导 311
D.2 适用范围和定义 311
D.3 逐字复制 313
D.4 大量复制 313
D.5 修改 314
D.6 组合文档 315
D.7 文档的合集 315
D.8 独立作品的聚集 315
D.9 翻译 316
D.10 终止 316
D.11 本许可证的未来修订版 316
D.12 重新许可 317
附录 E ASCII 字符集 319
附录 F 弃用功能
F.1 简介 321
F.2 预定义的常量 321
F.3 指定寄存器中的变量 322
F.4 更改非自动变量分配 324
附录 G 内置函数
G.1 简介 325
G.2 内置函数说明 327
附录 H 文档版本历史
支持 358
术语表 363
索引 383
全球销售及服务网点 395

注：

© 2015 Microchip Technology Inc.

前言

客户须知

所有文档均会过时，本文档也不例外。 Microchip 的工具和文档将不断演变以满足客户的需求，因此实际使用中有些对话框和 / 或工具说明可能与本文档所述之内容有所不同。

欲了解开发工具的最新信息，请参考 MPLAB® X IDE 帮助。从 Help （帮助）菜单选择 Topics （主题）或 Help Contents （帮助内容），打开现有帮助文件列表。

关于最新的 PDF，请访问我公司网站 （http://www.microchip.com）。文档通过 “DSXXXXXXXXA”进行标识，其中 “XXXXXXXX”是文档编号，“A”是文档的版本。文档编号位于每页底部的页码之前。

简介

以下几节介绍了 MPLAB XC16 C 编译器的文档和支持信息：

- 文档编排
- 本指南使用的约定
- 推荐读物

文档编排

本文档介绍如何使用 GNU 语言工具来编写 16 位单片机 / 数字信号控制器应用程序代码。内容安排如下：

- **第 1 章 “编译器概述”** ——介绍编译器、开发工具和功能集。
- **第 2 章 “通用 C 接口”** ——介绍可用于增强 MPLAB XC 编译器之间代码可移植性的通用 C 接口。
- **第 3 章 “操作指南”** ——列出了一些具体操作方面的问题和简要说明，以及指向手册中相关章节的链接。
- **第 4 章 “XC16 工具链和 MPLAB X IDE”** ——说明关于如何通过 MPLAB X IDE 设置和使用编译器及相关工具的基础知识。
- **第 5 章 “编译器命令行驱动程序”** ——介绍如何从命令行中使用编译器。
- **第 6 章 “与器件相关的特性”** ——介绍编译器头文件和寄存器定义文件，以及如何用于 SFR。
- **第 7 章 “MPLAB XC16 和 ANSI C 之间的差别”** ——介绍编译器语法支持的 C 语言与标准 ANSI-89 C 之间的差别。
- **第8章 “支持的数据类型和变量”** ——介绍编译器的整型、浮点型和指针数据类型。

- **第 9 章 “定点算术支持”**——说明编译器中的定点算术支持。
- **第 10 章 “存储器分配和访问”**——介绍编译器运行时模型，包括关于段、初始化、存储模型、软件堆栈和更多方面的信息。
- **第 11 章 “操作符和语句”**——介绍操作符和语句。
- **第 12 章 “寄存器使用”**——说明如何访问和使用 SFR。
- **第 13 章 “函数”**——详细介绍可用的函数。
- **第 14 章 “中断”**——介绍如何使用中断。
- **第 15 章 “main、运行时启动和复位”**——介绍 C 代码的重要元素。
- **第 16 章 “混合使用 C 代码和汇编代码”**——提供关于编译器与 16 位汇编语言模块配合使用的指导。
- **第 17 章 “库程序”**——说明如何使用库。
- **第 18 章 “优化”**——介绍优化选项。
- **第 19 章 “预处理”**——详细介绍预处理操作。
- **第 20 章 “链接程序”**——说明链接如何工作。
- **附录 A “实现定义的行为”**——详细介绍在 ANSI 标准中描述为实现定义的、特定于编译器的参数。
- **附录 B “嵌入式编译器兼容性模式”**——详细介绍编译器的兼容性模式。
- **附录 C “诊断”**——列出由编译器生成的错误和警告消息。
- **附录 D “GNU 自由文档许可证”**——自由软件基金会的使用许可证。
- **附录 E “ASCII 字符集”**——ASCII 字符集表。
- **附录 F “弃用功能”——**详细介绍被视为过时的功能。
- **附录 G “内置函数”**——列出 C 编译器的内置函数。
- **附录 H “文档版本历史”**——关于本文档的先前和当前版本的信息。

© 2015 Microchip Technology Inc.

本指南使用的约定

本指南采用以下文档约定：

文档约定

说明	表示	示例
Arial 字体：		
斜体字	参考书目	*MPLAB® IDE User's Guide*
	需强调的文字	……为*仅有*的编译器……
首字母大写	窗口	Output 窗口
	对话框	Settings 对话框
	菜单选择	选择 Enable Programmer
引用	窗口或对话框中的字段名	“Save project before build”
带右尖括号且有下划线的斜体文字	菜单路径	*File>Save*
粗体字	对话框按钮	单击 **OK**
	选项卡	单击 **Power** 选项卡
N‘Rnnnn	verilog 格式的数字，其中 N 为总位数， R 为基数， n 为其中一位。	4‘b0010， 2‘hF1
尖括号 < > 括起的文字	键盘上的按键	按下 <Enter>，<F1>
Courier New 字体：		
常规 Courier New	源代码示例	`#define START`
	文件名	`autoexec.bat`
	文件路径	`c:\mcc18\h`
	关键字	`_asm， _endasm， static`
	命令行选项	`-Opa+，-Opa-`
	二进制位值	`0， 1`
	常量	`0xFF， ‘A’`
斜体 Courier New	可变参数	***file***`.o`，其中 ***file*** 可以是任一有效文件名
方括号 []	可选参数	`mcc18 [`选项`] `***file*** `[`选项`]`
花括号和竖线：{|}	选择互斥参数； “或”选择	`errorlevel {0|1}`
侧边栏文字		
DD	取决于器件。 该功能并非所有器件都支持。 支持的器件会在标题或文字中列出。	`xmemory` 属性

推荐读物

本文档介绍了如何使用 MPLAB XC16 C 编译器。下面列出了其他有用的文档。以下 Microchip 文档均已提供，并建议读者作为补充参考资料。

发行说明 （自述文件）

关于 Microchip 工具的最新信息，请阅读软件附带的相关发行说明 （HTML 文件）。

《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）

关于使用 16 位汇编器、目标链接器、目标归档器 / 库管理器和各种实用程序的指南。

《16 位语言工具函数库》（DS51456D_CN）

该文档说明了 Microchip 16 位器件可使用的库。包括标准 （包括数学）库以及 C 编译器内置函数。DSP 和 16 位外设库在随每种外设库类型所提供的发行说明中叙述。

针对器件的文档

Microchip 网站上提供了许多描述 16 位器件功能和特性的文档，其中包含：

- 具体器件以及器件系列的数据手册
- 器件系列的参考手册
- 程序员参考手册

C 标准方面的信息

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

此标准规定了用 C 语言编写程序的格式，并对 C 程序进行了解释。其目的是提高 C 程序在多种计算机系统上的可移植性、可靠性、可维护性及执行效率。

C 语言参考手册

Harbison, Samuel P. and Steele, Guy L., *C A Reference Manual*，第四版。 Prentice-Hall, Englewood Cliffs, N.J. 07632.

Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*，第二版。 Prentice Hall, Englewood Cliffs, N.J. 07632.

Kochan, Steven G., *Programming In ANSI C*，修订版。 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*，第一版。 LLH Technology Publishing, Eagle Rock, Virginia 24085.

© 2015 Microchip Technology Inc.

第 1 章 编译器概述

1.1 简介

以下主题对 MPLAB XC16 C 编译器进行了定义和介绍：

- 器件说明
- 编译器说明和文档
- 编译器和其他开发工具

1.2 器件说明

MPLAB XC16 C 编译器完全支持所有 Microchip 16 位器件：

- dsPIC® 系列数字信号控制器结合了数字信号处理器 （Digital Signal Processor，DSP）应用中所需的高性能和嵌入式应用所需的标准单片机 （MCU）功能。
- PIC24 系列 MCU 与 dsPIC DSC 基本上是相同的，只是它们没有数字信号控制器模块或 DSP 指令子集。PIC24 系列 MCU 是针对不需要 DSC 功能的应用的高性能 MCU。

1.3 编译器说明和文档

MPLAB XC16 C 编译器是一款全功能的优化编译器，可将标准 ANSI C 程序转换为 16 位器件汇编语言源代码。该编译器还支持多种命令行选项和语言扩展，以充分利用 16 位器件的硬件功能，同时满足编译器代码生成器较高的控制要求。

该编译器是自由软件基金会的 GNU 编译器套件（GNU Compiler Collection，GCC）的移植版本。

该编译器可用于几种流行的操作系统，包括 32 位和 64 位 Windows®、 Linux 和 Apple OS X。

该编译器可以在 3 种工作模式下运行：免费 （Free） 、标准 （Standard）或专业（PRO）。标准和专业工作模式是许可模式，需要使用激活密钥和互联网连接来使能它们。无许可证的客户可以使用免费模式。基本的编译器操作、支持的器件和可用的存储器在所有模式之间是相同的。这些模式之间的差别仅在于编译器采用的优化级别（见**第 18 章 “优化”**）。

以下几节介绍了编译器的关键功能。

1.3.1 ANSI C 标准

该编译器是经过充分验证的编译器，符合 ANSI 规范（ANSI x3.159-1989）定义、并在 Kernighan 和 Ritchie 的 *The C Programming Language*（第二版）中介绍的 ANSI C 标准。 ANSI 标准包含了对原始 C 定义的一些扩展，现在它们已成为该语言的标准特性。这些扩展可以增强可移植性，并增强了功能和性能。此外，还包含用于 dsPIC DSC 嵌入式控制应用程序的语言扩展。

1.3.2 优化

该编译器使用了一组复杂的优化轮次，这些优化轮次采用许多先进的技术，基于 C 源代码生成高效紧凑的代码。优化轮次包括适用于所有 C 代码的高级优化，以及可利用器件架构特定功能的特定于 16 位器件的优化。

关于优化的更多信息，请参见**第 18 章 “优化”**。

1.3.3 ANSI 标准库支持

该编译器附带了一个完整的 ANSI C 标准库。所有库函数已经过验证且符合 ANSI C 库标准。该库包括字符串操作、动态存储器分配、数据转换、计时等函数以及数学函数（三角函数、指数函数和双曲函数）。用于文件处理的标准 I/O 函数亦包括在内，并且在分发时，它们支持使用命令行模拟器对主机文件系统进行完全访问。编译器分发版中提供了低级文件 I/O 函数全功能的源代码，可用作需要该功能的应用程序的起始点。

1.3.4 灵活的存储模型

该编译器支持大代码和小代码模型以及大数据和小数据模型。小代码模型采用效率更高的形式的调用和转移指令，而小数据模型支持使用紧凑指令来访问 SFR 空间中的数据。

该编译器支持两种访问常量数据的模型。“常量位于数据中”模型使用数据存储器，由运行时库初始化。“常量位于代码中”模型使用程序存储器，通过程序空间可视性（Program Space Visibility， PSV）窗口访问。

1.3.5 属性和限定符

编译器关键字 `__attribute__` 可用于指定变量、结构位域或函数的特殊属性。该关键字后跟包含在双括号内的属性说明，如下所示：

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

在其他编译器中，限定符用于创建限定类型：

```
persistent int last_mode;
```

MPLAB XC16 C 编译器具有一些非标准限定符，如**第 8.11 节“特定于编译器的类型限定符”**所述。

一般来说，限定符指示应如何访问某个对象，而属性指示将对象存放在何处。属性还具有许多其他用途。

1.3.6 编译器驱动程序

该编译器包含了功能强大的命令行驱动程序。使用该驱动程序，可以一步完成应用程序的编译、汇编和链接。

1.3.7 文档

MPLAB X IDE 和 MPLAB IDE v8.xx 及更高版本均支持该编译器。本文档中仅介绍了 MPLAB X IDE。

特定器件独有、因而特定编译器独有的功能会在章节旁注有 “DD”图标，以及指示信息所适用的特定器件的文字 （见**前言**）。

© 2015 Microchip Technology Inc.

1.4 编译器和其他开发工具

该编译器可与许多其他 Microchip 工具配合工作，包括：

- MPLAB XC16 汇编器和链接器——请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。
- MPLAB X IDE 和 MPLAB IDE v8.xx
- MPLAB SIM 模拟器和 MPLAB X 模拟器
- 命令行 MDB 模拟器——请参见 《Microchip 调试器 （MDB）用户指南》（DS50002102C_CN），
  它位于：
  <MPLAB X IDE Installation Directory>docs
- 所有 Microchip 调试工具和编程器
- 支持 16 位器件的演示板和入门工具包

注：

© 2015 Microchip Technology Inc.

第 2 章 通用 C 接口

2.1 简介

通用 C 接口（Common C Interface，CCI）可用于所有 MPLAB XC C 编译器，旨在用于增强这些编译器之间的代码可移植性。例如，符合 CCI 标准的代码可以更容易地从使用 MPLAB XC8 C 编译器的 PIC18 MCU 移植到使用 MPLAB XC16 C 编译器的 PIC24 MCU。

CCI 假定您的源代码已经符合 ANSI 标准。如果希望使用 CCI，您需要自行负责编写符合标准的代码。原有项目需要进行移植，以达到符合标准的要求。此外，还必须设置一个编译器选项，以确保在编译项目时编译器的操作与接口一致。

《MPLAB XC16 C 编译器用户指南》的这一章讨论了以下主题：

- 背景知识——对于可移植代码的追求
- 使用 CCI
- ANSI 标准细化
- ANSI 标准扩展
- 编译器特性

2.2 背景知识——对于可移植代码的追求

所有编程人员都希望编写可移植的源代码。

可移植性意味着相同的源代码可以在不同执行环境中编译和运行，而不仅仅是编写时所针对的环境。很少有代码可以实现百分百的可移植性，但代码对变化的容忍度越高，让它在新环境中运行所需的时间和工作就越少。

嵌入式工程师通常认为代码可移植性就是能够跨不同的目标器件，但这只是可移植性的一部分。相同的代码可能针对相同的目标器件、但使用不同的编译器进行编译。这些编译器之间的差异可能导致代码在编译时或运行时发生失败，所以也必须考虑到这一点。

您可以只针对一种目标器件编写代码，只使用一种品牌的编译器，但如果对于编译器的操作没有任何规定，简单更新一下编译器版本就可能会改变您的代码的行为。

代码必须具有跨目标器件、工具和时间的可移植性，才具有真正的灵活性。

很显然，这种可移植性无法由编程人员单独实现，因为不同编译器供应商的产品会基于不同的技术、实现不同的功能和代码语法，并且编译器供应商还会改善其产品的工作方式。许多很棒的编译器优化破坏了许多可信的项目。

C 语言标准的制定确保可以对变更进行管理，使代码具有更高的可移植性。美国国家标准学会（American National Standards Institute，ANSI）发布了许多领域的标准，包括编程语言。 ANSI C 标准是 C 编程语言普遍采用的标准。

2.2.1 ANSI 标准

ANSI C 标准必须调和两个对立的目标：编译器供应商可以自由地增加新器件支持和改进代码生成，而编程人员可以确信自己的源代码可正常工作。如果可以同时满足这两个目标，源代码就可以获得可移植性。

该标准以一组规则的形式实现，它们不仅详细规定了符合标准的 C 程序必须遵循的语法，而且详细规定了解释 C 程序的语义规则。因此，为了使编译器符合标准，必须确保符合标准的 C 程序按标准中的描述工作。

标准描述了*实现*、工具集和运行代码的运行时环境。如果其中任何一方面发生变化，例如，您针对不同的目标器件进行编译并在其上运行，或者如果您更新了用于编译的编译器版本，则说明您在使用不同的实现。

标准使用*行为*这一术语来表示程序的外观或操作。它与如何对程序进行编码无关。

由于标准试图实现一些可以被理解为相互冲突的目标，所以一些规定显得有点含糊不清。例如，标准规定 `int` 类型必须能够至少容纳一个 16 位值，但它并未直接规定 `int` 的实际大小；此外，对有符号整型进行右移的操作在不同实现上会产生不同的结果；但是，这些不同的结果仍然符合 ANSI C 标准。

如果标准太过严格，器件架构可能就无法让编译器符合标准。[1] 但如果标准太过薄弱，编程人员会在不同编译器和架构中看到差异巨大的结果，标准就会失去效力。

标准将行为未完全定义的源代码归类为几组，它们包括以下行为：

实现定义的行为

这是未规定的行为，每种实现用文档说明它是如何选择的。

未规定的行为

标准提供了两种或更多的可能性，但并未进一步要求在任何特定实例中选择哪种可能性。

未定义的行为

这是标准未做任何要求的行为。

严格遵从标准的代码不会产生依赖于任何未规定、未定义或实现定义的行为的输出。`int` 类型的大小（前面用作示例）可以归类为实现定义的行为。即，`int` 的大小将根据使用哪种编译器、如何使用该编译器，以及所面向的器件来定义。

所有 MPLAB XC 编译器都符合针对编程语言的 ANSI X3.159-1989 标准（无法支持递归的 MPLAB XC8 编译器除外，如脚注中所述）。它通常被称为 C89 标准。此外，还支持更新标准 C99 中的一些功能。

1. 例如，中档 PIC® 单片机没有数据堆栈。由于用于此类器件的编译器无法实现递归，它（严格说来）无法达到符合 ANSI C 标准的要求。该示例说明了标准对于中档器件和工具过于严格的情形。

© 2015 Microchip Technology Inc.

对于独立式实现 （或对于通常所称的嵌入式应用程序），标准允许对语言的非标准扩展，但显然并不强制要求如何规定它们或它们如何工作。在应用程序的工作与器件硬件密切相关时，编程人员需要能够通过一种方式来指定器件设置和中断，以及利用通常很复杂的小型器件存储器架构。标准无法以一致的方式提供这一点。

虽然 ANSI C 标准让编程人员和编译器供应商可以相互理解，但编程人员需要考虑其工具的实现定义的行为，以及需要使用 C 语言非标准扩展的可能性。这两种情况都会影响代码的可移植性。

2.2.2 通用 C 接口

通用 C 接口 （CCI）为 ANSI C 标准提供了补充，使编程人员在使用任意 MPLAB XC C 编译器时，可以在所有 Microchip 器件上更容易地实现一致的结果。

它提供了以下改进，在设计上全都充分考虑到了可移植性。

对 ANSI C 标准进行细化

CCI 文档规定了一些代码的具体行为，其操作在 ANSI C 标准下属于实现定义的行为。例如， CCI 详细定义了有符号整型进行右移操作的结果。请注意，许多实现定义的条目与器件特性密切关联， CCI 并未定义它们，例如 `int` 的大小。

对非标准扩展使用一致的语法

CCI 非标准扩展大多使用具有统一语法的关键字实现。它们可代替本机编译器实现的关键字、宏和属性。关键字的解释可能会因每种编译器而不同，关键字的任何参数也可能是特定于器件的。

编码准则

CCI 提供了关于如何编写代码的建议，以便代码可以移植到其他器件或编译器。虽然您可以选择不遵从建议，但代码将不符合 CCI 标准。

2.3 使用 CCI

CCI 通过细化实现定义的行为和标准化语言扩展的语法来支持更高的可移植性。

CCI 是您选择遵循并付诸实施的标准，因而它与新项目相关，虽然您可以选择修改现有的项目，使它们符合标准。

要使项目符合 CCI 标准，必须执行以下操作。

使能 CCI

在项目中选择 MPLAB IDE 窗口小部件 *Use CCI Syntax*（使用 CCI 语法）或使用等效的命令行选项。

在每个模块中包含 <xc.h>

一些 CCI 特性只有在编译器发现该头文件时才会使能。

确保符合 ANSI 标准

不符合 ANSI C 标准的代码不会符合 CCI。

遵从 CCI 对于 ANSI 的细化定义

对于一些 ANSI 实现定义的行为， CCI 进行了明确定义。

使用语言的 CCI 扩展

使用 CCI 扩展，而不是本机语言扩展。

下面几节详细介绍了与 CCI 关联的一些具体条目。这些条目可分为以下几类：细化标准、处理 ANSI C 标准扩展，以及其他一些编译器选项和用法。随这些条目还提供了一些准则。

如果任何实现定义的行为或任何非标准扩展未在本文档中讨论，则说明它不是 CCI 的一部分。例如，GCC case 语句范围、标号地址和 24 位 `short long` 类型不是 CCI 的一部分。使用这些特性的程序不符合 CCI 标准。如果使用了非 CCI 特性并使能 CCI，编译器可能会发出警告或错误进行提示。

© 2015 Microchip Technology Inc.

2.4 ANSI 标准细化

以下主题介绍了 CCI 如何细化 ANSI C 标准中概述的实现定义的行为。

2.4.1 源文件编码

在 CCI 下，源文件必须使用 7 位 ASCII 集合中的字符编写。代码行可以使用*换行符*（“\n”）或*回车符*（“\r”）紧跟*换行符*来终止。在字符常量或字符串字面值中可以使用转义字符来表示不处于基本字符集中的扩展字符。

2.4.1.1 示例

以下显示了使用转义字符来定义一个字符串常量。

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

2.4.1.2 差别

所有编译器都使用了该字符集。

2.4.1.3 移植到 CCI

无需任何操作。

2.4.2 `main` 的原型

`main()` 函数的原型为：

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

2.4.2.1 示例

以下显示了一个说明如何定义 `main()` 的示例：

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

2.4.2.2 差别

8 位编译器对于该函数使用 `void` 返回类型。

2.4.2.3 移植到 CCI

每个程序都具有一个 `main()` 函数定义。确认先前针对 8 位目标器件编译的所有项目中的 `main()` 的返回类型。

2.4.3 头文件规范

使用目录分隔符的头文件规范不符合 CCI 标准。

2.4.3.1 示例

以下示例显示了两条符合标准的包含伪指令。

```
#include <usb_main.h>
#include "global.h"
```

2.4.3.2 差别

在所有编译器的先前版本中，都允许使用目录分隔符的头文件规范。当使用 Windows 样式的分隔符 “\”，而代码在其他主机操作系统下编译时，将会出现兼容性问题。在 CCI 下，不应使用任何目录分隔符。

2.4.3.3 移植到 CCI

在头文件规范中使用目录分隔符的所有 `#include` 伪指令都需要进行更改。删除伪指令中除头文件名之外的所有内容。将目录路径添加到编译器的包含搜索路径或 MPLAB IDE 等效选项中。这将强制编译器搜索使用该选项指定的目录。

例如，以下代码：

```
#include <inc/lcd.h>
```

应更改为：

```
#include <lcd.h>
```

并在 MPLAB IDE 项目属性中将 `inc` 目录的路径添加到编译器的头文件搜索路径中或在命令行上添加 `inc` 目录，如下所示：

```
-Ilcd
```

2.4.4 包含搜索路径

在 CCI 下包含头文件时，文件应可以在由编译器搜索的路径中找到，下面将详细说明。

对于在尖括号分隔符 `< >` 中指定的头文件，应可在通过 `-I` 选项 （或等效的 MPLAB IDE 选项）指定的搜索路径中或在标准的编译器包含目录中找到。 `-I` 选项将按指定它们的顺序进行搜索。

对于在引号字符 `" "` 中指定的头文件，应可在当前工作目录中或在以尖括号分隔符指定头文件时搜索的相同目录中（如上所述）找到。对于 MPLAB X 项目，当前工作目录是 C 源文件所在的目录。如果搜索未成功，搜索路径应为在尖括号分隔符中指定头文件时搜索的相同目录。

使用任何其他选项指定头文件的搜索路径都不符合 CCI 标准。

2.4.4.1 示例

如果使用以下伪指令包含头文件：

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

头文件应可在当前工作目录中，或通过任意 `-I` 选项指定的路径中，或标准编译器目录中找到。头文件位于其他位置不符合 CCI 标准。

2.4.4.2 差别

CCI 下的编译器操作不会发生变化。这纯粹是一条编码准则。

2.4.4.3 移植到 CCI

删除除 `-I` 选项 （或等效的 MPLAB IDE 选项）之外指定头文件搜索路径的任何其他选项，并使用 `-I` 选项代替它。确保可以在本节指定的目录中找到头文件。

© 2015 Microchip Technology Inc.

2.4.5 标识符中的有效初始字符数量

标识符 （内部和外部）中的至少前 255 个字符是有效的。它扩展了 ANSI C 标准的要求， ANSI C 标准使用较少的有效字符数量来标识对象。

2.4.5.1 示例

以下示例显示了两个命名方式很差的变量，但其名称在 CCI 下会被视为唯一。

```
int stateOfPortBWhenTheOperatorHasSelectedAutomaticModeAndMotorIsRunningFast;
int stateOfPortBWhenTheOperatorHasSelectedAutomaticModeAndMotorIsRunningSlow;
```

2.4.5.2 差别

先前的 8 位编译器在默认情况下使用 31 个有效字符，但允许通过某个选项对它进行扩展。16 位和 32 位编译器未对有效字符数量做出任何限制。

2.4.5.3 移植到 CCI

无需任何操作。您可以采用限制较少的命名方案。

2.4.6 类型的大小

CCI *未*完全定义基本 C 类型（如 `char`、 `int` 和 `long`）的大小。按照设计，这些类型将反映目标器件中的寄存器大小和其他架构特性。它们使器件可以高效地访问相应类型的对象。但是， ANSI C 标准在 `<limits.h>` 中规定了这些类型的最低要求。

如果项目中需要固定大小的类型，请使用 `<stdint.h>` 中定义的类型，例如 `uint8_t` 或 `int16_t`。这些类型的定义在所有 XC 编译器之间都保持一致，即使这些类型在 CCI 中没有定义。

基本上，C 语言提供了两组可供选择的类型：大小和格式针对所用器件定制的类型，或无论目标器件如何都具有固定大小的类型。

2.4.6.1 示例

以下示例显示了变量 `native` 的定义，其大小将支持在目标器件上进行高效的访问；以及变量 `fixed`，其大小明确指定并保持固定，虽然它可能无法在每款器件上支持高效的访问。

```
int native;
int16_t fixed;
```

2.4.6.2 差别

这与编译器先前实现的类型一致。

2.4.6.3 移植到 CCI

如果需要具有固定大小的 C 类型，则无论目标器件如何，使用由 `<stdint.h>` 定义的类型之一。

2.4.7 普通 char 类型

普通 char 的类型为 unsigned char。通常建议 char 类型的所有定义都明确指定对象的符号。

2.4.7.1 示例

以下示例

```
char foobar;
```

定义了一个名为 foobar 的 unsigned char 对象。

2.4.7.2 差别

8 位编译器总是将普通 char 视为无符号类型。

16 位和 32 位编译器使用 signed char 作为默认的普通 char 类型。这些编译器上的 -funsigned-char 选项会将默认类型更改为 unsigned char。

2.4.7.3 移植到 CCI

如果使用 16 位或 32 位编译器，则需要复查定义为普通 char 的对象的所有定义。任何希望用作有符号量的普通 char 都应替换为显式定义，例如：

```
signed char foobar;
```

在 MPLAB XC16 和 XC32 上，可以使用 -funsigned-char 选项来更改普通 char 的类型，但由于 MPLAB XC8 上不支持该选项，因此这种代码并不严格符合标准。

2.4.8 有符号整型表示

有符号整型的值通过获取整型的二进制补码来确定。

2.4.8.1 示例

以下显示了变量 test，它被赋予十进制值 -28。

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

2.4.8.2 差别

所有编译器都使用本节介绍的方式来表示有符号整型。

2.4.8.3 移植到 CCI

无需任何操作。

2.4.9 整型转换

将一个整型转换为大小不足的有符号整型时，原始值将从最高有效位开始截断，以适应目标大小。

2.4.9.1 示例

以下代码显示了会被截断的赋值操作。

```
signed char destination;
unsigned int source = 0x12FE;
destination = source;
```

在 CCI 下， destination 赋值之后的值为 -2 （即，位模式 0xFE）。

© 2015 Microchip Technology Inc.

2.4.9.2 差别

所有编译器都使用与本节所介绍方式完全相同的方式来执行整型转换。

2.4.9.3 移植到 CCI

无需任何操作。

2.4.10 有符号值的按位运算

有符号值的按位运算将对二进制补码表示形式进行操作（包括符号位）。另请参见**第 2.4.11 节 “右移有符号值”**。

2.4.10.1 示例

以下显示了一个对负数进行按位与运算的示例。

```
signed char output, input = -13;
output = input & 0x7E;
```

在 CCI 下， `output` 赋值之后的值为 0x72。

2.4.10.2 差别

所有编译器都使用与本节所介绍方式完全相同的方式来执行按位运算。

2.4.10.3 移植到 CCI

无需任何操作。

2.4.11 右移有符号值

右移有符号值会涉及到符号扩展。这将保留原始值的符号。

2.4.11.1 示例

以下显示了一个对负数进行右移操作的示例。

```
signed char output, input = -13;
output = input >> 3;
```

在 CCI 下， `output` 赋值之后的值为 -2（即，位模式 0xFE）。

2.4.11.2 差别

所有编译器都按本节所述的方式执行右移。

2.4.11.3 移植到 CCI

无需任何操作。

2.4.12 使用具有不同类型的联合成员访问另一个联合成员时的成员转换

如果某个联合定义了几个不同类型的成员，并且您使用其中一个成员标识符来尝试访问另一个成员的内容（是否对结果应用任何转换）在标准中属于实现定义的行为。在 CCI 中，不应用任何转换，联合对象的字节会被解释为类型与所访问成员相同的对象，而不考虑对齐或其他可能的无效条件。

2.4.12.1 示例

以下显示了一个定义了几个成员的联合示例。

```
union {
   signed char code;
   unsigned int data;
   float offset;
} foobar;
```

尝试通过读取 `data` 来提取 `offset` 的代码不能保证读取到正确的值。

```
float result;
result = foobbar.data;
```

2.4.12.2 差别

所有编译器都不转换通过其他成员访问的联合成员。

2.4.12.3 移植到 CCI

无需任何操作。

2.4.13 默认的位域 `int` 类型

指定为普通 `int` 的位域的类型与使用 `unsigned int` 定义的位域的类型相同。对于其他对象，类型 `int`、`signed` 和 `signed int` 属于同义，所以这与其他对象完全不同。建议在所有位域定义中明确指定位域的符号。

2.4.13.1 示例

以下显示了一个结构标签的示例，其中包含类型为无符号整型并具有指定大小的位域。

```
struct OUTPUTS {
   int direction    :1;
   int parity    :3;
   int value     :4;
};
```

2.4.13.2 差别

如果对于位域使用了 `int` 类型，8 位编译器会发出一个警告，但会使用 `unsigned int` 类型实现位域。

16 位和 32 位编译器实现了使用 `int` 定义为具有 `signed int` 类型的位域，除非指定了选项 `-funsigned-bitfields`。

© 2015 Microchip Technology Inc.

2.4.13.3 移植到 CCI

需要复查使用普通 int 类型定义了位域的所有代码。如果希望它们作为有符号量，则应将它们的类型更改为 signed int。例如，在以下示例中：

```
struct WAYPT {
   int log          :3;
   int direction    :4;
};
```

位域类型应更改为 signed int，如下：

```
struct WAYPT {
   signed int log       :3;
   signed int direction :4;
};
```

2.4.14 位域跨越存储单元边界

标准规定实现可以决定位域是否跨越存储单元边界。在 CCI 中，位域不会跨越存储单元边界；将为结构分配新的存储单元，并通过填充位来填补空隙。

请注意，存储单元的大小对于每种编译器是不同的，因为它基于基本数据类型（如 int）的大小，而位域类型是基于基本数据类型派生的。在 8 位编译器上，该单元的大小为 8 位；对于 16 位编译器，它为 16 位；对于 32 位编译器，它为 32 位。

2.4.14.1 示例

以下显示了一个定义了位域的结构。

```
struct {
        unsigned first  : 6;
        unsigned second :6;
} order;
```

在 CCI 下且使用 MPLAB XC8 时，存储分配单元为字节大小。位域 second 会被分配一个新的存储单元，因为在第一个存储单元中分配 first 之后，只剩下 2 位。该结构 order 的大小为 2 个字节。

2.4.14.2 差别

这种分配方式与所有先前编译器使用的方式完全相同。

2.4.14.3 移植到 CCI

无需任何操作。

2.4.15 位域的分配顺序

ANSI C 标准未规定位域在其存储单元中的存储顺序。在 CCI 中，所定义的第一个位为它所分配到存储单元中的最低有效位。

2.4.15.1 示例

以下显示了一个定义了位域的结构。

```
struct {
        unsigned lo  : 1;
        unsigned mid :6;
        unsigned hi  : 1;
} foo;
```

在分配给结构 foo 的存储单元中，位域 lo 会被分配到最低有效位。位域 mid 会被分配到接下来的低 6 位， hi 会被分配到同一存储单元字节的最高有效位。

2.4.15.2 差别

这与所有编译器的先前操作是相同的。

2.4.15.3 移植到 CCI

无需任何操作。

2.4.16 NULL 宏

NULL 宏在 <stddef.h> 中定义；但是，其定义属于实现定义的行为。在 CCI 下，NULL 的定义为表达式 (0)。

2.4.16.1 示例

以下显示了通过 NULL 宏为指针赋予一个空指针常量。

```
int * ip = NULL;
```

NULL 和 (0) 的值会被隐式转换为目标类型。

2.4.16.2 差别

32 位编译器先前使用表达式 ((void*)0) 来为 NULL 赋值。

2.4.16.3 移植到 CCI

无需任何操作。

2.4.17 浮点型长度

在 CCI 下，浮点型的长度必须不小于 32 位。

2.4.17.1 示例

以下显示了 outY 的定义，其大小至少为 32 位。

```
float outY;
```

2.4.17.2 差别

8 位编译器允许使用 24 位的 float 和 double 类型。

2.4.17.3 移植到 CCI

使用 8 位编译器时，在使能 CCI 模式之后， float 和 double 类型的大小将自动设为 32 位。复查所有可能具有 float 或 double 类型，并且大小可能为 24 位的源代码。

对于其他编译器，不需要进行移植。

© 2015 Microchip Technology Inc.

2.5 ANSI 标准扩展

以下主题介绍了 CCI 如何提供对标准的特定于器件扩展。

2.5.1 通用头文件

必须使用单个头文件 <xc.h> 来声明所有特定于编译器和特定于器件的类型和 SFR。您*必须*将该文件包含到每个模块中，以符合 CCI 标准。一些 CCI 定义依赖于该头文件。

2.5.1.1 示例

以下显示了包含该头文件的示例，从而可以符合 CCI 标准，以及可以访问 SFR。

```
#include <xc.h>
```

2.5.1.2 差别

一些 8 位编译器使用了 <htc.h> 作为等效的头文件。16 位和 32 位编译器的先前版本使用了一系列的头文件来完成同样的工作。

2.5.1.3 移植到 CCI

将先前在 8 位编译器代码中使用的以下头文件：

```
#include <htc.h>
```

或特定于系列的头文件，如以下文件：

```
#include <p32xxxx.h>
#include <p30fxxxx.h>
#include <p33Fxxxx.h>
#include <p24Fxxxx.h>
#include "p30f6014.h"
```

更改为：

```
#include <xc.h>
```

2.5.2 绝对寻址

变量和函数可以通过使用 __at() 构造放置在绝对地址处。基于堆栈的 （auto 和参数）变量不能使用 __at() 说明符。

2.5.2.1 示例

以下显示了分配到绝对地址的两个变量和一个函数。

```
const char keys[] __at(123) = { 'r', 's', 'u', 'd'};
__at(0x1000)int modify(int x)  {
return x * 2 + 3;
}
```

2.5.2.2 差别

8 位编译器使用了 @ 符号来指定绝对地址。

16 位和 32 位编译器使用了 address 属性来指定对象的地址。

2.5.2.3 移植到 CCI

如果可能，避免将对象和函数分配到绝对寻址。

在 MPLAB XC8 中，更改绝对对象定义，例如将以下代码：

```
int scanMode @ 0x200;
```

更改为：

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

在 MPLAB XC16 和 XC32 中，更改代码，例如将以下代码：

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

更改为：

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

2.5.2.4 注意事项

在使用 MPLAB XC8 时，如果对某个对象同时应用了 `__at()` 和 `__section()` 说明符，`__section()` 说明符目前会被忽略。

2.5.3 far 对象和函数

`__far` 限定符可用于指示变量或函数位于 “far 存储区”中。究竟哪个存储区构成 far 存储区取决于目标器件，但它通常是需要较复杂代码来进行访问的存储区。如果表达式涉及的是使用 far 限定符的对象，表达式通常会生成较慢、较大的代码。

使用 “差别”一节中讨论的本机关键字来查找关于该限定符语义的信息。

一些器件可能未实现这种存储区；在这种情况下，使用该限定符会被忽略。基于堆栈的（`auto` 和函数参数）变量无法使用 `__far` 说明符。

2.5.3.1 示例

以下显示了使用 `__far` 限定的变量和函数。

```
__far int serialNo;
__far int ext_getCond(int selector);
```

2.5.3.2 差别

8 位编译器使用了 `far` 限定符来表示这种含义。函数不能使用 `far` 进行限定。

16 位编译器对于变量和函数都使用了 `far` 属性。

32 位编译器仅对于函数使用了 `far` 属性。

© 2015 Microchip Technology Inc.

2.5.3.3 移植到 CCI

对于 8 位编译器，更改每一处 far 限定符为 __far，例如将以下代码：

```
far char template[20];
```

更改为：

```
__far char template[20];
```

在 16 位和 32 位编译器中，更改每一处 far 属性，例如将以下代码：

```
void bar(void) __attribute__ ((far));
int tblIdx __attribute__ ((far));
```

更改为：

```
void __far bar(void);
int __far tblIdx;
```

2.5.3.4 注意事项

无。

2.5.4 near 对象

__near 限定符可用于指示变量或函数位于 “near 存储区”中。究竟哪个存储区构成 near 存储区取决于目标器件，但它通常是可以使用较简单代码来进行访问的存储区。如果表达式涉及的是使用 near 限定符的对象，表达式通常会生成较快、较小的代码。

使用 “差别”一节中讨论的本机关键字来查找关于该限定符语义的信息。

一些器件可能未实现这种存储区；在这种情况下，使用该限定符会被忽略。基于堆栈的（auto 和函数参数）变量无法使用 __near 说明符。

2.5.4.1 示例

以下显示了使用 __near 限定的变量和函数。

```
__near int serialNo;
__near int ext_getCond(int selector);
```

2.5.4.2 差别

8 位编译器使用了 near 限定符来表示这种含义。函数不能使用 near 进行限定。

16 位编译器对于变量和函数都使用了 near 属性。

32 位编译器仅对于函数使用了 near 属性。

2.5.4.3 移植到 CCI

对于 8 位编译器，更改每一处 near 限定符为 __near，例如将以下代码：

```
near char template[20];
```

更改为：

```
__near char template[20];
```

在 16 位和 32 位编译器中，更改每一处 near 属性，例如将以下代码：

```
void bar(void) __attribute__ ((near));
int tblIdx __attribute__ ((near));
```

更改为：

```
void __near bar(void);
int __near tblIdx;
```

2.5.4.4 注意事项

无。

2.5.5 持久性对象

__persistent 限定符可用于指示运行时启动代码不应清零的变量。

使用 “差别”一节中讨论的本机关键字来查找关于该限定符语义的信息。

2.5.5.1 示例

以下显示了使用 __ persistent 限定的变量。

```
__persistent int serialNo;
```

2.5.5.2 差别

8 位编译器使用 persistent 限定符来表示这种含义。

16 位和 32 位编译器对于变量使用 persistent 属性来指示不应将它们清零。

2.5.5.3 移植到 CCI

对于8位编译器，更改每一处persistent限定符为__persistent，例如将以下代码：

```
persistent char template[20];
```

更改为：

```
__persistent char template[20];
```

对于 16 位和 32 位编译器，更改每一处 persistent 属性，例如将以下代码：

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

更改为：

```
int __persistent tblIdx;
```

2.5.5.4 注意事项

无。

© 2015 Microchip Technology Inc.

2.5.6 X 和 Y 数据对象

__xdata 和 __ydata 限定符可用于指示变量位于特殊存储区。究竟哪个存储区构成 X 和 Y 存储区取决于目标器件，但它通常是可以在单独总线上进行独立访问的存储区。一些 DSP 指令通常需要此类存储区。

使用 “差别”一节中讨论的本机关键字来查找关于这些限定符语义的信息。

一些器件可能未实现这种存储区；在这种情况下，使用这些限定符会被忽略。

2.5.6.1 示例

以下显示了一个使用 __xdata 限定的变量，以及另一个使用 __ydata 限定的变量。

```
__xdata char data[16];
__ydata char coeffs[4];
```

2.5.6.2 差别

16 位编译器对于变量使用了 xmemory 和 ymemory 空间属性。

对于所有其他编译器，未定义等效的说明符。

2.5.6.3 移植到 CCI

对于16位编译器，更改每一处space属性xmemory或ymemory为__xdata或__ydata，例如将以下代码：

```
char __attribute__((space(xmemory)))template[20];
```

更改为：

```
__xdata char template[20];
```

2.5.6.4 注意事项

无。

2.5.7 分区数据对象

__bank(num) 限定符可用于指示变量位于某个特定的数据存储区（bank）。数字 num 代表存储区编号。究竟哪个存储区构成分区 （banked）存储器取决于目标器件，但它通常是数据存储器的一个分区，用于支持地址字段宽度有限的汇编指令。

使用 “差别”一节中讨论的本机关键字来查找关于这些限定符语义的信息。

一些器件可能未实现分区数据存储器，在这种情况下，使用该限定符会被忽略。实现的数据存储区数量会因器件而异。

2.5.7.1 示例

以下显示了使用 __bank() 限定的变量。

```
__bank(0) char start;
__bank(5) char stop;
```

2.5.7.2 差别

8 位编译器使用了 4 个限定符 `bank0`、`bank1`、`bank2` 和 `bank3` 来表示相同（然而更有限）的存储区布局方式。

对于所有其他编译器，未定义等效的说明符。

2.5.7.3 移植到 CCI

对于 8 位编译器，更改每一处 `bank`*x* 限定符为 `__bank`，例如将以下代码：

```
bank2 int logEntry;
```

更改为：

```
__bank(2) int logEntry;
```

2.5.7.4 注意事项

MPLAB XC8 中尚未实现该功能。

2.5.8 对象对齐

`__align(`*`alignment`*`)` 说明符可用于指示变量必须对齐到某个为所指定 alignment 倍数的存储器地址。 alignment 项必须为 2 的幂。正值表示对对象的起始地址进行对齐；负值表示对对象的结束地址进行对齐。

使用 “差别”一节中讨论的本机关键字来查找关于该说明符语义的信息。

2.5.8.1 示例

以下显示了使用 `__align()` 限定的变量，以分别确保它们的结束地址为 8 的倍数，起始地址为 2 的倍数。

```
__align(-8) int spacer;
__align(2) char coeffs[6];
```

2.5.8.2 差别

8 位编译器上未实现对齐功能。

16 位和 32 位编译器对于变量使用了 `aligned` 属性。

2.5.8.3 移植到 CCI

对于 16 位和 32 位编译器，更改每一处 `aligned` 属性为 `__align`，例如将以下代码：

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

更改为：

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

2.5.8.4 注意事项

MPLAB XC8 上尚未实现该功能。

© 2015 Microchip Technology Inc.

2.5.9 EEPROM 对象

__eeprom 限定符可用于指示应将变量定位到 EEPROM 中。

使用 “差别”一节中讨论的本机关键字来查找关于该限定符语义的信息。

一些器件可能未实现 EEPROM。对此类器件使用该限定符会生成警告。基于堆栈的（auto 和函数参数）变量无法使用 __eeprom 说明符。

2.5.9.1 示例

以下显示了使用 __eeprom 限定的变量。

```
__eeprom int serialNos[4];
```

2.5.9.2 差别

8 位编译器对于一些器件使用了 eeprom 限定符来表示这种含义。

16 位编译器使用了 space 属性来将变量分配到用于 EEPROM 的存储空间。

2.5.9.3 移植到 CCI

对于 8 位编译器，更改每一处 eeprom 限定符为 __eeprom，例如将以下代码：

```
eeprom char title[20];
```

更改为：

```
__eeprom char title[20];
```

对于 16 位编译器，更改每一处 eedata space 属性，例如将以下代码：

```
int mainSw __attribute__ ((space(eedata)));
```

更改为：

```
int __eeprom mainSw;
```

2.5.9.4 注意事项

对于所有 PIC18 器件，MPLAB XC8 均未实现 __eeprom 限定符；对于其他 8 位器件，该限定符会按预期工作。

2.5.10 中断函数

__interrupt(*type*) 说明符可用于指示函数将用作中断服务程序。*type* 是使用逗号分隔的关键字列表，指示关于中断函数的信息。

当前的中断类型包括：

< *空* >

实现默认的中断函数

low_priority

中断函数对应于低优先级中断源 （MPLAB XC8——仅限 PIC18）

high_priority

中断函数对应于高优先级中断源 （MPLAB XC8）

save(*symbol-list*)

在进入中断时保存并在退出中断时恢复所列的符号 （XC16）

irq(*irqid*)

指定与该中断关联的中断向量 （XC16）

altirq(*altirqid*)

指定与该中断关联的备用中断向量 （XC16）

preprologue(*asm*)

指定需要在任何编译器生成的中断代码之前执行的汇编代码 （XC16）

shadow

允许 ISR 利用影子寄存器进行现场切换 （XC16）

auto_psv

ISR 将设置 PSVPAG 寄存器，并在退出时恢复它 （XC16）

no_auto_psv

ISR 不会设置 PSVPAG 寄存器 （XC16）

使用 “差别”一节中讨论的本机关键字来查找关于该说明符语义的信息。

一些器件可能未实现中断。对此类器件使用该限定符会生成警告。如果 __interrupt 说明符的参数对于目标器件没有意义，编译器会发出警告或错误。

2.5.10.1 示例

以下显示了使用 __interrupt 限定的函数。

```
__interrupt(low_priority) void getData(void) {
    if (TMR0IE && TMR0IF) {
        TMR0IF=0;
        ++tick_count;
    }
}
```


© 2015 Microchip Technology Inc.

2.5.10.2 差别

8 位编译器对于一些器件使用了 `interrupt` 和 `low_priority` 限定符来表示这种含义。中断程序默认情况下为高优先级。

16 位和 32 位编译器使用了 `interrupt` 属性来定义中断函数。

2.5.10.3 移植到 CCI

对于 8 位编译器，更改每一处 `interrupt` 限定符，例如将以下代码：

```
void interrupt myIsr(void)
void interrupt low_priority myLoIsr(void)
```

分别更改为以下形式：

```
void __interrupt(high_priority) myIsr(void)
void __interrupt(low_priority) myLoIsr(void)
```

对于 16 位编译器，更改每一处 `interrupt` 属性，例如将以下代码：

```
void _attribute_((interrupt(auto_psv,irq(52))))
_T1Interrupt(void);
```

更改为：

```
void __interrupt(auto_psv,irq(52))) _T1Interrupt(void);
```

对于 32 位编译器， `__interrupt()` 关键字接受两个参数，即，向量编号和 （可选） IPL 值。更改使用 `interrupt` 属性的代码，如以下示例：

```
void __attribute__((vector(0), interrupt(IPL7AUTO), nomips16))
myisr0_7A(void) {}

void __attribute__((vector(1), interrupt(IPL6SRS), nomips16))
myisr1_6SRS(void) {}

/* Determine IPL and context-saving mode at runtime */
void __attribute__((vector(2), interrupt(), nomips16))
myisr2_RUNTIME(void) {}
```

更改为：

```
void __interrupt(0,IPL7AUTO) myisr0_7A(void) {}

void __interrupt(1,IPL6SRS) myisr1_6SRS(void) {}

/* Determine IPL and context-saving mode at runtime */
void __interrupt(2) myisr2_RUNTIME(void) {}
```

2.5.10.4 注意事项

无。

2.5.11 压缩对象

__pack 说明符可用于指示结构不使用存储器空隙，对齐结构成员；或不对齐各个结构成员。

使用 “差别”一节中讨论的本机关键字来查找关于该说明符语义的信息。

对于某些器件，一些编译器不会使用存储器空隙来填充结构，对齐结构成员，对此类器件使用该说明符会被忽略。

2.5.11.1 示例

以下显示了一个使用 __pack 限定的结构，以及有一个成员已被显式压缩的结构。

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

2.5.11.2 差别

对于 MPLAB XC8 来说，__pack 说明符是一个全新的 CCI 说明符。该说明符没有明显的作用，因为器件存储器对于所有数据对象都是可字节寻址的。

16 位和 32 位编译器使用 packed 属性来指示结构成员使用存储器空隙，不对齐。

2.5.11.3 移植到 CCI

对于 MPLAB XC8，不需要进行移植。

对于 16 位和 32 位编译器，更改每一处 packed 属性，例如将以下代码：

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

更改为：

```
struct DOT
{
   char a;
   __pack int x[2];
};
```

或者如果需要，您也可以压缩整个结构。

2.5.11.4 注意事项

无。

© 2015 Microchip Technology Inc.

2.5.12 指示过时的对象

__deprecate 说明符可用于指示某个对象寿命有限，不应在新设计中使用。编译器供应商经常使用它来指示编译器扩展或功能可能过时，或已开发了应优先使用的更好功能。

使用 “差别”一节中讨论的本机关键字来查找关于该说明符语义的信息。

2.5.12.1 示例

以下显示了一个使用 __deprecate 关键字的函数。

```
void __deprecate getValue(int mode)
{
//...
}
```

2.5.12.2 差别

8 位编译器上未实现任何弃用功能。

16 位和 32 位编译器使用 deprecated 属性 （请注意拼写的不同）来指示应尽可能避免使用这些对象。

2.5.12.3 移植到 CCI

对于 16 位和 32 位编译器，更改每一处 deprecated 属性，例如将以下代码：

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

更改为：

```
int __deprecate intMask;
```

2.5.12.4 注意事项

无。

2.5.13 将对象分配到段

__section() 说明符可用于指示应将某个对象定位到指定段（或 psect，使用 MPLAB XC8 术语）。它通常在对象具有特殊的独特链接要求、无法通过现有编译器特性解决的时候使用。

使用 “差别”一节中讨论的本机关键字来查找关于该说明符语义的信息。

2.5.13.1 示例

以下显示了一个使用 __section 关键字的变量。

```
int __section("comSec") commonFlag;
```

2.5.13.2 差别

8 位编译器使用 `#pragma psect` 伪指令来将对象重定位到一个新的段或 psect。`__section()` 说明符的操作与该 pragma 伪指令有几个方面的不同，如下所述。

不同于该 pragma 伪指令，使用 `__section()` 创建的新 psect 不会继承正常情况下该对象分配到的 psect 的标志。这意味着新 psect 可以链接到任意存储区中，包括任意数据存储区。编译器也不会对对象在新段中的位置做出任何假定。使用该 pragma 伪指令重定位到新 psect 的对象必须总是链接到相同的存储区中，但可以在该区域中的任意地址上。

`__section()` 说明符允许将已初始化对象放置在不同 psect 中。即使在新 psect 中，仍然会执行对象的初始化。这将需要自动分配额外的 psect （其名称为新 psect 名称加上字母 `i` 作为前缀），其中将包含初始值。 pragma 伪指令不能用于已初始化对象。

不同于使用 pragma 伪指令的对象，使用 `__section()` 分配了不同 psect 的对象会被运行时启动代码清零。

在当前的 MPLAB XC8 编译器实现中，必须为使用 `__section()` 说明符创建的所有新 psect 保留存储器，并通过一个链接器选项定位。

16 位和 32 位编译器使用 `section` 属性来指示不同的目标段名称。 `__section()` 说明符的工作方式类似于属性。

2.5.13.3 移植到 CCI

对于MPLAB XC8，更改每一处`#pragma psect`伪指令为`__section()`说明符，例如：

```
#pragma psect text%%u=myText
int getMode(int target) {
//...
}
```

更改为：

```
int __section ("myText") getMode(int target) {
//...
}
```

对于 16 位和 32 位编译器，更改每一处 `section` 属性，例如将以下代码：

```
int __attribute__((section("myVars"))) intMask;
```

更改为：

```
int __section("myVars") intMask;
```

2.5.13.4 注意事项

无。

2.5.14 指定配置位

`#pragma config` 伪指令可用于编程器件的配置位。该 pragma 伪指令具有以下形式：

`#pragma config` *`setting = state|value`*

其中，*`setting`* 是配置设置描述符（如 `WDT`），*`state`* 是一个描述性的值（如 `ON`），*`value`* 是一个数值。

使用 “差别”一节中讨论的本机关键字来查找关于该伪指令语义的信息。

© 2015 Microchip Technology Inc.

2.5.14.1 示例

以下显示了使用该 pragma 伪指令指定配置位的示例。

```
#pragma config WDT=ON, WDTPS = 0x1A
```

2.5.14.2 差别

对于一些尚不支持 `#pragma config` 的目标器件，8 位编译器使用 `__CONFIG()` 宏。

16 位编译器使用许多宏来指定配置设置。

32 位编译器支持使用 `#pragma config`。

2.5.14.3 移植到 CCI

对于 8 位编译器，更改每一处 `__CONFIG()` 宏为 `#pragma config` 伪指令，例如

```
__CONFIG(WDTEN & XT & DPROT)
```

更改为

```
#pragma config WDTE=ON, FOSC=XT, CPD=ON
```

如果已使用了 `#pragma config`，则无需进行移植。

对于 16 位编译器，更改每一处 `_FOSC()` 或 `_FBORPOR()` 宏属性，例如将以下代码：

```
_FOSC(CSW_FSCM_ON & EC_PLL16);
```

更改为：

```
#pragma config FCKSMEM = CSW_ON_FSCM_ON,  FPR = ECIO_PLL16
```

对于 32 位代码，无需进行移植。

2.5.14.4 注意事项

无。

2.5.15 列表宏

CCI 定义了列出编译器和目标器件特性的宏的一般格式。这些宏可用于基于编译器或目标器件条件性地编译备用源代码。

表 2-1 详细介绍了宏和宏系列。

表 2-1： 由 CCI 定义的列表宏

名称	定义时的含义	示例
`__XC__`	使用 MPLAB® XC 编译器编译	`__XC__`
`__CCI__`	编译器符合 CCI 标准，并使能了 CCI 强制实施	`__CCI__`
`__XC##__`	使用了特定的 XC 编译器 （## 可以为 8、 16 或 32）	`__XC8__`
`__DEVICEFAMILY__`	所选目标器件的系列	`__dsPIC30F__`
`__DEVICENAME__`	所选目标器件的名称	`__18F452__`

2.5.15.1 示例

以下显示了根据具有 EEPROM 存储器的器件进行条件性编译的代码。

```
#ifdef __XC16__
void __interrupt(__auto_psv__) myIsr(void)
#else
void __interrupt(low_priority) myIsr(void)
#endif
```

2.5.15.2 差别

其中一些 CCI 宏是全新的（如 __CCI__），其他一些与先前具有相同含义的符号具有不同的名称（如 __18F452 现在为 __18F452__）。

2.5.15.3 移植到 CCI

需要复查所有使用编译器定义的宏的代码。旧的宏可以继续按预期工作，但它们不符合 CCI 标准。

2.5.15.4 注意事项

无。

2.5.16 行内汇编

asm() 语句可用于在 C 代码中插入行内汇编代码。其参数是一个 C 字符串字面值，代表一条汇编指令。显然，参数中包含的指令是特定于器件的。

使用 “差别”一节中讨论的本机关键字来查找关于该语句语义的信息。

2.5.16.1 示例

以下显示了插入行内汇编 MOVLW 指令的示例。

```
asm("MOVLW _foobar");
```

2.5.16.2 差别

8 位编译器使用 asm() 或 #asm ... #endasm 构造来嵌入汇编代码。

这是 16 位和 32 位编译器使用的相同语法。

2.5.16.3 移植到 CCI

对于 8 位编译器，更改所有 #asm ... #endasm 实例，使 #asm 块中的每条指令都放入自己的 asm() 语句中，例如将以下代码：

```
#asm
   MOVLW  20
   MOVWF _i
   CLRF   Ii+1
#endasm
```

更改为：

```
asm("MOVLW20");
asm("MOVWF _i");
asm("CLRFIi+1");
```

16 位或 32 位编译器无需进行移植。

2.5.16.4 注意事项

无。

© 2015 Microchip Technology Inc.

2.6 编译器特性

以下小节详细介绍了用于控制 CCI 的编译器选项。

2.6.1 使能 CCI

假定您使用 MPLAB X IDE 来编译使用 CCI 的项目。MPLAB X IDE 的 Project Properties（项目属性）中用于确保符合 CCI 标准的窗口小部件是编译器类别中的 Use CCI Syntax（使用 CCI 语法）。

如果不使用该 IDE，则命令行选项为 `--EXT=cci`（对于 MPLAB XC8）或 `-mcci`（对于 MPLAB XC16 和 XC32）。

2.6.1.1 差别

该选项先前从未实现过。

2.6.1.2 移植到 CCI

使能该选项。

2.6.1.3 注意事项

无。

注：

© 2015 Microchip Technology Inc.

第 3 章 操作指南

3.1 简介

本章包含关于在为 Microchip 16 位器件编译项目时经常遇到的一些情况的帮助和参考。单击每一节开始处的链接，可以协助您找到与您的问题相关的主题。一些主题涉及多个章节中的内容。

从这里开始

- 安装和激活编译器
- 调用编译器
- 编写源代码
- 让应用程序执行所需的操作
- 了解编译过程
- 修复无法工作的代码

3.2 安装和激活编译器

本节详细介绍在安装或激活编译器时可能出现的问题。

- 如何安装和激活编译器？
- 如何确定编译器是否已成功安装并激活？
- 是否可以安装同一编译器的多个版本？

3.2.1 如何安装和激活编译器？

XC 编译器安装程序会同时执行许可证的安装和激活。关于指示信息，请参见以下文档。该文档可从 Microchip 网站 www.microchip.com 下载。

“Installing and Licensing MPLAB XC C Compilers”（DS52059）

3.2.2 如何确定编译器是否已成功安装并激活？

要验证编译器是否已安装并激活，必须编译代码。您可以使用编译器随附的示例代码。默认情况下，它位于以下文件夹中：`c:\Program Files\Microchip\xc16\examples`

然后，根据您所用的操作系统，在 `xc16_getting_started` 文件夹中找到文件 `run_hello.bat`（Windows 操作系统）或 `run_hello.sh`（Mac/Linux 操作系统）。编辑优化级别，使之对应于您预期的许可证级别。关于更多信息，请参见本用户指南的以下章节：

第 18 章 “优化”

第 5.7.6 节 “用于控制优化的选项”

运行编辑后的批处理或 shell 文件。如果工具已正确安装，输出中将显示编译和执行过程中的各个步骤，并以以下文本结束：

```
Hello, world!
```

3.2.3 是否可以安装同一编译器的多个版本？

可以，编译器和安装过程设计为允许您安装同一编译器的多个版本。在 MPLAB X IDE 中，可以通过在 IDE 中更改选项来简便地在编译器版本之间切换。关于详细信息，请参见本用户指南的以下章节：

第 3.3.3 节 “如何选择要用于编译项目的编译器版本？”

编译器应安装到根据编译器版本命名的目录中。这将反映在安装程序指定的默认目录中。例如，MPLAB XC16 编译器 v1.00 和 v1.10 通常放置在独立的目录中，如下所示：

```
C:\Program Files\Microchip\xc16\v1.00\
C:\Program Files\Microchip\xc16\v1.10\
```


© 2015 Microchip Technology Inc.

3.3 调用编译器

本节讨论如何在命令行上和从 IDE 中运行编译器。此外，还包含了关于如何使用编译器选项和编译过程来充分利用编译器获得最大收获的信息。

- 如何在 MPLAB X IDE 中进行编译？
- 如何在命令行上进行编译？
- 如何选择要用于编译项目的编译器版本？
- 如何更改编译器优化？
- 如何确定我所得到的优化功能？
- 如何确定哪些编译器选项可用，以及它们的功能？
- 如何编译库？
- 如何确定 MPLAB X IDE 中的编译选项的功能？
- MPLAB X IDE 调试编译有何差别？
- *另请参见*为什么 MPLAB X IDE 反汇编窗口中没有反汇编信息？
- *另请参见*默认情况下会包含哪些库？
- *另请参见*如何阻止编译器使用某些特定的存储单元？

3.3.1 如何在 MPLAB X IDE 中进行编译？

在 MPLAB X IDE 中，通过编译项目来编译代码。

关于在 MPLAB X IDE 中使用编译器的更多信息，请参见本用户指南的以下章节：

第 4 章 “XC16 工具链和 MPLAB X IDE”

第 3.3.3 节 “如何选择要用于编译项目的编译器版本？”

3.3.2 如何在命令行上进行编译？

要在命令行上编译代码，请参见本用户指南的以下章节：

第 5 章 “编译器命令行驱动程序”

第 3.3.3 节 “如何选择要用于编译项目的编译器版本？”

3.3.3 如何选择要用于编译项目的编译器版本？

编译和安装过程设计为允许您同时安装多个编译器版本。

在 MPLAB X IDE 中，通过打开 Project Properties 窗口（*File（文件）> Project Properties*），然后选择 Configuration（配置）类别（`Conf: [default]`）来选择用于编译项目的编译器。窗口最右侧的 Compiler Toolchain（编译器工具链）中将会显示 MPLAB XC16 编译器版本的列表。选择您需要的 MPLAB XC16 编译器。

选定之后，通过选择 XC16 (Global Options)（XC16（全局选项））、XC16 编译器和 XC16 链接器类别，将显示对应于该编译器的所有控件。它们会在右侧显示一个选项窗格；每个类别均具有几个窗格，可以从靠近窗格顶部的下拉菜单中进行选择。

3.3.4 如何更改编译器优化？

您只能选择您的许可证允许您使用的优化。关于编译器许可证和相关优化，以及设置优化的更多信息，请参见本用户指南的以下章节：

第 18 章 “优化”

第 5.7.6 节 “用于控制优化的选项”

3.3.5 如何确定我所得到的优化功能？

当您选择某个优化级别时，您会得到若干个优化功能。本用户指南的以下章节以表的形式列出了这些功能：

第 18.3 节 “优化功能汇总”

3.3.6 如何确定哪些编译器选项可用，以及它们的功能？

本用户指南的以下章节给出了所有编译器选项的列表：

第 5.7 节 “驱动程序选项说明”

3.3.7 如何编译库？

关于如何创建和编译您自己的库的信息，请参见本用户指南的以下章节：

第 5.4.1.2 节 “用户定义的库”

3.3.8 如何确定 MPLAB X IDE 中的编译选项的功能？

MPLAB X IDE 的 Project Properties 窗口中的大多数窗口小部件和控件以及 XC16 选项，均直接映射到相应的命令行驱动程序选项或子选项。关于这些选项和所有相应命令行选项的列表，请参见本用户指南的以下章节：

第 4.5 节 “项目设置”

3.3.9 MPLAB X IDE 调试编译有何差别？

对于调试编译， MPLAB X IDE 需要使用额外的存储器，因为调试需要额外的资源。关于详细信息，请参见 MPLAB X IDE 文档。

© 2015 Microchip Technology Inc.

3.4 编写源代码

本节列出与您编写的源代码有关的问题。它被细分为下面列出的章节。

- C 语言特定细节
- 特定于器件的功能
- 存储器分配
- 变量
- 函数
- 中断
- 汇编代码

3.4.1 C 语言特定细节

MPLAB XC16 C 编译器是符合 ANSI C 标准的编译器，因此遵循标准 C 语言的约定。关于更多信息，请参见本用户指南的以下章节：

第 1.3.1 节 “ANSI C 标准”

3.4.2 特定于器件的功能

本节讨论设置或控制某个特定于 Microchip 器件的功能需要编写的代码。

- 如何将代码移植到不同的器件架构？
- 如何设置配置位？
- 如何访问用户 ID 存储单元？
- 如何访问特殊功能寄存器 （SFR）？
- 编译器定义了哪些特定于器件的符号，我可以使用它们吗？
- *另请参见*如何阻止编译器使用某些特定的存储单元？

3.4.2.1 如何将代码移植到不同的器件架构？

为了减少在架构之间移植代码所需的工作，开发了通用 C 接口 （CCI）。如果使用这些编码样式，您的代码将可以更容易地向上移植。关于 CCI 的更多信息，请参见本用户指南的以下章节：

第 2 章 “通用 C 接口”

3.4.2.2 如何设置配置位？

它们应在代码中通过使用宏或 pragma 伪指令进行设置。先前版本的 MPLAB IDE 允许您通过对话框来设置这些位，但 MPLAB X IDE 要求在源代码中指定它们。请参见本用户指南的以下章节：

第 6.5 节 “配置位访问”

3.4.2.3 如何访问用户 ID 存储单元？

目前，访问器件（或系列）ID 单元的唯一方式是指定器件 ID 寄存器的固定地址。目前未提供相应的宏或 pragma 伪指令。关于器件 ID 寄存器的地址，请参见器件数据手册。

3.4.2.4 如何访问特殊功能寄存器 （SFR）？

编译器在分发时随附了定义变量的头文件。这些变量映射到存储器映射 SFR 的顶部。由于它们是 C 变量，所以它们可以像任何其他 C 变量一样使用。访问这些寄存器不需要任何新语法。

这些变量的名称与您所用器件的数据手册中指示的名称相同。

此外，还可以访问 SFR 内的位。可以在结构中包含位域通过结构来访问位域，这些结构作为一个整体映射到 SFR。例如，PORTCbits.RC1 表示 PORTC 的 RC1 位。关于头文件的更多信息，请参见本用户指南的以下章节：

第 6.3 节 “器件头文件”

3.4.2.5 编译器定义了哪些特定于器件的符号，我可以使用它们吗？

编译器定义了一些特定于器件的符号或宏，以及其他一些符号或宏。本用户指南的以下章节对它们进行了介绍：

第 19.4 节 “预定义的宏名称”

© 2015 Microchip Technology Inc.

3.4.3 存储器分配

以下这些问题涉及到您的源代码影响存储器分配的方式。

- 如何将变量或函数定位到指定的地址处？
- 如何将变量放入程序存储器？
- 如何为变量分配空间，但不初始化 / 装入任何值？
- 如何阻止编译器使用某些特定的存储单元？

3.4.3.1 如何将变量或函数定位到指定的地址处？

通过微调工具链来将变量或函数分配在特定存储区域会使链接器相对难以完成它的工作。提供了一些工具来解决可能存在的问题，但总是应小心使用它们。例如，使用 `section` 属性将变量组合在一起可能就足够了，而不是将对象固定在某个特定地址处（使用 `address` 属性或 `__at` 构造）。

3.4.3.2 如何将变量放入程序存储器？

关于如何将变量放入程序存储器空间的信息，请参见本用户指南的以下章节：

第 10.4 节 “程序空间中的变量”

3.4.3.3 如何为变量分配空间，但不初始化 / 装入任何值？

要在不初始化变量或不将变量装入存储器的情况下为变量分配存储空间，可以使用 `noload` 属性。关于变量属性的更多信息，请参见本用户指南的以下章节：

第 8.12 节 “变量属性”

3.4.3.4 如何阻止编译器使用某些特定的存储单元？

可以通过将 `address` 属性与 `noload` 属性一起使用来阻止编译器使用某些存储器段。此外，您也可以使用选项 `-mreserved`。关于变量属性和选项的更多信息，请参见本用户指南的以下章节：

第 8.12 节 “变量属性”

第 5.7.1 节 “特定于 16 位器件的选项”

3.4.4 变量

本节讨论与程序中的变量和类型的定义和用法相关的问题。

- 为什么浮点结果与所预期的结果不太相同？
- 如何保留变量的值，即使是在软复位后？
- 在调用 ISR 时，如何保存 C 变量？
- 变量名和宏名可以多长？
- 如何访问存储在 PSV 或 EDS 页中的值？
- 为什么需要将数据放入它自己的段？
- 如何将值装入闪存？
- 如何将数据压缩到闪存中？
- 如何定义大数组？
- *另请参见*如何在中断与主干代码之间共享数据？
- *另请参见*如何将变量或函数定位到指定的地址处？
- *另请参见*如何将变量放入程序存储器？
- *另请参见*如何将变量放入片外存储器？
- *另请参见*如何循环移位一个变量？
- *另请参见*如何确定变量和函数定位到何处？

3.4.4.1 为什么浮点结果与所预期的结果不太相同？

首先，确保使用您期望的浮点数据类型。希望使用 IEEE 双精度 （64 位）格式浮点值时，建议在程序中显式地使用 `long double` 类型；希望使用 IEEE 单精度 （32 位）格式值时，建议显式地使用 `float` 类型。默认情况下，编译器对于 `double` 类型使用 IEEE 单精度格式。使用 `-fno-short-double` 开关可指定对于 `double` 类型使用 IEEE 双精度 （64 位）格式。

接下来，要注意浮点格式的局限性和舍入的影响。并不是所有实数都可以用浮点格式精确地表示。例如，分数 1/10 无法精确地用单精度或双精度格式表示。

如果装入或计算的结果为 1/10，以浮点格式存储的值将为该格式可表示的最接近的近似值。在这种情况下，即说明“真实”值根据 IEEE 算术规则被“舍入”为最接近的近似值。在计算早期阶段引入的这种值的微小差异可能随着多次运算而累积，产生明显的误差。一般情况下，计算可能从可精确表示的数字 （如 1 和 10）开始，产生不可精确表示的结果（如 1/10）。这并不是由于编译器选择的生成代码导致的，也不是由于微处理器架构的任何细节导致的，它是 IEEE 浮点格式和算术规则的基本特性。所有编译器 / 微处理器平台都面临同样的问题。关于更多信息，请参见本用户指南的以下章节：

第 8.4 节 “浮点型数据类型”

© 2015 Microchip Technology Inc.

3.4.4.2 如何保留变量的值，即使是在软复位后？

首先，请参见器件数据手册，了解有哪些复位可用。然后，要在软件复位后保存变量的值，可以使用 `persistent` 属性，该属性指定不应在启动时初始化或清零该变量。关于该属性的更多信息，请参见本用户指南的以下章节：

第 8.12 节 “变量属性”

3.4.4.3 在调用 ISR 时，如何保存 C 变量？

您可以使用 `interrupt` 属性的 `save` 参数来保存变量和 SFR，从而可以在从中断返回时恢复它们的值。关于更多信息，请参见本用户指南的以下章节：

第 14.5 节 “中断服务程序现场保存”

3.4.4.4 变量名和宏名可以多长？

对于 MPLAB XC16，没有任何限制，但对于 CCI 则存在一个限制。关于详细信息，请参见本用户指南的以下章节：

第 2.4.5 节 “标识符中的有效初始字符数量”

3.4.4.5 如何访问存储在 PSV 或 EDS 页中的值？

16 位器件具有一种从闪存中访问数据存储器的方法，称为程序空间可视性 （Program Space Visibility， PSV）。可以通过使用 `__psv__` 限定符来访问 PSV 存储区中的值。另一种从程序存储器中访问数据空间的方法称为扩展数据空间 （Extended Data Space，EDS）。可以通过使用 `__eds__` 限定符来访问 EDS 中的值。关于以上每个限定符的更多信息，请参见本用户指南的以下章节：

第 8.11.1 节 “__psv__ 类型限定符”

第 8.11.3 节 “__eds__ 类型限定符”

3.4.4.6 为什么需要将数据放入它自己的段？

MPLAB XC16 目标链接器可以高效地将数据放入段中。但是，您可能希望手动将一些变量组放入一些段中，因为这会比手动将每个变量放在特定地址处简便得多。如果需要，可以在器件链接描述文件内的用户定义段中指定绝对起始地址。要将数据放入它自己的段中，可以使用 `section` 属性，本用户指南的以下章节介绍了该属性：

第 8.12 节 “变量属性”

要编辑链接描述文件内的用户定义段，请参见以下文档。它可从 Microchip 网站 www.microchip.com 下载。

《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）的第 9.5 节 “链接描述文件的内容”

3.4.4.7 如何将值装入闪存？

编译器提供了一些不同的方式来定义闪存变量。

- 变量可以通过相应的 space 属性**显式地**放入闪存。
- 在常量位于代码中 （const-in-code）的默认存储模型中，如果变量具有 C 语言的 const 类型限定符，则它们会被**隐式地**放入闪存中。

这些差异使您可以选择自己要完成多少工作来访问变量，以及希望编译器完成多少工作。仅指定属性 space(prog) 和初始值时，编译器需要完成的工作最少；这会将访问操作 （通常通过 tblrd 指令）留给您 （编程人员）来完成。将 space(prog) 属性与相应的访问限定符（如__eds__或__prog__）配合使用时，编译器要完成的工作最多。

此外，通常会有单页的闪存空间专供 const 限定对象使用。关于更多详细信息，请参见本用户指南以下章节中的 -mconst-in-code：

第 8.12 节 “变量属性”

第 10.4 节 “程序空间中的变量”

3.4.4.8 如何将数据压缩到闪存中？

要将变量的最高字节指定为存储到 space(prog) 段中，可以使用 -mfillupper 选项或 fillupper 变量属性。关于更多信息，请参见本用户指南的以下章节：

第 5.7.1 节 “特定于 16 位器件的选项”（-mfillupper）

第 8.12 节 “变量属性” （fillupper）

3.4.4.9 如何定义大数组？

默认情况下，会为数组分配 32K 的存储空间。如果需要更多，可以使用编译器选项 -mlarge-arrays，但请记住这会产生存储器开销。关于该选项的更多信息，请参见本用户指南的以下章节：

第 10.3.2.3 节 “非自动变量长度限制”

© 2015 Microchip Technology Inc.

3.4.5 函数

本节讨论与函数相关的问题。

- 如何防止删除某个函数？
- 为什么要对函数进行内联？
- 为什么我的函数不发生内联？
- 为什么应将函数放入它自己的段？
- 如何阻止编译器保存或恢复任何寄存器？
- 如何确定某个函数是否被使用？
- 如何确定编译器内包含了哪些函数？
- 传递给函数的参数位于存储器中的哪个位置？
- *另请参见*如何确定某个函数有多大？
- *另请参见*如何确定变量和函数定位到何处？
- *另请参见*如何在 C 中使用中断？

3.4.5.1 如何防止删除某个函数？

对函数应用 `keep` 属性可阻止链接器通过 `--gc-sections` 删除函数，即使在函数未被使用时。请参见本用户指南的以下章节：

第 13.2.2 节 “函数属性”

3.4.5.2 为什么要对函数进行内联？

本用户指南的以下章节介绍了为什么您可能希望对函数进行内联，以及如何这样做：

第 13.6 节 “内联函数”

3.4.5.3 为什么我的函数不发生内联？

除非您使用 `inline` 关键字特别对某个函数进行内联，否则将由编译器决定对哪些函数进行内联。一般情况下，小函数会被内联，而较大的函数不会。关于详细信息，请参见本用户指南的以下章节：

第 13.6 节 “内联函数”

3.4.5.4 为什么应将函数放入它自己的段？

MPLAB XC16 目标链接器可以高效地将函数放入段中。由于手动将函数放入程序存储器可能降低链接器以最高效率完成该工作的能力，所以大多数应用程序都不应通过手动方式将函数放入程序存储器。但是，自举应用程序是特殊的。它们要求应用程序固件在存储器中驻留在比自举程序存储空间高的位置，这需要通过手动方式来放置函数，以避免与自举应用程序发生冲突。

此外，需将代码放入安全段中的应用程序需要使用 boot 或 secure 属性来定制程序函数的放置方式。 address 属性也可以应用于函数。

要使用 `section` 属性将某个函数放入它自己的段，要使用 `boot` 或 `secure` 属性放置某个函数，或要使用 `address` 属性放置某个函数，请参见本用户指南中的以下章节：

第 13.2.2 节 “函数属性”

3.4.5.5 如何阻止编译器保存或恢复任何寄存器？

如果不希望在中断或复位后保存或恢复寄存器值，可以使用 `naked` 属性。但是，使用该属性时需要小心，因为您通常会希望保存这些值。关于详细信息，请参见本用户指南的以下章节：

第 13.2.2 节 “函数属性”

3.4.5.6 如何确定某个函数是否被使用？

在编译项目之后，查看映射文件中已用函数的列表。使用链接器选项 `--gc-sections` 可确保未用函数被删除。关于详细信息，请参见本用户指南的以下章节：

第 13.2.2 节 “函数属性”

关于链接器的更多信息，请参见以下文档。它可从 Microchip 网站 www.microchip.com 下载。

《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

3.4.5.7 如何确定编译器内包含了哪些函数？

通过运行并停止预处理器，然后检查输出，您可以查看编译器的预定义符号/宏和函数。本用户指南的以下章节介绍了实现该操作的选项：

5.7.2 “用于控制输出类型的选项”

3.4.5.8 传递给函数的参数位于存储器中的哪个位置？

您需要通过运行编译器代码来确定这一点。在运行编译器时，查看应用程序二进制接口。此外，一些内置函数可能可以提供帮助。请参见本用户指南的以下章节：

附录 G “内置函数”

© 2015 Microchip Technology Inc.

3.4.6 中断

本节讨论中断和中断服务程序 （ISR）问题。

- 如何在 C 中使用中断？
- 如何向项目中添加陷阱中断向量？
- 我的应用程序是否能够 / 应当从陷阱返回？
- 如何在两个中断程序之间共享数据？
- 什么是默认中断，它在哪里定义，以及如何使用它？
- *另请参见*如何让中断程序更快？
- *另请参见*如何在中断与主干代码之间共享数据？
- *另请参见*在调用 ISR 时，如何保存 C 变量？

3.4.6.1 如何在 C 中使用中断？

首先，需要了解目标器件上提供的中断硬件。16 位器件实现了几个独立的中断向量单元，并使用优先级方案。详细信息，请参见器件数据手册。然后，阅读本用户指南的以下章节来了解更多信息：

第 14 章 “中断”

3.4.6.2 如何向项目中添加陷阱中断向量？

编译器处理硬陷阱的方式与正常中断向量相同，它们的名称与外设处理程序的名称非常类似。在安装文件夹的 `docs` 文件夹中，可以找到特定器件支持的所有中断函数：`vector_index.html`。

一般格式为：

```
void __attribute__((interrupt)) ISR_fn_name(void) {
}
```

3.4.6.3 我的应用程序是否能够 / 应当从陷阱返回？

这个问题是取决于应用程序 / 陷阱。一般的答案是当发生此类事件时，可能应当安全地重新启动应用程序。

3.4.6.4 如何在两个中断程序之间共享数据？

由于 ISR 本身的性质，ISR 不发送结果或接收参数。共享数据的唯一方法是使用公共数据共享过程。这种情况的示例是使用可变全局变量或使用专用访问器函数，这些函数可以小心地控制对数据的访问，使您的应用程序变得更稳健。

每当横跨不同控制线程共享数据（这就是中断程序的真正面目）时，重要的是对共享数据访问进行保护，阻止之后发生的中断破坏数据，因为并不是所有访问都是原子的。

3.4.6.5 什么是默认中断，它在哪里定义，以及如何使用它？

当不存在其他命名向量时，“默认中断”将填充向量表。如果它未定义，编译器会创建一个将在调试环境中暂停或在正常执行 （非调试）环境中复位的中断。

您可以通过简单地定义名为 `_DefaultInterrupt` 的 ISR 来定义您自己的处理程序。

3.4.7 汇编代码

本节讨论在 C 项目中编写汇编代码时会出现的问题。

- 如何混合使用汇编代码和 C 代码？
- 汇编源文件中除了指令之外，还需要什么？
- 如何从汇编代码中访问 C 对象？
- 如何从汇编代码内访问 SFR？
- 何时应混合使用汇编代码和 C 代码？
- .s 和 .S 文件之间的区别是什么？
- 如何为汇编模块制作函数包装器？
- 何时应使用行内汇编代码而不是汇编模块？

3.4.7.1 如何混合使用汇编代码和 C 代码？

汇编代码可以编写为从 C 代码中调用的独立函数，也可以编写为处于 C 代码中的行内代码。关于详细信息，请参见本用户指南的以下章节：

第 16 章 “混合使用 C 代码和汇编代码”

3.4.7.2 汇编源文件中除了指令之外，还需要什么？

汇编代码通常需要汇编器伪指令，以及指令本身。以下文档介绍了这些伪指令的操作。它可从 Microchip 网站 www.microchip.com 下载。

《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

有两个在汇编代码中常用的伪指令。第一个是 `.section` 伪指令。所有汇编代码都必须使用该伪指令放在某个段中，以便链接器可以将它作为一个整体处理和放入存储器。第二个是 `.global` 伪指令。该伪指令用于使符号可被多个源文件访问。

3.4.7.3 如何从汇编代码中访问 C 对象？

大多数 C 对象都可从汇编代码中进行访问。C 源代码中使用的符号和基于该源代码生成的汇编代码中使用的那些符号之间存在一种映射。汇编代码应访问汇编等效符号，本用户指南中的以下章节详细介绍了这些符号：

第 16.2 节 “混合使用汇编语言与 C 语言变量及函数”

3.4.7.4 如何从汇编代码内访问 SFR？

在汇编代码中访问 SFR 的最简单方法是使用器件通用包含文件（`xc.h`），该文件会将相应的 SFR 地址赋值给这些符号。

无法保证您将能够访问通过编译 C 代码（即便是访问您所需 SFR 的代码）而生成的符号。请参见本用户指南的以下章节：

第 13.8 节 “函数调用约定”

© 2015 Microchip Technology Inc.

3.4.7.5 何时应混合使用汇编代码和 C 代码？

这是一个非常依赖于应用程序的问题。有一些特定于器件的操作无法用常规 C 代码完成，通常语言工具会通过内置函数来提供该功能。

如果您决定混合使用汇编和 C 代码，请确保代码符合运行时模型，即，在正确的寄存器中传递参数；并且正确地使用寄存器，使之不会被覆盖。

3.4.7.6 .s 和 .S 文件之间的区别是什么？

这两种文件都包含汇编语言。 `.S` 文件由 C 编译器进行预处理。这意味着它们可能包含 C 预处理语句（`#define` 和 `#ifdef` 等），但它们不应包含 C 语句。关于这些汇编文件的信息，请参见本用户指南的以下章节：

第 5.5.1 节 “输出文件”

3.4.7.7 如何为汇编模块制作函数包装器？

C 编译器期望所有 C 可见名称都以一个前导下划线开始。要将某个函数从汇编代码导出到 C 代码，它必须全局可见。当然，在 C 代码中还应具有一个外部原型，以便编译器可以正确地看到它。

```
foo.s:

    .text
    .global _foo
_foo:  retlw #0,w0

main.c:

    extern int foo(void);
```

关于将 C 代码与汇编模块配合使用的详细信息，请参见本用户指南的以下章节：

第 16.2 节 “混合使用汇编语言与 C 语言变量及函数”

3.4.7.8 何时应使用行内汇编代码而不是汇编模块？

如果编程人员决定混合使用汇编代码和 C 代码，请确保代码符合运行时模型；即正确地使用寄存器，使之不会被覆盖；并且代码使用 GNU 扩展行内汇编代码。长序列通常难以调试，因此请确保正确地遵循准则。

3.5 让应用程序执行所需的操作

本节提供了编程技术、应用程序和示例。此外，它还讨论了与让应用程序执行特定任务相关的问题。

- 如何生成调试信息？
- 为什么 MPLAB X IDE 反汇编窗口中没有反汇编信息？
- 如何在中断与主干代码之间共享数据？
- 如何在将代码编程到器件中之后保护代码？
- 如何在使用 Printf 时重定向标准 I/O？
- 如何将变量放入片外存储器？
- 如何在代码中实现延时？
- 如何循环移位一个变量？

3.5.1 如何生成调试信息？

对于编译器和汇编器，命令行选项 `-g` 用于生成调试信息。关于详细信息，请参见以下文档（可从 Microchip 网站访问），并请参见本用户指南的以下章节：

《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）

第 5.7.5 节 “用于调试的选项”

3.5.2 为什么 MPLAB X IDE 反汇编窗口中没有反汇编信息？

您必须先使能生成调试信息，然后才能在反汇编窗口中看到信息。请参见本用户指南的以下章节：

第 3.5.1 节 “如何生成调试信息？”

3.5.3 如何在中断与主干代码之间共享数据？

同时从中断和主干代码访问的变量可以很容易被程序损坏或误读。`volatile` 限定符指示编译器避免对此类变量执行优化。这可以解决与该问题相关的一些问题。

由于访问变量的方式产生的其他问题会因语句而不同。因此，通常最好是在主干代码中访问变量之前禁止中断，之后再重新允许中断，从而完全避免这些问题。关于这些解决方法的更多信息，请参见本用户指南的以下章节：

第 8.10.2 节 “volatile 类型限定符”

第 14.7 节 “允许 / 禁止中断”

3.5.4 如何在将代码编程到器件中之后保护代码？

许多带有闪存程序存储器的器件允许对全部或部分存储器进行写保护。为此需要正确地设置器件配置位。关于使用配置位的更多信息，请参见本用户指南的以下章节：

第 6.5 节 “配置位访问”

第 2.5.14 节 “指定配置位”

对于这个问题，器件数据手册也是一个很好的资源。

© 2015 Microchip Technology Inc.

3.5.5 如何在使用 Printf 时重定向标准 I/O？

printf 函数可以完成两件事：它基于您指定的格式字符串和占位符格式化文本，并将该格式化文本发送（打印）到目标（或流）。例如，您可以选择将 printf 输出发送到 LCD、SPI 模块或 USART。关于使用 ANSI C 函数 printf，包括如何定制输出以将它发送至另一个外设的更多信息，请参见以下文档。它可从 Microchip 网站 www.microchip.com 下载。

《16 位语言工具函数库》（DS51456D_CN）

3.5.6 如何将变量放入片外存储器？

要将变量定位到片外存储器，请使用 __external__ 类型限定符。要将变量定位到通过并行主端口 （PMP）访问的片外存储器中，请使用 __pmp__ 类型限定符。

访问这些变量需要的时间会多于内部数据存储器中的变量。关于访问片外存储器的详细信息，请参见本用户指南的以下章节：

第 8.11.6 节 “__external__ 类型限定符”

第 8.11.5 节 “__pmp__ 类型限定符”

3.5.7 如何在代码中实现延时？

使用器件定时器来产生延时是最好的方法。如果没有定时器可用，则可以使用库函数 _delay32、__delay_ms 或 __delay_us，如以下文档所述。它可从 Microchip 网站 www.microchip.com 下载。

《16 位语言工具函数库》（DS51456D_CN）

3.5.8 如何循环移位一个变量？

C 语言没有循环移位操作符，但可以通过移位和按位或操作符来执行循环移位。关于更多信息，请参见本用户指南的以下章节：

第 2.4.10 节 “有符号值的按位运算”

第 2.4.11 节 “右移有符号值”

3.6 了解编译过程

本节介绍如何了解编译器在编译过程中做了些什么、它如何对输出代码进行编码、对象放置在何处等方面。此外，还讨论了编译器支持的功能。

- 许可授权是如何影响功能和优化级别的？
- 如何让代码更小？
- 如何减少 RAM 使用量？
- 如何让代码更快？
- 如何控制语言工具将对象存放在存储器中的位置？
- 如何让中断程序更快？
- C 变量最大可以多大？
- 代码将被应用哪些优化？
- 编译器支持哪些器件？
- 如何知道编译器生成哪些代码？
- 如何确定某个函数有多大？
- 如何确定变量和函数定位到何处？
- 如何正确地保留存储器？
- 如何知道还有多少存储器可用？
- 默认情况下会包含哪些库？
- 如何创建我自己的库？
- 在选择调试器时，为什么会出现存储器不足错误？
- *另请参见*如何确定警告 / 错误消息的含义？
- *另请参见*如何编译库？
- *另请参见* MPLAB X IDE 调试编译有何差别？
- *另请参见*如何防止删除某个函数？

3.6.1 许可授权是如何影响功能和优化级别的？

不同的许可证可用的功能和优化不同。请参见本用户指南的以下章节：

第 18 章 “优化”

© 2015 Microchip Technology Inc.

3.6.2 如何让代码更小？

生成较小代码的一般建议：

- 不要混合使用数据类型。
- 以本机字宽度定义索引变量。
- 当整型足够时，不要使用浮点型变量。使用浮点型变量时，考虑使用较小的数学库。
- 可以将编译器选项 `-mpa` 与任意优化级别配合使用，以减小代码大小。请参见本用户指南的以下章节：

  第 18 章 “优化”

注： 经过优化的代码可能较难调试。

- 在初始化 SFR 时，使用 SFR 的完整名称而不是位名称。您可以使用这种技术一次初始化若干个位域。
- 将共用代码重新组织为函数，而不是逐句将代码复制到程序中的几个位置。
- 使用小代码、小标量和大数据存储模型。请参见本用户指南的以下章节：

  第 10.14 节 “存储模型”

3.6.3 如何减少 RAM 使用量？

尝试使用以下建议来减少 RAM 使用量：

1. 一些减小代码大小的相同建议对于减少 RAM 使用量也很有用，请参见本用户指南的以下章节：

   第 3.6.2 节 “如何让代码更小？”

2. 传递引用大对象的指针，而不是向函数或从函数传递这些对象（以节约堆栈资源）。
3. 在整个程序中不需要进行更改的对象可以通过使用 `-mconst-in-code` 选项和 `const` 限定符放入程序存储器。这可以释放宝贵的 RAM，但会降低执行速度。请参见本用户指南的以下章节：

   第 10.4 节 “程序空间中的变量”

3.6.4 如何让代码更快？

尝试使用以下建议来提高代码执行速度：

1. 代码长度越短，代码速度就会越快。减少执行一项任务所需的机器指令数将使这项任务的执行速度更快。关于产生更小和更快代码的更多信息，请参见本用户指南的以下章节：

   第 3.6.2 节 “如何让代码更小？”

   第 3.6.6 节 “如何让中断程序更快？”

2. 根据您的编译器许可证，您可能可以使用更高的优化级别来生成更快的代码。关于详细信息，请参见本用户指南的以下章节：

   第 18 章 “优化”

   第 5.7.6 节 “用于控制优化的选项”

3. 算法选择对于解决方案的代码大小和速度的影响会比所有其他因素更大。请选择适合于您的代码的适当算法。

3.6.5 如何控制语言工具将对象存放在存储器中的位置？

在大多数情况下，应允许语言工具为对象分配存储空间。如果您仍然希望控制对象的存储空间分配，请参见本用户指南的以下章节：

第 3.4.3 节 “存储器分配”

3.6.6 如何让中断程序更快？

尝试使用以下建议来提高 ISR 执行速度：

1. 代码长度越短，代码速度往往就越快。关于详细信息，请参见本用户指南的以下章节：

   第 3.6.2 节 “如何让代码更小？”

2. 提高代码速度的建议也适用于 ISR 代码。关于详细信息，请参见本用户指南的以下章节：

   第 3.6.4 节 “如何让代码更快？”

3. 考虑让 ISR 只是简单设置一个标志并返回。然后，可以在主干代码中通过检查该标志来处理中断。这种方式的优点是可以将复杂的中断处理代码移出 ISR，从而它不会再增加寄存器使用量。对于由中断和主干代码共享的变量，总是使用 `volatile` 限定符。请参见本用户指南的以下章节：

   第 8.10.2 节 “volatile 类型限定符”

   第 3.5.3 节 “如何在中断与主干代码之间共享数据？”

3.6.7 C 变量最大可以多大？

这个问题特别涉及到各个 C 对象 （如数组或结构）的大小。所有变量的总大小是另一回事。

要回答这个问题，需要知道变量将位于哪个存储空间中。使用 `-mconst-in-code` 选项时，使用 `const` 限定的对象将位于程序存储器中；其他对象将被放入数据存储器。本用户指南的以下章节介绍了程序存储器对象大小：

第 10.4.3 节 “程序存储器变量的长度限制”

数据存储器中的对象可大致分为“自动”和“非自动”。这些对象具有大小限制。关于自动和非自动变量及大小限制的更多信息，请参见本用户指南的以下章节：

第 10.3 节 “数据存储空间中的变量”

第 10.3.3.3 节 “自动变量长度限制”

第 10.3.2.3 节 “非自动变量长度限制”

3.6.8 代码将被应用哪些优化？

可用的代码优化取决于您的编译器许可证。关于更多信息，请参见本用户指南的以下章节：

第 18 章 “优化”

第 5.7.6 节 “用于控制优化的选项”

© 2015 Microchip Technology Inc.

3.6.9 编译器支持哪些器件？

通常在每次发布编译器版本时提供对新器件的支持。要了解您的编译器是否支持某个器件，请参见本用户指南的以下章节：

第 6.2 节 “器件支持”

3.6.10 如何知道编译器生成哪些代码？

可以使用汇编器列表文件选项设置汇编列表文件，使之包含关于代码的各种信息。这些信息可以包含几乎整个程序的汇编输出、链接到程序中的库程序、段信息、符号列表，以及更多。

列表文件可使用如下方式生成：

- 在命令行上，使用选项创建一个基本列表文件：
  `-Wa, -a=`*`projectname`*`.lst`
- 对于 MPLAB X IDE，右键单击您的项目并选择 “Properties”（属性）。在 Project Properties 窗口中，在 “Categories”（类别）下单击 “xc16-as” 。从 “Option categories”（选项类别）中，选择 “Listing file options”（列表文件选项），并确保选中 “List to file”（列表至文件）。

默认情况下，汇编列表文件将具有 `.lst` 扩展名。

关于列表文件的信息，请参见以下文档。它可从 Microchip 网站 www.microchip.com 下载。

《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

3.6.11 如何确定某个函数有多大？

可以通过汇编列表文件确定某个函数的大小（对于该函数生成的汇编代码量）。请参见本用户指南的以下章节：

第 3.6.10 节 “如何知道编译器生成哪些代码？”

3.6.12 如何确定变量和函数定位到何处？

xc16-objdump 实用程序可显示有关一个或多个目标文件的信息。使用 `-t` 选项可打印文件的符号表条目。

此外，您还可以通过链接器生成的映射文件确定变量和函数被定位到何处。映射文件中只会显示全局符号。

C 标识符和汇编代码中使用的符号之间存在一种映射。与某个变量关联的符号将被分配该变量最低字节的地址；对于函数，它是对于该函数生成的第一条指令的地址。关于 xc16-objdump 和链接器映射文件的更多信息，请参见以下文档。它可从 Microchip 网站 www.microchip.com 下载。

《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）

3.6.13 如何正确地保留存储器？

通过在链接描述文件中创建特定的段或通过使用属性来阻止编译器使用存储器段，可以保留存储器。如果未使用以上方法之一来保留存储器，则可能不会保留您认为已保留的存储器，链接器可能会在该区域中放置对象。关于保留存储器的更多信息，请参见以下文档。它可从 Microchip 网站 www.microchip.com 下载。另外，请参见本用户指南的以下章节。

《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）

第 3.4.3.4 节 “如何阻止编译器使用某些特定的存储单元？”

3.6.14 如何知道还有多少存储器可用？

在编译后，可通过选项 --report-mem 或在 MPLAB X IDE 的 Dashboard （仪表板）窗口中查看编译器输出的存储器使用情况摘要。所有这些摘要会指示已用的存储器量和仍然可用的量，但都不会指示该存储器是一个连续块还是分成许多小块。小的空闲存储器块不能用于较大的对象，所以即使可用的存储器总量明显足以用于要定位的对象，也可能会产生存储器不足的错误。

要确定每个链接器类中究竟哪些存储器仍然可用，请查看链接器映射文件。如果存在存储器分页，则该文件还会指示该类中最大的连续块。关于映射文件的信息，请参见以下文档。它可从 Microchip 网站 www.microchip.com 下载。

《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）

3.6.15 默认情况下会包含哪些库？

在编译时，编译器会自动在编译过程中包含任何适用的标准库。所以，您完全不需要控制这些文件。但是，有一些库您必须要记住包含，例如编译器未随附的任何库。用户可以通过检查映射文件确定在产生的编译映像中使用了哪些标准库。从标准库中包含的归档成员会被列入一个列表中，该列表与导致在映射内包含特定标准库归档的符号相关联。关于标准库的详细信息，请参见以下文档。它可从 Microchip 网站 www.microchip.com 下载。

《16 位语言工具函数库》（DS51456D_CN）

3.6.16 如何创建我自己的库？

要使用自己或同事编译的一个或多个库文件，请在命令行上将它们包含在要编译的文件列表中。库文件可以在文件列表中相对于源文件的任意位置指定。但是，如果有多个库文件，则会按命令行中所指定的顺序进行搜索。

在命令行上指定库 `lib`*`library`*`.a` 的一个示例：

```
xc16-gcc -mcpu=33FJ256GP710 main.c int.c lib library.a
```

如果希望使用 `-l` 选项，则：

```
xc16-gcc -mcpu=33FJ256GP710 main.c int.c -llibrary
```

如果要使用 MPLAB X IDE 来编译项目，则将库文件添加到项目中的 Libraries 文件夹中，并按照对它们进行搜索的顺序添加。IDE 将确保在编译序列中的适当时间点将它们传递给编译器。关于如何编译您自己的库文件的信息，请参见本用户指南的以下章节：

第 5.4.1.2 节 “用户定义的库”

3.6.17 在选择调试器时，为什么会出现存储器不足错误？

如果您使用硬件工具调试器（例如 PICkit 3 在线调试器、 MPLAB ICD 3 在线调试器或 MPLAB REAL ICE 在线仿真器），进行调试会需要一些 RAM。请参见本用户指南的以下章节：

第 3.5.2 节 “为什么 MPLAB X IDE 反汇编窗口中没有反汇编信息？”

© 2015 Microchip Technology Inc.

3.7 修复无法工作的代码

本节讨论与由于编译器错误而无法编译或可以编译但无法按预期工作的项目相关的问题。

- 如何确定警告 / 错误消息的含义？
- 如何在程序中查找导致编译器错误或警告的代码？
- 如何阻止产生警告？
- 如何知道堆栈是否溢出？
- 使用中断时，什么原因会导致变量损坏和代码失败？
- *另请参见*调用编译器。
- *另请参见*如何正确地保留存储器？

3.7.1 如何确定警告 / 错误消息的含义？

大多数警告和错误消息都是不言自明的；但是，有些消息需要进一步的说明。下面引用的附录中介绍了所有 MPLAB XC16 警告和错误消息。此外，本用户指南的以下章节包含了介绍如何控制消息输出的内容：

附录 C “诊断”

第 5.7.4 节 “用于控制警告和错误的选项”

3.7.2 如何在程序中查找导致编译器错误或警告的代码？

在大多数情况下，如果错误是与源代码相关的语法错误，则编译器产生的消息会指示引起问题的代码行。如果在 MPLAB X IDE 中进行编译，则可以双击消息，让编辑器将您带到引起问题的行。但确定引起问题的代码并不总是那么容易。

在某些情况下，所报告的错误位于需要注意的代码行之后。这是因为 C 语言允许 C 语句跨越源文件的多行。编译器有可能直到开始扫描下一条语句后才能确定前面的代码行存在错误。假设存在以下代码：

```
input = PORTB   // oops - forgot the semicolon
if(input>6)
  // ...
```

赋值语句缺少分号这一错误被标记在包含 `if()` 语句的下一行上。

在其他情况下，错误可能来自汇编器，而不是编译器。如果正在编译的源代码是汇编模块，则错误会直接指示引发错误的汇编代码行。

最后，还有一些错误不与任何特定代码行相关。编译器选项错误或链接器错误就是这些错误的示例。

如果需要查看由编译器生成的汇编代码，可以在汇编列表文件中查找。关于链接器尝试将对象定位到何处的信息，请参见映射文件。关于列表文件和映射文件的信息，请参见以下文档。它可从 Microchip 网站 www.microchip.com 下载。

《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）

3.7.3 如何阻止产生警告？

一般情况下，不应该忽略警告。警告指示可能会导致代码失败的情况。请总是检查您的代码，以确认它不是可能的错误源。

但是，如果您觉得自己希望禁止警告消息，可以执行以下操作：

- 通过使用选项的 `-Wno-` 版本来禁止特定警告。
- 通过 `-w` 选项禁止所有警告。
- 在 MPLAB X IDE 中，在 Project Properties 窗口的每个工具类别下禁止警告。此外，还可以查看 Tool Options（工具选项）窗口中， Embedded（已安装工具）按钮下的 Suppressible Messages （可禁止的消息）选项卡。

关于详细信息，请参见本用户指南的以下章节：

第 5.7.4 节 “用于控制警告和错误的选项”

3.7.4 如何知道堆栈是否溢出？

16 位器件使用可以在 SPLIM 寄存器中设置上边界地址的堆栈。因此，可以通过设置堆栈深度来防止溢出。

溢出以及其他堆栈错误可以产生陷阱并在代码中识别到。关于使用软件堆栈的更多信息，请参见本指南的以下章节：

第 10.3.3.1 节 “软件堆栈”

第 10.3.3.2 节 “C 堆栈使用”

器件数据手册中关于软件堆栈的章节

3.7.5 使用中断时，什么原因会导致变量损坏和代码失败？

这通常是由于中断和主干代码同时使用变量而引起的。如果编译器优化了对某个变量的访问，或者访问被中断程序中断，则可能会发生损坏。请参见本用户指南的以下章节：

第 3.5.3 节 “如何在中断与主干代码之间共享数据？”

© 2015 Microchip Technology Inc.

第 4 章 XC16 工具链和 MPLAB X IDE

4.1 简介

16 位语言工具可以与 MPLAB X IDE 配合使用，为 dsPIC® DSC 和 PIC24 MCU 器件系列的应用程序代码提供 GUI 开发环境。这些工具包括：

- MPLAB XC16 C 编译器
- MPLAB XC16 汇编器
- MPLAB XC16 目标链接器
- MPLAB XC16 目标归档器 / 库管理器和其他 16 位实用程序

本章讨论的主题包括：

- MPLAB X IDE 和工具安装
- MPLAB X IDE 设置
- MPLAB X IDE 项目
- 项目设置
- 项目示例

4.2 MPLAB X IDE 和工具安装

为了配合使用 16 位语言工具与 MPLAB X IDE，必须安装：

- MPLAB X IDE，它可从 Microchip 网站上免费获取。
- MPLAB XC16 C 编译器，它包含所有 16 位语言工具。该编译器可从 Microchip 网站上免费获取 （免费与评估许可证级别）或购买 （标准或专业许可证级别）。

默认情况下， 16 位语言工具将安装在以下目录中：

- Windows 操作系统 32 位——`C:\Program Files\Microchip\xc16\`*`x.xx`*
- Windows 操作系统 64 位——`C:\Program Files (x86)\Microchip\xc16\`*`x.xx`*
- Mac 操作系统——`Applications/microchip/xc16/`*`x.xx`*
- Linux 操作系统——`/opt/microchip/xc16/`*`x.xx`*

其中， *x.xx* 是版本号。

每个工具的可执行文件将位于 `bin` 子目录中：

- C 编译器——`xc16-gcc.exe`
- 汇编器——`xc16-as.exe`
- 目标链接器——`xc16-ld.exe`
- 目标归档器 / 库管理器——`xc16-ar.exe`
- 其他实用程序——`xc16-`*`utility`*`.exe`

所有器件包含 （头）文件均位于 `support/`*`family`*`/h` 子目录，其中的 *family* 是对应于选定器件的器件系列 （例如， dsPIC30F 是对应于 dsPIC30F6015 器件的器件系列）。关于这些文件的更多信息，请参见**第 6.3 节 “器件头文件”**。

所有器件连接器脚本文件均位于 `support/family/gld` 子目录，其中的 *`family`* 是对应于选定器件的器件系列（例如，dsPIC30F 是对应于 dsPIC30F6015 器件的器件系列）。关于这些文件的更多信息，请参见链接器文档。

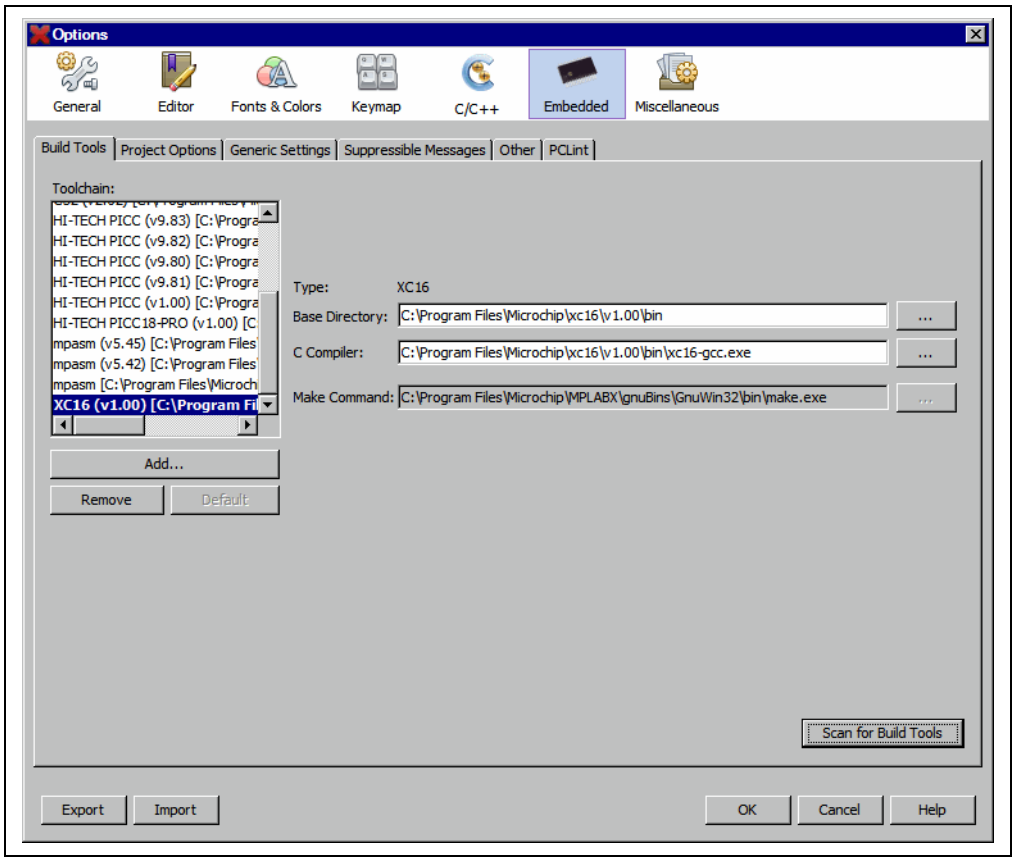
代码示例和模板文件位于 `support/templates/assembly` 子目录。

4.3 MPLAB X IDE 设置

在 PC 上安装 MPLAB X IDE 之后，启动该应用程序，并检查下面的设置，以确保正确识别 16 位语言工具。

1. 从 MPLAB X IDE 菜单栏中，选择 *Tools（工具）>Options（选项）*打开 Options 对话框。单击“Embedded”按钮，并选择“Build Tools”（编译工具）选项卡。
2. 单击“Tool Collection”（工具集合）下的“XC16”。确保安装的路径正确。
3. 单击**确定**。

图 4-1：　　Windows® 操作系统中的 XC16 工具套件位置

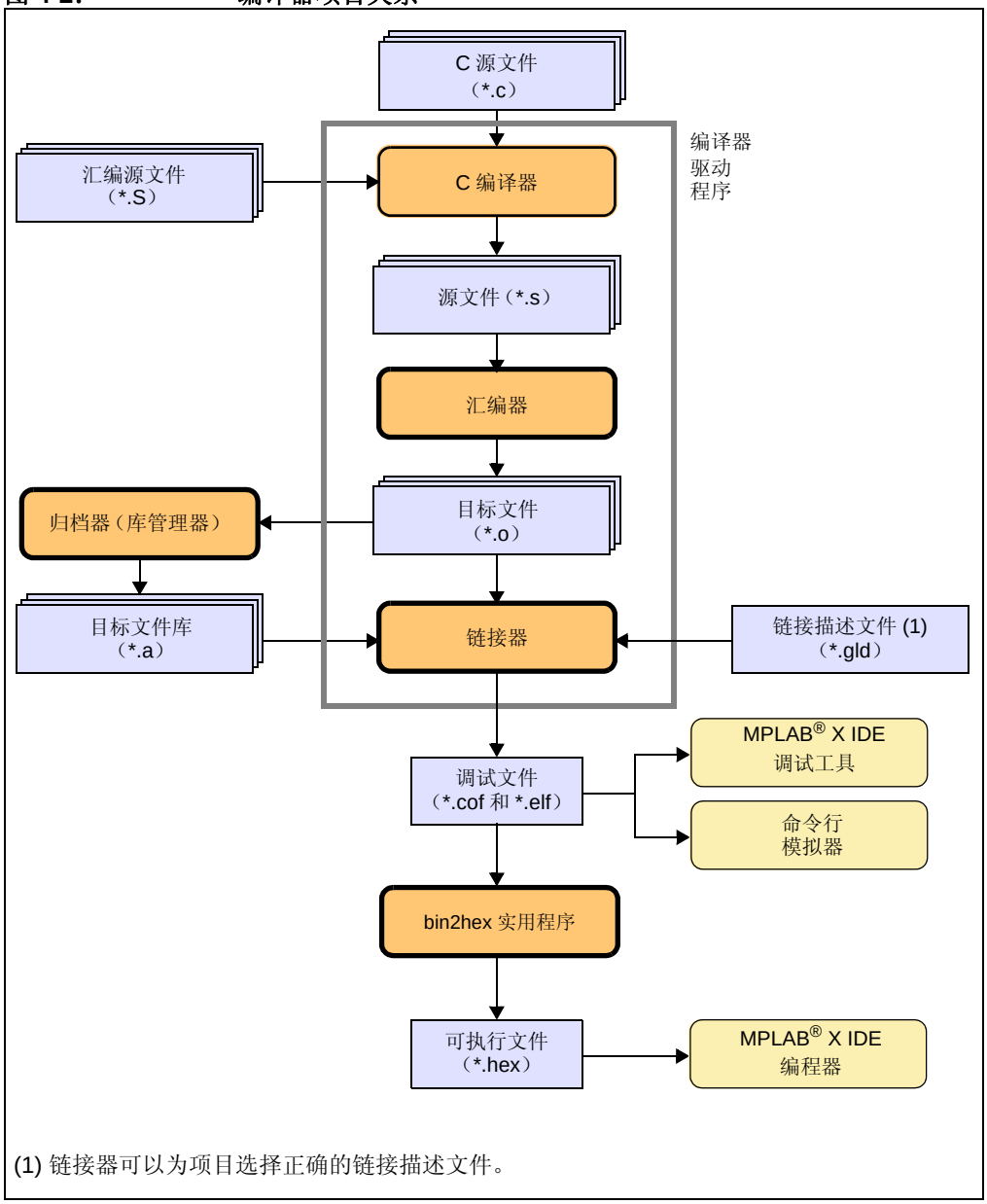


© 2015 Microchip Technology Inc.

4.4 MPLAB X IDE 项目

MPLAB X IDE 中的项目是编译应用程序所需的一组文件，以及这些文件与各种编译工具的关联。下面是一个常规 MPLAB X IDE 项目。

图 4-2：　　编译器项目关系



(1) 链接器可以为项目选择正确的链接描述文件。

在该 MPLAB X IDE 项目中，C 源文件为编译器的输入。编译器将生成输入到汇编器的源文件。关于编译器的更多信息，请参见编译器文档。

汇编源文件为 C 预处理器的输入。产生的源文件是汇编器的输入。汇编器将生成目标文件，作为链接器或归档器的输入。关于汇编器的更多信息，请参见汇编器文档。

目标文件可以使用归档器 / 库管理器归档到库中。关于归档器的更多信息，请参见归档器 / 库管理器文档。

目标文件和所有库文件，以及链接描述文件（会自动添加通用链接描述文件）用于通过链接器生成项目输出文件。链接器生成的输出文件为 ELF 或 COF 文件，由模拟器和调试工具使用。该文件可以输入到 bin2hex 实用程序来生成可执行文件 （.hex）。关于链接描述文件和使用目标链接器的更多信息，请参见链接器文档。

关于项目的更多信息，请参见 MPLAB X IDE 文档。

© 2015 Microchip Technology Inc.

4.5 项目设置

要首次设置 MPLAB X IDE 项目，请使用内置的 Project Wizard（项目向导）（*File>New Project*（新建项目））。在该向导中，可以选择使用 16 位语言工具的语言工具套件。关于该向导和 MPLAB X IDE 项目的更多信息，请参见 MPLAB X IDE 文档。

建立项目之后，可以在 MPLAB X IDE 中设置工具的属性。

1. 从 MPLAB X IDE 菜单栏中，选择 *File>Project Properties* 打开一个窗口来设置 / 检查项目编译选项。
2. 在 “Conf:[*default*]”下，从工具集合中选择要设置的工具。
   - XC16 (Global Options)
   - xc16-as （16 位汇编器）
   - xc16-gcc （16 位 C 编译器）
   - xc16-ld （16 位链接器）

4.5.1 XC16 (Global Options)

设置所有 16 位语言工具的全局选项。另请参见 “选项页面功能”。

表 4-1： ALL OPTIONS （所有选项）类别

选项	说明	命令行
Output file format （输出文件格式）	选择 ELF/DWARF 或 COFF。	`-omf=elf` `-omf=cof`
Define common macros （定义通用宏）	向编译器、汇编器和链接器添加通用宏。	`-D`*`macro`*
Generic build （通用编译）	针对通用内核器件 （无外设）进行编译。	
Use legacy libc （使用旧式库）	选中则使用 v3.25 之前格式的库。 取消选中则使用新的 （HI-TECH）库格式。	`-legacy-libc`
Fast floating point math （快速浮点运算）	选中则使用速度较快的单精度和双精度浮点库，它们会消耗更多的 RAM。 取消选中则使用原始库，它们较慢，但产生的代码较小。	`-fast-math`
Relaxed floating point math （放宽浮点运算）	选中则使用放宽遵从性的数学库。这是一个遵循的规则与 IEEE 标准对于无穷大、 NaN 和非规格化 （极小）数规定的规则略有不同的数学库。 取消选中则使用标准浮点数学库。	`-relaxed-math`
Don’t delete intermediate files （不删除中间文件）	选中则不删除中间文件。将它们放在目标文件目录中，并基于源文件命名它们。 取消选中则在编译后删除中间文件。	`-save-temps=obj`
Common include dirs （公共包含目录）	在此处输入的目录路径会被追加到编译器现有包含路径后。 相对路径是相对于 MPLAB X IDE 项目目录。	`-Idir`

4.5.2 xc16-as （16 位汇编器）

可以在 MPLAB X IDE 中指定命令行选项的一个子集。选择某个类别，然后设置汇编器选项。关于更多选项，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。另请参见 “选项页面功能”。

表 4-2： GENERAL OPTIONS （常规选项）类别

选项	说明	命令行
Define ASM macros (.S only) （定义 ASM 宏（仅对于 .S））	添加汇编器宏。	`-Dmacro`
Assembler symbols （汇编器符号）	将符号 “sym”定义为给定值 “value”。	`--defsym sym=value`
ASM include dirs （ASM 包含目录）	将某个目录添加到汇编器搜索 `.include` 伪指令中指定文件的目录列表中。 更多信息，请参见**第 4.5.7 节 “附加的搜索路径和目录”**。	`-I"dir"`
Preprocessor include dirs （预处理器包含目录）	将某个目录添加到编译器预处理器搜索 `.include` 伪指令中指定文件的目录列表中。 更多信息，请参见**第 4.5.7 节 “附加的搜索路径和目录”**。	`-I"dir"`
Allow call optimization （允许调用优化）	选中则开启放宽 （relaxation）。 取消选中则关闭放宽。	`--relax` `--no-relax`
Keep local symbols （保留局部符号）	选中则保留局部符号，即以 `.L` （仅大写）开头的标号。 取消选中则丢弃局部符号。	`--keep-locals （-L）`

Diagnostics level （诊断级别）	选择要在 Output （输出）窗口中显示的警告。
	- Generate warnings （生成警告）
	- Suppress warnings （禁止警告）
	- Fatal warnings （致命警告）
		
		`--warn`
		`--no-warn`
		`--fatal-warnings`

Additional driver options （额外的驱动程序选项）	输入 GUI 中不存在的任何额外驱动程序选项。在此处引入的字符串将像在驱动程序调用命令中一样发出。	

表 4-3： LISTING FILE OPTIONS （列表文件选项）类别

选项	说明	命令行
Include source code （包含源代码）	选中则产生高级语言列表。高级列表要求由编译器生成汇编源代码、向编译器传递诸如 `-g` 之类的调试选项，并请求产生汇编列表 （`-al`）。 取消选中则产生普通列表。	`-ah`
Include macros expansions （包含宏展开）	选中则展开列表中的宏。 取消选中则折叠宏。	`-am`
Omit false conditionals （省略虚假条件语句）	选中则在列表中省略为 false 的条件分支代码 （`.if` 和 `.ifdef`）。 取消选中则包含为 false 的条件分支代码。	`-ac`
Omit forms processing （省略表单处理）	选中则关闭由列表伪指令 `.psize`、`.eject`、`.title` 和 `.sbttl` 执行的所有表单处理。 取消选中则由列表伪指令进行处理。	`-an`
Include assembly （包含汇编）	选中则包含汇编列表。该 `-a` 子选项可与其他子选项配合使用。 取消选中则排除汇编列表。	`-al`

© 2015 Microchip Technology Inc.

表 4-3： LISTING FILE OPTIONS （列表文件选项）类别 （续）

选项	说明	命令行
Include symbols （包含符号）	选中则包含符号表列表。 取消选中则从列表中排除符号表。	`-as`
Omit debugging directives （省略调试伪指令）	选中则从列表中省略调试伪指令。这可以使列表更简洁。 取消选中则包含调试伪指令。	`-ad`
Include section information （包含段信息）	选中则显示关于每个代码段和数据段的信息。该信息包含每个段的大小的详细信息，之后是程序存储器和数据存储器总使用量的信息。 取消选中则不显示该信息。	`-ai`
List to file （列表至文件）	选中则将列表信息发送到一个文件。 取消选中则将列表信息发送到 Output 窗口。	`-a=`*`asmfilename`*`.lst`

4.5.3 xc16-gcc （16 位 C 编译器）

虽然 MPLAB XC16 C 编译器与 MPLAB X IDE 配合工作，但它必须单独购买。可以购买完整版本，也可以免费下载学生 （有限功能）版本。关于详细信息，请参见 Microchip 网站 （www.microchip.com）。

可以在 MPLAB X IDE 中指定命令行选项的一个子集。选择某个类别，然后设置编译器选项。关于其他选项，请参见《MPLAB® XC16 汇编器，链接器和实用程序用户指南》（DS52106A_CN），它也可从 Microchip 网站上获取。

另请参见 “选项页面功能”。

表 4-4： GENERAL （常规）类别

选项	说明	命令行
Generate debugging info （生成调试信息）	创建 COFF 或 ELF 文件，文件中包含用于在 MPLAB X IDE 中进行代码调试的信息。 **注：** COFF 仅支持在 `.text` 段中进行调试。	`-g`
Isolate each function in a section （将每个函数单独放在一个段中）	选中则在输出文件中将每个函数放入它自己的段。函数的名称决定输出文件中段的名称。 **注：** 指定该选项时，汇编器和链接器可能会创建更大的目标文件和可执行文件，速度也会较慢。 取消选中则将多个函数放在一个段中。	`-ffunction-sections`
Place data into its own section （将数据放入它自己的段中）	在输出文件中将每个数据项放入它自己的段中。 数据项的名称决定段的名称。指定该选项时，汇编器和链接器可能会创建更大的目标文件和可执行文件，速度也会较慢。	`-fdata-sections`
Use 64-bit double （使用 64 位双精度）	使用 `long double` 而不是等效于 float 的 `double` 类型。如果模块通过参数传递直接或通过共享缓冲空间间接共用 `double` 数据，跨模块混合使用该选项可能会产生异常结果。	`-fno-short-double`
Fillupper value for data in flash （填充闪存中数据的高字节值）	使用指定的值填充闪存中数据的高字节。	`-mfillupper=`*`value`*
Name the text section （命名 text 段）	将 text（程序代码）放入名为 *name* 的段，而不是默认的 `.text` 段。	`-mtext=`*`name`*

© 2015 Microchip Technology Inc.

表 4-5： MEMORY MODEL （存储模型）类别

选项	说明	命令行
Code Model （代码模型）	选择代码 （程序存储器 /ROM）模型。 - default （默认） - large （大， >32 K 字） - small （小， ≤32 K 字）	 -msmall-code -mlarge-code -msmall-code
Data Model （数据模型）	选择数据 （数据存储器 /RAM）模型。 - default - large （>8 KB） - small （≤8 KB）	 -msmall-data -mlarge-data -msmall-data
Scalar model （标量模型）	选择标量模型。 - default - large （>8 KB） - small （≤8 KB）	 -msmall-scalar -mlarge-scalar -msmall-scalar
Location of constant model （常量模型的位置）	选择常量的存储位置。 - default - Data （数据） - Code （代码）	 -mconst-in-code -mconst-in-data -mconst-in-code
Place all code in auxiliary flash （将所有代码放入辅助闪存）	将来自当前翻译单元的所有代码都放入辅助闪存。该选项仅在具有辅助闪存的器件上可用。	-mauxflash
Put constants into auxiliary flash （将常量放入辅助闪存）	当与 -mconst-in-code 配合使用时，将常量放入辅助闪存。	-mconst-in-auxflash
Put char vars into near data space （将 char 变量放入 near 数据空间）	将 char 变量放入 near 数据空间，无论存储模型如何。	-mnear-chars
Allow arrays larger than 32K （允许大于 32K 的数组）	允许大于 32K 的数组，无论存储模型如何。	-menable-large-arrays
Aggregate data model （聚合数据模型）	使用聚合数据模型。	-mlarge-aggregate

表 4-6： OPTIMIZATIONS （优化）类别

选项	说明	命令行
Optimization Level （优化级别）	选择优化级别。您的编译器许可证可能仅支持某些优化。等效于 -0*n* 选项，其中的 *n* 是以下选项之一： • 0——不优化。编译器的目标是降低编译开销，以及使调试产生预期的结果。 • 1——优化。优化编译需要的时间稍长一些，对于大函数需要更大的主机存储器。编译器会尝试减小代码长度和执行时间。 • 2——更多优化。编译器执行几乎所有不涉及空间 - 速度权衡的受支持优化。 • 3——优化但更倾向于速度 （O2 的超集）。 • S——优化但更倾向于长度 （O2 的超集）。	-0*n*
Do not override ‘inline’ （不覆盖 “inline”）	选中则不允许编译器决定何时覆盖 “inline” 关键字。 取消选中则允许编译器决定何时覆盖 “inline” 关键字。	-mno-override-inline

表 4-6： OPTIMIZATIONS （优化）类别 （续）

选项	说明	命令行
Unroll loops （展开循环）	选中则执行循环展开优化。这仅仅针对可以在编译时或运行时确定迭代次数的循环。 取消选中则不展开循环。	`-funroll-loops`
Omit frame pointer （省略帧指针）	选中时，对于不需要帧指针的函数，不会在寄存器中保存帧指针。 取消选中则保存帧指针。	`-fomit-frame-pointer`
Unlimited procedural abstraction （无限的过程抽象）	使能过程抽象优化。对嵌套深度没有任何限制。	`-mpa`
Procedural abstraction （过程抽象）	使能最高到级别 *n* 的过程抽象优化。等效于 `-mpa=`*n* 选项，其中 *n* 等于： • 0——禁止优化。 • 1——允许第一层抽象；即，可以将源代码中的指令序列抽象为子程序。 • 2 或更高——允许第二层抽象；即，在第一层中被放入子程序的指令可以抽象为更深一层的子程序。对于更大的 *n* 值，这种模式将不断延续。实际结果就是将子程序调用嵌套深度限制为最大为 *n*。	`-mpa=`*n*
Align arrays （对齐数组）	将数组变量的最小对齐值设置为小于等于其总存储大小的最大的 2 的幂，或机器上使用的最大对齐值，取两者的较小者。	`-falign-arrays`

© 2015 Microchip Technology Inc.

表 4-7： PREPROCESSING AND MESSAGES （预处理和消息）类别

选项	说明	命令行
Include C dirs （包含 C 目录）	将目录 *dir* 添加到要在其中搜索头文件的目录列表的开头。 更多信息，请参见**第 4.5.7 节 “附加的搜索路径和目录”**。	-I"*dir*"
Define C macros （定义 C 宏）	使用字符串 1 定义宏 *macro*。	-D*macro*
ANSI-std C support （ANSI 标准 C 支持）	选中则支持所有 （并且仅支持） ASCI C 程序。 取消选中则支持 ASCI 和非 ASCI 程序。	-ansi
Use CCI syntax （使用 CCI 语法）	如果您的代码是按照通用 C 接口 （CCI）语法 （见**第 2 章 “通用 C 接口”**）编写的，则选中。 如果不是则取消选中。	-mcci
Use IAR syntax （使用 IAR 语法）	如果您的代码是按照 IAR 的嵌入式编译器兼容性模式语法 （见**附录 B “嵌入式编译器兼容性模式”**）编写的，则选中。 如果不是则取消选中。	-mext=IAR
Errata （勘误表）	该选项用于使能由 ID 标识的特定勘误表变通方法。 ID 的有效值会随时更改，可能对于某个特定形式是不需要的。 ID 为 `all` 将使能当前支持的所有勘误表变通方法。ID 为 `list` 将显示当前支持的勘误表标识符与勘误表的简要说明。	-merrata=*id*
Smart IO forwarding level （智能 IO 转发级别）	该选项会尝试静态分析传递给 printf、 scanf 以及这些函数的 “f”和 “v”版本的格式字符串。使用非浮点格式参数将被转换为使用库函数的仅整型版本。 等效于 `-msmart-io=`*n* 选项，其中 *n* 等于： • 0——禁止该选项。 • 1——仅转换它可以验证的字面值。 • 2——使编译器变得乐观，并转换带有变量或未知格式参数的函数调用。	-msmart-io=*n*
Smart IO format strings （智能 IO 格式字符串）	指定当编译器无法确定格式参数时的格式参数。	
Make warnings into errors （使警告变为错误）	选中则基于警告以及错误来暂停编译。 取消选中则仅基于错误来暂停编译。	-Werror
Additional warnings （额外的警告）	选中则使能所有警告。 取消选中则禁止警告。	-Wall
Strict ANSI warnings （严格 ANSI 警告）	选中则发出严格 ANSI C 要求的所有警告。 取消选中则发出所有警告。	-pedantic
Disable ISR warn （禁止 ISR 警告）	禁止关于不正确使用 ISR 函数名称的警告。默认情况下，如果识别到的中断向量名称未附加 `interrupt`，编译器会产生警告。该选项会禁止该功能。	-mno-isr-warn
Enable SFR warnings （使能 SFR 警告）	使能与 SFR 相关的警告。	-msfr-warn=on|off

4.5.4 xc16-ld （16 位链接器）

可以在 MPLAB X IDE 中指定命令行选项的一个子集。选择某个类别，然后设置链接器选项。关于更多选项，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。另请参见 “选项页面功能”。

表 4-8： GENERAL 类别

选项	说明	命令行
Heap size （堆大小）	指定堆的大小，以字节为单位。分配指定字节大小运行时堆，供 C 程序使用。堆从未用数据存储器中分配。如果没有足够的存储器可用，则会报告错误。	`--heap` *size*
Min stack size （最小堆栈大小）	指定堆栈的最小大小，以字节为单位。默认情况下，链接器会分配所有未用的数据存储器作为运行时堆栈。或者，程序员也可以通过声明两个全局符号来分配堆栈：`__SP_init` 和 `__SPLIM_init`。使用该选项可以确保至少有最小大小的堆栈可用。链接映射输出文件中会报告实际堆栈大小。如果没有最小大小可用，则会报告错误。	`--stack` *size*
Use local stack （使用局部堆栈）	允许将堆栈分配在扩展数据空间 （EDS）存储器中。	
Allow overlapped sections （允许重叠的段）	选中则不检查段地址是否重叠。 取消选中则检查是否有重叠。	`--check-sections` `--no-check-sections`
Init data sections （初始化数据段）	选中则支持初始化数据。 取消选中则不支持。	`--data-init` `--no-data-init`
Pack data template （压缩数据模板）	选中则压缩初始数据值。 取消选中则不压缩。	`--pack-data` `--no-pack-data`
Create handles （创建句柄）	选中则支持 far 代码指针。 取消选中则不支持。	`--handles` `--no-handles`
Create default ISR （创建默认 ISR）	选中则创建用于未用向量的中断函数。 取消选中则不创建默认 ISR。	`--isr` `--no-isr`
Remove unused sections （删除未用的段）	选中则不使能对未用输入段的垃圾回收 （在某些目标器件上）。 取消选中则使能垃圾回收。	`--no-gc-sections` `--gc-sections`
Fill value for upper byte of data （数据最高字节的填充值）	输入数据最高字节的填充值。将数据编码到程序存储器中时，使用该值作为最高字节 （bit 16-23）。该选项会影响使用 `psv` 或 `eedata` 属性创建的段的编码，如果使能了 `--no-pack-data` 选项，则还会影响数据初始化模板。	`--fill-upper=`*value*
Stack guardband size （堆栈安全余量大小）	输入堆栈安全余量大小来确保有足够的堆栈空间来处理堆栈溢出异常。	`--stackguard=`*size*
Additional driver options （额外的驱动程序选项）	在此处输入该 GUI 其他位置不存在的任何额外驱动程序选项。在此处引入的字符串将像在驱动程序调用命令中一样发出。	

© 2015 Microchip Technology Inc.

表 4-8： GENERAL 类别 （续）

选项	说明	命令行
Use response file to link （使用响应文件进行链接）	选中则创建对于链接步骤使用响应文件的 makefile。在 Windows 中，最大命令行长度为 8191 个字符。在链接很长的程序时，链接行可能会超过该限制。MPLAB XC16 提供了响应文件变通办法。关于详细信息，请参见 《MPLAB® X IDE 用户指南》（DS50002027C_CN）的 “疑难解答”章节。 取消选中则不使用响应文件。	

表 4-9： SYMBOLS AND MACROS （符号和宏）类别

选项	说明	命令行
Linker symbols （链接器符号）	在输出文件中创建包含绝对地址 （*expr*）的全局符号。您可以根据需要在命令行中任意多次地使用该选项来定义多个符号。对于该上下文中的 *expr*，支持一种有限形式的算术运算：您可以提供一个十六进制常量或现有符号的名称，或使用 + 和 - 来加上或减去十六进制常量或符号。	`--defsym=`*sym*
Define Linker macros （定义链接器宏）	添加链接器宏。	`-D`*macro*

Symbols （符号）	指定输出中的符号信息。
	- Keep all （全部保留）
	- Strip debugging info （去除调试信息）
	- Strip all symbol info （去除所有符号信息）
		
		—
		`--strip-debug (-S)`
		`--strip-all (-s)`

表 4-10： FILL FLASH MEMORY （填充闪存）类别

选项	说明	命令行
Which areas to fill （要填充的区域）	指定填充闪存的哪些区域。 **No Fill** （无填充）——无 （默认）。 **Fill All Unused** （填充所有未用）——填充所有未用的存储空间。 **Provide Range to fill** （提供要填充的范围）——填充某个存储器范围。在 “Memory Address Range”（存储器地址范围）下输入范围。	
How to fill it （如何填充它）	指定如何填充闪存。 **Provide sequence of values** （提供值的序列）——在 Sequence （序列）选项下提供一个序列。 **Constant or incrementing value** （常量或递增值）——在同名选项下提供一个常量、递增 / 递减或递增 / 递减常量。	
Sequence	选择 **Provide sequence of values** 时，输入序列。其形式为 n1, n2, ...，其中的 n1 使用 C 语法。 示例：0x10, 25, 0x3F, 16。	`--fill=`*sequence*
Constant （常量）	选择 **Constant or incrementing value** 时，输入一个常量。使用 C 语法指定常量 （例如， 0x 为十六进制， 0 为八进制）。 示例：0x10 等同于 020 或 16。	`--fill=`*constant*

表 4-10： FILL FLASH MEMORY （填充闪存）类别 （续）

选项	说明	命令行
Increment/Decrement （递增 / 递减）	选择 **Constant or incrementing value** 时，您可以选择对于每个连续地址递增或递减常量初始值。 **No Incrementing** （无递增）——不改变常量值 **Increment Const** （递增常量）——按 “Increment/Decrement Constant”（递增 / 递减常量）下指定的数量递增常量值。 **Decrement Const** （递减常量）——按 “Increment/Decrement Constant”下指定的数量递减常量值。	
Increment/Decrement Constant	选择 **Increment Const** 或 **Decrement Const** 时，输入一个常量递增或递减值。使用 C 语法指定常量（例如， 0x 为十六进制， 0 为八进制）。示例：0x10 等同于 020 或 16。	`--fill=`*`constant`*`+=`*`incr`* `--fill=`*`constant`*`-=`*`decr`*
Memory Address Range （存储器地址范围）	选择 **Provide Range to fill** 时，在此处输入范围。以 “Start:End （开始 : 结束）”的形式指定范围，其中的 “Start （开始）”和 “End （结束）”使用 C 语法。示例：0x100:0x1FF 等于 256:511。	`--fill=`*`value`*`@`*`range`*

表 4-11： LIBRARIES （库）类别

选项	说明	命令行
Libraries	添加要与项目文件链接的库。您可以添加多个库。	`--library=`*`name`*
Library directory （库目录）	将某个库目录添加到库搜索路径中。您可以添加多个库目录。	`--library-path="`*`name`*`"`
Force linking of objects that might not be compatible （强制链接可能不兼容的目标文件）	选中则强制链接可能不兼容的目标文件。链接器会将项目所用器件与链接期间组合的目标文件中包含的信息进行比较。如果检测到可能发生冲突，则会报告错误 （指令集可能不兼容情况）或警告 （寄存器可能不兼容情况）。指定该选项可撤销此类错误或警告。 取消选中则不强制链接。	`--force-link` `--no-force-link`
Don’t merge I/O library functions （不合并 I/O 库函数）	选中则不合并 I/O 库函数。不尝试通过合并 I/O 库函数调用来节省存储器。在某些情况下，使用该选项会增加存储器使用量。 取消选中则合并 I/O 库函数，以节省存储器。	`--no-smart-io` `--smart-io`
Exclude standard libraries （排除标准库）	选中则在链接时不使用标准系统启动文件或库。仅使用在命令行上指定的库目录。 取消选中则使用标准系统启动文件和库。	`--nostdlib`

© 2015 Microchip Technology Inc.

表 4-12： DIAGNOSTICS （诊断）类别

选项	说明	命令行
Generate map file （生成映射文件）	创建映射文件。	-Map="*file*"
Display memory usage （显示存储器使用情况）	选中则打印存储器使用情况报告。 取消选中则不打印报告。	--report-mem
Generate cross-reference file （生成交叉引用文件）	选中则创建交叉引用表。 取消选中则不创建该表。	--cref
Warn on section realignment （产生关于段重新对齐的警告）	选中则在段起始位置由于对齐而发生变化时产生警告。 取消选中则不产生警告。	--warn-section-align
Trace Symbols （跟踪符号）	添加 / 删除跟踪符号。	--trace-symbol=*symbol*

表 4-13： CODE GUARD （代码保护）类别

选项	说明	命令行
Boot RAM （引导 RAM）	指定引导 RAM 段：none （无）、 small （小型）、 medium （中型）或 large （大型）。	--boot=*option*_ram
引导闪存	指定引导闪存段：none、 small、 medium 或 large standard （大型标准）或 none、 small、 medium 或 large high （大型高）。	--boot=*option*_flash_std --boot=*option*_flash_high
Boot EEPROM （引导 EEPROM）	指定引导 EEPROM 段。	--boot=eeprom
Boot write-protect （引导写保护）	指定引导写保护段。	--boot=write_protect
Secure RAM （安全 RAM）	指定安全 RAM 段：none、 small、 medium 或 large。	--secure=*option*_ram
Secure Flash （安全闪存）	指定安全闪存段：none、 small、 medium 或 large standard 或 none、 small、 medium 或 large high。	--secure=*option*_flash_std --secure=*option*_flash_high
Secure EEPROM （安全 EEPROM）	指定安全 EEPROM 段。	--secure=eeprom
Secure write-protect （安全写保护）	指定安全写保护段。	--secure=write_protect
General write-protect （通用写保护）	指定通用写保护段。	--general=write_protect
General code-protect （通用保护代码）	指定安全代码保护段：standard 或 high。	--general=code_protect_std --general=code_protect_high

关于 CodeGuard™ 选项的更多信息，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）中的第 8.8 节 “指定 CodeGuard 安全功能的选项”。

注：不是所有开发工具都支持 CodeGuard 编程。更多信息请参见工具文档。

4.5.5 xc16-ar （16 位归档器 / 库管理器）

可以在 MPLAB X IDE 中指定命令行选项的一个子集。选择某个类别，然后设置链接器选项。关于更多选项，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。另请参见 “选项页面功能”。

表 4-14： GENERAL 类别

选项	说明	命令行
Break line into multiple lines （行拆分为多行）	对于 Windows 操作系统，最大命令行长度为 8191 个字符。如果要归档到库中的文件很多，链接行可能会超过该限制。编译器可以将归档行拆分为较小的行，以避开该限制。	`true`

4.5.6 选项页面功能

对于所有工具， Properties 页面的 Options 部分具有以下功能：

表 4-15： 页面功能选项

Reset （复位）	将页面复位为默认值。
Additional options （其他选项）	以命令行 （非 GUI）格式输入选项。
Option Description （选项说明）	单击某个选项名称，可在该窗口中查看关于该选项的信息。并非所有选项都在该窗口中具有信息。
Generated Command Line （生成的命令行）	单击某个选项名称，可在该窗口中查看该选项的等效命令行。

4.5.7 附加的搜索路径和目录

对于编译器、汇编器和链接器，您可以设置用于搜索包含文件和库的附加目录路径。

您可以根据需要添加任意多个目录，以包含一系列路径。总是最先搜索当前的工作目录，然后搜索附加目录，附加目录按指定它们的顺序进行搜索。

指定的所有路径都应相对于项目目录，即包含 `nbproject` 的目录。

© 2015 Microchip Technology Inc.

4.6 项目示例

在该示例中，您将创建一个具有两个 C 代码文件的 MPLAB X IDE 项目。

- 运行项目向导
- 向项目添加文件
- 设置编译选项
- 编译项目
- 输出文件
- 进一步开发

4.6.1 运行项目向导

在 MPLAB X IDE 中，选择 *File>New Project* 来启动向导。

1. **选择项目：**对于类别选择 “Microchip Embedded”（Microchip 嵌入式），对于项目选中 “Standalone Project”（独立项目）。单击 **Next>** （下一步）继续。
2. **选择器件：**选择 dsPIC30F6014。单击 **Next>** （下一步）继续。
3. **选择调试头：**该器件没有调试头，所以跳过该步骤。
4. **选择工具：**从列表中选择一种开发工具。选定器件的工具支持会在工具旁以彩色圆圈的形式显示。将鼠标悬停在圆圈上，可以文本形式查看支持。单击 **Next>** （下一步）继续。
5. **选择编译器：**选择 XC16 工具链的一个版本。单击 **Next>** （下一步）继续。
6. **选择项目名称和文件夹：**输入项目名称，例如 `MyXC16Project`。然后选择项目文件夹的位置。单击 **Finish** （完成）来完成项目创建和设置。

完成项目向导时， Project （项目）窗口中将包含项目树。关于项目更多信息，请参见 MPLAB X IDE 文档。

4.6.2 向项目添加文件

将 C 代码文件 `tmp6014.c` 和 `traps.c` 添加到项目中：

1. 右键单击项目树中的 “Source Files”（源文件）文件夹。选择 “Add Existing Item”（添加现有项）打开 Select Item （选择项）对话框。
2. 转至目录：
   `C:\Program Files\Microchip\xc16\`*`vx.xx`*`\support\templates\c`
   其中， *vx.xx* 是版本号。
3. 单击文件 `tmp6014.c`。然后，按住 shift 键并单击文件 `traps.c`。单击 **Select** （选择）。

当您完成时，项目树中的 Source Files 文件夹现在会打开，其中包含两个添加的文件。

4.6.3 设置编译选项

选择 *File>Project Properties*，或右键单击项目名称并选择 “Properties”来打开 Project Properties 对话框。

1. 在 “Conf:[default]>XC16 (Global Options)”下，选择 “xc16-gcc”。
2. 从 “Option Categories”中选择 “Preprocessing and messages”。输入以下路径作为 “C Include Dirs”：
   `C:\Program Files\Microchip\xc16\vx.xx\support\dsPIC30F\h`
   其中， *vx.xx* 是版本号。
3. 在 “Conf:[default]>XC16 (Global Options)”下，选择 “xc16-ld”。
4. 从 “Option Categories”中选择 “Diagnostics”。然后，在 “Generate map file”中输入文件名，例如， `example.map`。
5. 单击对话框底部的 **OK** （确定），接受编译选项并关闭对话框。

4.6.4 编译项目

在项目树中右键单击项目名称 “MyXC16Project”，然后从弹出菜单中选择 “Build”（编译）。 Output 窗口将显示编译结果。

如果编译未成功完成，请检查以下各项：

1. 复核该示例中的先前步骤。确保已正确设置语言工具，并具有所有正确的项目文件和编译选项。
2. 如果您修改了示例源代码，则检查 Output 窗口中的 Build 选项卡，确定源代码中是否存在语法错误。如果发现任何错误，则单击错误转到包含该错误的源代码行。修正该错误，然后尝试重新编译。

4.6.5 输出文件

通过在 MPLAB X IDE 中打开文件来查看项目输出文件。

1. 选择 *File>Open File*。在 Open （打开）对话框中，查找项目目录。
2. 在 “Files of type”（文件类型）下选择 “All Files”（所有文件）来查看所有项目文件。
3. 选择 *File>Open File*。在 Open 对话框中，选择 “example.map”。单击 **Open** 在 MPLAB X IDE 编辑器窗口中查看链接器映射文件。关于该文件的更多信息，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。
4. 选择 *File>Open File*。在 Open 对话框中，返回到项目目录，然后转到 *dist （分发）>default （默认）>production （生产）*目录。请注意，其中只有一个 Hex 文件 “MyXC16Project.X.production.hex”。它是主输出文件。单击 **Open**，在 MPLAB X IDE 编辑器窗口中查看 Hex 文件。关于该文件的更多信息，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。另外还有一个文件 “MyXC16Project.X.production.cof”或 “MyXC16Project.X.production.elf”。该文件包含调试信息，由调试工具使用来调试代码。关于选择调试文件类型的信息，请参见**第 4.5.1 节 “XC16 (Global Options)”**。

© 2015 Microchip Technology Inc.

4.6.6 进一步开发

通常，您的应用程序代码将会包含错误，第一次会无法工作。因此，您需要一个调试工具来帮助开发代码。有几个调试工具使用前面介绍的输出文件，可以与 MPLAB X IDE 配合工作来帮助您完成这项工作。您可以从由 Microchip 或第三方供应商制造的软件模拟器、在线仿真器或在线调试器中进行选择。要了解这些工具可以如何帮助您，请参见这些工具的文档。在调试时，需要使用 ***Debug（调试）>Debug Project*** 来运行和调试代码。关于更多信息，请参见 《MPLAB® IDE 用户指南》（DS50002027C_CN）。

开发完代码之后，您会希望将代码编程到器件中。同样，有几个编程器可以与 MPLAB X IDE 配合工作，帮助您完成这项工作。要了解这些工具可以如何帮助您，请参见这些工具的文档。在编程时，您需要使用调试工具栏上的 “Make and Program Device Project”（Make 项目并对器件编程）按钮。关于该控件的信息，请参见《MPLAB® IDE 用户指南》（DS50002027C_CN）。

注：

© 2015 Microchip Technology Inc.

第 5 章 编译器命令行驱动程序

5.1 简介

编译器命令行驱动程序（xc16-gcc）是调用 MPLAB XC16 C 编译器操作的应用程序。该驱动程序可以编译、汇编和链接 C 语言和汇编语言模块和库归档。大多数编译器命令行选项对于GCC工具集的所有实现都是通用的。一些选项特定于编译器，将在下面介绍。

编译器驱动程序也可以与 MPLAB X IDE 配合使用。编译器选项在 GUI 中选择，并传递给编译器驱动程序来执行。

下面将讨论有关驱动程序的命令行使用的主题。

- 调用编译器
- 编译序列
- 运行时文件
- 编译器输出
- 编译器消息
- 驱动程序选项说明
- MPLAB X IDE 工具链或 MPLAB IDE 工具套件等效选项

5.2 调用编译器

编译器在下一节中介绍的命令行上调用和运行。此外，随后几节还介绍了编译器所使用的环境变量和输入文件。

5.2.1 驱动程序命令行格式

编译器命令行的基本形式为：

```
xc16-gcc [options] files
```

其中：

options：关于可用的选项，请参见**第 5.7 节 “驱动程序选项说明”**。

files：详细信息请参见**第 5.2.3 节 “输入文件类型”**。

注： 命令行选项和文件扩展名是区分大小写的。

在本手册中，将假定编译器应用程序处于控制台的搜索路径中（见**第 5.2.2 节“环境变量”**），或者在执行任意应用程序时指定了完整路径。

约定是提供 *options*（通过文件名之前的前导*短划线*“-”表示），虽然这不是强制性的。

files 可以是 C 和汇编源文件以及预编译的中间文件（如可重定位目标文件（.o））的任意混合。文件的顺序并不重要，只是它可能会影响代码或数据在存储器中出现的顺序。

例如，要编译、汇编和链接 C 源文件 hello.c，从而创建可重定位目标输出文件 hello.elf。

```
xc16-gcc -mcpu=30f2010 -o hello.elf hello.c
```

5.2.2 环境变量

本节中的变量是可选的，但如果定义了它们，它们将由编译器使用。如果以下一些环境变量未设置，编译器驱动程序或其他子程序可以选择确定它们适当的值。驱动程序或其他子程序可以利用有关编译器安装的内部知识。只要安装结构保持不变，所有子目录和可执行文件保持在相同的相对位置中，驱动程序或子程序就能够确定一个可用的值。

© 2015 Microchip Technology Inc.

表 5-1：　　编译器相关的环境变量

变量	定义
XC16_C_INCLUDE_PATH PIC30_C_INCLUDE_PATH	该变量的值是以分号分隔的目录列表，非常类似于 PATH。在编译器搜索头文件时，它会尝试在使用 -I 指定的目录之后、但在标准头文件目录之前搜索在该变量中列出的目录。如果该环境变量未定义，则预处理器会基于标准安装选择一个适当的值。默认情况下，会在以下目录中搜索包含文件： <install-path>\include 和 <install-path>\support\h
XC16_COMPILER_PATH PIC30_COMPILER_PATH	该变量的值是以分号分隔的目录列表，非常类似于 PATH。在搜索子程序时，如果使用 PIC30_EXEC_PREFIX 无法找到子程序，编译器会尝试由此指定的目录。
XC16_EXEC_PREFIX PIC30_EXEC_PREFIX	如果设置了该环境变量，则它指定在由编译器执行的子程序的名称中使用的前缀。将该前缀与子程序名称组合时，不会添加目录分隔符，但如果需要，您可以指定一个以斜杠（/）结尾的前缀。如果编译器无法使用指定的前缀找到子程序，它将尝试在 PATH 环境变量中查找。 如果该环境变量未设置或设置为空值，编译器驱动程序将基于标准安装选择一个适当的值。如果安装未进行任何修改，这将会使驱动程序能够找到所需的子程序。 使用 -B 命令行选项指定的其他前缀将优先于用户或驱动程序定义的变量值。 在正常情况下，最好将该值保留未定义，让驱动程序自己查找子程序。
XC16_LIBRARY_PATH PIC30_LIBRARY_PATH	该变量的值是以分号分隔的目录列表，非常类似于 PATH。该变量指定要传递给链接器的目录列表。驱动程序对该变量的默认求值为： <install-path>\lib; <install-path>\support\gld。
XC16_OMF PIC30_OMF	指定编译器使用的 OMF（目标模块格式）。默认情况下，这些工具会创建 ELF 目标文件。如果环境变量的值为 coff，工具将创建 COFF 目标文件。
TMPDIR	如果设置了该变量，则它指定要用于临时文件的目录。编译器使用临时文件来存放某个编译阶段的输出，用作下一个阶段的输入：例如，预处理器的输出，它是相应编译器的输入。

5.2.3 输入文件类型

编译驱动程序完全根据文件类型或扩展名来区分源文件、中间文件和库文件。它可以识别以下文件扩展名，它们区分大小写。

表 5-2： 文件名

扩展名	定义
file.c	必须进行预处理的 C 源文件。
file.h	头文件 （不进行编译或链接）。
file.i	不应进行预处理的 C 源文件。
file.o	目标文件。
file.p	预过程抽象汇编语言文件。
file.s	汇编代码。
file.S	必须进行预处理的汇编代码。
其他	要传递给链接器的文件。

编译器对于源文件的名称不存在任何限制，但需要注意操作系统对于大小写形式、名称长度和其他方面的限制。如果使用的是 IDE，则需要避免汇编源文件的基本名称与使用该文件的任何项目的基本名称相同。如果相同可能导致在编译过程中源文件被某个临时文件覆盖。

在谈及计算机程序时，经常会使用术语“源文件”和“模块”。它们通常可以互换，但它们指代编译序列中不同点的源代码。

源文件是包含全部或部分程序的文件。源文件最初由驱动程序传递给预处理器。

模块是给定源文件在包含通过 `#include` 预处理器伪指令指定的所有头文件 （或其他源文件）之后的预处理器输出。然后，这些模块将传递给其余的编译器应用程序。因此，模块可能由几个源文件和头文件组成。模块也常被称为翻译单元。这些术语也适用于汇编文件，因为它们也可以包含其他头文件和源文件。

© 2015 Microchip Technology Inc.

5.3 编译序列

以下几节将介绍编译器如何与其他应用程序配合工作，以及如何执行不同类型的编译。

5.3.1 编译器应用程序

MPLAB XC16 C 编译器编译 C 源文件，生成汇编语言文件。这些编译器生成的文件与其他目标文件和库进行汇编和链接，以生成可执行的 ELF 或 COFF 文件格式的最终应用程序。ELF 或 COFF 文件可以装入 MPLAB X IDE 中，在其中可以对它进行测试和调试，也可以使用转换实用程序将 ELF 或 COFF 文件转换为适合于装入命令行模拟器或器件编程器的 Intel® hex 格式。**第 4.4 节 “MPLAB X IDE 项目”**给出了一个软件开发工具数据流程图。

驱动程序将调用所需的内部编译器应用程序。这些应用程序在驱动程序大框内显示为一些小框。在该图中，还可以看到每个应用程序所生成的临时文件。

表 5-3 列出了编译器应用程序。所显示的名称是可执行文件的名称，可以在编译器的安装目录下的 `bin` 目录中找到它们。您的 PATH 环境变量应包含该目录。

表 5-3： 编译器应用程序名称

名称	说明
`xc16-gcc`	命令行驱动程序；编译器的接口
`xc16-as`	汇编器 （基于目标器件）
`xc16-ld`	链接器
`xc16-bin2hex`	用于创建 HEX 文件的转换实用程序
`xc16-strings`	字符串提取器实用程序
`xc16-strip`	符号去除器实用程序
`xc16-nm`	符号列表实用程序
`xc16-ar`	归档器 / 库管理器
`xc16-objdump`	目标文件显示实用程序
`xc16-ranlib`	归档索引器实用程序

5.3.2 单步编译

可以使用单个命令行来编译一个文件或多个文件。

5.3.2.1 编译单个文件

本节演示如何编译和链接单个文件。为了进行以下的讨论，将假定编译器安装在标准目录位置，并且您的 PATH 或其他环境变量（见**第 5.2.2 节 “环境变量”**）设置为在运行编译器时不需要指定完整的编译器路径。

以下是一个将两个数字相加的简单 C 程序。

使用任意文本编辑器创建以下程序，并将其保存为 ex1.c。

```
#include <xc.h>
int main(void);
unsigned int Add(unsigned int a, unsigned int b);
unsigned int x, y, z;
int
main(void)
{
  x = 2;
  y = 5;
  z = Add(x,y);
  return 0;
}
unsigned int
Add(unsigned int a, unsigned int b)
{
  return(a+b);
}
```

该程序的第一行包含了头文件 xc.h，该文件将包含提供目标器件上所有特殊功能寄存器定义的相应头文件。关于头文件的更多信息，请参见**第 6.3 节 “器件头文件”**。

使用您偏好的终端，通过在提示符下输入以下命令来编译该程序。

```
xc16-gcc -mcpu=30f2010 -o ex1.elf ex1.c
```

命令行选项 -o ex1.elf 用于命名输出可执行文件（如果未指定 -o 选项，则输出文件命名为 a.out）。该可执行文件可以装入 MPLAB X IDE。

如果需要 hex 文件（例如，装入到器件编程器），则使用以下命令：

```
xc16-bin2hex ex1.elf
```

这将创建一个名为 ex1.hex 的 Intel hex 文件。

© 2015 Microchip Technology Inc.

5.3.2.2 编译多个文件

将 Add() 函数移入一个名为 add.c 的文件中，以演示如何在应用程序中使用多个文件。即：

File 1

```
/* ex1.c */
#include <xc.h>
int main(void);
unsigned int Add(unsigned int a, unsigned int b);
unsigned int x, y, z;
int main(void)
{
  x = 2;
  y = 5;
  z = Add(x,y);
  return 0;
}
```

File 2

```
/* add.c */
#include <xc.h>
unsigned int
Add(unsigned int a, unsigned int b)
{
  return(a+b);
}
```

通过在终端程序中输入以下命令在一个命令中编译这两个文件。

```
xc16-gcc -mcpu=30f2010 -o ex1.elf ex1.c add.c
```

该命令将编译模块 ex1.c 和 add.c。编译的模块将与编译器库进行链接，并生成可执行文件 ex1.elf。

5.3.3 多步编译

make 实用程序和集成开发环境（如 MPLAB IDE）支持对包含多个源文件的项目进行增量编译。在编译项目时，它们会注意哪些源文件自上次编译以来发生了更改，并使用该信息来加快编译速度。

例如，如果要编译两个源文件，但只有一个文件自上次编译以来发生了更改，则不需要重新生成未发生更改的源文件对应的中间文件。

如果所调用的编译器使用 make 实用程序，则需要配置 make 文件，以识别不同的中间文件格式和用于生成中间文件的选项。 make 实用程序通常会多次调用编译器：对于每个源文件调用一次来生成中间文件，以及调用一次来执行编译的第二阶段。

此外，您还可能希望生成中间文件来构建您自己的库文件，不过 MPLAB XC16 能够构建库，所以这通常是不必要的。关于库创建的更多信息，请参见《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

© 2015 Microchip Technology Inc.

例如，文件 ex1.c 和 add.c 需要使用 make 实用程序进行编译。使用 make 实用程序来编译这些文件的命令行可能类似于：

```
xc16-gcc -mcpu=30f6014 -c ex1.c
xc16-gcc -mcpu=30f6014 -c add.c
xc16-gcc -mcpu=30f6014 -o ex1 ex1.o add.o
```

-c 选项会将指定文件编译为中间 （目标）文件格式，但不进行链接。在所有文件都按 make 的指定进行编译之后，所产生的目标文件在最后的步骤中进行链接，产生最终的输出 ex1。以上示例使用命令行驱动程序 xc16-gcc 来执行最终的链接步骤。您可以显式地调用链接器应用程序 xc16-ld，但建议不要这么做。在驱动链接器应用程序时，您必须指定链接器选项，而不是驱动程序选项。关于使用链接器的更多信息，请参见《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

在编译调试代码时，目标模块格式 （OMF）必须对于编译、汇编和链接是一致的。默认情况下使用 ELF/DWARF 格式，但也可以使用 -omf=coff 或环境变量 XC16_OMF 选择 COFF 格式。

5.3.4 汇编编译

混合的 C 和汇编代码可以使用编译器一起编译 （图）。更多详细信息，请参见**第 16 章 “混合使用 C 代码和汇编代码”**。

此外，可以通过 -S 选项使用编译器从 C 代码 （.c）生成汇编代码 （.s）。然后，可以使用命令行驱动程序在后续的编译中使用汇编输出。

5.4 运行时文件

除了源文件、链接器文件和头文件之外，编译器还使用以下文件。

5.4.1 库文件

编译器可以按照图 4-2 将库文件包含到输出中。

默认情况下，`xc16-gcc`会在编译器安装目录下的已知位置搜索编译过程中所需的库文件。

5.4.1.1 标准库

C 标准库包含一组标准化的函数，例如字符串、数学和输入 / 输出程序。《16 位语言工具函数库》（DS51456D_CN）介绍了这些函数的范围。

5.4.1.2 用户定义的库

用户可以创建自己的库。库可用于捆绑和预编译选定的函数，使应用程序文件管理变得更简便，应用程序编译时间变得更短。

库可以使用编译器和库管理器手动创建。要创建之后用作 16 位库管理器 （`xc16-ar`）输入的文件，请使用 `-c` 编译器选项在链接器阶段之前停止编译。关于使用库管理器的信息，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

库应命名为 `lib`*`library`*`.a`，可以通过指定其路径名 （`-L`*`dir`*）和 `-l`*`library`* 添加到编译器命令行中。关于这些选项的详细信息，请参见**第 5.7.9 节 “用于链接的选项”**。

所有库文件都不应包含 `main()` 函数，也不应包含配置位的设置或任何其他此类数据。

以下给出了一个将库 `libmyfns.a` 添加到命令行中的简单示例：

```
xc16-gcc -mcpu=30f2010 -lmyfns example.c
```

与标准 C 库函数一样，用户定义的库中包含的所有函数都需要在头文件中添加声明。常见的做法是创建一个或多个与库文件打包在一起的头文件。然后，可以在需要时将这些头文件包含到源代码中。

将会先扫描在命令行上指定库文件中的未解析符号，所以这些文件可能会重新定义在 C 标准库中定义的任何方面。

5.4.2 启动和初始化

有两种启动模块可以用来初始化 C 运行时环境：

- 默认情况下 （或 `-Wl`, `--data-init` 选项）链接的主启动模块。
- 指定 `-Wl`, `--no-data-init` 选项 （无数据初始化）时链接的备用启动模块。

这些模块包含在 `libpic30-`*`omf`*`.a` 归档 / 库中。为了支持器件系列之间的架构差别，提供了这些模块的多个版本。编译器会自动使用正确的模块。

关于启动模块的更多信息，请参见**第 15.3 节 “运行时启动和初始化”**。

© 2015 Microchip Technology Inc.

5.5 编译器输出

编译器在编译过程中会创建许多文件。其中的大量文件是中间文件，在编译完成后会被删除，但许多文件会保留，用于对器件进行编程或用于调试目的。

5.5.1 输出文件

编译驱动程序可以产生具有以下扩展名的输出文件，这些扩展名是区分大小写的。

表 5-4： 文件名

扩展名	定义
file.hex	可执行文件
file.cof	COF 调试文件 （默认）
file.elf	ELF 调试文件
file.o	目标文件 （中间文件）
file.S	汇编代码文件 （需要预处理）
file.s	汇编代码文件 （中间文件）
file.i	经过预处理的文件 （中间文件）
file.p	预过程抽象汇编语言文件 （中间文件）
file.map	映射文件

许多输出文件使用与它们的源文件相同的基本文件名。例如，使用 -c 选项时，源文件 input.c 将创建一个名为 input.o 的目标文件。

默认输出文件是名为 a.out 的 ELF 文件，除非使用 -o 选项改写了文件名。

如果使用 MPLAB X IDE 来指定编译器选项，则通常会有一个为每个应用程序创建的项目文件。该项目的名称将用作项目中输出文件的基本文件名，除非用户另外指定。请查看您使用的 IDE 的手册，了解更多详细信息。

注： 在本手册中，*项目名称*一词将指代在 IDE 中创建的项目的名称。

编译器能够直接生成一些由 Microchip 开发工具使用的输出文件格式。

xc16-gcc 的默认行为是产生 ELF 输出。要更改文件输出或文件名，请参见**第 5.7 节 “驱动程序选项说明”**。

5.5.2 诊断文件

编译器生成的两个重要文件是由汇编器生成的汇编列表文件和由链接器生成的映射文件。

汇编列表文件包含原始源代码和生成的汇编代码之间的映射。它对于了解一些信息很有用，例如 C 源代码如何进行编码，或汇编源代码如何进行优化。在确认编译器生成的访问对象的代码是否是原子操作时，以及显示存放所有对象和代码的区域时，它是必需的。

在汇编器中创建列表文件的选项是 `-a`。该选项具有许多变化形式，《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）中进行了介绍。要从编译器中传递选项，请参见**第 5.7.8 节 “用于汇编的选项”**。

每次编译都会产生一个列表文件。因此，如果需要为每个源文件产生一个列表文件，则必须单独编译这些源文件，请参见**第 5.3.3 节 “多步编译”**。如果使用 MPLAB IDE 进行编译，就是如此。每个列表文件将被分配模块名称和扩展名 `.lst`。

映射文件将显示与对象在存储器中的放置位置相关的信息。对于确认用户定义的链接器选项是否已正确处理，以及确定对象和函数的确切位置，它很有用。

在链接器应用程序中创建映射文件的链接器选项是 `-Map` *`file`*，《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）对它进行了介绍。要从命令行驱动程序中指定选项，请参见**第 5.7.9 节 “用于链接的选项”**。

在编译项目时，如果执行了链接器并运行完成，则会生成一个映射文件。

5.6 编译器消息

附录 C “诊断”介绍了编译器输出消息：错误、警告和建议性消息。

关于控制编译器错误、警告或建议性消息输出的选项的信息，请参见**第 5.7.4 节 “用于控制警告和错误的选项”**。

没有可直接控制由编译器发出的消息的 pragma 伪指令。

© 2015 Microchip Technology Inc.

5.7 驱动程序选项说明

编译器具有许多用于控制编译的选项，所有这些选项均区分大小写。它们根据功能进行分组，如以下所示。请记住，这些是用于命令行驱动程序的选项；关于对编译器指定用于这些工具的选项的信息，请参见**第 5.7.8 节 “用于汇编的选项”**或**第 5.7.9 节 “用于链接的选项”**。

- 特定于 16 位器件的选项
- 用于控制输出类型的选项
- 用于控制 C 方言的选项
- 用于控制警告和错误的选项
- 用于调试的选项
- 用于控制优化的选项
- 用于控制预处理器的选项
- 用于汇编的选项
- 用于链接的选项
- 用于目录搜索的选项
- 用于代码生成约定的选项

5.7.1 特定于 16 位器件的选项

关于存储模型的更多信息，请参见**第 10.14 节 “存储模型”**。

表 5-5： 特定于 16 位器件的选项

选项	定义
`-mconst-in-code`	将使用 `const` 限定的变量放入 `auto_psv` 空间。编译器将使用 PSV 窗口访问这些变量。**（这是默认设置。）**
`-mconst-in-data`	将使用 `const` 限定的变量放入数据存储空间。
`-mconst-in-auxflash`	当与 `-mconst-in-code` 配合使用时，将所有使用 `const` 限定的文件作用域变量放入辅助闪存。所有使用辅助闪存的模块都应使用该选项进行编译；否则可能发生链接错误。
`-merrata=`*`id`*`[,`*`id`*`]*`	该选项用于使能由 *`id`* 标识的特定勘误表变通方法。 *`id`* 的有效值会随时更改，可能对于某个特定形式是不需要的。 *`id`* 为 `list` 将显示当前支持的勘误表标识符与勘误表的简要说明。 *`id`* 为 `all` 将使能当前支持的所有勘误表变通方法。
`-mfillupper`	指定存储到 `space(prog)` 段中的变量的高字节。 `fillupper` 属性将对每个变量执行相同的功能。
`-mlarge-arrays`	指定数组可以大于 32K 的默认最大大小。更多信息，请参见**第 6.7 节 “位反转寻址和模寻址”**。
`-mlarge-code`	使用大代码模型进行编译。对于被调用函数的所在位置不作任何假定。 选择该选项时，单个大于 32k 的函数不受支持，由于函数内的所有转移均为短转移形式，它可能会导致汇编时错误。

注 1： 过程抽象器的行为是对函数进行内联的反向操作。该轮次设计为从整个翻译单元内的多个位置提取公共代码序列，并将它们放入一个公共代码区域。虽然该选项通常不会提高所生成代码的运行时性能，但它可以显著减小代码大小。使用 `-mpa` 编译的程序会较难调试；在使用 COFF 目标格式进行调试时，建议不要使用该选项。在生成汇编文件之后，将以编译的一个独立阶段的方式调用过程抽象器。该阶段不会进行跨翻译单元的优化。使能过程优化阶段时，行内汇编代码必须限制为使用有效的机器指令。一定不能使用无效的机器指令或指令序列，或者汇编器伪指令 （段伪指令、宏和包含文件等），否则过程抽象阶段将会发生失败，从而无法创建输出文件。

表 5-5： 特定于 16 位器件的选项 （续）

选项	定义
-mlarge-data	使用大数据模型进行编译。对于静态变量和外部变量的位置不作任何假定。
-mcpu=*target*	该选项用于选择目标处理器 ID （并且如果调用了汇编器和链接器，则将其传递给这些工具）。该选项会影响一些预定义常量的设置方式；关于更多信息，请参见**第 19.4 节 “预定义的宏名称”**。发行版随附的 Readme.htm 文件提供了可接受的目标处理器的完整列表。
-mpa(1)	使能过程抽象优化。对嵌套深度没有任何限制。 优化级别取决于编译器版本 （见**第 18 章 “优化”**）。
-mpa=*n*(1)	使能最高到层级 *n* 的过程抽象优化。如果 *n* 为零，则禁止优化。如果 *n* 为 1，则允许第一层抽象；即，可以将源代码中的指令序列抽象为子程序。如果 *n* 为 2，则允许第二层抽象；即，在第一层中被放入子程序的指令可以抽象为更深一层的子程序。对于更大的 *n* 值，这种模式将不断延续。实际结果就是将子程序调用嵌套深度限制为最大为 *n*。 优化级别取决于编译器版本 （见**第 18 章 “优化”**）。
-mno-pa(1)	不使能过程抽象优化。 **（这是默认设置。）**
-mno-isr-warn	默认情况下，如果识别到的中断向量名称未附加 __interrupt__，编译器会产生警告。该选项会禁止该功能。
-omf	选择编译器使用的 OMF （目标模块格式）。 *omf* 说明符可以为以下之一： elf 生成 ELF 目标文件。**（这是默认设置。）** coff 生成 COFF 目标文件。 用于 ELF 目标文件的调试格式为 DWARF 2.0。
-msmall-code	使用小代码模型进行编译。假定被调用函数处于邻近位置 （处于调用方的 32 K 字范围内）。 **（这是默认设置。）**
-msmall-data	使用小数据模型进行编译。假定所有静态变量和外部变量位于数据存储空间的低 8 KB 范围内。 **（这是默认设置。）**
-msmall-scalar	类似于 -msmall-data，只是仅假定静态标量和外部标量位于数据存储空间的低 8 KB 范围内。 **（这是默认设置。）**
-mtext=*name*	指定 -mtext=*name* 将导致 text （程序代码）被放入名为 *name* 的段，而不是默认的 .text 段。 = 两侧不应出现空格。
-mauxflash	将来自当前翻译单元的所有代码都放入辅助闪存。该选项仅在具有辅助闪存的器件上可用。
-msmart-io [=0|1|2]	该选项会尝试静态分析传递给 printf、scanf 以及这些函数的 “f” 和 “v” 版本的格式字符串。使用非浮点格式参数将被转换为使用库函数的仅整型版本。 -msmart-io=0 会禁止该选项，而 -msmart-io=2 会使编译器变得乐观，并转换带有变量或未知格式参数的函数调用。 **-msmart-io=1 是默认设置**，并且只会转换它可以验证的字面值。
--partition *n*	该选项选择双分区器件中的单个分区 *n* 作为目标，将输出文本约束为包含在一个分区中。

注 1： 过程抽象器的行为是对函数进行内联的反向操作。该轮次设计为从整个翻译单元内的多个位置提取公共代码序列，并将它们放入一个公共代码区域。虽然该选项通常不会提高所生成代码的运行时性能，但它可以显著减小代码大小。使用 -mpa 编译的程序会较难调试；在使用 COFF 目标格式进行调试时，建议不要使用该选项。在生成汇编文件之后，将以编译的一个独立阶段的方式调用过程抽象器。该阶段不会进行跨翻译单元的优化。使能过程优化阶段时，行内汇编代码必须限制为使用有效的机器指令。一定不能使用无效的机器指令或指令序列，或者汇编器伪指令 （段伪指令、宏和包含文件等），否则过程抽象阶段将会发生失败，从而无法创建输出文件。

© 2015 Microchip Technology Inc.

5.7.2 用于控制输出类型的选项

以下选项控制编译器产生的输出的类型。

表 5-6： 输出类型控制选项

选项	定义
-c	编译或汇编源文件，但不进行链接。默认的文件扩展名为 .o。
-E	在预处理阶段之后停止 （即，在运行相应编译器之前）。默认的输出文件为 stdout。
-o *file*	将输出放入 *file*。
-S	在正确编译之后 （即调用汇编器之前）停止。默认的输出文件扩展名为 .s。
-v	打印在每个编译阶段中执行的命令。
-x	您可以使用 -x 选项显式地指定输入语言： -x *language* 显式指定以下输入文件的语言 （而不是让编译器基于文件名后缀选择默认值）。该选项应用于随后的所有输入文件，直到遇到下一个 -x 选项为止。编译器支持以下值： c c-header cpp-output assembler assembler-with-cpp -x none 关闭指定的任何语言，从而根据文件名后缀来处理后续的文件。这是默认的行为，但如果使用了另一个 -x 选项，则需要使用该选项。 例如： xc16-gcc -x assembler foo.asm bar.asm -x none main.c mabonga.s 如果没有 -x none，编译器将假定所有输入文件均用于汇编器。
--help	打印命令行选项的说明。

5.7.3 用于控制 C 方言的选项

以下选项定义编译器使用的 C 方言的类型。

表 5-7： C 方言控制选项

选项	定义
-ansi	支持 （且仅支持）所有 ANSI 标准 C 程序。
-aux-info filename	将翻译单元中 （包括头文件中）声明和 / 或定义的所有函数的原型声明输出到给定文件名中。在除 C 之外的任何其他语言中，均会静默地忽略该选项。除了声明之外，该文件还会在注释中指示每个声明的来源 （源文件和行），声明是隐式、有原型还是无原型的 （在行号和冒号后的第一个字符中，分别为 I、N （对于新样式）或 O （对于旧样式）），以及它是来自声明还是定义（在随后的字符中，分别为 C 或 F）。对于函数定义，还会在声明之后的注释内提供 K&R 样式的参数列表及其声明。
-menable-fixed [=*rounding mode*]	使能定点变量类型和算术运算支持。（可选）将默认 *rounding mode*（舍入模式）设置为截断、常规或收敛之一。如果未指定 *rounding mode*，则默认为 **truncation**。
-ffreestanding	声明编译是在独立式环境中发生的。这隐含着 -fno-builtin。独立式环境是一个可能不存在标准库的环境，并且程序可能不一定从 main 启动。最明显的示例是 OS 内核。这等效于 -fno-hosted。
-fno-asm	不将 asm、inline 或 typeof 识别为关键字，从而使代码可以使用这些词作为标识符。您可以改为使用关键字 __asm__、 __inline__ 和 __typeof__。-ansi 隐含着 -fno-asm。
-fno-builtin -fno-builtin-*function*	不识别不以 __builtin_ 作为前缀开头的内置函数。
-fsigned-char	使类型 char 为有符号，类似于 signed char。 **（这是默认设置。）**
-fsigned-bitfields -funsigned-bitfields -fno-signed-bitfields -fno-unsigned-bitfields	这些选项控制在声明未使用 signed 或 unsigned 时位域为有符号还是无符号。默认情况下，这种位域为有符号，除非使用了 -traditional，在这种情况下位域总是为无符号。
-funsigned-char	使类型 char 为无符号，类似于 unsigned char。

© 2015 Microchip Technology Inc.

5.7.4 用于控制警告和错误的选项

警告是一些诊断消息，它们报告本质上不属于错误但具有风险的程序结构，或者指示可能存在错误。

您可以通过以 -W 开始的选项来请求产生许多特定的警告；例如，使用 -Wimplicit 来请求产生关于隐式声明的警告。这些特定警告选项中的每个选项也具有以 -Wno- 开始的否定形式，用于关闭警告；例如，-Wno-implicit。本手册仅列出两种形式中不属于默认设置的形式。

5.7.4.1 用于控制警告数量和类型的选项

以下选项控制编译器产生的警告的类型和数量。

表 5-8： -WALL 隐含的警告 / 错误选项

选项	定义
-fsyntax-only	检查代码的语法，但除此之外不执行任何操作。
-pedantic	发出严格 ANSI C 要求的所有警告；拒绝所有使用已禁止扩展的程序。
-pedantic-errors	类似于 -pedantic，只是产生错误而不是警告。
-w	禁止所有警告消息。
-Wall	结合本表中列出的所有 -W 选项。这将使能关于一些用户视为有问题且易于避免 （或通过修改来阻止警告）的程序结构的所有警告，即使是与宏配合使用。
-Wchar-subscripts	数组下标具有 char 类型时产生警告。
-Wcomment -Wcomments	每当/* 注释中出现注释开始序列/*，或每当//注释中出现反斜杠换行符时，产生警告。
-Wdiv-by-zero	产生关于编译时整型除以零的警告。要禁止该警告消息，请使用 -Wno-div-by-zero。不会产生关于浮点型除以零的警告，因为它是获取无穷大和 NaN 的合法方式。 **（这是默认设置。）**
-Werror-implicit-function-declaration	每当函数在声明之前使用时产生错误。
-Wformat	检查对于 printf 和 scanf 等的调用，以确保所提供参数具有对应于指定格式字符串的类型。
-Wimplicit	等效于同时指定 -Wimplicit-int 和 -Wimplicit-function-declaration。
-Wimplicit-function-declaration	每当函数在声明之前使用时产生警告。
-Wimplicit-int	声明未指定类型时产生警告。
-Wmain	main 的类型可疑时产生警告。 main 应为具有外部链接的函数，返回 int，接受 0 个、2 个或 3 个适当类型的参数。
-Wmissing-braces	聚合或联合的初始化中大括号不完整时产生警告。在以下示例中， a 的初始化中大括号不完整，但 b 的完整。 int a[2][2] = { 0, 1, 2, 3 }; int b[2][2] = { { 0, 1 }, { 2, 3 } };

-Wmultichar
-Wno-multichar
	如果使用了多字符 char 常量，则产生警告。此类常量通常为输入错误。由于它们具有实现定义的值，所以不应在可移植代码中使用它们。以下示例说明了多字符 char 常量的使用：
	char
	xx(void)
	{
	return('xx');
	}

表 5-8： -WALL 隐含的警告 / 错误选项 （续）

选项	定义
-Wparentheses	如果在某些上下文中省略了括号 （例如，在期望得到真值的上下文中进行赋值，或者对操作符进行了嵌套，而其优先顺序不明确），则产生警告。
-Wreturn-type	如果函数定义了返回类型，且返回类型默认为 int，则产生警告。此外，对于返回类型不是 void 的函数，如果 return 语句没有返回值，也产生警告。
-Wsequence-point	对由于违反 C 标准中的序列点规则而可能具有未定义语义的代码产生警告。 C 标准以序列点的形式定义了对 C 程序中的表达式进行求值的顺序，序列点代表程序各个部分之间的执行偏序：在序列点之前执行的那些部分和在它之后执行的那些部分。它们出现在以下位置：在完整表达式 （该表达式不属于更大表达式的一部分）的求值之后；在 &&、||、? ：或 , （逗号）操作符第一个操作数的求值之后；在调用函数之前（但在其参数和表示被调用函数的表达式的求值之后）；以及在某些其他位置。除了序列点规则所表述的之外，并未规定表达式的子表达式的求值顺序。所有这些规则仅描述偏序，而不是全序，因为，举例来说，如果在表达式中调用了两个函数，它们之间不存在序列点，则调用这两个函数的顺序是未指定的。但是，标准委员会已经规定，函数调用不发生重叠。并未规定对象值的修改在各序列点间何时生效。行为取决于这一点的程序具有未定义的行为； C 标准规定在上一个和下一个序列点之间，对象的存储值最多只能被表达式求值修改一次。此外，只有在确定将要保存的值时才能读取前一个值”。如果程序违反了这些规则，则任何特定实现上的结果将是完全不可预测的。 具有未定义行为的代码示例：a = a++;、a[n] = b[n++] 和 a[i++] = i;。该选项不会诊断一些更复杂的情况，并且它偶尔可能会给出误报结果，但通常情况下，它对于检测程序中的此类问题是相当有效的。
-Wswitch	每当 switch 语句的参数为枚举类型，但该枚举的一个或多个指定成员缺少对应的 case 语句时，产生警告。（使用 default 标号会阻止该警告。）在使用该选项时，超出枚举范围的 case 标号也会产生警告。
-Wsystem-headers	打印关于在系统头文件中找到的程序结构的警告消息。通常假定来自系统头文件的警告并不表明真正的问题，只会使编译器输出更难以阅读，所以通常会禁止这些警告。使用该命令行选项将指示编译器发出来自系统头文件的警告，如同它们是出现在用户代码中。但请注意，将 -Wall 与该选项一起使用并不会对系统头文件中的未知 pragma 伪指令产生警告；因此，还必须使用 -Wunknown-pragmas。
-Wtrigraphs	遇到任何三字符组合 （假设已使能它们）时产生警告。
-Wuninitialized	如果未先进行初始化而使用了自动变量，则产生警告。 只有在使能优化时才可能产生这些警告，因为它们需要只有在优化时才会计算的数据流信息。 只有对于将分配到寄存器的变量，才会出现这些警告。因此，对于声明为 volatile、已获取地址，或大小不为 1、2、4 或 8 字节的变量，不会出现这些警告。此外，对于结构、联合或数组 （即使它们处于寄存器中时），也不会出现这些警告。 请注意，对于仅用于计算一个从未被使用的值的变量，可能不会出现任何警告，因为数据流分析可能会在打印警告之前已删除这种计算。

© 2015 Microchip Technology Inc.

表 5-8： -WALL 隐含的警告 / 错误选项 （续）

选项	定义
-Wunknown-pragmas	当遇到编译器无法理解的 #pragma 伪指令时，产生警告。如果使用了该命令行选项，则甚至会对于系统头文件中的未知 pragma 伪指令发出警告。如果只是通过 -Wall 命令行选项使能警告，则不会发生这种情况。
-Wunused	每当一个变量在声明之外未被使用过，每当函数声明为静态但从未定义，每当声明了标号但未使用过，以及每当某个语句计算一个明显未被使用的结果时，产生警告。 要获取关于未用函数参数的警告，必须同时指定 -W 和 -Wunused。 将表达式强制类型转换为 void 会禁止对表达式产生该警告。类似地，unused 属性会禁止对未用变量、参数和标号产生该警告。
-Wunused-function	每当声明了静态函数但未定义或非内联静态函数未被使用过时，产生警告。
-Wunused-label	每当声明了标号但未使用过时，产生警告。要禁止该警告，请使用 unused 属性 （见**第 8.12 节 “变量属性”**）。
-Wunused-parameter	每当函数参数除了声明之外未被使用过时，产生警告。要禁止该警告，请使用 unused 属性 （见**第 8.12 节 “变量属性”**）。
-Wunused-variable	每当局部变量或非常量静态变量除了声明之外未被使用过时，产生警告。要禁止该警告，请使用 unused 属性 （见**第 8.12 节 “变量属性”**）。
-Wunused-value	每当语句计算一个明显未被使用的结果时，产生警告。要禁止该警告，请将表达式强制类型转换为 void。

5.7.4.2 -Wall 不隐含的选项

-Wall 不隐含以下 -W 选项。其中一些选项产生关于此类程序结构的警告：用户一般不视为可疑但有时可能希望检查的程序结构。其他一些选项则产生关于此类程序结构的警告：在某些情况下必需或难以避免的程序结构，且没有简单的方式可以通过修改代码来禁止警告。

表 5-9： -WALL 不隐含的警告 / 错误选项

选项	定义
-W	打印特定事件的额外警告消息。详细信息请参见**第 5.7.4.3 节 “-W 选项”**。
-Waggregate-return	如果定义或调用了返回结构或联合的函数，则产生警告。
-Wbad-function-cast	每当函数调用被强制类型转换为非匹配类型时，产生警告。例如，如果 int foof() 被强制类型转换为任何 * 类型，则产生警告。
-Wcast-align	每当对指针进行强制类型转换，从而增加了目标所需的对齐时，产生警告。例如，如果 char * 被强制类型转换为 int *，则产生警告。
-Wcast-qual	每当对指针进行强制类型转换，从而从目标类型中去除一个类型限定符时，产生警告。例如，如果 const char * 被强制类型转换为普通 char *，则产生警告。
-Wconversion	如果原型导致的类型转换与不存在原型时同一个参数发生的类型转换不同，则产生警告。这包括定点到浮点的转换 （以及反之）和更改定点参数宽度或符号的转换，但与默认提升相同时除外。 此外，如果某个负整型常量表达式隐式转换为无符号类型，则产生警告。例如，如果 x 为无符号，则对赋值 x=-1 产生警告。但不对诸如 (unsigned) -1 之类的显式强制类型转换产生警告。
-Werror	使所有警告变为错误。
-Winline	如果某个函数不能进行内联，但将它声明为内联或提供了 -finline-functions 选项，则产生警告。
-Wlarger-than-*len*	每当定义的对象大于 *len* 个字节时产生警告。
-Wlong-long -Wno-long-long	如果使用了 long long 类型，则产生警告。这是默认设置。要禁止警告消息，请使用 -Wno-long-long。只有在使用 -pedantic 标志时，才会考虑标志 -Wlong-long 和 -Wno-long-long。
-Wmissing-declarations	如果定义了某个全局函数，但无事先声明，则产生警告。即使定义本身提供了原型也如此。
-Wmissing-format-attribute	如果使能了 -Wformat，则也对于可指定为 format 属性的函数产生警告。请注意，这些只是可指定 format 属性的函数，不是已指定了该属性的函数。除非使能了 -Wformat，否则该选项不起作用。
-Wmissing-noreturn	对于可能指定为 noreturn 属性的函数产生警告。这些只是可指定 noreturn 的函数，不是已指定了该属性的函数。应仔细地手动验证这些函数。事实上，请不要在添加 noreturn 属性之前返回，否则可能会引入微小的代码生成错误。
-Wmissing-prototypes	如果定义了某个全局函数，但无事先的原型声明，则产生警告。即使定义本身提供了原型，也会发出该警告。（该选项可以用于检测未在头文件中声明的全局函数）。
-Wnested-externs	如果在函数内遇到 extern 声明，则产生警告。
-Wno-deprecated-declarations	不产生关于使用通过 deprecated 属性标记为已弃用的函数、变量和类型的警告。
-Wpadded	如果某个结构中包含填充 （用于对齐结构的某个成员或对齐整个结构），则产生警告。

© 2015 Microchip Technology Inc.

表 5-9： -WALL 不隐含的警告 / 错误选项 （续）

选项	定义
-Wpointer-arith	如果任何对象依赖于函数类型的大小或 void 的大小，则产生警告。编译器会将这些类型的大小赋值为 1，以方便 void * 指针和指向函数指针的计算。
-Wredundant-decls	如果任何对象在同一作用域内声明多次 （即使多个声明均有效，不改变任何设置），则产生警告。
-Wshadow	每当某个局部变量遮蔽另一个局部变量时，产生警告。
-Wsign-compare -Wno-sign-compare	如果在有符号值和无符号值之间进行比较，当有符号值被转换为无符号时，可能产生错误的结果，则产生警告。-W 也会使能该警告；要获得 -W 的其他警告而不产生该警告，请使用 -W -Wno-sign-compare。
-Wstrict-prototypes	如果某个函数在声明或定义时未指定参数类型，则产生警告。（如果前面具有指定参数类型的声明，则允许旧样式的函数定义，而不产生警告。）
-Wtraditional	对某些在传统和 ANSI C 中具有不同行为的程序结构产生警告。 • 宏主体中的字符串常量内出现的宏参数。在传统 C 中，它们将替换参数，但在 ANSI C 中，它们是常量的一部分。 • 某个函数在一个块中声明为外部函数，然后在该块结束之后使用了该函数。 • 具有 long 类型操作数的 switch 语句。 • 非静态函数声明之后跟随一个静态函数声明。一些传统 C 编译器不接受该程序结构。
-Wundef	如果在 #if 指令中对未定义的标识符进行求值，则产生警告。
-Wwrite-strings	为字符串常量赋予 const char[length] 类型，从而在将一个字符串常量的地址复制到非 const char* 指针时将获得警告。在编译时，这些警告帮助您查找可以尝试写入字符串常量的代码，但前提是您在声明和原型中使用 const 时很小心。否则，它只会令人烦扰，这也是 -Wall 不请求产生这些警告的原因。

5.7.4.3 -W 选项

使用 -W 命令行选项可以打印以下事件的额外警告消息：

- nonvolatile 自动变量可能被对 longjmp 的调用更改。只有在优化编译时才可能出现这些警告。编译器只能检测到对 setjmp 的调用。它无法知道 longjmp 会在哪里被调用；事实上，信号处理程序可能会在代码中的任意一点调用它。因此，即使事实上不存在任何问题，因为事实上无法在会导致问题的位置调用 longjmp，也可能会产生警告。
- 函数有可能通过 return *value*; 退出，也有可能通过 return; 退出。不传递任何返回语句而结束函数主体的情况会被当做 return; 处理。
- 表达式语句或逗号表达式的左侧不包含副作用。要禁止该警告，请将未用表达式强制类型转换为 void。例如，诸如 x[i,j] 之类的表达式会产生警告，但 x[(void)i,j] 不会。
- 无符号值通过 < 或 <= 与零进行比较。
- 出现诸如 x<=y<=z 之类的比较；这等效于 (x<=y ? 1 : 0) <= z，这是一种不同于普通数学表示的解释。
- 诸如 static 之类的存储类别说明符在声明中没有放在最前面。根据 C 标准，这种用法是过时的。
- 如果还指定了 -Wall 或 -Wunused，则会产生关于未用参数的警告。
- 在有符号值和无符号值之间进行比较，当有符号值被转换为无符号时，可能产生错误的结果。（但如果也指定了 -Wno-sign-compare，则不产生警告）。
- 聚合数据类型的初始化大括号不完整。例如，以下代码将会产生此类警告，因为 x.h 的初始化两边缺少大括号：

```
struct s { int f, g; };
struct t { struct s h; int i; };
struct t x = { 1, 2, 3 };
```

- 聚合数据类型的初始化未对所有成员都进行初始化。例如，以下代码将产生此类警告，因为 x.h 会隐式地初始化为零：

```
struct s { int f, g, h; };
struct s x = { 3, 4 };
```


© 2015 Microchip Technology Inc.

5.7.5 用于调试的选项

以下选项用于调试。

表 5-10： 调试选项

选项	定义
-g	产生调试信息。 编译器支持将 -g 与 -O 配合使用，使得可以调试经过优化的代码。优化代码所采取的捷径有时可能会产生令人惊讶的结果： • 一些已声明的变量可能根本不存在； • 控制流可能短暂地异常转移； • 某些语句可能因为计算常量结果或已得到其值而不会被执行； • 某些语句可能因为被移出循环而在不同的位置执行。 尽管如此，证明还是可以对优化输出进行调试。这使得对可能存在问题的程序使用优化器是合理的。
-Q	使编译器在编译每个函数时打印其名称，并在完成时打印一些关于每个轮次的统计信息。
-save-temps	不删除中间文件。将它们放在当前目录中，并基于源文件命名它们。因而，使用 -c -save-temps 编译 foo.c 会产生以下文件： foo.i （预处理文件） foo.p （预过程抽象汇编语言文件） foo.s （汇编语言文件） foo.o （目标文件）

5.7.6 用于控制优化的选项

以下选项控制编译器优化。可用的优化级别取决于编译器版本 （见**第 18 章 “优化”**）。

表 5-11： 常规优化选项

选项	版本	定义
-O0	所有版本	不优化。**（这是默认设置。）** 不使用 -O 时，编译器的目标是降低编译开销，以及使调试产生预期的结果。语句是独立的：如果使用语句之间的断点停止程序，则可以为任意变量赋予新值或将程序计数器更改为指向函数中的任何其他语句，并得到预期从源代码获得的结果。 编译器只会将声明为 register 的变量分配到寄存器中。
-O -O1	所有版本	优化。优化编译需要的时间稍长一些，对于大函数需要更多的主机存储器。 使用 -O 时，编译器会尝试减小代码长度和执行时间。 指定 -O 时，编译器会开启 -fthread-jumps 和 -fdefer-pop。编译器会开启 -fomit-frame-pointer。
-O2	标准版和专业版	更多优化。编译器执行几乎所有不涉及空间 - 速度权衡的受支持优化。 -O2 会开启除循环展开（-funroll-loops）、函数内联 （-finline-functions）和严格别名优化（-fstrict-aliasing）之外的所有可选优化。此外，它还会开启帧指针消除（-fomit-frame-pointer）。与 -O 相比，该选项会增加编译时间但可提高生成代码的性能。
-O3	专业版	针对速度优化。 -O3 会开启 -O2 指定的所有优化，还会开启 inline-functions 选项。
-Os	专业版	针对长度优化。 -Os 会使能通常不会增加代码长度的所有 -O2 优化。它还会执行旨在减小代码长度的进一步优化。

5.7.6.1 用于特定优化控制的选项

以下选项控制特定的优化。 `-O2` 选项会开启除 `-funroll-loops`、 `-funroll-all-loops` 和 `-fstrict-aliasing` 之外的所有这些优化。

在极少数情况下，需要执行优化 “微调”时，可以使用以下标志。

表 5-12： 特定优化选项

选项	定义
`-falign-functions` `-falign-functions=`*n*	将函数起始位置对齐到下一个大于 *n* 的 2 的幂，最多跳过 *n* 个字节。例如，`-falign-functions=32` 会将函数对齐到下一个 32 字节边界处，但对于 `-falign-functions=24`，只有在跳过的字节数为 23 字节或更少字节时，才会对齐到下一个 32 字节边界处。 `-fno-align-functions` 和 `-falign-functions=1` 是等效的，表示不对函数进行对齐。 只有 *n* 为 2 的幂时，汇编器才支持该标志，所以 *n* 会被向上舍入。如果未指定 *n*，则使用依赖机器的默认值。
`-falign-labels` `-falign-labels=`*n*	将所有分支的目标地址对齐到 2 的幂的边界处，最多跳过 *n* 个字节，类似于 `-falign-functions`。该选项很容易会使代码速度变慢，因为它必须在以通常的代码流到达分支的目标地址时插入空操作。 如果 `-falign-loops` 或 `-falign-jumps` 适用，并且大于该值，则改为使用它们的值。 如果未指定 *n*，则使用依赖于机器的默认值，该默认值很可能为 1，意味着不进行对齐。
`-falign-loops` `-falign-loops=`*n*	将循环对齐到属于 2 的幂的边界处，最多跳过 *n* 个字节，类似于 `-falign-functions`。预期目的是循环会被执行许多次，以补偿所执行的所有空操作。 如果未指定 *n*，则使用依赖机器的默认值。
`-fcaller-saves`	允许将值分配到会被函数调用破坏的寄存器中，通过在此类调用前后发出额外的指令来保存和恢复寄存器。只有所产生的代码优于其他情况下产生的代码时，才会进行这种分配。
`-fcse-follow-jumps`	在公共子表达式消除中，当跳转的目标除了该跳转没有任何其他路径可达时，扫描该跳转指令。例如，当 CSE 遇到一个带有 `else` 子句的 `if` 语句时， CSE 会在条件测试为假时跟随跳转。
`-fcse-skip-blocks`	它类似于 `-fcse-follow-jumps`，但会使 CSE 跟随条件性跳过语句块的跳转。当 CSE 遇到没有 `else` 子句的简单 `if` 语句时， `-fcse-skip-blocks` 会使 CSE 跟随 `if` 语句主体前后的跳转。
`-fexpensive-optimizations`	执行许多开销相对较大的次要优化。
`-ffunction-sections` `-fdata-sections`	在输出文件中将每个函数或数据项放入它自己的段中。函数的名称或数据项的名称决定输出文件中段的名称。 只有使用这些选项具有明显的好处时，才使用这些选项。指定这些选项时，汇编器和链接器可能会创建更大的目标文件和可执行文件，速度也会较慢。 另请参见**第 5.7.6.2 节 “`-ffunction-sections` 选项”**。
`-fgcse`	执行全局公共子表达式消除轮次。该轮次也会执行全局常量和复制传播。
`-fgcse-lm`	使能 `-fgcse-lm` 时，全局公共子表达式消除将尝试移动只能被向其中存储破坏的装入。这允许将包含装入 / 存储序列的循环更改为循环外的装入，以及循环内的复制 / 存储。

© 2015 Microchip Technology Inc.

表 5-12： 特定优化选项 （续）

选项	定义
-fgcse-sm	使能 -fgcse-sm 时，将在全局公共子表达式消除之后运行存储移动轮次。该轮次会尝试将存储移出循环。与 -fgcse-lm 配合使用时，包含装入 / 存储序列的循环可以更改为循环之前的装入和循环之后的存储。
-fno-defer-pop	总是在函数返回时立即弹出每个函数调用的参数。编译器通常允许几个函数调用的参数累积在堆栈中，并一次将所有参数弹出堆栈。
-fno-peephole -fno-peephole2	禁止特定于机器的窥孔优化。窥孔优化发生在编译期间的不同时间点。 -fno-peephole 会禁止对机器指令进行窥孔优化，而 -fno-peephole2 会禁止高级窥孔优化。要完全禁止窥孔，请同时使用这两个选项。
-foptimize-register-move -fregmove	尝试重新分配转移指令中的寄存器编号，并作为其他简单指令的操作数，以最大程度提高关联的寄存器数量。 -fregmove 和 -foptimize-register-moves 是相同的优化。
-frename-registers	尝试通过使用在寄存器分配之后留下的寄存器来避免经过调度的代码中的假依赖性。这种优化对于具有许多寄存器的处理器比较有用。但它可能使调试无法进行，因为变量将不会存储在固定的寄存器中。
-frerun-cse-after-loop	在执行循环优化之后，重新运行公共子表达式消除。
-frerun-loop-opt	运行循环优化器两次。
-fschedule-insns	尝试重新排列指令的顺序，以消除 dsPIC DSC “先写后读”停顿 （关于更多详细信息，请参见您所用器件的系列参考手册。通常可以在不影响代码大小的情况下提高性能。
-fschedule-insns2	类似于 -fschedule-insns，但请求在完成寄存器分配之后再执行一次指令调度。
-fstrength-reduce	执行降低循环强度和消除迭代变量的优化。

表 5-12： 特定优化选项 （续）

选项	定义

-fstrict-aliasing	允许编译器采用适用于所编译语言的最严格别名规则。对于 C，这会基于表达式的类型激活优化。特别是，假定一种类型的对象不会与不同类型的对象位于同一地址，除非类型几乎相同。例如， unsigned int 类型可以为 int 类型的别名，但不能为 void* 或 double 类型的别名。字符类型可以为任何其他类型的别名。
	请特别注意类似以下形式的代码：
	union a_union {
	  int i;
	  double d;
	};
	
	int f() {
	  union a_union t;
	  t.d = 3.0;
	  return t.i;
	}
	不读取最近写入的联合成员，而读取其他联合成员的做法 （称为 “类型双关”）很常见。即使对于 -fstrict-aliasing，也允许类型双关，前提是通过联合类型来访问存储器。所以，上面的代码会按预期方式运行。但是，以下代码可能不会：
	int f() {
	  a_union t;
	  int * ip;
	  t.d = 3.0;
	  ip = &t.i;
	  return *ip;
	}

-fthread-jumps	执行优化，检查一个跳转分支的目标位置是否存在另一个条件比较，并且该条件比较包含在前一个比较语句之内。如果是这样，则将第一个分支重定向到第二个分支的目标位置或紧随其后的位置，这取决于条件是真还是假。
-funroll-loops	执行循环展开优化。这仅仅针对可以在编译时或运行时确定迭代次数的循环。 -funroll-loops 隐含着 -fstrength-reduce 和 -frerun-cse-after-loop。
-funroll-all-loops	执行循环展开优化。对于所有循环执行该优化，并且通常会使程序运行速度变慢。 -funroll-all-loops 隐含着 -fstrength-reduce 和 -frerun-cse-after-loop。

© 2015 Microchip Technology Inc.

5.7.6.2 -ffunction-sections 选项

-ffunction-sections 命令行选项会尝试将所有函数都放入自己的段中。但是，有许多条件会影响其实际操作。以下进行了总结：

- 正常（非中断）函数将在当前段名前加“.”，再在当前段名后加上函数名，例如：
  ```
  void foo() {}
  ```
  会被放入段 .text.foo（默认代码段名为 .text）。
- 默认段名可以使用 -mtext 选项进行修改。如果使用了该选项，则当前段名会被更改。例如，如果指定了 -mtext=mytext，则以上函数会被放入 mytext.foo。
- 如果函数具有段属性，则它会被放入指定的段，没有例外。因此：
  ```
  void __attribute__((section("mytext"))) foo() {}
  ```
  总是会被放入段 mytext，无论是否指定 -ffunction-sections。
- 中断函数通常会被放入在正常段名（如上所述）前加 .isr 的特殊段中。因此，如果当前段名为 .text（默认），则 ISR 会被放入 .isr.text.*function_name*。

  如果使用 -mtext 来更改默认段名的名称，则它会被替换，而不是使用 .text。但是，如果指定的段使用了段属性， .isr 仍然会被加到段名前。

  前面加上 .isr 是为了使 --gc-sections 选项不丢弃中断函数。它们必须保留。

5.7.6.3 用于指定独立于机器的标志的选项

`-f`*`flag`* 形式的选项指定独立于机器的标志。大多数标志同时具有肯定和否定两种形式； `-ffoo` 的否定形式为 `-fno-foo`。在下表中，仅列出了两种形式之一 （不是默认设置的形式）。

表 5-13： 独立于机器的优化选项

选项	定义
`-finline-functions`	将所有简单函数合并到调用它们的函数中。编译器会试探性地确定哪些函数足够简单，值得以这种方式合并。如果对某个给定函数的所有调用均被合并，并且该函数声明为 `static`，则通常该函数本身不作为汇编代码输出。
`-finline-limit=`*`n`*	默认情况下，编译器会限制可以内联的函数的大小。该标志用于控制显式标记为内联的函数 （即，使用 `inline` 关键字标记）的这一限制。 *n* 是可以内联的函数的大小 （以伪指令数量表示，不包含参数处理）。 n 的默认值为 10000。增大该值会导致被内联的代码更多，代价是增加编译时间和存储器消耗。 减小该值通常会使编译速度更快，更少的代码被内联 （可能意味着程序的速度变慢）。该选项对于使用内联的程序特别有用。 **注：**在该特定上下文中，伪指令代表函数大小的抽象测量。它不代表汇编指令条数，因而，其确切含义可能会因编译器版本而变化。
`-fkeep-inline-functions`	即使对某个给定函数的所有调用都进行了合并，并且该函数声明为 `static`，仍然输出该函数的一个独立的运行时可调用版本。该开关不会影响 `extern` 内联函数。
`-fkeep-static-consts`	在未开启优化时，发出声明为 static const 的变量，即使变量未被引用。 在默认情况下，编译器会使能该选项。如果希望无论是否开启了优化，均强制编译器检查变量是否被引用，请使用 `-fno-keep-static-consts` 选项。
`-fno-function-cse`	不将函数地址放入寄存器；使调用常量函数的每条指令显式地包含该函数的地址。该选项产生的代码效率较低，但不使用该选项时执行的优化可能会扰乱一些改变汇编器输出的特殊技巧。
`-fno-inline`	忽略 `inline` 关键字。通常该选项用于阻止编译器展开任何内联函数。如果未使能优化，则不会展开任何内联函数。
`-fomit-frame-pointer`	对于不需要帧指针的函数，不在寄存器中保存帧指针。这可以免去保存、设置和恢复帧指针的指令；这也使许多函数中有一个额外的寄存器可用。
`-foptimize-sibling-calls`	优化同属调用和尾递归调用。

© 2015 Microchip Technology Inc.

5.7.7 用于控制预处理器的选项

以下选项控制编译器预处理器。

表 5-14： 预处理器选项

选项	定义

-A*question* (*answer*)	在使用预处理条件语句 （如 #if #*question*(*answer*)）对问题 *question* 进行测试的情况下，断言该问题的答案 *answer*。 -A 会禁止通常描述目标机器的标准断言。
	例如，可以如下声明 main 的函数原型：
	#if #environ(freestanding)
	int main(void);
	#else
	int main(int argc, char *argv[]);
	#endif
	然后，可以使用 -A 命令行选项在两个原型之间进行选择。例如，要选择两者中的第一个，可以使用以下命令行选项：
	-Aenviron(freestanding)

-A -*predicate* =*answer*	取消带谓词 *predicate* 和答案 *answer* 的断言。
-A *predicate* =*answer*	使用谓词 *predicate* 和答案 *answer* 作出断言。这种形式优于旧形式 -A *predicate*(*answer*) （它仍然受支持），因为它不使用 shell 特殊字符。
-C	指示预处理器不要丢弃注释。与 -E 选项配合使用。
-dD	指示预处理器不要按正确的顺序将宏定义移动到输出中。
-D*macro*	使用字符串 1 作为定义来定义宏 *macro*。
-D*macro*=*defn*	将宏 *macro* 定义为 *defn*。在任何 -U 选项之前处理命令行中的所有 -D 实例。
-dM	指示预处理器仅输出在预处理结束时有效的宏定义的列表。与 -E 选项配合使用。
-dN	类似于 -dD，只是省略宏参数和内容。仅在输出中包含 #define name。
-fno-show-column	不在诊断中打印列号。如果诊断是由不理解列号的程序 （例如 dejagnu）进行扫描的，则可能需要如此。
-H	打印使用的每个头文件的名称以及其他正常活动。
-I-	只有对于 #include "*file*" 的情形，才搜索在 -I- 选项之前使用 -I 选项指定的所有目录；对于 #include <*file*>，不搜索它们。 如果在 -I- 选项之后，使用 -I 选项指定了其他目录，则对于所有 #include 伪指令搜索这些目录。（通常以这种方式使用所有 -I 目录。） 此外， -I- 选项会禁止使用当前目录 （当前输入文件所在的位置）作为 #include "*file*" 的第一个搜索目录。没有任何办法可以覆盖 -I- 的这种作用。通过 -I，您可以指定搜索调用编译器时为当前目录的目录。这与预处理器默认情况下执行的操作并不完全相同，但它往往可以令人满意。 -I- 不会禁止使用头文件的标准系统目录。因此， -I- 和 -nostdinc 是独立的。
-I*dir*	将目录 *dir* 添加到要在其中搜索头文件的目录列表的开头。这可用于覆盖系统头文件，替换为您自己的版本，因为这些目录在系统头文件目录之前搜索。如果使用了多个 -I 选项，目录将按从左到右的顺序扫描；然后扫描标准系统目录。

表 5-14： 预处理器选项 （续）

选项	定义
-idirafter *dir*	将目录 *dir* 添加到第二个包含路径中。在主包含路径 （-I 添加到的路径）的所有目录中都找不到头文件时，搜索第二个包含路径上的目录。
-imacros *file*	在处理常规输入文件之前，将文件作为输入处理，并丢弃所产生的输出。由于从该文件生成的输出被丢弃， -imacros *file* 的唯一作用是使文件中定义的宏在主输入中可用。 命令行上的所有 -D 和 -U 选项总是在 -imacros *file* 之前处理，无论这些选项的写入顺序如何。所有 -include 和 -imacros 选项按这些选项的写入顺序处理。
-include *file*	在处理常规输入文件之前，将文件作为输入进行处理。实际上，将先编译文件的内容。 命令行上的所有 -D 和 -U 选项总是在 -include *file* 之前处理，无论这些选项的写入顺序如何。所有 -include 和 -imacros 选项按这些选项的写入顺序处理。
-iprefix *prefix*	指定 *prefix* 作为随后 -iwithprefix 选项的前缀。
-isystem *dir*	将目录添加到第二个包含路径的开头，从而将其标记为系统目录，使它得到应用于标准系统目录的相同特殊处理。
-iwithprefix *dir*	将目录添加到第二个包含路径中。产生目录名称的方式是将前缀和 *dir* 连接，其中的前缀是先前使用 -iprefix 指定的。如果尚未指定前缀，则使用包含编译器安装路径的目录作为默认值。
-iwithprefixbefore *dir*	将目录添加到主包含路径中。产生目录名称的方式是将前缀和 *dir* 连接，与 -iwithprefix 的情形一样。
-M	指示预处理器输出适合于 make 的规则，该规则描述每个目标文件的依赖性。对于每个源文件，预处理器会输出一个 make 规则，其目标为该源文件的目标文件名，其依赖性为它使用的所有 #include 头文件。该规则可以为单行，如果它很长，也可以使用 \-newline 换行。规则的列表打印到标准输出中，而不打印到经预处理的 C 程序。 -M 隐含着 -E （见**第 5.7.2 节 “用于控制输出类型的选项”**）。
-MD	类似于 -M，但依赖性信息写入一个文件，并且编译继续进行。包含依赖性信息的文件的名称与源文件相同，并带有 .d 扩展名。
-MF *file*	与 -M 或 -MM 配合使用时，指定用于写入依赖性的文件。如果未给出 -MF 开关，预处理器会将规则发送到与预处理输出发送到的相同位置。 与驱动程序选项 -MD 或 -MMD 和 -MF 配合使用时，覆盖默认的依赖性输出文件。
-MG	将缺少的头文件视为生成的文件，并假定它们位于与源文件相同的目录中。如果指定了 -MG，则也必须指定 -M 或 -MM。 -MG 不支持与 -MD 或 -MMD 配合使用。
-MM	类似于 -M，但输出仅提及使用 #include “*file*” 包含的用户头文件。通过 #include<*file*> 包含的系统头文件会被省略。
-MMD	类似于 -MD，只是仅提及用户头文件，不提及系统头文件。
-MP	该选项指示 CPP 为除主文件之外的每个依赖性添加一个假目标，使每个依赖性均不依赖于任何对象。如果删除头文件时未更新 make-file 而使之匹配，这些假规则可以避开 make 发出的错误。 以下是典型的输出： test.o:test.c test.h test.h:

© 2015 Microchip Technology Inc.

表 5-14： 预处理器选项 （续）

选项	定义
-MQ	与 -MT 相同，但它会为对于 make 具有特殊含义的字符加上引号。 -MQ '$(objpfx)foo.o' 得到 $$(objpfx)foo.o：foo.c 默认目标会自动加引号，如同为它指定了 -MQ。
-MT *target*	更改依赖性生成发出的规则的目标。默认情况下， CPP 会采用主输入文件的名称，包括任何路径，删除任何文件后缀 （例如 .c），并追加平台的常用目标后缀。结果就是目标。 -MT 选项会将目标设置为与您指定的字符串完全相同。如果需要多个目标，可以将它们指定为 -MT 的单个参数，或使用多个 -MT 选项。 例如： -MT '$(objpfx)foo.o' 可能得到 $(objpfx)foo.o：foo.c
-nostdinc	不在标准系统目录中搜索头文件。仅搜索使用 -I 选项指定的目录 （以及当前目录，如适用）。（关于 -I 的信息，请参见**第 5.7.10 节 “用于目录搜索的选项”**）。 通过同时使用 -nostdinc 和 -I-，可以将包含文件搜索路径限制为仅包含显式指定的目录。
-P	指示预处理器不生成 #line 伪指令。与 -E 选项配合使用 （见**第 5.7.2 节 “用于控制输出类型的选项”**）。
-trigraphs	支持 ANSI C 三字符组合。 -ansi 选项也具有这种作用。
-U*macro*	取消定义宏 *macro*。 -U 选项在所有 -D 选项之后，但在任何 -include 和 -imacros 选项之前起作用。
-undef	不要预定义任何非标准宏 （包括架构标志）。

5.7.8 用于汇编的选项

以下选项控制汇编器操作。关于可用选项的更多信息，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

表 5-15： 汇编选项

选项	定义
-Wa,*option*	将*option*作为一个选项传递给汇编器。如果*option*中包含逗号，则在逗号处将其拆分为多个选项。例如，要生成汇编列表文件，请使用 -Wa,-a。

5.7.9 用于链接的选项

如果使用了 -c、 -S 或 -E 中的任何选项，则链接器不会运行，并且不应将目标文件名用作参数。关于可用选项的更多信息，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

表 5-16： 链接选项

选项	定义
--fill=*options*	填充未用的程序存储器。格式为： --fill=[wn:]expression[@address[:end_address] | unused] address 和 end_address 将指定要填充的程序存储器地址范围。如果未提供 end_address，则 expression 将被写入地址为 address 的特定存储单元。可指定 unused 的可选立即数值以指示将填充所有未使用的存储器。如果未提供存储单元参数，则将填充所有未使用的存储器。 expression 将说明如何填充指定的存储器。以下选项可用： 单个值 xc16-ld --fill=0x12345678@unused 范围值 xc16-ld --fill=1,2,3,4,097@0x9d000650:0x9d000750 递增值 xc16-ld --fill=7+=911@unused 默认情况下，链接器将使用指令字长度的数据进行填充。对于 16 位器件，默认的填充宽度为 24 位。但是，您可以使用 [wn:] 指定值宽度，其中 w 为填充值的宽度， n 属于 [1, 3]。 可在命令行上指定多个填充选项；链接器将始终先处理特定存储单元的填充选项。
--gc-sections	在链接时从代码中移除僵死函数。 仅支持ELF项目。要充分利用该功能，可在编译器命令行中添加-ffunction-sections选项。
-L*dir*	将目录 *dir* 添加到命令行选项 -l 指定、要用于搜索库的目录列表中。
-legacy-libc	使用旧式包含文件和库 （v3.24 和先前版本）。 包含文件和库的格式在 v3.25 中发生了更改，以便与 HI-TECH C 编译器格式匹配。
-l*library*	链接时搜索名为 *library* 的库。 链接器会在标准目录列表中搜索库，它实际上是一个名为 lib*library*.a 的文件。然后链接器会使用该文件，如同通过名称精确地指定了它。 它会对写入该选项的命令产生作用；链接器将按指定库和目标文件的顺序来处理它们。因此，foo.o -lz bar.o 会在文件 foo.o 之后但在 bar.o 之前搜索库 z。如果 bar.o 引用了 libz.a 中的函数，则可能不会装入这些函数。 搜索的目录包括几个标准系统目录，再加上使用 -L 指定的所有目录。 通常，通过这种方式找到的文件是库文件 （其成员为目标文件的归档文件）。链接器通过扫描归档文件来处理它，即查找定义了目前已引用但未定义的符号的成员。但如果找到的文件是一个普通目标文件，则它以通常的方式链接。使用 -l 选项 （例如， lmylib-）和指定文件名 （例如， libmylib.a）之间的唯一区别在于 -l 会按照指定搜索几个目录。 默认情况下，会指示链接器在 <install-path>\lib 中搜索使用 -l 选项指定的库。 使用**第 19.4 节 “预定义的宏名称”**中定义的环境变量可以改写该行为。

© 2015 Microchip Technology Inc.

表 5-16： 链接选项 （续）

选项	定义
-nodefaultlibs	在链接时，不要使用标准系统库。仅将您指定的库传递给链接器。编译器可能会生成对 memcmp、 memset 和 memcpy 的调用。这些入口通常由标准编译器库中的入口解析。指定该选项时，应通过其他某种机制提供这些入口点。
-nostdlib	链接时不要使用标准系统启动文件或库。不传递启动文件给链接器，只传递指定的库给链接器。编译器可能会生成对 memcmp、memset 和 memcpy 的调用。这些入口通常由标准编译器库中的入口解析。指定该选项时，应通过其他某种机制提供这些入口点。
-s	从可执行文件中删除所有符号表和重定位信息。
-T *script*	指定在链接时使用的链接描述文件 *script*。该选项会被转换为等效的 -T 链接器选项。
-u *symbol*	假定 *symbol* 未定义，通过强制链接库模块来定义该符号。可对不同符号多次使用 -u 来强制装入其他库模块，这是合法的。
-Wl,*option*	将 *option* 作为一个选项传递给连接器。如果 *option* 中包含逗号，则在逗号处将其拆分为多个选项。 例如，要生成映射文件，请使用 -W1, -Map=Project.map。
-Xlinker *option*	将 *option* 作为一个选项传递给连接器。可以使用该选项来提供编译器不知道如何识别的特定于系统的链接器选项。

5.7.10 用于目录搜索的选项

以下选项用于向编译器指定在何处查找要搜索的目录和文件。

表 5-17： 目录搜索选项

选项	定义
-specs=*file*	在编译器读入标准 spces 文件之后处理文件，以覆盖 xc16-gcc 驱动程序在确定将哪些开关传递给 xc16-cc1、 xc16-as、 xc16-ld 等时使用的默认设置。可以在命令行上指定多个 -specs=*file*，它们将按从左到右的顺序处理。

5.7.11 用于代码生成约定的选项

-f*flag* 形式的选项指定独立于机器的标志。大多数标志都具有肯定和否定两种形式； -ffoo 的否定形式为 -fno-foo。在下表中，仅列出了两种形式之一 （不是默认设置的形式）。

表 5-18： 代码生成约定选项

选项	定义
-fargument-alias -fargument-noalias -fargument- noalias-global	指定参数之间以及参数与全局数据之间可能存在的关系。 -fargument-alias 指定实参 （形参）可以互为别名，并且可以为全局存储的别名。 -fargument-noalias 指定实参不互为别名，但可以为全局存储的别名。 -fargument-noalias-global 指定实参不互为别名，也不为全局存储的别名。 每种语言会自动使用语言标准所要求的任何选项。您不需要自己使用这些选项。
-fcall-saved-*reg*	将名为 *reg* 的寄存器视为由函数保存的可分配寄存器。甚至可以在其中分配临时变量或跨调用存活的变量。如果函数使用了寄存器 *reg*，则通过这种方式编译的函数会保存和恢复该寄存器。 对帧指针或堆栈指针使用该标志是错误的。对在机器的执行模型中具有固定重要作用的其他寄存器使用该标志会产生灾难性的结果。 对可能返回函数值的寄存器使用该标志会产生一种不同的灾难结果。 该标志应在所有模块中一致地使用。
-fcall-used-*reg*	将名为 *reg* 的寄存器视为被函数调用破坏的可分配寄存器。可以在其中分配临时变量或不会跨调用存活的变量。通过这种方式编译的函数不会保存和恢复寄存器 *reg*。 对帧指针或堆栈指针使用该标志是错误的。对在机器的执行模型中具有固定重要作用的其他寄存器使用该标志会产生灾难性的结果。 该标志应在所有模块中一致地使用。
-ffixed-*reg*	将名为 *reg* 的寄存器作为固定寄存器处理；生成的代码应永远不引用它 （除了可能用作堆栈指针、帧指针或用于其他某种固定作用）。 *reg* 必须为寄存器的名称 （例如， -ffixed-w3）。
-fno-ident	忽略 #ident 伪指令。
-fpack-struct	将所有结构成员无空隙地压缩在一起。通常不希望使用该选项，因为它会使代码不是最优化的，且结构成员的偏移量与系统库不相符。 dsPIC® DSC 器件要求将字对齐到偶字节边界处，所以在使用 packed 属性时必须小心，以避免运行时寻址错误。

© 2015 Microchip Technology Inc.

表 5-18： 代码生成约定选项 （续）

选项	定义
-fpcc-struct-return	像长值一样，将短 struct 和 union 值返回到存储器中，而不是返回到寄存器中。该约定的效率较低，但其优点是可以使 16 位编译器编译的文件与其他编译器编译的文件兼容。短结构和联合指长度和对齐都与整型匹配的结构和联合。
-fno-short-double	默认情况下，编译器使用与 float 等效的 double 类型。该选项使 double 与 long double 等效。如果模块通过参数传递直接或通过共享缓冲空间间接共用 double 数据，跨模块混合使用该选项可能会产生异常结果。无论使用哪个开关设置，随产品提供的库都可正常工作。
-fshort-enums	按照 enum 类型声明的可能值范围的需要，为其分配字节。具体地说， enum 类型等效于具有足够空间的最小整型。
-fverbose-asm -fno-verbose-asm	在生成的汇编代码中放入额外的注释信息，使它更具可读性。 默认设置为 -fno-verbose-asm，它会省略额外的信息，这在比较两个汇编文件时很有用。

5.8 MPLAB X IDE 工具链或 MPLAB IDE 工具套件等效选项

关于 MPLAB X IDE 中的相关编译器选项的信息，请参见**第 4 章 “XC16 工具链和 MPLAB X IDE”**

© 2015 Microchip Technology Inc.

第 6 章 与器件相关的特性

6.1 简介

MPLAB XC16 C 编译器提供了一些纯粹与器件相关的特性。

- 器件支持
- 器件头文件
- 堆栈
- 配置位访问
- 使用 SFR （MCU 中）
- 位反转寻址和模寻址

6.2 器件支持

如**第 1 章 “编译器概述”**所述，该编译器支持所有 Microchip 16 位器件：dsPIC30/33 数字信号控制 （DSC）和 PIC24 单片机 （MCU）。

要确定您所用编译器版本的器件支持，请参见编译器安装文件夹的 `docs` 子文件夹中的文件 `Readme_XC16.html`。例如：

```
C:\Program Files (x86)\Microchip\xc16\v1.10\docs\Readme_XC16.html
```

6.3 器件头文件

在您编写的每个 C 源文件中通常包含的一个头文件是 `xc.h`，它是一个通用头文件，在您编译项目时，它将会包含其他特定于器件和架构的头文件。

包含该文件之后，将可以通过一些特殊变量来访问 SFR，以及通过一些宏来进行特殊存储器访问或包含一些特殊指令。

请避免在代码中包含特定于芯片的头文件，因为这会降低可移植性。但是，特定于器件的编译器头文件存储在 `support/`*`family`*`/h` 目录以供参考。

关于汇编包含文件 （`*.inc`）的信息，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

6.3.1 寄存器定义文件

第 6.3 节 “器件头文件”介绍的处理器头文件指定了每种器件的所有 SFR，但它们未定义 SFR 的地址。`support/family/gld` 目录中包含一组特定于器件的链接描述文件，每种器件对应一个链接描述文件。这些链接描述文件定义了 SFR 地址。要使用这些文件之一，请指定链接器命令行选项：

```
-T p30fxxxx.gld
```

其中， `xxxx` 对应于器件部件编号。

例如，假定有一个名为 `app2010.c` 的文件，它包含用于 dsPIC30F2010 器件的应用程序，则它可以使用以下命令行进行编译和链接：

```
xc16-gcc -mcpu=30f2010 -o app2010.out -T p30f2010.gld app2010.c
```

`-o` 命令行选项指定了输出可执行文件，`-T` 选项给出了 dsPIC30F2010 器件的链接描述文件名称。如果在当前目录中找不到 `p30f2010.gld`，则链接器会在它的已知库路径中搜索。默认搜索路径包括预安装的库和链接描述文件的所有位置。

在进行任何针对项目的修改之前，应先将相应的链接描述文件（随编译器提供）复制到您的项目目录中。

6.3.2 编译时存储器信息

每个器件头文件均包含一些宏，用以帮助确定存储器大小。对于每个存储区域（RAM、闪存、向量表和配置字等），头文件将定义两个符号：基址和以字节为单位的长度。

符号名称按以下模板构造：`__<region_id>_BASE` 或 `__<region_id>_LENGTH`。这些符号可以在使用预处理符号的任意位置使用。

例如：

```
#if __DATA_LENGTH < 0x1000
#error Please use a device with at least 4K of data memory
#endif
```


© 2015 Microchip Technology Inc.

6.4 堆栈

16 位器件使用本用户指南中所称的 “软件堆栈”作为堆栈。它是大多数计算机采用的典型堆栈安排，它是通过入栈和出栈类型的指令以及一个堆栈指针寄存器进行访问的普通数据存储器。“硬件堆栈”一词用于描述 Microchip 8 位器件采用的堆栈，它仅用于存储函数返回地址。

16 位器件将寄存器 W15 专门用作软件堆栈指针。所有处理器堆栈操作 （包括函数调用、中断和异常）都使用软件堆栈。堆栈朝着更高存储器地址方向向上增长。

dsPIC DSC 器件还支持堆栈溢出检测。如果堆栈指针限制寄存器 SPLIM 已初始化，器件会在所有堆栈操作时测试是否溢出。如果发生溢出，处理器会发出堆栈错误异常。默认情况下，这将导致处理器复位。应用程序也可以通过定义一个名为 `_StackError` 的中断函数来安装一个堆栈错误异常处理程序。详情请参见**第 14 章 “中断”**。

C 运行时启动模块会在启动和初始化序列期间初始化堆栈指针（W15）和堆栈指针限制寄存器。初始值通常由链接器提供，链接器会从未用数据存储器中分配可能的最大堆栈。链接映射输出文件中会报告堆栈的位置。应用程序可以通过使用 `--stack` 链接器命令行选项来确保至少有最小大小的堆栈可用。关于详细信息，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

或者，也可以通过汇编源文件中的用户定义的段分配特定大小的堆栈。在以下示例中，为堆栈保留 0x100 字节的数据存储器：

```
.section *,data,stack
.space 0x100
```

链接器会分配一个相应大小的段，并初始化 `__SP_init` 和 `__SPLIM_init`，使运行时启动代码可以正确地初始化堆栈。请注意，由于它是一个正常的汇编代码段，所以可以使用诸如 `address` 之类的属性来进一步定义堆栈。关于更多信息，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

6.5 配置位访问

Microchip 器件具有几个包含配置位或熔丝的存储单元。这些位指定基本的器件操作，例如振荡器模式、看门狗定时器、编程模式和代码保护。未正确设置这些位可能导致代码失败或器件无法运行。

6.5.1 使用 #pragma config 指定设置配置

配置设置可以使用预处理器伪指令 `#pragma config` 和编译器安装目录的 `docs` 子目录下指定的设置宏进行设置。

伪指令格式选项为：

```
#pragma config setting = state|value
#pragma config register = value
```

其中，`setting` 是配置设置描述符（如 WDT），`state` 是一个描述性的值（如 ON），`value` 是一个数值。寄存器标记可能代表整个配置字寄存器，例如， `CONFIG1L`。

通过 MPLAB X IDE 的 Dashboard 窗口中的 **Compiler Help** （编译器帮助）按钮或通过命令行，可找到按器件列出的所有可用设置的列表：

```
<MPLAB XC16 Installation folder>/vx.xx/docs/config_index.html
```

6.5.2 使用宏指定配置设置

提供了可以用于设置配置位的配置设置宏。例如，要使用宏设置 FOSC 位，可以在 C 源代码开始之前插入以下代码行：

```
_FOSC(CSW_FSCM_ON & EC_PLL16);
```

这将使能外部时钟，将 PLL 设置为 16x，并使能时钟切换和故障保护时钟监视。

类似地，要设置 FBORPOR 位：

```
_FBORPOR(PBOR_ON & BORV_27 & PWRT_ON_64 & MCLR_DIS);
```

这将使能在 2.7V 下发生欠压复位，将上电延时定时器初始化为 64 毫秒，并将 $\overline{\text{MCLR}}$ 引脚配置为用于 I/O。

配置设置宏在每种器件的编译器头文件中定义。关于相关宏的完整列表，请参见器件的头文件。默认情况下，头文件位于：

<MPLAB XC16 Installation folder>/vx.xx/support/device/h

其中， v*x.xx* 是编译器版本， *device* 是 16 位器件系列。

© 2015 Microchip Technology Inc.

6.6 使用 SFR

特殊功能寄存器 （Special Function Register， SFR）是一些控制器件上 MCU 操作或外设模块各个方面的寄存器。这些寄存器是存储器映射的，这意味着它们会位于器件数据存储空间中的一些特定地址处，并可以使用这些地址进行访问。一些寄存器中的各个位控制独立的特性。一些寄存器是只读的；一些是只写的。更多信息，请查看您所使用器件的数据手册。

存储器映射的 SFR 通过一些置于寄存器地址处的特殊 C 变量进行访问。这些变量可以像任何普通 C 变量一样进行访问，所以访问 SFR 不需要任何特殊的语法。

SFR 变量标识符在头文件中预定义，在源代码中包含 <xc.h> 头文件（见**第 6.3 节“器件头文件”**）之后即可访问它们。此外还定义了带有位域的结构，使您可以在源代码中访问寄存器内的位。

在您的项目中必须链接相应器件的链接描述文件，以确保 SFR 变量标识符链接到正确的地址。MPLAB IDE 会链接默认链接描述文件，但如果您要驱动命令行工具链，则必须显式指定链接描述文件。链接描述文件具有 .gld 扩展名（如 p30F6014.gld），并随编译器提供了一些基本链接描述文件。

处理器头文件中的约定是使用器件数据手册中 SFR 的相同名称来命名每个 SFR——例如，CORCON 对应于内核控制寄存器。如果某个寄存器中的各个位值得关注，那么也存在一个为该 SFR 定义的结构，该结构的名称为 SFR 名称后加上“bits”。例如，CORCONbits 对应于内核控制寄存器。在结构中各个位 （或位域）的命名使用数据手册中的名称——例如， PSV 对应于 CORCON 寄存器的 PSV 位。

以下是 CORCON 的完整定义 （可能发生变更）：

```
/* CORCON: CPU Mode control Register */
extern volatile unsigned int CORCON __attribute__((__sfr__));
typedef struct tagCORCONBITS {
  unsigned IF      :1;    /* Integer/Fractional mode             */
  unsigned RND     :1;    /* Rounding mode                       */
  unsigned PSV     :1;    /* Program Space Visibility enable     */
  unsigned IPL3    :1;
  unsigned ACCSAT  :1;    /* Acc saturation mode                 */
  unsigned SATDW   :1;    /* Data space write saturation enable  */
  unsigned SATB    :1;    /* Acc B saturation enable             */
  unsigned SATA    :1;    /* Acc A saturation enable             */
  unsigned DL      :3;    /* DO loop nesting level status        */
  unsigned           : 4;
} CORCONBITS;
extern volatile CORCONBITS CORCONbits __attribute__((__sfr__));
```

> **注：** 符号 CORCON 和 CORCONbits 指代同一寄存器，在链接时解析为相同的地址。

关于使用链接描述文件的更多信息，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

例如，以下是一个实时时钟示例代码。它使用一个 SFR （如 TMR1）以及 SFR 内的位（如 T1CONbits.TCS）。这些 SFR 的说明位于 p30F6014.h 文件中 （该文件通过 <xc.h> 自动包含，所以不需要在源代码中包含它）。该文件将使用特定于器件的链接描述文件 （即 p30F6014.gld）进行链接。

例 6-1： 实时时钟示例代码

```
/*
** Sample Real Time Clock for dsPIC
**
**  Uses Timer1, TCY clock timer mode
**  and interrupt on period match
*/

#include <xc.h>

/* Timer1 period for 1 ms with FOSC = 20 MHz */
#define TMR1_PERIOD 0x1388

struct clockType
  {
   unsigned int timer;     /* countdown timer, milliseconds */
   unsigned int ticks;     /* absolute time, milliseconds   */
   unsigned int seconds;   /* absolute time, seconds        */
  } volatile RTclock;

void reset_clock(void)
  {
   RTclock.timer = 0;      /* clear software registers  */
   RTclock.ticks = 0;
   RTclock.seconds = 0;

   TMR1 = 0;               /* clear timer1 register     */
   PR1 = TMR1_PERIOD;      /* set period1 register      */
   T1CONbits.TCS = 0;      /* set internal clock source */
   IPC0bits.T1IP = 4;      /* set priority level        */
   IFS0bits.T1IF = 0;      /* clear interrupt flag      */
   IEC0bits.T1IE = 1;      /* enable interrupts         */

   SRbits.IPL = 3;         /* enable CPU priority levels 4-7*/
   T1CONbits.TON = 1;      /* start the timer*/
  }
void __attribute__((__interrupt__,__auto_psv__)) _T1Interrupt(void)
  { static int sticks=0;

    if (RTclock.timer > 0)   /* if countdown timer is active */
        RTclock.timer -= 1;  /*  decrement it               */
    RTclock.ticks++;         /* increment ticks counter     */
    if (sticks++ > 1000)
    {                           /* if time to rollover         */
    sticks = 0;              /*  clear seconds ticks     */
    RTclock.seconds++;       /*  and increment seconds   */
    }

    IFS0bits.T1IF = 0;      /* clear interrupt flag      */
    return;
  }
```


© 2015 Microchip Technology Inc.

6.7 位反转寻址和模寻址

所有 dsPIC DSC 器件均支持位反转寻址和模寻址。

位反转寻址用于简化和加快 FFT （快速傅里叶变换）算法中对 X 空间数据数组的写操作。当使能时，预递增或后递增寻址模式将反转由指令使用的低位地址位。

模寻址（又称循环寻址）提供了一种使用 dsPIC 硬件自动支持循环数据缓冲区的方法。当使用时，软件不再需要对数组执行数据地址边界检查。

编译器并不直接支持使用位反转寻址和模寻址；也即，它无法基于假定在访问存储器时已使能这两种寻址模式的C源代码来生成代码。如果在目标器件上设置了这两种寻址方式之一，则编程人员需要负责确保编译器不使用指定为使用模寻址或位反转寻址作为指针的那些寄存器。如果在使能这两种寻址模式之一时可能发生中断，则必须特别小心。

可以在C代码中定义将在存储器中适当对齐供手工编写的汇编语言函数进行模寻址的数组。`aligned` 属性可用于定义定位为用作递增模缓冲区的数组。起始和结束地址的初始化，以及应用模寻址的寄存器，必须手工编写，使之与数组属性说明匹配。`reverse` 属性可用于定义定位为用作递减模缓冲区的数组。关于这些属性的更多信息，请参见**第 13.2.1 节“函数说明符”**。关于位反转寻址或模寻址的更多信息，请参见您所用器件的系列参考手册的 “数据存储器”一章。

注：

© 2015 Microchip Technology Inc.

第 7 章 MPLAB XC16 和 ANSI C 之间的差别

该编译器符合针对编程语言的 ANSI X3.159-1989 标准。它通常被称为 C89 标准。在本手册中将它称为 ANSI C 标准。此外，还支持更新标准 C99 中的一些特性。

- 不符合 ANSI C 标准的方面
- ANSI C 标准的扩展
- 实现定义的行为

7.1 不符合 ANSI C 标准的方面

不存在任何不符合 ANSI C 标准的方面。

7.2 ANSI C 标准的扩展

MPLAB XC16 C 编译器在以下方面提供了对 ANSI C 标准的扩展：关键字和表达式。

7.2.1 关键字差别

新的关键字属于基本 GCC 实现的一部分，所引用章节中的讨论基于标准 GCC 文档，并针对 GCC 的 16 位编译器移植版本的特定语法和语义进行讨论。

- 指定变量的属性——**第 8.12 节 “变量属性”**
- 指定函数的属性——**第 13.2.1 节 “函数说明符”**
- 内联函数——**第 13.6 节 “内联函数”**
- 指定寄存器中的变量——**第 10.11 节 “变量到寄存器的分配”**
- 复数——**第 8.8 节 “复数数据类型”**

7.2.2 表达式差别

表达式差别包括：

- 二进制常量——**第 8.9 节 “字面常量类型和格式”**。

7.3 实现定义的行为

ANSI C 标准的某些特性具有实现定义的行为。这意味着一些 C 代码的确切行为会因编译器不同而不同。

本文档将详细介绍 MPLAB XC16 C 编译器的确切行为，**附录 A “实现定义的行为”**也对其进行了全面总结。

注：

© 2015 Microchip Technology Inc.

第 8 章 支持的数据类型和变量

8.1 简介

MPLAB XC16 C 编译器支持多种数据类型和限定符 （属性）。此处将介绍这些数据类型和变量。关于变量在存储器中的存储位置的信息，请参见**第 10 章 “存储器分配和访问”**。

- 标识符
- 整型数据类型
- 浮点型数据类型
- 定点型数据类型
- 结构和联合
- 指针类型
- 复数数据类型
- 字面常量类型和格式
- 标准类型限定符
- 特定于编译器的类型限定符
- 变量属性

8.2 标识符

C 变量标识符 （以及函数标识符）是由字母和数字构成的序列，其中，下划线字符 “_” 算作字母。标识符不能以数字开头。虽然它们可以使用下划线开头，但这种标识符仅保留供编译器使用，不应由您的程序定义。汇编域标识符则不是如此，它们通常以下划线开头，请参见《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

标识符区分大小写，所以 `main` 不同于 `Main`。

标识符中的所有字符都具有意义，虽然长度超过31个字符的标识符的可移植性会降低。

8.3 整型数据类型

表 8-1 列出了该编译器支持的整型数据类型。所有未指定符号或有符号的整型数据类型均为有符号整型算术类型。所有无符号的整型数据类型均为无符号整型算术类型。

表 8-1： 整型数据类型

类型	位数	最小值	最大值
char, signed char	8	-128	127
unsigned char	8	0	255
short, signed short	16	-32768	32767
unsigned short	16	0	65535
int, signed int	16	-32768	32767
unsigned int	16	0	65535
long, signed long	32	-2^{31}	$2^{31} - 1$
unsigned long	32	0	$2^{32} - 1$
long long*, signed long long*	64	-2^{63}	$2^{63} - 1$
unsigned long long*	64	0	$2^{64} - 1$

* ANSI-89 扩展

不存在用于存储单个位的量的类型。

所有整型值均指定为以小尾数法格式存储，这意味着：

- 最低字节 （LSB）存储在最低地址
- 最低位 （LSb）存储在编号最低的位位置

例如，长整型值 0x12345678 以如下方式存储在地址 0x100 处：

0x100	0x101	0x102	0X103
0x78	0x56	0x34	0x12

再举一个示例，长整型值 0x12345678 存储在寄存器 w4 和 w5 中：

w4	w5
0x5678	0x1234

有符号值以二进制补码整型值的形式存储。

在源代码中包含 <limits.h> 之后，可使用指定整型最小值和最大值的预处理器宏；默认情况下，该文件位于：

<install directory>\include

由于 ANSI 标准未详细规定数据类型的长度，所以使用这些宏可以实现可移植性更高的代码，它们可以检查该实现中的类型可以存放的值的范围限制。

关于整型的实现定义的行为的信息，请参见**第 A.6 节 “整型”**。

8.3.1 双字整型

该编译器支持长度为 long int 两倍的整型数据类型。对于有符号整型，只需书写 long long int，对于无符号整型为 unsigned long long int。要指定 long long int 类型的整型常量，请在整数后添加后缀 LL。要指定 unsigned long long int 类型的整型常量，请在整数后添加后缀 ULL。

您可以像任何其他整型类型一样在算术运算中使用这些类型。

© 2015 Microchip Technology Inc.

8.3.2 char 类型

该编译器支持 char 数据类型，它默认为 signed char。可以通过一个选项来使用 unsigned char 作为默认设置，请参见**第 5.7.3 节 “用于控制 C 方言的选项”**。

一种常见的误解是， C char 类型纯粹用于 ASCII 字符操作。这是不正确的；事实上，C 语言甚至不保证默认字符表示形式为 ASCII（但是，该实现确实使用 ASCII 作为字符表示形式）。 char 类型只是多位整型中长度最小的，并且在所有方面的行为都类似于整型。使用名称“char”是出于历史原因，这并不意味着 char 只能用于表示字符。在 C 表达式中，可以自由地将 char 类型的值与其他类型的值混合使用。对于 MPLAB XC16 C 编译器，char 类型通常用于多种用途：用作 8 位整数、用于存储 ASCII 字符，以及用于访问 I/O 存储单元。

8.4 浮点型数据类型

编译器使用 IEEE-754 格式。表 8-2 列出了支持的浮点型数据类型。所有浮点型数据类型都是实数算术类型。

表 8-2： 浮点型数据类型

类型	位数	E 最小值	E 最大值	N 最小值	N 最大值
float	32	-126	127	2^{-126}	2^{128}
double*	32	-126	127	2^{-126}	2^{128}
long double	64	-1022	1023	2^{-1022}	2^{1024}

E = 指数
N = 规范化的 （近似值）
* 如果使用了 -fno-short-double，则 double 等效于 long double。

所有浮点值均指定为以小尾数法格式存储，这意味着：

- 最低字节 （LSB）存储在最低地址
- 最低位 （LSb）存储在编号最低的位位置

例如， double 类型值 1.2345678 以如下方式存储在地址 0x100 处：

0x100	0x101	0x102	0X103
0x51	0x06	0x9E	0x3F

再举一个示例， double 类型值 1.2345678 存储在寄存器 w4 和 w5 中：

w4	w5
0x0651	0x3F9E

浮点型总是为有符号的，在指定浮点型时， unsigned 关键字是非法的。

在源代码中包含 <float.h> 之后，将可使用指定有效范围的预处理器宏。

关于浮点型数的实现定义的行为的信息，请参见**第 A.7 节 “浮点型”**章节。

© 2015 Microchip Technology Inc.

8.5 定点型数据类型

表 8-3 列出了在指定 `-menable-fixed` 命令行选项时编译器支持的定点型数据类型。关于对定点C语言方言的编译器支持的更多详细信息，请参见**第9章“定点算术支持”**。如果不存在 signed 或 unsigned 类型说明符，则假定类型为有符号。

表 8-3： 定点整型数据类型

类型	位数	最小值	最大值
`_Fract`	16	-1.0	1.0 - 2^-15
`short _Fract`	16	-1.0	1.0 - 2^-15
`signed _Fract`	16	-1.0	1.0 - 2^-15
`signed short _Fract`	16	-1.0	1.0 - 2^-15
`unsigned _Fract`	16	0.0	1.0 - 2^-15
`unsigned short _Fract`	16	0.0	1.0 - 2^-15
`long _Fract`	32	-1.0	1.0 - 2^-31
`signed long _Fract`	32	-1.0	1.0 - 2^-31
`unsigned long _Fract`	32	0.0	1.0 - 2^-31
`_Accum`	40	-256.0	256.0 - 2^-31
`short _Accum`	40	-256.0	256.0 - 2^-31
`long _Accum`	40	-256.0	256.0 - 2^-31
`signed _Accum`	40	-256.0	256.0 - 2^-31
`signed short _Accum`	40	-256.0	256.0 - 2^-31
`signed long _Accum`	40	-256.0	256.0 - 2^-31
`unsigned _Accum`	40	0.0	256.0 - 2^-31
`unsigned short _Accum`	40	0.0	256.0 - 2^-31
`unsigned long _Accum`	40	0.0	256.0 - 2^-31

与整型和浮点型数据类型一样，所有定点值都以小尾数法格式表示，这意味着：

- 最低字节 （LSB）存储在最低地址
- 最低位 （LSb）存储在编号最低的位位置

8.6 结构和联合

MPLAB XC16 C 编译器支持 `struct` 和 `union` 类型。结构和联合的唯一区别在于对每个成员应用的存储器偏移量。

这两种类型至少为 1 个字节宽。结构中完全支持位域。

结构和联合可作为函数参数和函数返回值自由地传递。完全支持指向结构和联合的指针。

第 A.10 节 “结构、联合、枚举和位域”介绍了结构、联合和位域的实现定义的行为。

8.6.1 结构和联合限定符

MPLAB XC16 C 编译器支持对于结构使用类型限定符。对结构应用限定符时，其所有成员都会继承该限定。在以下示例中，结构使用 `const` 进行限定。

```
const struct foo {
  int number;
  int *ptr;
} record = { 0x55, &i };
```

在此例中，可以将整个结构放入程序空间，并且每个成员都是只读的。请记住，如果结构使用 `const` 限定，则通常需要对所有成员进行初始化，因为它们不能在运行时初始化。

如果对结构的成员单独使用 `const` 进行限定，但不对结构进行这种限定，则结构将被定位到 RAM，但每个成员仍然是只读的。将以下结构与上面的结构进行比较。

```
struct {
  const int number;
  int * const ptr;
} record = { 0x55, &i };
```


© 2015 Microchip Technology Inc.

8.6.2 结构中的位域

MPLAB XC16 C 编译器完全支持结构中的位域。

默认情况下，位域为 signed int 类型。通过使用一个命令行选项，可以将它们设为 unsigned int 位域，请参见**第 5.7.3 节 “用于控制 C 方言的选项”**。

所定义的第一个位将为用于存储它的字的 LSb。

该编译器支持长度为任意位数的位域，最高为基础类型的长度。可以将任何整型类型指定为位域。分配通常不会跨越基础类型的自然位边界。例如：

```
struct foo {
  long long i:40;
  int j:16;
  char k:8;
} x;

struct bar {
  long long I:40;
  char J:8;
  int K:16;
} y;
```

使用该编译器，struct foo 的长度将为 10 字节。i 会被分配在位偏移 0 处（至 39）。j 之前存在 8 位的填充数据，它分配在位偏移 48 处。如果 j 被分配在下一个可用位偏移 40 处，则它就会跨越 16 位整型的存储边界。 k 会被分配在 j 之后，位于位偏移 64 处。该结构将在末尾处包含 8 位的填充数据，以维持在使用数组情况下所需的对齐。对齐量为 2 个字节，因为结构中的最大对齐量为 2 个字节。

使用该编译器，struct bar 的长度将为 8 字节。I 会被分配在位偏移 0 处（至 39）。不需要在 J 前进行填充，因为它不会跨越 char 类型的存储边界。J 被分配在位偏移 40 处。 K 可以从位偏移 48 处开始分配，完成结构的分配，不会浪费任何空间。

可以通过声明未命名位域来填充控制寄存器中有效位之间的未用空间。例如：

```
struct foo {
  unsigned lo  : 1;
  unsigned           : 6;
  unsigned hi  : 1;
} x;
```

带有位域的结构可以通过提供由每个位域的初始值构成的*逗号*分隔列表进行初始化。例如：

```
struct foo {
  unsigned lo  : 1;
  unsigned        mid : 6;
  unsigned hi  : 1;
} x = {1, 8, 0};
```

带有未命名位域的结构可以进行初始化。不应为未命名成员提供初始值，例如：

```
struct foo {
    unsigned        lo  : 1;
    unsigned            : 6;
    unsigned        hi  : 1;
} x = {1, 0};
```

将正确初始化成员 lo 和 hi。

8.7 指针类型

MPLAB XC16 C 编译器支持两种基本指针类型：数据指针和函数指针。数据指针存放可以由程序间接读取，并可能可间接写入的变量的地址。函数指针存放可通过指针间接调用的可执行函数的地址。

8.7.1 指针与类型限定符

先回顾一下 ANSI C 标准对于指针类型定义的约定会很有帮助。

指针可以像任何其他 C 对象一样进行限定，但这样做时必须小心，因为会有两个量与指针相关联。第一个量是实际指针本身，它像任何普通 C 变量一样对待，并且会为它保留存储空间。第二个量是指针引用的目标，或指针所指向的目标。指针定义的一般形式如下：

*target_type_&_qualifiers * pointer's_qualifiers pointer's_name*;

* 右侧（即，指针名称旁）的所有限定符都与指针变量本身相关。* 左侧的类型和所有限定符都与指针的目标相关。这是有道理的，因为 * 操作符用于指针解引用，从而使您可以通过指针变量获取它的当前目标。

以下给出了 3 个使用 volatile 限定符的指针定义示例。定义中的各个字段使用空白进行了强调：

```
volatile int *           vip ;
int          * volatile  ivp ;
volatile int * volatile  vivp ;
```

第一个示例是一个名为 vip 的指针。它包含使用 volatile 限定的 int 对象的地址。指针本身（存放地址的变量）*不是* volatile 类型；但是，进行指针解引用时访问的对象被视为 volatile 类型。即，可通过指针访问的目标对象可在外部进行修改。

第二个示例是一个名为 ivp 的指针，它也包含一个 int 对象的地址。在该示例中，指针本身是 volatile 类型，即指针包含的地址可在外部进行修改；但是，进行指针解引用时可以访问的对象不是 volatile 类型。

最后一个示例是一个名为 vivp 的指针，它本身使用 volatile 限定，同时它包含 volatile 对象的地址。

请记住，一个指针可以被赋予许多对象的地址；例如，对于作为某个函数参数的指针，每次调用该函数时都会为其赋予一个新的对象地址。指针的定义必须对于赋予的每个目标地址都是有效的。

> **注：** 描述指针时必须小心。“const 指针”是指向 const 对象的指针，还是本身即为 const 类型的指针？您可以使用 “指向 const 的指针”和 “const 指针”这种描述来帮助阐明定义，但此类术语可能并不是每个人都理解。

© 2015 Microchip Technology Inc.

8.7.2 数据指针

所有标准数据指针均为 16 位宽。这对于访问整个数据存储空间足够了。

这些指针还能够访问使用 const 限定的对象，即使这些对象在程序存储空间中，使用 const 限定的对象出现在使用 PSV 窗口从数据空间中访问独特程序存储范围。在这种情况下，不应使用 -mconst-in-data 选项（见**第 5.7.1 节“特定于 16 位器件的选项”**）。

访问受管PSV空间的指针为32位宽。额外的指针宽度使这些指针可以访问任意PSV页。

此外，还有一组特殊用途的 32 位数据指针可用。更多信息，请参见**第 10 章 “存储器分配和访问”**。

8.7.3 函数指针

MPLAB XC16 C 编译器完全支持指向函数的指针，它们可以用于间接地调用函数。函数指针总是为 16 位宽。

由于函数指针只有 16 位宽，所以这些指针不能指向超出闪存前 64K 之外的范围。如果获取了分配到超出闪存前 64K 范围的函数的地址，链接器会安排生成一个 handle 段。handle 段总是被分配在前 64K 范围内。每个句柄提供间接寻址，使 16 位指针可以访问完整的闪存范围。可以使用 --no-handles 链接器选项禁止该操作。

8.7.4 特殊指针目标

指针和整数是不可互换的。将整数值赋予指针将会产生对于该操作的警告。例如：

```
const char * cp = 0x123;  // the compiler will flag this as bad code
```

整数 0x123 不具有与目标的类型、长度或存储器位置有关的任何信息。请总是避免将整数（无论它是常量还是变量）赋值给一个指针。赋值给指针的地址应通过 C 提供的地址运算符 “&”获得。

在需要指针引用一个看似任意的地址或地址范围的情况下，可以考虑在所需位置定义一个对象或标号。如果对象是在汇编代码中定义的，请使用 C 声明（使用 extern 关键字）创建一个与外部对象链接并可以获取其地址的 C 对象。

进行指针比较（相减）时需要小心。例如：

```
if(cp1 == cp2)
  ; take appropriate action
```

ANSI C 标准仅允许在两个指针目标为相同对象时进行指针比较。地址可能会延伸至超出数组末尾的元素。

将指针与整型常量进行比较的风险更高，例如：

```
if(cp1 == 0x246)
  ; take appropriate action
```

NULL 指针是一种可以将常量值安全地赋值给指针的情况。 NULL 指针在数值上等于 0（零），但由于它们并不保证指向任何有效对象，所以不应进行解引用，这是 ANSI C 标准规定的一种特殊情况。此外，也允许与宏 NULL 进行比较。

8.8 复数数据类型

该编译器支持复数数据类型。您可以使用关键字 __complex__ 声明复数整型类型和复数浮点类型。

例如， __complex__ float x; 将 x 声明为实部和虚部均为 float 类型的变量。__complex__ short int y; 将 y 声明为实部和虚部均为 short int 类型。

要书写具有复数数据类型的常量，请使用后缀“i”或“j”（任意一个；它们是等效的）。例如，2.5fi 具有 __complex__ float 类型，3i 具有 __complex__ int 类型。此类常量是一个纯虚数值，但您可以通过将它与实数常量相加来构成任意复数值。

要提取复数值表达式 *exp* 的实部，可以书写 __real__ *exp*。类似地，使用 __imag__ 可提取虚部。例如：

```
__complex__ float z;
float r;
float i;

r = __real__ z;
i = __imag__ z;
```

当用于具有复数类型的值时，运算符 ~ 执行复共轭。

编译器可以用非连续方式分配复数自动变量；甚至可能实部处于寄存器中，而虚部处于堆栈中（或反之）。调试信息格式没有任何方式来表示类似于这样的非连续分配，因此编译器以两个非复数类型的独立变量的形式来描述非连续复数变量。如果变量的实际名称为 foo，则两个虚拟变量命名为 foo$real 和 foo$imag。

© 2015 Microchip Technology Inc.

8.9 字面常量类型和格式

字面常量用于表示源代码中的数值，例如 123 是一个常量。与所有值一样，字面常量必须具有 C 类型。除了字面常量的类型之外，还可以使用几种格式之一来指定实际值。整型字面常量的格式会指定其基数。MPLAB XC16 支持 ANSI 标准基数说明符，以及用于在 C 代码中指定二进制常量的说明符。

表 8-4 列出了用于指定基数的格式。与用于指定十六进制数字的字母相同，用于指定二进制或十六进制基数的字母也不区分大小写。

表 8-4： 基数格式

基数	格式	示例
二进制	`0b` *数字*或 `0B` *数字*	0b10011010
八进制	`0` *数字*	0763
十进制	*数字*	129
十六进制	`0x` *数字*或 `0X` *数字*	0x2F

整型字面常量将具有 `int`、`long int` 或 `long long int` 类型，以便类型可以在不发生溢出的情况下存放其值。对于以八进制或十六进制指定的字面常量，如果它们对应的有符号常量太小而无法存放其值，则还可以分配 `unsigned int`、`unsigned long int` 或 `unsigned long long int` 类型。

可以通过在数字后添加一个后缀来更改字面常量的默认类型；例如 23U，其中的 `U` 为后缀。表 8-5 列出了在分配类型时考虑的后缀和类型的可能组合。例如，如果指定了后缀 `l`，并且值为一个十进制字面常量，并且如果 `long int` 类型可以存放该字面常量，则编译器会分配该类型；否则，它会分配 `long long int` 类型。如果该字面常量被指定为一个八进制或十六进制常量，则也会考虑无符号类型。

表 8-5： 后缀和分配的类型

后缀	十进制	八进制或十六进制

`u` 或 `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` 或 `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` 或 `U`，以及 `l` 或 `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` 或 `LL`	`long long int`	`long long int` `unsigned long long int`
`u` 或 `U`，以及 `ll` 或 `LL`	`unsigned long long int`	`unsigned long long int`

以下给出了一个可能由于分配给字面常量的默认类型不适合而发生失败的代码示例：

```
unsigned long int result;
unsigned char shifter;

void main (void)
{
   shifter = 20;
   result = 1 << shifter;
   // code that uses result
}
```

字面常量 1 将被分配 int 类型，因而移位运算的结果将为 int 类型，无论对字面常量进行多少次移位，long 变量 result 的高 16 位都永远不会被设置。在此例中，值 1 被左移 20 位将产生结果 0，而不是 0x100000。

以下代码使用后缀来更改字面常量的类型，从而确保移位结果具有 unsigned long 类型。

```
   result = 1UL << shifter;
```

浮点型字面常量具有 double 类型，除非加上后缀 f 或 F，在后面这种情况下它是一个 float 常量。后缀 l 或 L 指定 long double 类型。在 MPLAB XC16 中，double 类型等同于 32 位 float 类型。可以使用命令行选项 -fno-short-double 将 double 指定为 64 位 long double 类型。

定点型字面常量看起来类似于浮点型数，使用 [u][h,l]<r,k> 的组合作为后缀。后缀 u 表示无符号。后缀 h 和 l 分别表示短和长。后缀 r 表示 _Fract 类型，k 则指定 _Accum 类型。例如，-1.0r 是有符号的 _Fract 类型，0.5uhk 是无符号的短 _Accum 类型。

字符字面常量使用单引号字符 ' 包围，例如 'a'。字符字面常量具有 int 类型，虽然在编译之后可能会将其优化为 char 类型。

该实现不支持多字节字符字面常量。

字符串常量或字符串字面值使用双引号字符 " " 包围，例如 "hello world"。字符串字面常量的类型为 const char *，构成字符串的字符可以存储在程序存储器中。

为了符合 ANSI C 标准，编译器不支持在字符或字符数组中使用扩展字符集。实际上，它们需要使用反斜杠字符进行转义，如以下示例所示：

```
const char name[] = "Bj\xf8k";
printf("%s's Resum\xe9", name); \\ prints "Bjørk's Resumé"
```

定义并使用字符串初始化非 const 数组 （即，不是指针定义），例如：

```
char ca[]= "two";         // "two"  different to the above
```

是一种特殊情形，将产生一个位于数据空间中的数组，在启动时使用字符串 "two" 初始化，而在其他上下文中使用的字符串字面常量则代表一个直接从其存储位置访问的未命名数组。

对于具有相同字符序列的字符串，编译器将使用相同的存储单元和标号（用于初始化驻留在数据空间中的数组的字符串除外，如上一个示例中的最后一条语句所示）。

C预处理器会连接两个相邻的字符串字面常量（即，两个字符串 *仅*使用空格进行分隔）。因而：

```
const char * cp = "hello"  "world";
```

将为指针赋予字符串 "hello world" 的地址。

© 2015 Microchip Technology Inc.

8.10 标准类型限定符

类型限定符可以提供关于如何使用对象的附加信息。MPLAB XC16 编译器支持 ANSI C 限定符和一些额外的特殊限定符，这些特殊限定符对于嵌入式应用非常有用，并充分利用了 PIC MCU 和 dsPIC DSC 架构。

8.10.1 const 类型限定符

该编译器支持使用 ANSI 类型限定符 `const` 和 `volatile`。

`const` 类型限定符用于指示编译器，某个对象是只读的，不会被修改。如果尝试修改声明为 `const` 类型的对象，编译器将发出警告或错误。

默认情况下，使用 `const` 声明的用户定义对象会被放入程序空间，可以通过程序可视性空间访问，请参见**第 10.4 节 “程序空间中的变量”**。通常， `const` 对象必须在声明时进行初始化，因为它不能在运行时的任何时间点赋值。例如：

```
const int  version = 3;
```

会将 `version` 定义为将放置在程序存储器中的 `int` 变量，并且将总是包含值 3，程序永远无法修改它。

存储模型 `-mconst-in-data` 会将使用 `const` 限定的对象分配到数据空间中，该空间是可写的。

8.10.2 volatile 类型限定符

`volatile` 类型限定符用于指示编译器，无法保证某个对象在两次连续访问之间会保留其值。这可以防止优化器删除对声明为 `volatile` 对象看起来多余的引用，因为这可能会改变程序执行这种引用的行为。

所有可以由硬件修改或驱动硬件的 SFR 都限定为 `volatile` 类型，可能由中断程序修改的所有变量也应使用该限定符。例如：

```
extern volatile unsigned int INTCON1 __attribute__((__sfr__));
```

编译器生成的用于访问 `volatile` 对象的代码可能会与访问普通变量的代码不同，并且用于 `volatile` 对象的代码通常会较长且较慢，因此仅在需要时才使用该限定符。但是，在需要时未使用该限定符可能会导致代码失败。

`volatile` 关键字的另一个用途是防止未在 C 源代码中使用的变量被删除。如果某个非 `volatile` 变量从不使用，或者其使用方式对程序的功能没有任何影响，那么它可能会在编译器生成代码之前被删除。

如果某条 C 语句仅包含一个 `volatile` 变量的名称，则会生成读取变量的存储单元并丢弃结果的代码。例如，以下整条语句：

```
PORTB;
```

将生成读取 `PORTB` 但不对该值执行任何操作的汇编代码。这对于一些需要通过读取来复位中断标志状态的外设寄存器，比较有用。通常，这种语句将不进行编码，因为它没有任何作用。

一些变量虽然并未在源代码中进行限定，但它们会被视为 `volatile` 类型。如果项目中使用汇编代码，请参见**第 16 章 “混合使用 C 代码和汇编代码”**。

8.11 特定于编译器的类型限定符

MPLAB XC16 C 编译器支持一些特殊的类型限定符，用户可以使用它们来控制变量的访问方式。

8.11.1 __psv__ 类型限定符

__psv__ 限定符可以应用于被分配到程序存储空间的变量或指针目标。它指示如何访问 / 读取变量或指针目标。将变量分配到程序存储空间是一个独立的过程，使用 space 属性来指定，所以该限定符通常在定义变量时与该属性配合使用。例如：

```
__psv__ unsigned int __attribute__((space(psv))) myPSVvar = 0x1234;
__psv__ char * myPSVpointer;
```

该示例中的指针未使用 space 属性，因为它被定位到数据存储器中，但该限定符指示该如何访问指针目标。关于 space 属性以及如何将变量分配到闪存的更多信息，请参见**第 8.12 节 “变量属性”**。关于存储器布局和器件如何访问程序存储器的基本信息，请参见**第 10.2 节 “地址空间”**。

读取限定为 __psv__ 的变量时，编译器将管理在数据存储器窗口中可见的程序存储器页的选择。这意味着您不需要显式地在源代码中调整 PSVPAG SFR，但所生成代码的效率可能会比手工管理该窗口时生成的代码的效率稍低。

编译器将假定用 __psv__ 限定的任何对象或指针目标均可完全装入单个 PSV 页。使用 psv 或 auto_psv 空间属性分配存储器的对象就是如此。如果不是如此，则应使用 __prog__ 限定符 （见**第 8.11.2 节 “__prog__ 类型限定符”**）和适当的空间属性。

8.11.2 __prog__ 类型限定符

__prog__ 限定符类似于 __psv__ 限定符（见**第 8.11.1 节“__psv__ 类型限定符”**），但它向编译器指示限定的变量或指针目标可能跨越 PSV 页。因此，编译器将生成可以正确读取这些限定对象的代码，无论它们被分配到哪个页中。该代码的长度可能大于访问使用 __psv__ 限定的变量或指针目标的代码。例如：

```
__prog__ unsigned int __attribute__((space(prog))) myPROGvar = 0x1234;
__prog__ char * myPROGpointer;
```

该示例中的指针未使用 space 属性，因为它被定位到数据存储器中，但该限定符指示该如何访问指针目标。关于 space 属性以及如何将变量分配到闪存的更多信息，请参见**第 8.12 节 “变量属性”**。此外，关于存储器布局和器件如何访问程序存储器的基本信息，请参见**第 10.2 节 “地址空间”**。

© 2015 Microchip Technology Inc.

8.11.3 __eds__ 类型限定符

__eds__ 限定符指示限定对象已被定位到可访问的 EDS 存储空间，编译器应管理用于访问该存储器的相应寄存器。

当用于指针时，它意味着编译器应对于指针目标所在的存储空间作出一些假定，以及假定目标可能处于几个存储空间之一中，包括：space(data)（及其子集）、eds、space(eedata)、space(prog)、space(psv) 和 space(auto_psv)，以及某些器件上的 space(pmp)。并非所有器件都支持所有存储空间。例如

```
__eds__ unsigned int __attribute__((eds)) myEDSvar;
__eds__ char * myEDSpointer;
```

编译器会自动为标量变量声明 page 属性；这使编译器可以在访问较大数据类型时生成更高效的代码。请记住，标量变量不包含结构或数组。要强制结构或数组分页，请手动使用 page 属性，编译器会阻止对象跨越页边界。

对于对 __eds__ 限定变量的读访问，将会自动（适当地）处理 PSVPAG 或 DSRPAG 寄存器。对于支持扩展数据空间存储器的器件，编译器还会处理 DSWPAG 寄存器。

> **注：** 一些器件使用 DSRPAG 来表示对闪存或扩展数据空间（EDS）的扩展读访问。

关于该限定符的更多信息，请参见**第 10.7 节 “扩展数据空间访问”**。

8.11.4 __pack_upper_byte 类型限定符

该限定符允许使用闪存的最高字节来进行数据存储。对于 16 位器件，在闪存中使用了 24 位的字。该架构支持将闪存区域映射到数据空间，但这种映射只有 16 位宽，以适应数据空间大小，除非使用了 __pack_upper_byte 限定符。

关于该限定符的更多信息，请参见**第 10.10 节 “压缩存储在闪存中的数据”**。

8.11.5 __pmp__ 类型限定符

一些器件含有并行主端口（Parallel Master Port，PMP）外设，使得可以将各种存储器和非存储器器件直接与这些器件连接，该限定符可以用于这些器件。访问使用 __pmp__ 限定的变量或指针目标时，编译器将生成通过器件上的 PMP 外设访问这些对象的适当序列。例如：

```
____pmp__ int auxDevice
          __attribute__((space(pmp(external_PMP_memory))));
__pmp__ char * myPMPpointer;
```

除了该限定符之外，int 变量还使用了需要预定义的存储空间。该示例中的指针未使用 space 属性，因为它被定位到数据存储器中，但该限定符指示该如何访问指针目标。关于 space 属性的更多信息，请参见**第 8.12 节 “变量属性”**。关于存储器布局和器件如何访问程序存储器的基本信息，请参见**第 10.2 节 “地址空间”**。

关于该限定符的更多信息，请参见**第 10.5 节 “并行主端口访问”**。

8.11.6 __external__ 类型限定符

该限定符用于指示编译器应访问已被定位在外部存储器中的变量或指针目标。这些存储器包括任何已连接到器件，但不使用或无法使用并行主端口（PMP）外设（见**第 8.11.5 节 “__pmp__ 类型限定符”**）访问的所有存储器。对外部存储器中对象的访问类似于 PMP 访问，但执行这种访问的程序是完全可配置的，事实上需要先定义才能进行任何访问。关于如何配置存储空间和定义访问程序的完整信息，请参见**第 10.6 节 “外部存储器访问”**。

该限定符的使用方式如以下示例所示。

```
____external__ int external_array[256]
                __attribute__((space(external(external_memory))));
__external__ char * myExternalPointer;
```

除了该限定符之外，数组还使用了需要预定义的存储空间。该示例中的指针未使用 space 属性，因为它被定位到数据存储器中，但该限定符指示该如何访问指针目标。关于 space 属性的更多信息，请参见**第 8.12 节 “变量属性”**。关于存储器布局和器件如何访问程序存储器的基本信息，请参见。

关于该限定符的更多信息，请参见**第 10.6 节 “外部存储器访问”**。

© 2015 Microchip Technology Inc.

8.12 变量属性

MPLAB XC16 C 编译器使用属性来指示变量、结构成员和类型的存储器分配、类型和其他配置。对于函数提供了其他属性，**第 13.2.2 节 “函数属性”**对它们进行了介绍。**第 8.11 节 “特定于编译器的类型限定符”**中列出的限定符独立于属性使用。它们仅指示如何访问对象，但必须在需要确保正确的代码操作时使用。

编译器关键字 `__attribute__` 可用于指定对象的属性。该关键字后跟包含在双括号内的属性说明。变量当前支持以下属性：

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

您还可以用关键字前后加上 `__`（双下划线）的方式指定属性（例如，`__aligned__` 而不是 `aligned`）。这使您可以在头文件中使用它们，而无需担心可能存在同名宏的问题。

要指定多个属性，请在双括号内通过逗号分隔它们，例如：

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

注：	在整个项目中一致地使用变量属性非常重要。例如，如果某个变量在文件 A 中定义时带有 `far` 属性，而在文件 B 中声明为 `extern` 时未带有 `far`，则可能会产生链接错误。

address (*addr*)

address 属性指定变量的绝对地址。该属性可以与 section 属性配合使用。这种组合可以用于在某个特定地址处开始一组变量：

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

带有 address 属性的变量不能放入 auto_psv 空间 （见 space() 属性或 -mconst-in-code 选项）；试图这样做会导致警告，编译器会将变量放入 PSV 空间。如果要将变量放入 PSV 段，地址应为程序存储器地址。

aligned (alignment)

该属性指定变量的最小对齐量，以字节为单位。对齐量必须为 2 的幂。例如，以下声明：

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

会使编译器将全局变量 x 分配在 16 字节边界上。在 dsPIC DSC 器件上，它可以与 asm 表达式配合使用，以访问需要对齐操作数的 DSP 指令和寻址模式。

如以上示例所示，您可以显式指定希望编译器对于给定变量使用的对齐量（以字节为单位）。或者，您也可以省略对齐因数，仅要求编译器将变量对齐到 dsPIC DSC 器件的最大有用对齐量处。例如，可以书写：

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

每当在 aligned 属性说明中省略对齐因数时，编译器会自动将所声明变量的对齐量设置为目标机器上任意数据类型的最大对齐量——对于 dsPIC DSC 器件为两个字节（一个字）。

aligned 属性只能增大对齐量；您可以通过指定 packed （见下文）来减小它。aligned 属性与 reverse 属性冲突。同时指定两者是一种错误情况。

aligned 属性可以与 section 属性配合使用。这会使指定的段中发生对齐。默认情况下，当未指定段时，编译器会为变量生成一个唯一的段。这使链接器可以有最好的机会满足对齐限制，而无需使用内部填充，如果同一个对齐段内出现其他定义，则可能会发生内部填充。

boot

该属性可以用于在引导段 （BS） RAM 中定义受保护的变量：

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

在启动时，将不会初始化在 BS RAM 中定义的变量。因此，必须使用行内代码来初始化 BS RAM 中的所有变量。如果对 boot 变量指定初始值，则会报告诊断消息。

初始化示例如下：

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


© 2015 Microchip Technology Inc.

deprecated

deprecated 属性会导致编译器对附加了该属性的声明进行特殊识别。当使用 deprecated 函数或变量时，编译器会发出警告。

deprecated 定义仍然会被定义，因此可以存在于任何目标文件中。例如，编译以下文件：

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

将产生警告：

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

i 仍然以正常方式在生成的目标文件中定义。

eds

在属性上下文中，eds（扩展数据空间）属性向编译器指示该变量将分配在数据存储器内的任意位置。为使编译器正确地生成正确的访问序列，带有该属性的变量很可能也具有 __eds__ 类型限定符 （见**第 10.7 节 “扩展数据空间访问”**）。请注意， __eds__ 限定符和 eds 属性密切相关，但不相同。在一些器件上，将变量分配到某些存储空间（如 space (ymemory) 或 space (dma)）时，可能需要指定 eds，因为该存储空间可能仅存在于扩展数据空间中。

fillupper

该属性可用于指定一个存储到 space(prog) 段的变量的最高字节。

例如：

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

将使用 0x23 而不是 0x00 填充数组 foo 的最高字节。 foo[0] 仍然会被初始化为 0xDEAD。

命令行选项 -mfillupper=0x23 将执行相同的功能。

far

far 属性向编译器指示该变量不一定会分配在 near（前 8 KB）数据空间中（即，该变量可能被定位在数据存储器中 0x0000 和 0x7FFF 之间的任意位置）。

mode (*mode*)

该属性将所声明变量的数据类型指定为对应于模式 *mode* 的类型。这实际上让您可以根据变量宽度请求整型或浮点型类型。 *mode* 的有效值如下：

模式	宽度	编译器类型
QI	8 位	char
HI	16 位	int
SI	32 位	long
DI	64 位	long long
SF	32 位	float
DF	64 位	long double

该属性对于编写可在所有支持的编译器之间移植的代码很有用。例如，以下函数将两个 32 位有符号整型相加，并返回一个 32 位有符号整型结果：

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

您还可以指定模式 byte 或 __byte__，指示对应于 1 字节整型的模式； word 或 __word__ 对应于 1 字整型的模式； pointer 或 __pointer__ 则对应于用于表示指针的模式。

near

near 属性向编译器指示该变量分配在 near 数据空间 （数据存储器的前 8 KB）。此类变量的访问效率有时会比不是分配到 （或不知道要分配到） near 数据空间的变量高。

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

noload

noload 属性指示应为变量分配空间，但不应装入初始值。如果应用程序设计为在运行时将某个变量装入存储器 （例如从串行 EEPROM），该属性会很有用。

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

packed

packed 属性指定结构成员应具有可能的最小对齐量，除非使用 aligned 属性指定了较大的值。

以下是一个对成员 x 进行压缩的结构，从而使它紧随 a，不进行填充来实现对齐：

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

> **注：** 器件架构要求字在偶字节边界处对齐，所以在使用 packed 属性时必须小心，以避免运行时寻址错误。

page

page 属性用于实现不跨越页边界的变量分配。页大小取决于通过 space 属性选择的存储器的类型。驻留在 RAM 中的对象会被约束在 32K 页中，而驻留在闪存中的对象会被约束在 64K 页中 （不包括最高字节）。

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

默认情况下， space(auto_psv) 或 space(psv) 属性将使用单个存储器页。

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

在使用 eds 时，请参见**第 10.7 节 “扩展数据空间访问”**了解更多信息。

© 2015 Microchip Technology Inc.

persistent

persistent 属性指定不应在启动时初始化或清零变量。带 persistent 属性的变量可用于存储在器件复位后仍然有效的状态信息。

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

持久性数据不会被 C 运行时正常初始化。但是，在冷重启之后，持久性数据可能不具有任何有意义的值。以下代码示例说明了如何安全地初始化此类数据：

```
#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*)

reverse 属性为变量的结束地址加 1 指定最小对齐量。对齐量以字节为单位指定，必须为 2 的幂。反向对齐的变量可以用于在 dsPIC DSC 汇编语言中递减模缓冲区。如果应用程序在 C 程序中定义了将从汇编语言中访问的变量，该属性会很有用。

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

reverse 属性与 aligned 属性和 section 冲突。试图为反向对齐的变量指定一个段时，将会被忽略，并产生警告。对同一变量同时指定 reverse 和 aligned 属性是一种错误情况。带有 reverse 属性的变量不能放入 auto_psv 空间 （见 space() 属性或 -mconst-in-code 选项）；试图这样做会导致警告，编译器会将变量放入 PSV 空间。

section (*"section-name"*)

默认情况下，编译器会将它生成的对象放在诸如 .data 和 .bss 之类的段中。section 属性使您可以通过指定变量 （或函数）存放到特定段中来改写该行为。

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

secure

该属性可以用于在安全段 （SS） RAM 中定义受保护的变量：

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

在启动时，将不会初始化在 SS RAM 中定义的变量。因此，必须使用行内代码初始化 SS RAM 中的所有变量。如果对 secure 变量指定初始值，则会报告诊断消息。

可以使用行内代码将字符串字面值赋给 secure 变量，但需要由编译器进行额外的处理。例如：

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

在这种情况下，必须为字符串字面值在包含它的 secure 函数可访问的存储空间中分配存储位置。编译器会将字符串分配到为安全段指定的 psv 常量段中。

sfr (*address*)

sfr 属性向编译器指示该变量是 SFR，并且还可以使用 *address* 参数指定变量的运行时地址。

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

为了不产生错误，需要使用 extern 说明符。

> **注：** 按照约定， sfr 属性仅在处理器头文件中使用。要定义处于特定地址处的通用用户变量，请使用 address 属性配合 near 或 far 来指定正确的寻址模式。

space (*space*)

通常情况下，编译器会将变量分配在通用数据空间中。 space 属性可以用于指示编译器将变量分配到特定存储器空间中。**第 10.2 节 “地址空间”**对存储空间进行了进一步的介绍。 space 属性接受以下参数：

data

将变量分配到通用数据空间中。通用数据空间中的变量可以使用普通 C 语句访问。这是默认分配方式。

dataflash

将变量分配在 dataflash 中。

DD **xmemory**——仅对于 dsPIC30F 和 dsPIC33EP/F DSC

将变量分配到 X 数据空间中。 X 数据空间中的变量可以使用普通 C 语句访问。 xmemory 空间分配的示例：

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

DD **ymemory**——仅对于 dsPIC30F 和 dsPIC33EP/F DSC

将变量分配到 Y 数据空间中。 Y 数据空间中的变量可以使用普通 C 语句访问。 ymemory 空间分配的示例：

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


© 2015 Microchip Technology Inc.

prog

将变量分配到程序空间中为可执行代码指定的段中。程序空间中的变量无法使用普通 C 语句访问。它们必须由编程人员显式地访问，通常使用表访问行内汇编指令、程序空间可视性窗口，或通过**第 10.4.2 节“访问程序存储器中的对象”**介绍的方法。

auto_psv

将变量分配到程序空间中为自动程序空间可视性窗口访问指定的编译器管理段中。可以使用普通 C 语句读取（但不能写入）auto_psv 空间中的变量，并且最多分配 32K 的总空间。指定 space(auto_psv) 时，无法使用 section 属性指定段名；任何段名都会被忽略，并产生警告。auto_psv 空间中的变量无法放置在特定地址处，或指定反向对齐量。

> **注：** 放入 auto_psv 段的变量不会在启动时被装入数据存储器。该属性可用于减少 RAM 使用量。

DD **dma**——仅对于 PIC24E/H MCU 和 dsPIC33E/F DSC

将变量分配到 DMA 存储区中。DMA 存储区中的变量可以使用普通 C 语句并通过外设 DMA 访问。 __builtin_dmaoffset() 和 __builtin_dmapage() 可以用于确定配置 DMA 外设的正确偏移量。详情请参见**附录 G “内置函数”**。

```
#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

将变量分配到程序空间中为程序空间可视性窗口访问指定的段中。链接器会将该段定位在单次设置 PSVPAG 寄存器即可访问整个变量的位置。PSV 空间中的变量不由编译器管理，不能使用普通 C 语句访问。它们必须由编程人员显式地访问，通常使用表访问行内汇编指令，或使用程序空间可视性窗口。

DD **eedata**——仅对于 PIC24F 和 dsPIC30F/33F DSC

将变量分配到 EEPROM Data（EEData）空间中。EEData 空间中的变量无法使用普通 C 语句访问。它们必须由编程人员显式地访问，通常使用表访问行内汇编指令，或使用程序空间可视性窗口。

pmp

将变量分配在到 PMP 外设关联的片外存储器中。关于完整详细信息，请参见**第 10.5 节 “并行主端口访问”**。

external

将变量分配到用户定义的存储空间中。关于完整详细信息，请参见**第 10.6 节 “外部存储器访问”**。

transparent_union

该属性附加在属于 union 类型的函数参数上，即意味着对应的参数可能具有任意联合成员的类型，但传递参数时它的类型视为第一个联合成员的类型。参数使用透明联合的第一个成员的调用约定传递给函数，而不是使用联合本身的调用约定。联合的所有成员必须具有相同的机器表示形式；为使这种参数传递正确工作，这是必需的。

unordered

unordered 属性指示该变量的位置可能相对于当前 C 源文件内的其他变量移动。

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

unsupported(message)

当使用对象时，该属性将显示一条定制消息。

```
int foo __attribute__((unsupported(“This object is unsupported”));
```

对 foo 进行访问会产生一条警告消息。

unused

该属性附加到一个变量上，表示该变量的本意就是可能不会被使用。对于该变量，编译器不会产生未用变量警告。

weak

weak 属性会使声明作为一个弱符号发出。弱符号可能会被全局定义取代。当向对外部符号的引用应用 weak 时，链接时可以不需要该符号。例如：

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

在以上程序中，如果其他模块未定义 s，程序将仍可以进行链接，但不会为 s 分配地址。条件语句会验证 s 是否已定义（如果已定义，则返回其值）。否则返回 0。该功能有很多用途，主要是为了提供能与可选库链接的通用代码。

weak 属性可应用于函数和变量：

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

在以上代码中，只有在其他模块中链接了函数 compress_data 的情况下，才会使用该函数。确定是否使用该功能会成为一个链接时的决定，而不是编译时的决定。

© 2015 Microchip Technology Inc.

weak 属性对定义的影响更为复杂，需要使用多个文件来说明：

```
/* weak1.c */
int __attribute__((__weak__)) i;

void foo() {
  i = 1;
}

/* weak2.c */
int i;
extern void foo(void);

void bar() {
  i = 2;
}

main() {
  foo();
  bar();
}
```

此处， weak2.c 中 i 的定义会导致该符号成为一个强定义。不会发出任何链接错误，两个 i 将引用相同的存储单元。会为 weak1.c 版本的 i 分配存储单元，但该空间是不可访问的。

不会进行任何检查来确保 i 的两个版本具有相同的类型；将 weak2.c 中的 i 更改为 float 类型将仍然允许链接，但 foo 函数的行为将是非预期的。foo 将值写入 32 位浮点值的低半部分。相反，将 weak1.c 中 i 的弱定义更改为 float 类型可能会导致灾难性的后果。我们会将一个 32 位浮点值写入一个 16 位整型存储单元中，覆盖存储在紧接 i 之后位置的任何变量。

在仅存在弱定义的情况下，链接器将选择第一个这种定义的存储单元。剩余的定义将变为不可访问。

无论符号的类型如何，行为都是相同的；函数和变量的行为方式相同。

注：

© 2015 Microchip Technology Inc.

第 9 章 定点算术支持

9.1 简介

MPLAB XC16 C 编译器根据 ISO/IEC TR 18037 的 N1169 草案（关于嵌入式 C 语言的 ISO C99 技术报告，可从以下位置获取）支持定点算术：

http://www.open-std.org/JTC1/SC22/WG14/www/projects#18037

本章介绍了编译器根据该草案标准支持的类型和操作的特定于实现的详细信息。

- 使能定点算术支持
- 数据类型
- 舍入
- 外部定义
- 混合使用 C 代码和汇编代码

9.2 使能定点算术支持

在 MPLAB XC16 C 编译器中，默认情况下不使能定点算术支持；它必须通过 `-menable-fixed` 编译器开关显式地使能，**第 5.7 节 “驱动程序选项说明”**对该选项进行了介绍。

9.3 数据类型

通过对应于 Microchip 16 位器件固有硬件功能的三种定点格式支持所有 12 种主要定点类型及其别名（见 N1169 的第 4.1 节“Overview and principles of the fixed-point data types”）。

表 9-1： 定点格式——16 位器件

格式	说明
1.15	1 位符号， 15 位小数
1.31	1 位符号， 31 位小数
9.31	9 位有符号整数， 31 位小数

这些格式表示定点 C 数据类型，如下所示。

表 9-2： 定点格式——C 数据类型

类型	格式
`_Fract`	1.15
`short _Fract`	1.15
`signed _Fract`	1.15
`signed short _Fract`	1.15
`unsigned _Fract`	1.15（符号位为 0）
`unsigned short _Fract`	1.15（符号位为 0）
`long _Fract`	1.31
`signed long _Fract`	1.31
`unsigned long _Fract`	1.31（符号位为 0）
`_Accum`	9.31
`short _Accum`	9.31
`long _Accum`	9.31
`signed _Accum`	9.31
`signed short _Accum`	9.31
`signed long _Accum`	9.31
`unsigned _Accum`	9.31（符号位为 0）
`unsigned short _Accum`	9.31（符号位为 0）
`unsigned long _Accum`	9.31（符号位为 0）

`_Sat` 类型说明符指示使值发生饱和，它可用于 N1169 中介绍的任何类型。

无符号类型的表示形式与有符号类型相同，但负数（符号位为 1）在无符号类型中不是有效值。有符号类型在基础格式可表示的最小负数和最大正数处饱和。无符号类型在 0 和格式可表示的最大正数处饱和。

有符号或无符号类型溢出时的默认行为不是饱和 （由 N1169 的第 4.1.3 节 “Rounding and Overflow”介绍的 pragma 伪指令定义）。因此，将可能发生溢出的表达式的结果赋值给未使用 `_Sat` 说明符声明为饱和的有符号或无符号类型的变量时，可能会接收到无效值 （非饱和溢出的结果未定义）。

© 2015 Microchip Technology Inc.

9.4 舍入

支持三种舍入模式，对应于 16 位器件定点乘法功能支持的三种舍入模式。

表 9-3： 舍入模式

模式	说明
截断	截断有符号结果——向负饱和舍入
常规	将有符号结果舍入为最接近的值，趋向于正饱和
收敛	将有符号结果舍入为最接近的值，趋向于偶数

对定点变量的所有操作（无论是否是硬件固有支持的）都按照所选择的通行舍入模式执行。舍入模式可以通过 `-menable-fixed` 编译器开关（如**第 5.7 节“驱动程序选项说明”**所述）全局指定，也可以通过 `-round` 属性（如**第 13.2.2 节 “函数属性”**所述）逐个函数地指定。

《16 位 MCU 和 DSC 程序员参考手册》（DS70157E_CN）更详细地介绍了这些模式。

9.5 被零除

`_Fract` 或 `_Accum` 类型的值被零除的结果未定义，可能会也可能不会导致算术错误陷阱。无论是否存在 `_Sat` 关键字，被零除都不会产生结果类型的最小负饱和值或最大正饱和值。

9.6 外部定义

MPLAB XC16 C 编译器提供了包含文件 `stdfix.h`，其中提供 N1169 第 7.18a 节介绍的常量、 pragma 伪指令、 typedef 和函数定义。

当前 MPLAB XC16 标准 C 库不支持用于格式化 I/O 的定点转换说明符 （在 N1169 第 4.1.9 节“Formatted I/O functions for fixed-point arguments”中介绍）。定点变量可以通过 `(s)printf` 显示，方法是将它们强制类型转换为相应的浮点表示形式 （对于 `_Fract` 为 `double`，对于 `long_Fract` 和 `_Accum` 为 `long double`），然后以该格式显示值。要通过 `(s)scanf` 扫描定点数，需要先将它作为相应的 `double` 或 `long double` 浮点数进行扫描，然后将获得的值强制类型转换为所需的定点类型。

当前 MPLAB XC16 标准 C 库中不提供 N1169 第 4.1.7 节介绍的定点函数。

MPLAB XC16 C编译器支持N1169第4.1.5节介绍的使用后缀`k (K)`和`r (R)`的定点常量。

9.7 混合使用 C 代码和汇编代码

MPLAB XC16 C 编译器会生成假定某些 16 位器件资源由编译器的启动和运行时代码管理的定点代码。嵌入在同一程序中的手工编写汇编代码可能会影响由编译器生成的代码假定的 CPU 状态。

MPLAB XC16 程序可以同时包含直接利用 16 位器件固有定点功能的定点 C 语言和汇编语言代码，但为了使这两种代码安全地互操作，编译器必须在调用可能会改变某些 dsPIC 寄存器状态的汇编语言函数前后保存这些 dsPIC 寄存器。对于需要这样做的汇编语言函数，可以通过提供汇编语言函数的原型来指示 C 编译器这样做。这些原型应指定目标汇编语言函数的 `save(CORCON)` 属性，如**第 13.2.2 节“函数属性”**所述。以这种方式构造的程序可以正常工作，代价是在调用上述的汇编程序前后执行一些状态保存和恢复操作。

© 2015 Microchip Technology Inc.

第 10 章 存储器分配和访问

10.1 简介

存在两大组基于 RAM 的变量：自动 / 参数变量，它们会被分配到某种形式的堆栈中；以及全局 / 静态变量，它们可以自由地定位到整个数据存储空间。以下几节分别介绍了这两组变量的存储器分配。

- 地址空间
- 数据存储空间中的变量
- 程序空间中的变量
- 并行主端口访问
- 外部存储器访问
- 扩展数据空间访问
- dataflash 访问
- 双分区存储器访问
- 压缩存储在闪存中的数据
- 变量到寄存器的分配
- EEPROM 中的变量
- 动态存储器分配
- 存储模型

© 2015 Microchip Technology Inc.

10.2 地址空间

16 位器件结合了传统 PIC® 单片机 （MCU）的特性 （外设、哈佛架构和 RISC）和最新的 DSP 功能 （dsPIC DSC 器件）。这些器件具有两个不同的存储区域：

- 程序存储器包含可执行代码以及 （可选）常量数据。
- 数据存储器包含外部变量、静态变量、系统堆栈和文件寄存器。数据存储器由 near 数据 （它是数据存储空间前 8 KB 的存储区）和 far 数据 （它是数据存储空间的 56 KB 高地址部分）组成。

虽然程序和数据存储区域是明确分开的，dsPIC 和 PIC24 系列处理器的硬件支持通过通常称为程序空间可视性 （PSV）的硬件功能，从程序闪存内访问数据。关于 PSV 如何工作的更多详细信息，请参见器件数据手册或器件系列参考手册。另请参见**第 10.3 节 “数据存储空间中的变量”**和**第 14.8.2 节 “中断服务程序中的 PSV 使用”**。

简单来说，该架构支持通过特殊功能寄存器 （SFR） PSVPAG 将一个 32K 闪存页映射到数据地址空间的高 32K。支持扩展数据空间 （EDS）的器件使用 DSRPAG 寄存器进行映射。此外，还可以映射闪存和其他区域。更多详细信息，请参见**第 10.7 节 “扩展数据空间访问”**。

默认情况下，编译器仅支持直接访问单个 PSV 页，称为 `auto_psv` 空间。在该模型中，可以使用 16 位数据指针。但是，在较大容量的器件中，这使得难以管理闪存中存储的大量常量数据。

此处介绍的扩展使得可以将变量定义为 “受管” PSV 变量。这意味着编译器将自己处理偏移 （PSV 页内）和页。因此，数据指针必须为 32 位。与单一 PSV 页模型相比，编译器可能会产生更多的指令，但这是为了更灵活、更快速地访问闪存中的更多数据需要付出的代价。

© 2015 Microchip Technology Inc.

10.3 数据存储空间中的变量

大多数变量最终会被定位到数据存储空间。例外情况是使用 const 限定的非 auto 变量，它们可能会被放入程序存储空间。

由于 auto 变量和非 auto 变量的存储器分配方式在根本上不同，以下将分别对它们进行讨论。使用 C 语言术语来描述，这两组变量就是分别具有 “自动存储持续时间”和“永久存储持续时间”的变量。

从存储器分配的角度来说，变量基于是否为 auto 类型来分配空间；从而有下面各小节中的分组。

10.3.1 自动和非自动变量与局部和全局变量

术语 “局部”和 “全局”经常用于描述变量，但它们不是语言标准定义的术语。术语“局部变量”通常表示作用域为函数内部的变量，“全局变量”是作用域为整个程序的变量。但是， C 语言具有三种常用的作用域：块、文件 （即内部链接）和程序 （即外部链接）。所以，只使用两个术语来描述它们可能会造成混淆。

例如，在函数之外定义的 static 变量的作用域仅为该文件内部，所以它不可全局访问，但它可以由文件内的多个函数访问，所以它也不是任何一个函数的局部变量。

10.3.2 非自动变量分配和访问

非 auto （static 和 external）变量具有永久存储持续时间，由编译器定位到数据存储空间。如果选择了 constants-in-data 存储模型，编译器也会将使用 const 限定（见 **第 8.10.1 节 “const 类型限定符”**）的非 auto 变量分配到数据存储空间中；否则，它们会被定位到程序存储器中。

10.3.2.1 非自动变量的默认分配

编译器会考虑几种类别的 static 和 external 变量，它们全都与变量在程序开始时应包含的值相关，也即，应在程序启动时清零的变量 （未初始化变量）、应包含非零值的变量 （初始化变量），以及在程序启动时根本不应更改的变量 （持久性变量）。限定为 const 类型的对象通常会被赋予一个初始值，因为它们是只读的。如果未为它们赋予初始值，它们会被与其他未初始化变量组合在一起。

放入 RAM 中的数据可能会在启动时通过从程序存储器复制初始化值进行初始化。

10.3.2.2 静态变量

所有 static 变量都具有永久存储持续时间，即使是在函数内定义的那些“局部静态”变量。局部 static 变量的作用域仅为定义它们的函数或块内部，但与 auto 变量不同，在程序的整个持续时间中都会为它们保留存储器。因而，它们将像其他非自动变量一样分配存储器。静态变量可由其他函数通过指针访问，因为它们具有永久持续时间。

程序将确保 static 变量在函数调用之间保留其值，除非通过一个指针显式地进行修改。

属于 static 类型且进行初始化的变量在程序执行过程中仅进行一次初始值赋值。因此，它们可能优于已初始化 auto 对象，后者在每次定义它们的块开始执行时都需要进行赋值。运行时启动代码初始化所有初始化静态变量的方式与其他非 auto 初始化对象相同，请参见**第 5.4.2 节 “启动和初始化”**。

10.3.2.3 非自动变量长度限制

编译器选项 -mlarge-arrays 允许您定义和访问大于 32K 的数组。您必须通过指定大小足以包含此类对象的存储空间来确保有足够的空间来分配此类数组。

使用该选项会对代码生成方式产生一些影响，因为它会影响 size_t 类型的定义，将其长度增大为 unsigned long int 类型。如果用作全局选项，它会影响索引中使用的许多操作（使操作变得更复杂）。局部使用该选项可能会影响访问变量的方式。考虑到这些因素，使用大型数组是需要仔细规划的。本节介绍一些供其使用的技术。

选择 -mlarge-arrays 选项时会发生两个情况：

1. 编译器以不同的方式生成用于访问数组的代码。
2. 编译器将 size_t 类型定义为 unsigned long int。

如果在整个程序中使用，第 1 项会对代码长度产生负面影响。可以仅使用该选项编译单个模块并让它工作，但存在一些局限性，将在稍后讨论。

第 2 项会影响外部函数接收或返回 size_t 类型的对象时的调用约定。编译器提供了已编译的库来处理更长的 size_t，这些对象将由链接器自动选择 （如果它们存在）。

将 -mlarge-arrays 和正常大小的数组混合在一起是相对简单的，可能是使用该功能的最好方式。存在几个使用限制：在此类模块中定义的函数不应调用使用 size_t 的外部程序，并且此类模块中定义的函数不应接收 size_t 作为参数。

例如，用户可以定义一个大数组和一个访问函数，然后由其他代码模块使用该访问函数来访问该数组。优点是只有一个模块需要使用 -mlarge-arrays 进行编译，缺点是需要使用访问函数来访问该数组。在使用 -mlarge-arrays 编译整个程序会对代码长度和速度产生负面影响的情况下，这会很有用。

© 2015 Microchip Technology Inc.

代码示例：

file1.c

```
/* to be compiled -mlarge-arrays */
__prog__ int array1[48000] __attribute__((space(prog)));
__prog__ int array2[48000] __attribute__((space(prog)));

int access_large_array(__prog__ int *array, unsigned long index) {
  return array[index];
}
```

file2.c

```
/* to be compiled without -mlarge-arrays */
extern __prog__ int array1[] __attribute__((space(prog)));
extern __prog__ int array2[] __attribute__((space(prog)));

extern int access_large_array(__prog__ int *array,
  unsigned long index);

main() {
  fprintf(stderr,"Answer is: %d\n", access_large_array(array1,
          39543));
  fprintf(stderr,"Answer is: %d\n", access_large_array(array2,
          16));
}
```

10.3.2.4 更改非自动变量分配

如**第 10.2 节 “地址空间”**所述，编译器根据使用的存储模型和数据是否初始化来安排要放入段中的数据。当模块在链接时进行合并时，链接器将基于各个段的属性确定它们的起始地址。

可能会出现特定变量必须定位到特定地址处，或处于某个地址范围内的情况。实现这一点的最简单方法是使用 address 属性，如**第 7 章 “MPLAB XC16 和 ANSI C 之间的差别”**所述。例如，要将变量 Mabonga 定位在数据存储器中的地址 0x1000 处：

```
int __attribute__ ((address(0x1000))) Mabonga = 1;
```

可以将一组公共变量分配到指定的段，并加上地址说明：

```
int __attribute__ ((section("mysection"), address(0x1234))), foo;
```

10.3.2.5 数据存储器分配宏

下面介绍了可用于在数据存储器中分配空间的宏。有以下两种类型：需要参数的宏和不需要参数的宏。

下面的宏需要指定对齐量的参数 N。 N 必须为 2 的幂，最小值为 2。

```
#define _XBSS(N)   __attribute__((space(xmemory), aligned(N)))
#define _XDATA(N)  __attribute__((space(xmemory), aligned(N)))
#define _YBSS(N)   __attribute__((space(ymemory), aligned(N)))
#define _YDATA(N)  __attribute__((space(ymemory), aligned(N)))
#define _EEDATA(N) __attribute__((space(eedata),  aligned(N)))
```

例如，要声明一个位于 X 存储区中、对齐到 32 字节地址处的未初始化数组：

```
int _XBSS(32) xbuf[16];
```

要声明一个位于 EEPROM 中、无特殊对齐的初始化数组：

```
int _EEDATA(2) table1[] = {0, 1, 1, 2, 3, 5, 8, 13, 21};
```

下面的宏不需要参数。它们可以用于将变量定位到持久性数据存储区或 near 数据存储区。

```
#define _PERSISTENT __attribute__((persistent))
#define _NEAR       __attribute__((near))
```

例如，要声明两个在器件复位后保留其值的变量：

```
int _PERSISTENT var1,var2;
```

10.3.3 自动变量分配和访问

本节介绍 Auto 变量（那些具有自动存储持续时间的变量）分配。这也包括函数参数变量（其行为类似于 Auto 变量）以及由编译器定义的临时变量。

auto（自动，*automatic* 的缩写）变量是局部变量的默认类型。除非显式地声明为 static 类型，否则局部变量将设为 auto 类型。如果需要，可以使用 auto 关键字。

正如其名称所暗示的，auto 变量会在代码块执行时自动产生，然后在代码块退出之后消失。因为它们不是在程序的整个持续时间中都存在，所以可以在变量不存在时回收存储器，并将其分配给程序中的其他变量。

通常此类变量存储在某种形式的数据堆栈中，可以根据每个函数的要求简便地分配然后释放存储器。**第 10.3.3.1 节 “软件堆栈”**对该堆栈进行了介绍。

标准限定符 const 和 volatile 可同时用于 auto 变量，它们不会影响这些变量在存储器中的定位。这意味着，使用 const 限定的局部对象仍是 auto 对象，因此将在堆栈中分配存储器，而不是像非 auto const 对象一样在程序存储器中分配。

10.3.3.1 软件堆栈

dsPIC DSC 器件将寄存器 W15 专门用作软件堆栈指针。所有处理器堆栈操作（包括函数调用、中断和异常）都使用软件堆栈。堆栈朝着更高存储器地址方向向上增长。

dsPIC DSC 器件还支持堆栈溢出检测。如果堆栈指针限制寄存器 SPLIM 已初始化，器件会在所有堆栈操作时测试是否溢出。如果将发生溢出，处理器会发出堆栈错误异常。默认情况下，这将导致处理器复位。应用程序也可以通过定义一个名为 _StackError 的中断函数来安装一个堆栈错误异常处理程序。详情请参见**第 14 章 “中断”**。

C 运行时启动模块会在启动和初始化序列期间初始化堆栈指针（W15）和堆栈指针限制寄存器。初始值通常由链接器提供，链接器会从未用数据存储器中分配可能的最大堆栈。链接映射输出文件中会报告堆栈的位置。应用程序可以通过使用 --static 链接器命令行选项来确保至少有最小大小的堆栈可用。关于详细信息，请参见《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

© 2015 Microchip Technology Inc.

或者，也可以通过汇编源文件中的用户定义的段分配特定大小的堆栈。在以下示例中，为堆栈保留 0x100 字节的数据存储器：

```
.section *,data,stack
.space 0x100
```

链接器会分配一个相应大小的段，并初始化 `__SP_init` 和 `__SPLIM_init`，使运行时启动代码可以正确地初始化堆栈。请注意，由于它是一个正常的汇编代码段，所以可以使用诸如`address`之类的属性来进一步定义堆栈。关于更多信息，请参见《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

10.3.3.2 C 堆栈使用

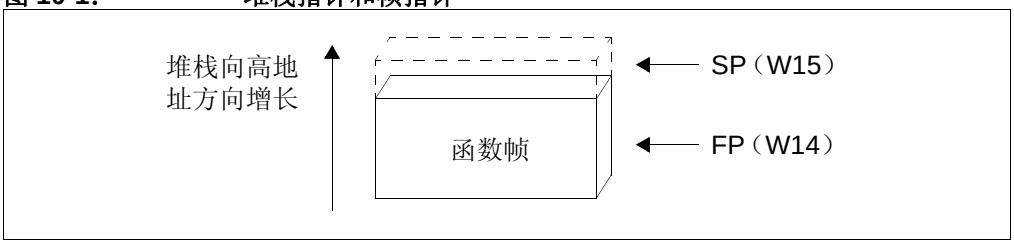
C 编译器使用软件堆栈来：

- 分配自动变量
- 向函数传递参数
- 在中断函数中保存处理器状态
- 保存函数返回地址
- 存储临时结果
- 跨函数调用保存寄存器

运行时堆栈从低地址到高地址向上增长。编译器使用两个工作寄存器来管理堆栈：

- W15——它是堆栈指针 （SP）。它指向栈顶，即定义为堆栈中的第一个未用单元。
- W14——它是帧指针 （FP）。它指向当前函数的帧。如果需要，每个函数会在栈顶创建一个新的帧，并在其中分配自动和临时变量。编译器选项 `-fomit-frame-pointer` 可以用于限制 FP 的使用。

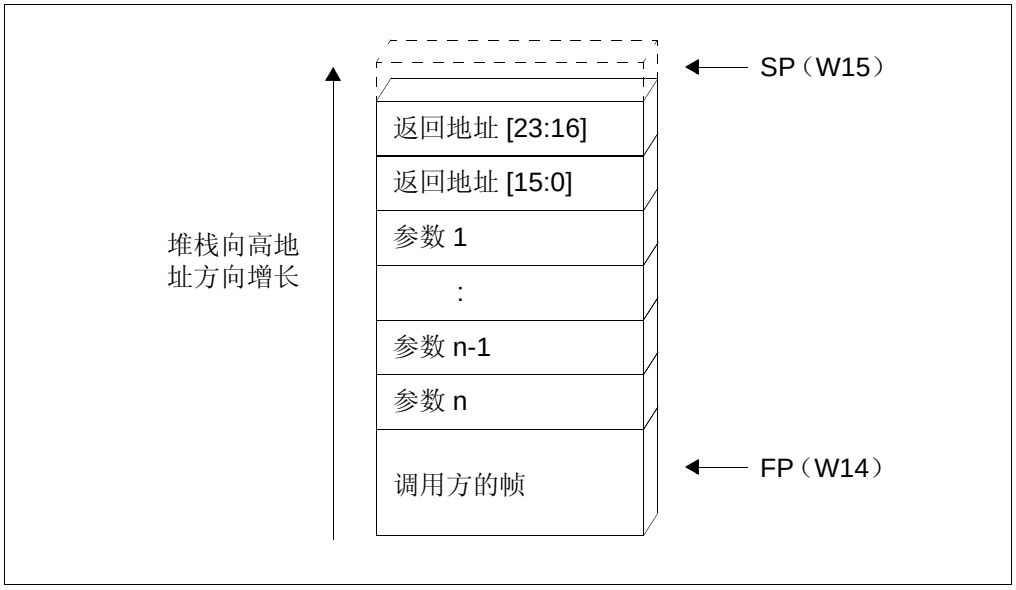
图 10-1： 堆栈指针和帧指针



`libpic30-`*`omf`*`.a` 中的 C 运行时启动模块会将堆栈指针 W15 初始化为指向栈底，将堆栈指针限制寄存器初始化为指向栈顶。堆栈向上增长，如果它将增长到超出堆栈指针限制值寄存器中的值，则将产生一个堆栈错误陷阱。用户可以通过初始化堆栈指针限制寄存器来进一步限制堆栈增长。

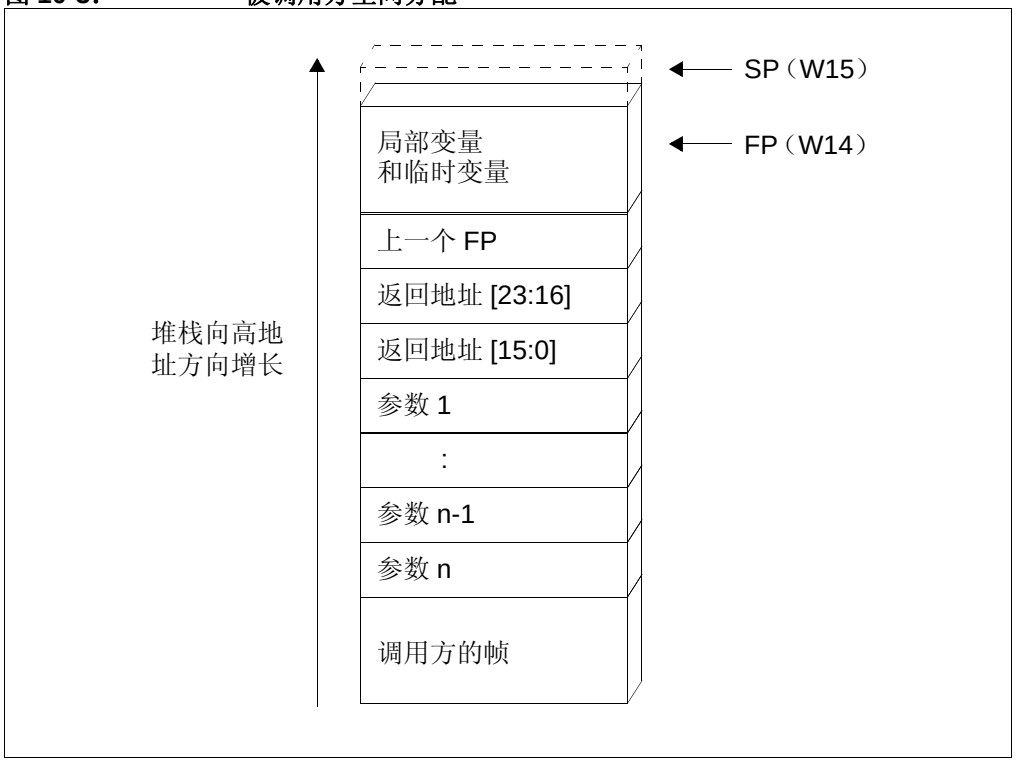
下图说明了在调用函数时涉及的步骤。执行 `CALL` 或 `RCALL` 指令会将返回地址压入软件堆栈。请参见图 10-2。

图 10-2： CALL 或 RCALL



被调用的函数 （被调用方）现在可以为其局部现场分配空间 （图 10-3）。

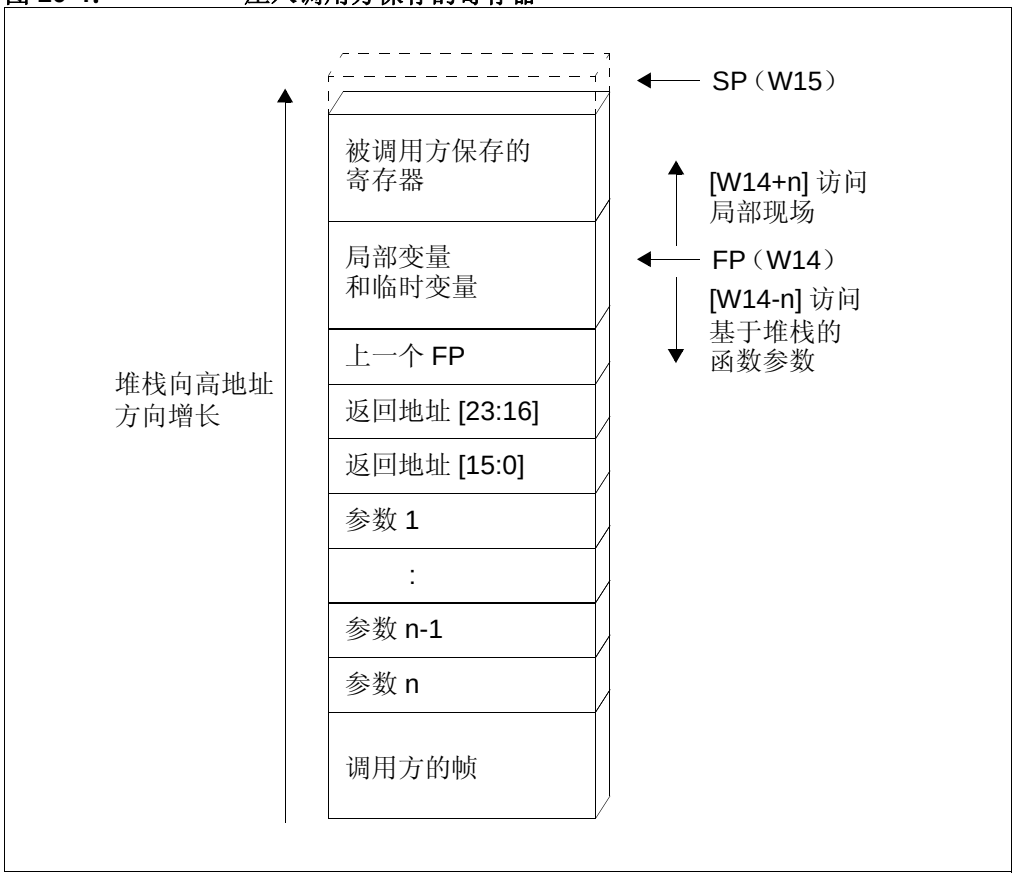
图 10-3： 被调用方空间分配



© 2015 Microchip Technology Inc.

最后，函数中使用的任何被调用方保存的寄存器会被压入堆栈 （图 10-4）。

图 10-4：　　压入调用方保存的寄存器



10.3.3.3　自动变量长度限制

如果程序需要不是在整个程序中都可访问的大对象，可以考虑将它们设为局部对象，但使用 `static` 说明符。此类变量仍然是函数的局部变量，但不再是 `auto` 类型，会被分配不处于软件堆栈中的永久存储空间。

在最大长度方面，`auto` 对象与非 `auot` 对象一样存在类似的约束，但它们分配到软件堆栈中而不是固定的存储器位置。介绍定义和使用大型数组的**第 10.3.2.3 节“非自动变量长度限制”**也适用于 `auto` 对象。

10.3.4　更改自动变量分配

由于 `auto` 变量是在软件堆栈中动态地分配空间，所以使用 `address` 属性或其他机制来使它们分配在非默认位置是不允许的。

10.4 程序空间中的变量

16 位内核系列处理器的硬件支持通过通常称为程序空间可视性 （PSV）的硬件功能，从程序闪存内访问数据。关于 PSV 如何工作的更多详细信息，请参见器件数据手册或器件系列参考手册。另请参见**第 10.4.1 节“程序存储器对象的分配和访问”**和**第 14.8.2 节 “中断服务程序中的 PSV 使用”**。

简单来说，该架构支持通过特殊功能寄存器（SFR）PSVPAG 或 DSRPAG 将一个 32K 闪存页映射到数据地址空间的高 32K。默认情况下，编译器仅支持直接访问单个 PSV 页，称为 auto_psv 空间。在该模型中，可以使用 16 位数据指针。但是，在较大容量的器件中，这使得难以管理闪存中存储的大量常量数据。

使能选项 -mconst-in-code 时，使用 const 限定的不属于 auto 类型的变量会被放入程序存储器。使用const限定的所有auto变量都会与其他auto变量一起放入堆栈。

任何使用 const 限定的 （自动或非自动）变量都永远是只读的，尝试在源代码中写入这些变量将导致编译器发出错误。

const 对象通常会定义初始值，因为程序无法在运行时写入这些对象。但是，这并不是一项要求。未初始化 const 对象与其他未初始化 RAM 变量一起分配空间，但仍然是只读的。以下给出了 const 对象定义的示例。

```
const char IOtype = 'A';  // initialized const object
const char buffer[10];    // I reserve memory in RAM
```

关于用于表示程序存储器中 const 限定变量的等效汇编符号，请参见**第 16 章 “混合使用 C 代码和汇编代码”**。

10.4.1 程序存储器对象的分配和访问

存在许多由编译器分配到程序存储器的对象。以下几节说明了这些对象，以及它们如何由编译器分配到它们最终的存储器位置和如何访问它们。

10.4.1.1 字符串和常量对象

默认情况下，编译器会自动安排将字符串和使用 const 限定的初始化变量分配在 auto_psv 段中，该段会被映射到 PSV 窗口。指定 -mconst-in-data 选项可指示编译器不使用 PSV 窗口，这些对象将与其他基于 RAM 的变量分配在一起。

在默认存储模型中， PSV 页固定为 1 页，通过 auto_psv 存储空间表示。访问单个自动 PSV 页是非常高效的，因为不需要任何页处理。使用**第 10.4.2.1 节 “受管 PSV 访问”**章节介绍的技术，可以访问额外的闪存。

© 2015 Microchip Technology Inc.

10.4.1.2 安全闪存中使用 CONST 限定的变量

使用由编译器管理的 PSV 常量段，可以在安全闪存段中支持 `const` 限定变量并设定初始值。例如：

```
const int __attribute__((boot)) time_delay = 55;
```

如果从 `time_delay` 的定义中省略 `const` 限定符，该语句将被拒绝，并产生一条错误消息。（不支持安全 RAM 中的初始化变量）。

由于指定了 `const` 限定符，所以变量 `time_delay` 可以分配在由引导段拥有的 PSV 常量段中。此外，也可以使用 `space(auto_psv)` 属性显式指定 PSV 常量段：

```
int __attribute__((boot,space(auto_psv))) bebop = 20;
```

使用字符串字面值初始化的指针变量需要进行特殊处理。例如：

```
char * const foo __attribute__((boot)) = "eek";
```

编译器会识别字符串字面值 "eek" 必须分配在与指针变量 `foo` 相同的 PSV 常量段中。

无论是选择了 constants-in-code 还是 constants-in-data 的存储模型，编译器都会根据安全段的需要创建并管理 PSV 常量段。对于用户管理的 PSV 段的支持通过下面说明的对象兼容性模型来维护。

在进入引导或安全函数时，PSVPAG 会被设置为正确的值。该值将在任何外部函数调用后恢复。

10.4.1.3 字符串字面值作为参数

除了用作初始值设定项之外，字符串字面值还可以用作函数参数。例如：

```
myputs("Enter the Dragon code:\n");
```

此处，字符串字面值的分配取决于其周围的代码。如果该语句出现在引导或安全函数中，字面值将被分配在相应的 PSV 常量段中。否则，它会根据常量存储模型被放入通用 （非安全）存储器。

回忆一下，存储在安全段中的数据不能由任何其他段读取。例如，无法使用被分配到安全段中的字符串调用标准 C 库函数 `puts()`。因此，作为函数参数出现的字面值只能传递给处于同一安全段中的函数。对于通过指针和数组引用的对象，也是如此。按值传递的 `char`、 `int` 和 `float` 等简单标量类型可以传递给处于不同段中的函数。

10.4.2 访问程序存储器中的对象

将对象分配到程序存储器中和访问它们被视为两个独立的问题。编译器要求您对变量进行限定，以指示如何访问它们。您可以选择让编译器管理对这些对象的访问，也可以自己完成，这会更有效率，但是更复杂。

10.4.2.1 受管 PSV 访问

编译器引入了几个新的限定符（针对受众中的语言专家的 CV 限定符）。类似于 `const volatile` 限定符，这些新限定符可以应用于对象或指针目标。这些限定符是：

- `__psv__`，用于访问不跨越 PSV 边界的对象，例如在 `space(auto_psv)` 或 `space(psv)` 中分配的那些对象
- `__prog__`，用于访问可能跨越 PSV 边界的对象，特别是在 `space(prog)` 中分配的那些对象，但它可以应用于闪存中的任意对象
- `__eds__`，用于访问可能处于闪存或扩展数据空间（对于具有 >32K RAM 的器件）的对象，请参见**第 10.7 节 “扩展数据空间访问”**中的 `__eds__`。

对于放入 `space(auto_psv)` 中的对象，通常不需要指定 `__psv__` 或 `__prog__`。

定义处于编译器管理的闪存空间中的变量通过以下方式完成：

```
__psv__ unsigned int Flash_variable __attribute__((space(psv)));
```

现在，读取变量将导致编译器生成会根据需要调整相应 PSV 页 SFR 的代码，以正确访问变量。这些限定符同样可以修饰指针：

```
__psv__ unsigned int *pFlash;
```

产生指向 PSV 中某个数据的指针，该指针可以通过正常方式指定给受管 PSV 对象。例如：

```
pFlash = &Flash_variable;
```

10.4.2.2 对象兼容性模式

由于处于安全段中的函数会将 PSVPAG 设置为它们相应的 `psv` 常量段，所以必须为管理多个 PSVPAG 寄存器值设立约定。在先前版本的编译器中，如果选择了默认的 `constants-in-code` 存储模型，则在程序启动期间会设置单个 PSVPAG 值。编译器依靠该预设值来访问 `const` 变量和字符串字面值，以及任何使用 `space(auto_psv)` 特别指定的变量。

MPLAB XC16 提供了对于多个 PSVPAG 值的支持。使用 `space(auto_psv)` 声明的变量可以与安全段常量变量和 / 或同一源文件中的受管 `psv` 变量组合在一起。采用单个 PSVPAG 预设值的预编译对象与定义安全段 `psv` 常量或受管 `psv` 变量的对象是链接兼容的。

虽然 PSVPAG 被视为是编译器管理的资源，但函数调用约定没有任何改变。

© 2015 Microchip Technology Inc.

10.4.2.3 ISR 注意事项

使用受管 PSV 指针进行的数据访问一定不是原子操作，意味着需要执行几条指令才能完成访问。如果访问不应中断，则必须小心。

此外，中断服务程序（ISR）总是不会真正知道 PSVPAG 寄存器当前将是什么状态。不幸的是，编译器无从确定该信息是否在所有情况下都很重要。

编译器将简单地假设，中断服务程序的编写者知道 ISR 是否需要自动的、编译器管理的 PSVPAG。当选择默认的 `-mconst-in-code` 选项时，需要它才能访问 `auto_psv` 空间中的任何常量数据，或者任意字符串字面值或字符串常量。定义中断服务程序时，最好指定是否需要定义特殊功能寄存器 PSVPAG 的默认设置。

这可以通过为中断函数定义另外添加一个属性实现：

- `auto_psv`——编译器会将 PSVPAG 寄存器设置为正确值，以访问 `auto_psv` 空间，并确保退出 ISR 时，它会被恢复
- `no_auto_psv`——编译器不会设置 PSVPAG 寄存器

例如：

```
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void) {
  IFS0bits.T1IF = 0;
}
```

提供上述选择是为了，如果您特别关心中断延时，则可以选择最佳的选项。进入函数时，保存和设置 PSVPAG 需要大约 3 个周期；从函数中退出时，恢复设置还需要一个周期。

请注意，`boot` 或 `secure` 中断服务程序将对它们的常量数据使用不同的 PSVPAG 寄存器设置。

10.4.3 程序存储器变量的长度限制

任何类型的数组（包括聚合类型的数组）都可以使用 `const` 限定并放入程序存储器中。结构和联合聚合类型也是，请参见**第 8.6 节 “结构和联合”**。这些对象的长度往往会变得很大，并可能影响存储器分配。

对于分配在编译器管理的 PSV 窗口 （`auto_psv` 空间）中的对象，可供分配的存储器总量受 PSV 窗口本身的大小限制。因此，单个对象的大小不能大于 PSV 窗口的大小，并且所有此类对象的总大小不得大于该窗口。

在最大大小方面，分配到程序存储器中的变量存在与数据空间对象类似的约束，但它们分配到较大的程序空间而不是数据空间存储器。介绍定义和使用大型数组的**第 10.3.2.3 节 “非自动变量长度限制”**也适用于程序空间存储器中的对象。

10.4.4 更改程序存储器变量分配

在某种程度上，分配到程序存储器的变量可以分配到特定存储地址。**第 10.3.2.4 节“更改非自动变量分配”**介绍了的特定地址和段也适用于程序存储空间中的对象。请注意，您不能对处于 auto_psv 空间中的对象使用 address 属性。

space 属性可以用于定义定位为在 PSV 窗口中使用的变量。要指定分配到编译器管理的 PSV 空间中的变量，请使用属性 space(auto_psv)。要将变量分配到不由编译器管理的段中来进行 PSV 访问，请使用 space(psv) 属性。关于这些属性的更多信息，请参见**第 7 章 “MPLAB XC16 和 ANSI C 之间的差别”**。

例如，要将变量放入 auto_psv 空间（这将导致在闪存中以方便的方式分配存储单元，并通过*单一* PSVPAG 设置访问），可以指定：

```
unsigned int Flash_variable __attribute__((space(auto_psv)));
```

与闪存相关的其他用户空间有：

- space(psv)——编译器不会自动访问的 PSV 空间
- space(prog)——编译器不会自动访问的闪存中任意单元

请注意，psv 和 auto_psv 空间都会被适当地分块或对齐，以便单一 PSVPAG 设置适合用于访问整个变量。

关于 PSV 使用的更多信息，请参见《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

© 2015 Microchip Technology Inc.

10.5 并行主端口访问

一些器件含有并行主端口 （PMP）外设，使得可以将各种存储器和非存储器器件直接与这些器件连接。对外设的访问通过外设选择进行控制。关于该外设的更多信息，请参见器件系列参考手册或具体器件的数据手册。

注： 带有 EPMP 的器件上不支持 PMP 属性。请改为使用扩展数据空间（EDS）。请参见**第 10.7 节 “扩展数据空间访问”**。

该外设可能还需要进行一些后续的配置，具体取决于所连接的存储器的类型和品牌。编译器不会自动执行这些配置。

此处介绍的扩展使得可以将变量定义为 PMP。这意味着编译器将与 PMP 外设进行通信，以访问变量。

要使用该功能：

- 初始化 PMP——定义初始化函数：`void __init_PMP(void)`
- 声明新的存储空间
- 在 PMP 空间中定义变量

10.5.1 初始化 PMP

PMP 外设需要先进行初始化，然后才能正确处理访问。请参见所连接的器件的相应文档，以及所使用的 16 位器件的数据手册。

如果使用 PMP，在正常的 C 运行时初始化期间，工具套件将调用 `void __init_PMP(void)`。如果使用了自定义的初始化，请确保调用该函数。

该函数将在 PMMODE 和 PMCON 特殊功能寄存器中进行必要的设置。特别是：

- 外设不应配置为产生中断：
  `PMMODEbits.IRQM = 0`
- 外设不应配置为产生增量：
  `PMMODEbits.INCM = 0`
  在运行时，编译器会根据需要修改该设置。
- 外设应初始化为 16 位模式：
  `PMMODEbits.MODE16 = 1`
  在运行时，编译器会根据需要修改该设置。
- 外设应配置为一种主模式：
  `PMMODEbits.MODE = 2` 或 `PMMODEbits.MODE = 3`
- 对于所连接的器件，根据需要设置等待状态 `PMMODEbits.WAITB`、`PMMODEbits.WAITM` 和 `PMMODEbits.WAITE`。
- 特殊功能寄存器 PMCON 应进行相应的配置，确保在定义存储空间时，片选功能位 `PMCONbits.CSF` 与传递给编译器的信息匹配。

部分示例为：

```
void __init_PMP(void) {
  PMMODEbits.IRQM = 0;
  PMMODEbits.INCM = 0;
  PMMODEbits.MODE16 = 1;
  PMMODEbits.MODE = 3;
  /* device specific configuration of PMMODE and PMCCON follows */
}
```

10.5.2 声明新的存储空间

编译器工具套件需要通过 PMP 连接的每个附加存储器的信息。为了 16 位器件链接器可以正确地分配存储器，必须提供关于可用存储器大小和需要的片选数量的信息。

在**第 7 章“MPLAB XC16 和 ANSI C 之间的差别”**中，介绍了 pmp 存储空间。该属性用于两种用途：声明扩展存储空间和为外部存储器分配 C 变量声明 （这将在下一个子节介绍）。

声明扩展存储器需要提供存储空间的大小。您可以选择将存储器指定到特定的片选引脚；如果未指定，则将假定未使用片选。这些存储器声明看起来类似正常的外部 C 声明：

```
  extern int external_PMP_memory
__attribute__((space(pmp(size(1024),cs(0)))));
```

以上定义了大小为 1024 字节的外部存储器，并且没有片选。除非使用片选，否则编译器仅支持一个 PMP 存储器：

```
  extern int PMP_bank1 __attribute__((space(pmp(size(1024),cs(1)))));
  extern int PMP_bank2 __attribute__((space(pmp(size(2048),cs(2)))));
```

以上 PMP_bank1 将使用片选引脚 1 激活 （访问该存储区中的变量时，地址引脚 14 将置为有效）。 PMP_bank2 将使用片选引脚 2 激活 （地址引脚 15 将置为有效）。

请注意，使用片选时，每个存储区的最大存储空间为 16 KB。建议将声明放入公共头文件中，以便所有翻译单元都可以使用声明。

10.5.3 在 PMP 空间中定义变量

pmp 空间属性还可以用于向空间分配各个变量。这要求提供存储空间声明。给定前一子节的声明，可以构造以下变量声明：

```
  __pmp__ int external_array[256]
          __attribute__((space(pmp(external_PMP_memory))));
```

external_array 将分配到前面声明的存储器 external_PMP_memory 中。如果只有一个 PMP 存储器，并且未使用片选，则可以省略对存储器的显式引用。但是，始终显式引用存储器是一种好的做法，这可以使代码更容易维护。

请注意，类似于受管 PSV 指针，以上使用了新类型的限定符 __pmp__ 来修饰变量。将该限定符附加到变量或指针时，它指示编译器生成通过PMP外设进行访问的正确序列。

现在，已经声明了一个变量，可以使用正常的 C 语法来访问它。编译器将生成可与 PMP 外设进行正确通信的代码。

© 2015 Microchip Technology Inc.

10.6 外部存储器访问

不是 Microchip 的所有 16 位器件都有并行主端口外设 （见**第 10.5 节 “并行主端口访问”**），也不是所有存储器都适合于连接 PMP（例如，Microchip 销售的串行存储器）。工具套件提供了针对所有所称的外部存储器的更通用接口，虽然可以看到，存储器不一定要是外部存储器。

类似于 PMP 访问，工具链需要知道关于所连接的外部存储器的信息。但是不同于 PMP 访问，编译器不知道如何访问这些存储器。提供了一种机制，应用程序可以通过该机制指定应如何访问这些存储器。

外部对象的地址长度均为 32 位。可连接的最大存储器为 64K（16 位）；地址中的其他 16 位用于唯一地标识存储器。总共可以连接 64K （16 位）的存储器 （理论上）。

要使用该功能，请仔细阅读以下几节。

10.6.1 声明新的存储空间

这非常类似于为 PMP 访问声明新的存储空间。

16 位工具套件需要知道关于每个外部存储器的信息。为了 16 位器件链接器可以正确地分配存储器，需要提供关于可用存储器大小和 （可选）存储器起点的信息。

在**第 7 章 “MPLAB XC16 和 ANSI C 之间的差别”**中，介绍了 `external` 存储空间。该属性用于两种用途：声明扩展存储空间和为外部存储器分配 C 变量声明 （这将在下一个子节介绍）。

声明扩展存储器需要提供存储空间的大小。您可以选择指定该存储器的起点；如果未指定起点，则假定为 0x0000。

```
  extern int external_memory
__attribute__((space(external(size(1024)))));
```

以上定义了大小为 1024 字节的外部存储器。该存储器可以通过它的给定名称 `external_memory` 唯一地标识。

10.6.2 在外部空间中定义变量

`external` 空间属性还可以用于向空间分配各个变量。这要求提供存储空间声明。给定前一子节的声明，可以构造以下变量声明：

```
  __external__ int external_array[256]
               __attribute__((space(external(external_memory))));
```

`external_array` 将分配到前面声明的存储器 `external_memory` 中。

请注意，类似于受管 PSV 对象，以上使用了新类型的限定符 `__external__` 来修饰变量。将该限定符附加到变量或指针目标时，它指示编译器生成用于访问这些对象的正确序列。

声明外部存储器变量之后，可以使用正常的 C 语法来访问它。编译器将生成代码，通过程序员必须定义的特殊辅助函数来访问变量。它们在下一子节中介绍。

10.6.3 定义如何访问存储空间

对放入外部存储器中的变量的引用，通过使用几个辅助函数进行控制。最多可以定义 5 个函数用于读取数据，定义 5 个函数用于写入数据。这些函数中有一个通用函数，如果其他 4 个函数中有任何一个函数未定义，但是又需要该函数，则会调用通用函数。如果未定义任何函数，编译器将会产生一条错误消息。在下一子节中，将会介绍一个简短的示例。通常，定义各个函数可以生成更有效率的代码。

10.6.3.1 用于读取的函数

read_external

```
void __read_external(unsigned int address,
  unsigned int memory_space,
  void *buffer,
  unsigned int len)
```

该函数是通用读取函数，如果需要以下某个函数，但该函数未定义，则将调用通用函数。该函数应执行一些必需的步骤，从名为 `memory_space` 的外部存储器中，从地址 `address` 开始读取数据，填充 `buffer` 中的 `len` 字节的存储空间。

read_external8

```
unsigned char __read_external8(unsigned int address,
  unsigned int memory_space)
```

从外部存储空间 `memory_space`，从地址 `address` 开始读取 8 位。尝试访问 8 位大小的对象时，编译器将调用该函数。

read_external16

```
unsigned int __read_external16(unsigned int address,
  unsigned int memory_space)
```

从外部存储空间 `memory_space`，从地址 `address` 开始读取 16 位。尝试访问 16 位大小的对象时，编译器将调用该函数。

read_external32

```
unsigned long __read_external32(unsigned int address,
  unsigned int memory_space)
```

从外部存储空间 `memory_space`，从地址 `address` 开始读取 32 位。尝试访问 32 位大小的对象时 （例如 `long` 或 `float` 类型），编译器将调用该函数。

read_external64

```
unsigned long long __read_external64(unsigned int address,
  unsigned int memory_space)
```

从外部存储空间 `memory_space`，从地址 `address` 开始读取 64 位。尝试访问 64 位大小的对象时 （例如 `long long` 或 `long double` 类型），编译器将调用该函数。

© 2015 Microchip Technology Inc.

10.6.3.2 用于写入的函数

write_external

```
void __write_external(unsigned int address,
  unsigned int memory_space,
  void *buffer,
  unsigned int len)
```

该函数是通用写入函数，如果需要以下某个函数，但该函数未定义，则将调用通用函数。该函数应执行一些必需的步骤，将 buffer 中 len 字节的数据写入名为 memory_space 的外部存储器中，从地址 address 开始。

write_external8

```
void __write_external8(unsigned int address,
  unsigned int memory_space,
  unsigned char data)
```

将 data 的 8 位写入外部存储空间 memory_space，从地址 address 开始。尝试写 8 位大小的对象时，编译器将调用该函数。

write_external16

```
void __write_external16(unsigned int address,
  unsigned int memory_space,
  unsigned int data)
```

将 data 的 16 位写入外部存储空间 memory_space，从地址 address 开始。尝试写 16 位大小的对象时，编译器将调用该函数。

write_external32

```
void __write_external32(unsigned int address,
  unsigned int memory_space,
  unsigned long data)
```

将 data 的 32 位写入外部存储空间 memory_space，从地址 address 开始。尝试写 32 位大小的对象时 （例如 long 或 float 类型），编译器将调用该函数。

write_external64

```
void __write_external64(unsigned int address,
  unsigned int memory_space,
  unsigned long long data)
```

将 data 的 64 位写入外部存储空间 memory_space，从地址 address 开始。尝试写 64 位大小的对象时（例如 long long 或 long double 类型），编译器将调用该函数。

10.6.4 外部示例

以下示例片断来自工作示例 （在 Examples 文件夹中）。

该示例使用外部存储器实现可寻址的位存储器。在此例中，每个位存储在真正的数据存储器中，而不是片外。代码将定义 512 个单元的 external 存储器，并使用相应的读写函数将访问映射到本地数据存储器中的一个 64 字节区域。

首先，定义外部存储器：

```
 extern unsigned int bit_memory
__attribute__((space(external(size(512)))));
```

接着，定义相应的读写函数。这些函数将使用以正常方式保留的存储器数组。

```
 static unsigned char real_bit_memory[64];
 unsigned char __read_external8(unsigned int address,
                                unsigned int memory_space) {
   if (memory_space == bit_memory) {
     /* an address within our bit memory */
     unsigned int byte_offset, bit_offset;
     byte_offset = address / 8;
     bit_offset = address % 8;
     return (real_bit_memory[byte_offset] >> bit_offset) & 0x1;
   } else {
     fprintf(stderr,"I don't know how to access memory space:%d\n",
             memory_space);
   }
   return 0;
 }
 void __write_external8(unsigned int address,
                        unsigned int memory_space,
                        unsigned char data) {
   if (memory_space == bit_memory) {
     /* an address within our bit memory */
     unsigned int byte_offset, bit_offset;
     byte_offset = address / 8;
     bit_offset = address % 8;
     real_bit_memory[byte_offset] &= (~(1 << bit_offset));
     if (data & 0x1) real_bit_memory[byte_offset] |=
        (1 << bit_offset);
   } else {
     fprintf(stderr,"I don't know how to access memory space:%d\n",
             memory_space);
   }
 }
```

这些函数以类似的方式工作：

- 如果访问 bit_memory，则
  - 确定正确的字节偏移和位偏移
  - 读或写 real_bit_memory 中的相应位置
- 否则，访问另一个存储器 （其访问是未知的）

© 2015 Microchip Technology Inc.

解决问题的两个主要部分后，产生一些变量和访问代码：

```
__external__ unsigned char bits[NUMBER_OF_BITS]
             __attribute__((space(external(bit_memory))));
// inside main
  __external__ unsigned char *bit;
  bit = bits;
  for (i = 0; i < 512; i++) {
    printf("%d ",*bit++);
  }
```

除了 __external__ CV 限定符之外，也可以使用一般的 C 语句来定义和访问外部存储空间中的变量。

10.7 扩展数据空间访问

将变量或指针目标限定为可通过扩展数据空间窗口访问，这样您可以简便地访问被放入各种不同存储空间中的对象。它们包括： space(data)（及其子集）、eds、space(eedata)、space(prog)、space(psv)、space(auto_psv)，以及某些器件上的 space(pmp)。并非所有器件都支持所有存储空间。

要使用该功能：

- 声明处于相应存储空间中的对象
- 使用 __eds__ 限定符限定对象

例如：

```
__eds__ int var_a __attribute__((space(prog)));
__eds__ int var_b [10] __attribute__((eds));
__eds__ int *var_c;
__eds__ int *__eds__ *var_d __attribute__((space(psv)));
```

var_a——在闪存中声明一个自动访问的 int

var_b——声明一个位于 eds 中的 int 数组；将会自动访问数组的元素

var_c——声明一个指向 int 的指针，其目标可能存在于扩展数据空间指针支持的任意一个存储空间，并在解引用时自动访问；指针本身必须处于正常数据空间中

var_d——声明一个指向 int 的指针，其目标可能存在于扩展数据空间指针支持的任意一个存储空间，并在解引用时自动访问；指针值存在于闪存中，并自动访问。

编译器会自动为标量变量声明维护 page 属性；这使编译器可以在访问较大数据类型时生成更高效的代码。请记住，标量变量不包含结构或数组。要强制结构或数组分页，请手动使用 page 属性，编译器会阻止对象跨越页边界。

对于对 __eds__ 限定变量的读访问，编译器将会自动适当地处理 PSVPAG 或 DSRPAG 寄存器。对于支持扩展数据空间存储器的器件，编译器还会处理 DSWPAG 寄存器。

注： 一些器件使用 DSRPAG 来表示对闪存或扩展数据空间（EDS）的扩展读访问。

10.8 dataflash 访问

语言工具无法像生成对其他片内存储器的访问一样生成对这种存储器的访问。关于建议的访问程序，请参见器件数据手册或系列参考手册。

编译器定义了以下功能，用于帮助定义和使用一些器件上提供的 dataflash。

1. 该工具定义了一个新的存储空间 space(dataflash)，它可作为属性应用于任何变量。
2. 该工具提供了一个新的内置函数，用于确定正确加上属性的 dataflash 变量的地址：

```
int foo[5] __attribute__((space(dataflash))) = { 1,2,3,4,5 };
offset = __builtin_dataflashoffset(&foo);
```

10.9 双分区存储器访问

语言工具链支持一个新的选项，该选项指示编译器和链接器选择双分区器件中的单个分区作为目标。该选项会将输出文本约束为包含在一个分区中，可从 MPLAB X IDE 中或从命令行使用 `--partition` *n* 选择。

10.10 压缩存储在闪存中的数据

16 位内核系列使用 24 位闪存字大小。该架构支持将闪存区域映射到数据空间中，如**第 10.4 节 “程序空间中的变量”**所述。不幸的是，这种映射只有 16 位宽，以适应数据空间大小。

编译器支持通过压缩存储来使用闪存的最高字节。使用这种最高字节可以对大型结构实现代码大小节约，但进行访问的开销会更高。添加到声明中的类型限定符 __pack_upper_byte 指示应将该变量放入闪存并使用最高字节。不同于用于 MPLAB XC16 C 编译器的其他限定符（如 __psv__），该限定符同时结合了定位和访问控制。

10.10.1 压缩示例

```
__pack_upper_byte char message[] = "Hello World!\n";
```

会将消息字符串分配到闪存中，使存储器的最高字节包含有效数据。

对于 __pack_upper_byte 数据的类型没有任何限制。编译器将如同指定了 __attribute__((packed)) 一样 “压缩” 结构。这可以进一步消除由于填充而浪费的空间。

与其他扩展类型限定符一样， __pack_upper_byte 类型限定符会强制编译器使用一个独特的寻址空间；因此，在将值作为参数传递时维护该限定符是很重要的。不要试图抛弃 __pack_upper_byte 限定符——这行不通。

© 2015 Microchip Technology Inc.

10.10.2 使用注意事项

使用该限定符时，需要考虑以下方面：

1. `__pack_upper_byte` 数据最好用于不需要频繁访问或访问时序不重要的大型数据集。
2. 顺序访问 `__pack_upper_byte` 数据对象可以提高访问性能。
3. `libpic30.h` 中定义了 mempcy 的一个版本，它的原型为：
```
void _memcpy_packed(void *dst, __pack_upper_byte void *src,
                    unsigned int len);
```
4. 以下声明样式对于压缩的存储器是无效的：
`__pack_upper_byte char *message = "Hello World!\n";` 此处，*`message`* 是一个指向 `__pack_upper_byte` 空间的指针，但字符串 “Hello World!\n” 处于正常常量数据空间，与 `__pack_upper_byte` 不兼容。为字面字符串指定不同的源地址空间并没有标准的 C 方式。改为将 *`message`* 声明为对象 （如**第 10.10.1 节 “压缩示例”**中的数组声明）。
5. 在访问压缩变量期间， TBLPAG SFR 可能会被破坏。

10.10.3 寻址信息

闪存的最高字节不具有唯一地址，而这是 C 语言的一个要求。因此，编译器必须构造一个。工具链会将闪存重新映射到线性地址，使所有字节从程序地址字 0 开始。这意味着 `__pack_upper_byte` 变量的真正闪存地址不是存储在指针或符号表中的地址。闪存地址可以通过以下方式确定：

1. 字偏移量 = 地址除以 3
2. 程序地址偏移量 = 字偏移量 * 2
3. 字节偏移量 = 地址对 3 求模

要引用的字节位于闪存中的*程序地址偏移量*处。

`__pack_upper_byte` 对象的重映射寻址方案会阻止编译器接受固定地址请求。

10.11 变量到寄存器的分配

> **注：** 使用在编译器分配的寄存器（固定寄存器）中指定的变量通常是不必要的，有时也是危险的。该功能已被弃用，建议不要使用。

您可以为特定 C 变量指定固定寄存器分配。建议不要这样做。

DD 10.12 EEPROM 中的变量

编译器提供了一些方便的宏定义，用于将数据定位到器件 EE 数据区域。这可以非常简单地实现：

```
int _EEDATA(2) user_data[] =  { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

user_data 将被放入 EE 数据空间（space(eedata)），保留 10 个字并具有给定的初始值。

器件提供了两种方式来让编程人员访问该存储区域。第一种是通过程序空间可视性窗口。第二种是使用特殊的机器指令（TBLRDx）。

10.12.1 通过用户管理的 PSV 访问 EEDATA

编译器通常通过管理 PSV 窗口来访问存储在程序存储器中的常量。如果不是这种情况，可以使用 PSV 窗口来访问 EEDATA 存储器。

要使用 PSV 窗口：

- 必须将 psv 页寄存器设置为要访问的程序存储器的相应地址。对于 EE 数据，它为 0xFF，但最好是使用 __builtin_psvpage() 函数。
- 在一些器件中，还应通过将 CORCON 寄存器中的 PSV 位置 1 来使能 PSV 窗口。如果该位未置 1，使用 PSV 窗口将总是读为 0x0000。

例 10-1： 通过 PSV 进行 EEDATA 访问

```
#include <xc.h>
int main(void) {
  PSVPAG = __builtin_psvpage(&user_data);
  CORCONbits.PSV = 1;

  /* ... */

  if (user_data[2]) ;/* do something */

  }
```

这些步骤只需执行一次。除非 psv 页发生改变，否则可以通过以正常 C 变量的形式引用它们来读取 EE 数据空间中的变量，如示例中所示。

> **注：** 这种访问模型与编译器管理的 PSV（-mconst-in-code）模型不兼容。您应当小心，以防止冲突。

© 2015 Microchip Technology Inc.

10.12.2 使用 TBLRDx 指令访问 EEDATA

编译器不直接支持TBLRDx指令，但可以通过行内汇编代码或编译器内置函数来使用它们。类似于 PSV 访问，通过 SFR 值和已编码为指令一部分的地址来构造 23 位地址。要访问上一个示例中给出的相同存储器，可以使用以下代码：

要使用 TBLRDx 指令：

- 必须将 TBLPAG 寄存器设置为要访问的程序存储器的相应地址。对于 EE 数据，它为 0x7F，但最好是使用 `__builtin_tblpage()` 函数。
- TBLRDx 指令可以通过 `__asm__` 语句或通过 `__builtin_tblrd` 函数之一访问；关于该指令的信息，请参见 《16 位 MCU 和 DSC 程序员参考手册》（DS70157E_CN）。

例 10-2： 通过表读进行 EEDATA 访问

```
#include <xc.h>
#define eedata_read(src, offset, dest) {  \
  register int eedata_addr;               \
  register int eedata_val;                \
                                          \
  eedata_addr = __builtin_tbloffset(&src)+offset; \
  eedata_val = __builtin_tblrdl(eedata_addr); \
  dest = eedata_val;                      \
  }

char user_data[] __attribute__((space(eedata))) = { /* values */ };

int main(void) {
  int value;

  TBLPAG = __builtin_tblpage(&user_data);

  eedata_read(user_data,2*sizeof(user_data[0]), value);
  if (value) ; /* do something */

  }
```

10.12.3 使用受管访问来访问 EEDATA

在大多数器件上， EE 数据空间是程序地址空间的一部分。因此， EEData 可以使用受管访问限定符 __psv__ 或 __eds__ 之一自动访问。

例 10-3： 使用受管 PSV 访问

```
#include <xc.h>

__eds__ char user_data[] __attribute__((space(eedata))) = { /* values
*/ };

int main(void) {
  int value;

  value = user_data[0];
  if (value) ; /* do something */
}
```

10.12.4 其他信息来源

器件系列参考手册很好地介绍了如何使用闪存程序存储器和 EE 数据存储器。这些手册也提供了关于对程序存储器和 EE 数据存储器进行运行时编程的信息。

编译器随附了许多库程序。关于更多信息请参见《16 位语言工具函数库》（DS51456D_CN）手册。

10.13 动态存储器分配

C 运行时堆是数据存储器中的未初始化区域，用于使用标准 C 库动态存储器管理函数 calloc、malloc 和 realloc 进行动态存储器分配。如果不使用以上任何函数，则不需要分配堆。默认情况下，不会创建堆。

如果希望直接 （通过调用存储器分配函数之一）或间接 （通过使用标准 C 库输入 / 输出函数）使用动态存储器分配，则必须创建堆。使用 --heap 链接器命令行选项在链接器命令行上指定堆大小来创建堆。使用命令行分配 512 字节堆的示例如下：

```
xc16-gcc foo.c -Wl,--heap=512
```

链接器会紧邻在堆栈下面分配堆。

您可以使用标准 C 库输入 / 输出函数来创建打开的文件（fopen）。如果您打开一些文件，则堆大小必须对于同时打开的每个文件包含 40 个字节。如果堆存储器不足，则 open 函数将返回一个错误指示。对于应进行缓冲的每个文件，需要 4 个字节的堆空间。如果没有足够的堆存储器来用作缓冲区，则将以非缓冲模式打开文件。默认缓冲区可以使用 setvbuf 或 setbuf 修改。

© 2015 Microchip Technology Inc.

10.14 存储模型

编译器支持几种存储模型。根据您使用的特定器件和存储器使用类型，可以使用命令行选项来选择适用于应用程序的最佳存储模型。

表 10-1： 存储模型命令行选项

选项	存储器定义	说明
-msmall-data	最高 8 KB 的数据存储器。**这是默认设置。**	允许使用 PIC18 类似指令来访问数据存储器。
-msmall-scalar	最高 8 KB 的数据存储器。**这是默认设置。**	允许使用 PIC18 类似指令来访问数据存储器中的标量。
-mlarge-data	大于 8 KB 的数据存储器。	使用间接寻址进行数据引用。
-msmall-code	最高 32K 字的程序存储器。**这是默认设置。**	函数指针不会经过跳转表。函数调用使用 RCALL 指令。
-mlarge-code	大于 32K 字的程序存储器。	函数指针可能会经过跳转表。函数调用使用 CALL 指令。
-mconst-in-data	常量定位在数据存储器中。	值由启动代码从程序存储器复制。
-mconst-in-code	常量定位在程序存储器中。**这是默认设置。**	值通过程序空间可视性 （PSV）数据窗口访问。
-mconst-in-auxflash	常量位于辅助闪存中。	值通过程序空间可视性窗口访问。

命令行选项全局应用于被编译的模块。各个变量和函数可以声明为 near、far 或处于 eds 中，以更好地控制代码生成。关于设置各个变量或函数属性的信息，请参见**第 8.12 节 “变量属性”**和**第 13.2.1 节 “函数说明符”**。

10.14.1 near 或 far 数据

如果变量分配在 near 数据空间中，编译器通常可以比变量不分配在 near 数据中时生成更好 （更紧凑）的代码。如果应用程序的所有变量可以装入 8 KB 的 near 数据中，则可以在编译每个模块时通过使用默认 `-msmall-data` 命令行选项请求编译器将变量放入其中。如果标量类型 （无数组或结构）消耗的总数据量少于 8 KB，则可以将默认的 `-msmall-scalar` 与 `-mlarge-data` 配合使用。这会请求编译器进行安排，仅将应用程序的标量分配在 near 数据空间。

如果这两个全局选项都不适用，则可用以下替代方法。

1. 可以使用 `-mlarge-data` 或 `-mlarge-scalar` 命令行选项编译应用程序的某些模块。在这种情况下，只有这些模块使用的变量会被分配在 far 数据段中。如果使用这种替代方法，则使用外部定义的变量时必须小心。如果使用这些选项之一编译的模块使用的某个变量是在外部定义的，则定义了该变量的模块也必须使用相同的选项进行编译，或者必须用 far 属性来标记变量声明和定义。
2. 如果使用了命令行选项 `-mlarge-data` 或 `-mlarge-scalar`，可以通过使用 near 属性标记各个变量来将其从 `far` 数据空间中排除。
3. 可以通过使用 far 属性标记各个变量来将它们放入 `far` 数据段，而不是使用命令行选项，命令行选项的作用域为模块范围。

如果应用程序的所有 near 变量无法装入 8K 的 near 数据空间，链接器将产生一条错误消息。

© 2015 Microchip Technology Inc.

第 11 章 操作符和语句

11.1 简介

MPLAB XC16 C 编译器支持所有 ANSI 操作符。其中一些操作符的确切结果是实现定义的，**附录 A“实现定义的行为”**详细介绍了这种行为。以下几节介绍了常被误解的代码操作，以及编译器能够执行的其他操作。

- 内置函数
- 整型提升

11.2 内置函数

内置函数使 C 语言程序员可以访问目前只能通过行内汇编访问但却十分有用、适用于一系列广泛应用程序的汇编器操作符或机器指令。内置函数在 C 源文件中编写，语法类似于函数调用，但它们被编译为直接实现函数的汇编代码，通常不涉及函数调用或库程序。

关于内置函数的更多信息，请参见**附录 G “内置函数”**。

11.3 整型提升

当操作符具有多个操作数时，它们通常必须是完全相同的类型。如果需要，编译器会自动转换操作数，使它们具有相同的类型。类型转换将转换为较长的类型，所以不会丢失任何信息；但是，类型变化可能导致与所预期不同的代码行为。这些构成了标准类型转换。

在这些类型转换之前，一些操作数会无条件地转换为较长的类型，即使操作符的两个操作数具有相同的类型。这种转换称为*整型提升*，它是标准 C 行为的一部分。编译器会根据需要执行整型提升，不存在可以控制或禁止该操作的选项。如果不知道类型已发生变化，一些表达式的结果将不是正常情况下预期的结果。

整型提升指的是将枚举类型、`char`、`short int` 或位域类型的有符号或无符号形式隐式转换为 `signed int` 或 `unsigned int` 类型。如果转换结果可以通过 `signed int` 表示，则它就是目标类型，否则则转换为 `unsigned int`。

假设存在以下示例：

```
unsigned char count, a=0, b=50;
if(a - b < 10)
  count++;
```

`a - b` 的 `unsigned char` 结果为 206（它大于 10），但在执行减法运算之前， `a` 和 `b` 都会通过整型提升转换为 `signed int`。对这些数据类型执行减法运算的结果为 -50（它小于 10），因此会执行 `if()` 语句的主体。

如果减法运算的结果为无符号量，则应用强制类型转换。例如：

```
if((unsigned int)(a - b) < 10)
  count++;
```

在此例中，将使用 unsigned int 类型进行比较，并且将不会执行 if() 的主体。

经常发生的另一个问题是使用按位取反操作符 ~。该操作符会翻转值中的每个位。假设存在以下代码。

```
unsigned char count, c;
c = 0x55;
if( ~c == 0xAA)
  count++;
```

如果 c 包含值 0x55，则通常假定 ~c 会产生 0xAA；但是，其结果为 0xFFAA，所以以上示例中的比较会失败。在某些情况下，编译器可以对于这种问题发出比较不匹配错误。同样，可以使用强制类型转换来改变这种行为。

如上例所述，整型提升的结果就是，运算不是使用 char 类型的操作数执行的，而是使用 int 类型的操作数执行的。但在某些情况下，无论操作数是 char 类型还是 int 类型，运算的结果都是相同的。在这些情况下，编译器不会执行整型提升，以提高代码效率。假设存在以下示例：

```
unsigned char a, b, c;
a = b + c;
```

严格地说，该语句要求将 b 和 c 的值提升为 unsigned int 类型，执行加法运算，将加法运算的结果强制转换为 a 的类型，然后进行赋值。虽然 b 和 c 的提升值的 unsigned int 加法运算结果不同于这些值不进行提升情况下的 unsigned char 加法运算结果，但在 unsigned int 结果转换回 unsigned char 之后，最终结果是相同的。如果 8 位加法运算的效率比 16 位加法运算高，编译器将编码为前者。

如果在以上示例中，a 的类型为 unsigned int，则必须执行整型提升，以符合 ANSI C 标准。

© 2015 Microchip Technology Inc.

第 12 章 寄存器使用

12.1 简介

一些寄存器在 C 运行时环境中具有重要作用。因此，创建涉及这些寄存器的代码需要知道编译器是如何使用它们的。

- 寄存器变量
- 更改寄存器内容

12.2 寄存器变量

寄存器变量使用一个或多个工作寄存器，如表 12-1 所示。

表 12-1： 寄存器约定

变量	工作寄存器
char、 signed char 和 unsigned char	W0-W13 和 W14 （如果不用作帧指针）。
short、 signed short 和 unsigned short	W0-W13 和 W14 （如果不用作帧指针）。
int、 signed int 和 unsigned int	W0-W13 和 W14 （如果不用作帧指针）。
void * （或任意指针）	W0-W13 和 W14 （如果不用作帧指针）。
long、 signed long 和 unsigned long	一对邻接寄存器，其中的第一个是来自集合{W0, W2, W4, W6, W8, W10, W12} 的寄存器。
long long、signed long long 和 unsigned long long	4 个邻接寄存器，其中的第一个是来自集合 {W0, W4, W8} 的寄存器。后续的更高编号寄存器包含后续的更高位。
float	一对邻接寄存器，其中的第一个是来自集合{W0, W2, W4, W6, W8, W10, W12} 的寄存器。
double[1]	一对邻接寄存器，其中的第一个是来自集合{W0, W2, W4, W6, W8, W10, W12} 的寄存器。
long double	4 个邻接寄存器，其中的第一个是来自集合 {W0, W4, W8} 的寄存器。
structure	结构中的每 2 个字节对应 1 个邻接寄存器。
_Fract _Sat _Fract	W0-W13 和 W14 （如果不用作帧指针）。
long _Fract _Sat long _Fract	一对邻接寄存器，其中的第一个是来自集合{W0, W2, W4, W6, W8, W10, W12} 的寄存器。
_Accum _Sat _Accum	3 个邻接寄存器，其中的第一个寄存器来自集合 {W0, W4, W8, W12} ；仅当 W14 不用作帧指针时才包含 W12。

注 1： 如果使用了 -fno-short-double，则 double 等效于 long double。

12.3 更改寄存器内容

编译器基于 C 源代码生成的汇编代码将使用 16 位器件上存在的一些寄存器。最重要的是，编译器会假定只有它生成的代码可更改这些寄存器的内容。因此，如果汇编代码向某个寄存器装入一个值，并且没有后续代码生成需要使用该寄存器，则编译器将假定该寄存器的内容在稍后的输出序列中仍然有效。

编译器管理的特殊寄存器包括：W0-W15、RCOUNT、STATUS（SR）、PSVPAG 和 DSRPAG。如果使能了定点支持，则编译器可能分配 A 和 B，这种情况下编译器可能会调整 CORCON。

永远不能直接通过 C 代码，或通过嵌入到 C 代码中的任何汇编代码更改这些寄存器的状态。以下示例显示了违反这些规则并更改 STATUS 寄存器中的 ZERO 位的 C 语句和行内汇编代码。

```
#include <xc.h>

void badCode(void)
{
 asm (“mov #0, w8”);
 WREG9 = 0;
}
```

编译器无法解释在 C 代码中遇到的行内汇编代码的含义。它也无法将映射到 SFR 的变量与实际寄存器本身关联。使用这两种方法中的任一方法写入 SFR 寄存器都不会将寄存器标记为已更改，可能导致代码失败。

© 2015 Microchip Technology Inc.

第 13 章 函数

13.1 简介

编译器支持 C 代码函数，并可处理汇编代码函数，如以下主题所述：

- 编写函数
- 函数长度限制
- 函数代码的分配
- 更改默认的函数分配
- 内联函数
- 存储模型
- 函数调用约定

13.2 编写函数

以下几节介绍了与函数相关联的实现和特殊特性。

13.2.1 函数说明符

唯一对函数具有作用的说明符为 `static`。

使用 `static` 说明符定义的函数只会影响函数的作用域；即，限制源代码中可以调用函数的位置。静态类型的函数只能从定义了该函数的文件中的代码直接调用。该说明符不会改变对函数进行编码的方式。

13.2.2 函数属性

关键字 __attribute__ 可用于在进行声明时指定特殊的属性。该关键字后跟包含在双括号内的属性说明。函数当前支持以下属性：

- address (addr)
- alias ("target")
- auto_psv 和 no_auto_psv
- boot
- const
- context
- deprecated
- far
- format (archetype, string-index, first-to-check)
- format_arg (string-index)
  tinterrupt [([save(list)] [, irq(irqid)] [, altirq(altirqid)] [, preprologue(asm)])]
- keep
- naked
- near
- no_instrument_function
- noload
- noreturn
- round(mode)
- save(list)
- section ("section-name")
- secure
- shadow
- unsupported("message")
- unused
- user_init
- weak

您还可以用关键字前后加上 __（双下划线）的方式指定属性（例如，__shadow__ 而不是 shadow）。这使您可以在头文件中使用它们，而无需担心可能存在同名宏的问题。

声明中可以指定多个属性，方法是在双括号内通过逗号隔开它们，或者在一个属性声明后紧跟另一个属性声明。

address (*addr*)

address 属性指定函数的绝对地址。

```
void __attribute__ ((address(0x100))) foo() {
...
}
```

或者，您也可以在函数原型中定义地址：

```
void foo() __attribute__ ((address(0x100)));
```


© 2015 Microchip Technology Inc.

alias ("*target*")

alias 属性会导致声明被忽略，视为另一个符号的别名，而另一个符号是必须指定的。使用该属性会产生对目标的外部引用，目标必须在链接阶段解析。

auto_psv 和 no_auto_psv

auto_psv 属性与 interrupt 属性配合使用时，会导致编译器在函数序言（prologue）中生成额外的代码来将 psv 页 SFR 设置为正确的值，以便访问 space(auto_psv)（或 constants-in-code 存储模型中的常量）变量。这种情况下请使用该选项：使用 24 位指针，在 psv 页被修改时可能发生中断，而中断程序或它调用的函数使用了 auto_psv 变量。可将它与 no_auto_psv 进行比较。

当 no_auto_psv 属性与 interrupt 属性配合使用时，将导致编译器不生成额外的代码来访问 space(auto_psv)（或 constants-in-code 存储模型中的常量）变量。如果 auto_psv 情况下的所有条件都不为真，则使用该选项。

如果对于中断程序既不指定 auto_psv 也不指定 no_auto_psv 选项，编译器将发出警告并采用 auto_psv。

boot

该属性指示编译器将函数分配在程序闪存的引导段中。

例如，要声明一个受保护的函数：

```
void __attribute__((boot)) func();
```

可以使用一个可选参数来指定引导段内的受保护访问入口点。该参数可以是范围 0 至 31（16 除外）的字面整型，或者 unused 一词。整型参数对应于段快速操作存储区中的 32 个指令槽隙，它占据每个安全段的最低地址范围。值 16 被排除在外是因为入口 16 保留供安全段中断向量使用。值 unused 用于指定用于快速操作存储区中所有未用槽隙的函数。

访问入口点可方便创建来自不同供应商的应用程序段，在运行时合并在一起。可以对外部函数以及局部定义的函数指定它们。

例如：

```
/* an external function that we wish to call */
extern void __attribute__((boot(3))) boot_service3();
/* local function callable from other segments */
void __attribute__((secure(4))) secure_service4()
{
  boot_service3();
}
```

> **注：** 为了分配带有 boot 或 secure 属性的函数，必须保留用于引导和 / 或安全段的存储器。这可以通过在源代码中设置配置字，或通过指定链接器命令选项来实现。关于更多信息，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）的第 8.8 节 “指定 CodeGuard™ 安全功能选项”。
>
> 如果使用了属性 boot 或 secure，并且未保留存储器，则会产生链接错误。

要指定安全中断处理程序，请将 boot 属性与 interrupt 属性配合使用：

```
void __attribute__((boot,interrupt)) boot_interrupts();
```

为某个外部安全函数指定访问入口点时，成功链接并不需要在项目中包含该函数。对该函数的所有引用都会被解析为闪存中的一个固定位置，具体取决于在链接时选择的安全模型。

当为一个局部定义的函数指定了访问入口点时，链接器会在安全段快速操作存储区中插入一条转移指令。快速操作存储区入口 16 属于例外，它以向量（即，指令地址）而不是指令的形式表示。实际的函数定义会被定位到快速操作存储区之外；因此快速操作存储区将包含一个跳转表，通过它可以将控制从安全段转移到具有所定义入口点的函数。

自动变量由包含它的函数所有，不需要 boot 属性。它们可以赋予初始值，如下所示：

```
void __attribute__((boot)) chuck_cookies()
{
  int hurl;
  int them = 55;
  char *where = "far";
  splat(where);
  /* ... */
}
```

请注意，where 的初始值基于一个分配在 PSV 常量段 .boot_const 中的字符串字面值。编译器会在进入函数时将 psv 页 SFR 设置为正确的值。如果需要，编译器也会在对 splat() 的调用后恢复它。

const

许多函数不检查除其参数之外的任何值，除返回值之外没有任何其他影响。这种函数可能会被执行公共子表达式消除和循环优化，就如算术运算符一样。这些函数应使用属性 const 声明。例如：

```
int square (int) __attribute__ ((const int));
```

指示调用假想函数 square 的次数少于程序指示的次数是安全的。

请注意，具有指针参数并会检查所指向数据的函数*不得*使用 const 声明。同样地，调用非 const 函数的函数通常不得为 const 类型。const 函数具有 void 返回类型是没有意义的。

© 2015 Microchip Technology Inc.

context

context 属性可用于将当前程序与备用寄存器集相关联。它通常用于中断服务程序，用以减少必须保护的现场量，这可以改善中断响应延时。

deprecated

关于 deprecated 属性的信息，请参见**第 8.12 节 “变量属性”**。

far

far 属性向编译器指示函数可能被定位到太远位置而无法使用短调用指令。

format (*archetype*, *string-index*, *first-to-check*)

format 属性指定函数接受 printf、scanf 或 strftime 样式的参数，应根据格式字符串对这些参数进行类型检查。例如，假设具有以下声明：

```
extern int
my_printf (void *my_object, const char *my_format, ...)
     __attribute__ ((format (printf, 2, 3)));
```

这会导致编译器检查对 my_printf 的调用中的参数，确定是否与 printf 样式的格式字符串参数 my_format 一致。

参数 *archetype* 决定如何解释格式字符串，应为 printf、 scanf 或 strftime 之一。参数 *string-index* 指定哪个参数是格式字符串参数 （参数从左到右进行编号，从 1 开始），而 *first-to-check* 是要根据格式字符串检查的第一个参数的编号。对于参数不可用于检查的函数 （如 vprintf），请将第三个参数指定为零。在这种情况下，编译器仅检查格式字符串的一致性。

在前面的示例中，格式字符串 （my_format）是函数 my_print 的第二个参数，要检查的参数从第三个参数开始，所以格式属性的正确参数为 2 和 3。

format 属性使您可以标识自己的接受格式字符串作为参数的函数，使编译器可以检查对这些函数的调用是否存在错误。每当请求产生这种警告（使用 -Wformat）时，编译器总是会检查 ANSI 库函数 printf、 fprintf、 sprintf、 scanf、 fscanf、 sscanf、strftime、vprintf、vfprintf 和 vsprintf 的格式，所以不需要修改头文件 stdio.h。

format_arg (*string-index*)

format_arg 属性指定一个函数接受 printf 或 scanf 样式的参数，对它进行修改（例如，将它译成另一种语言），然后将它传递给 printf 或 scanf 样式的函数。例如，假设具有以下声明：

```
extern char *
my_dgettext (char *my_domain, const char *my_format)
      __attribute__ ((format_arg (2)));
```

这会导致编译器检查对 my_dgettext 的调用中的参数 （调用结果传递给 printf、scanf 或 strftime 类型的函数），确定参数是否与 printf 样式的格式字符串参数 my_format 一致。

参数 *string-index* 指定哪个参数是格式字符串参数 （从 1 开始）。

format-arg属性使您可以标识自己的可修改格式字符串的函数，从而使编译器可以检查对 printf、 scanf 或 strftime 函数的调用，这些函数的操作数是一个对您自己的函数的调用。

interrupt [([save(*list*)] [, irq(*irqid*)]
[, altirq(*altirqid*)] [, preprologue(*asm*)])]

使用该选项可以指示指定函数是一个中断处理程序。存在该属性时，编译器会生成适合在中断处理程序中使用的函数序言和结语序列。可选参数 save 指定分别要在函数序言和结语中保存和恢复的变量的列表。可选参数 irq 和 altirq 指定要使用的中断向量表 ID。可选参数 preprologue 指定要在编译器生成的序言代码之前发出的汇编代码。关于完整说明 （包括示例），请参见**第 14 章 “中断”**。

使用 interrupt 属性时，请指定 auto_psv 或 no_auto_psv。如果未指定，将会产生警告并采用 auto_psv。

keep

keep 属性将阻止链接器通过 ELF 链接器选项 --gc-sections 删除函数，即使它未被使用。

```
void __attribute__((keep)) func();
```

naked

naked 属性将阻止编译器保存或恢复任何寄存器。该属性应慎用，因为未保存或恢复寄存器可能会导致问题。出于安全性，可以考虑将该属性与 noreturn 配合使用——任何返回尝试都会导致复位。

```
void __attribute__((naked)) func();
```

near

near 属性指示编译器可以使用效率更高的形式的调用指令来调用函数。

no_instrument_function

如果给出了命令行选项 -finstrument-function，则会在大多数用户编译的函数的入口和出口生成性能分析函数调用。带有该属性的函数不会被如此处理。

noload

noload 属性指示应为函数分配空间，但不应将实际代码装入存储器。如果应用程序设计为在运行时将某个函数装入存储器 （例如从串行 EEPROM），该属性会很有用。

```
void bar() __attribute__ ((noload)) {
...
}
```


© 2015 Microchip Technology Inc.

noreturn

一些标准库函数 （如 abort 和 exit）不能返回。编译器自动知道这一点。一些程序定义了自己从不返回的函数。您可以将它们声明为 noreturn，将这一点告知给编译器。例如：

```
void fatal (int i) __attribute__ ((noreturn));

void
fatal (int i)
{
  /* Print error message. */
  exit(1);
}
```

noreturn关键字指示编译器假定fatal不能返回。然后，它可以在不考虑如果fatal真的返回时会如何的情况下进行优化。这可以获得略微较好的代码。此外，它也有助于避免关于未初始化变量的虚假警告。

noreturn 函数具有 void 之外的返回类型是没有意义的。

如果 noreturn 函数试图返回，它将发生复位。

round(mode)

round 属性用于控制函数内的 C 语言定点支持 （_Fract 和 _Accum 变量类型）方言代码 （-menable-fixed 命令行选项）的舍入模式。将 mode 指定为截断、常规或收敛之一。对于带有该属性的函数中的 C 语言代码，该属性会改写由 -menable-fixed 设置的默认舍入模式，但不影响可能由该函数调用的函数。

save(list)

使用save(list)属性声明的函数将指示编译器保存和恢复在list中表示的C变量。

section (*"section-name"*)

通常情况下，编译器会将它生成的代码放入 .text 段。但有时您需要额外的段，或者您需要某些函数出现在特殊的段中。 section 属性指定一个函数处于某个特定段中。例如，假设具有以下声明：

```
extern void foobar (void) __attribute__ ((section (".libtext")));
```

这会将函数 foobar 放入 .libtext 段。

链接器将按顺序分配保存的命名段。这可能使您可以确保代码是可局部相互引用的，即使是跨模块。这可以确保调用相互之间足够近，从而可以使用效率更高的调用指令。

secure

该属性指示编译器将函数分配在程序闪存的安全段中。

例如，要声明一个受保护的函数：

```
void __attribute__((secure)) func();
```

可以使用一个可选参数来指定安全段内的受保护访问入口点。该参数可以是范围 0 至 31（16 除外）的字面整型，或者 unused 一词。整型参数对应于段快速操作存储区中的 32 个指令槽隙，它占据每个安全段的最低地址范围。值 16 被排除在外是因为入口 16 保留供安全段中断向量使用。值 unused 用于指定用于快速操作存储区中所有未用槽隙的函数。

访问入口点可方便创建来自不同供应商的应用程序段，在运行时合并在一起。可以对外部函数以及局部定义的函数指定它们。例如：

```
/* an external function that we wish to call */
extern void __attribute__((boot(3))) boot_service3();
/* local function callable from other segments */
void __attribute__((secure(4))) secure_service4()
{
  boot_service3();
}
```

注： 为了分配带有 boot 或 secure 属性的函数，必须保留用于引导和 / 或安全段的存储器。这可以通过在源代码中设置配置字，或通过指定链接器命令选项来实现。关于更多信息，请参见 《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）的第 8.8 节 “指定 CodeGuard™ 安全功能选项”。

如果使用了属性 boot 或 secure，并且未保留存储器，则会产生链接错误。

要指定安全中断处理程序，请将 secure 属性与 interrupt 属性配合使用：

```
void __attribute__((secure,interrupt)) secure_interrupts();
```

为某个外部安全函数指定访问入口点时，成功链接并不需要在项目中包含该函数。对该函数的所有引用都会被解析为闪存中的一个固定位置，具体取决于在链接时选择的安全模型。

当为一个局部定义的函数指定了访问入口点时，链接器会在安全段快速操作存储区中插入一条转移指令。快速操作存储区入口 16 属于例外，它以向量（即，指令地址）而不是指令的形式表示。实际的函数定义会被定位到快速操作存储区之外；因此快速操作存储区将包含一个跳转表，通过它可以将控制从安全段转移到具有所定义入口点的函数。

© 2015 Microchip Technology Inc.

自动变量由包含它的函数所有，不需要 secure 属性。它们可以赋予初始值，如下所示：

```
void __attribute__((secure)) chuck_cookies()
{
  int hurl;
  int them = 55;
  char *where = "far";
  splat(where);
  /* ... */
}
```

请注意，where 的初始值基于一个分配在 PSV 常量段 .secure_const 中的字符串字面值。编译器会在进入函数时将 PSVPAG 设置为正确的值。如果需要，编译器也会在对 splat() 的调用后恢复 PSVPAG。

shadow

shadow 属性会导致编译器使用影子寄存器而不是软件堆栈来保存寄存器。该属性通常与 interrupt 属性配合使用。

```
void __attribute__ ((interrupt, shadow)) _T1Interrupt (void);
```

unsupported(”message”)

当使用对象时，该属性将显示一条定制消息。

```
int foo __attribute__((unsupported(“This object is unsupported”));
```

对 foo 进行访问会产生一条警告消息。

unused

该属性附加到一个函数上，表示该函数的本意就是可能不会被使用。对于该函数，编译器不会产生未用函数警告。

user_init

user_init 属性可以应用于任何具有 void 参数和返回类型的非中断函数。应用该属性会导致默认 C 启动模块在执行用户主函数之前调用该函数。并不保证函数的运行顺序，所以这些函数不能依赖于先前已运行的其他 user_init 函数；这些函数将在 PSV 和数据初始化之后调用。 user_init 仍可能会被执行中的程序调用。例如：

```
void __attribute__((user_init)) initialize_me(void) {
   // perform initalization sequence alpha alpha beta
}
```

weak

关于 weak 属性的信息，请参见**第 8.12 节 “变量属性”**。

13.3 函数长度限制

对于所有器件，为函数生成的代码的长度可能会大于一页大小，这仅受限于可用程序存储器。但是，产生的代码长度大于一页的函数的效率可能不够高，因为需要较长的调用序列来跳转到和调用处于其他页中的目标。更多详细信息，请参见**第 13.4 节 “函数代码的分配”**。

13.4 函数代码的分配

与函数相关联的代码总是放置在目标器件的程序存储器中。如**第 10.2 节 “地址空间”**所述，编译器根据使用的存储模型和数据是否初始化来安排要放入 .text 段中的代码。当模块在链接时进行合并时，链接器将基于各个段的属性确定它们的起始地址。

13.5 更改默认的函数分配

可能会出现特定函数必须定位到特定地址处，或处于某个地址范围内的情况。实现这一点的最简单方法是使用 address 属性，如**第 13.2.1 节 “函数说明符”**所述。例如，要将函数 PrintString 定位在程序存储器中的地址 0x8000 处：

```
int __attribute__ ((address(0x8000))) PrintString (const char *s);
```

定位代码的另一种方式是将函数放入一个用户定义的段，并在定制链接描述文件中指定该段的起始地址。其实现方式如下：

1. 在 C 源代码中修改代码声明，指定一个用户定义的段。
2. 将这个用户定义的段添加到定制链接描述文件中，指定该段的起始地址。

例如，要将函数 PrintString 定位到程序存储器中的地址 0x8000 处，需要先在 C 源代码中如下声明函数：

```
int __attribute__((__section__(".myTextSection")))
PrintString(const char *s);
```

section 属性指定该函数应放入名为 .myTextSection 的段，而不是默认的 .text 段。它并不指定将用户定义的段定位到何处。这必须在定制链接描述文件中完成，如下所示。使用特定于器件的链接描述文件作为基础，添加以下段定义：

```
.myTextSection 0x8000 :
  {
        *(.myTextSection);
  } >program
```

该定义指定输出文件应包含名为 .myTextSection、从地址 0x8000 开始的段，并包含名为 .myTextSection 的所有输入段。该示例中，由于该段中存在单个函数 PrintString，所以该函数会被定位到程序存储器中的地址 0x8000 处。

© 2015 Microchip Technology Inc.

13.6 内联函数

通过将函数声明为内联，可以指示编译器将该函数的代码合并到其调用方的代码中。这可以消除函数调用开销，通常可以使执行速度变快。此外，如果任何实际参数值为常量，其已知值可能允许在编译时进行简化，从而不需要包含内联函数的所有代码。对代码长度的影响较难预测。使用内联函数时的机器代码可能变长也可能变短，这取决于具体情况。

> **注：** 仅当函数的定义在调用位置可见时（不仅仅是原型）才会发生函数内联。要将某个函数内联到多个源文件中，可以将函数定义放入每个源文件包含的头文件中。

要将函数声明为内联，请在其声明中使用 `inline` 关键字，如下：

```
inline int
inc (int *a)
{
  (*a)++;
}
```

（如果使用了 `-traditional` 选项或 `-ansi` 选项，则写为 `__inline__` 而不是 `inline`。）此外，还可以通过命令行选项 `-finline-functions` 将所有“足够简单”的函数设为内联。编译器会基于函数长度的估计值，试探性地决定哪些函数足够简单，值得以这种方式合并。

> **注：** 只有使用 `-finline` 选项或使能优化时，才会识别 `inline` 关键字。

函数定义中的某些用法可能会使它不适合于内联替换。这些用法包括：使用 `varargs`、使用 `alloca`、使用长度可变的数据、使用计算 `goto`，以及使用非局部 `goto`。使用命令行选项 `-Winline` 将在无法替换标记为 `inline` 的函数时产生警告，并给出失败的原因。

在编译器语法中， `inline` 关键字不会影响函数的链接。

当某个函数同时为 `inline` 和 `static` 时，如果对该函数的所有调用都被合并到调用方中，并且从不使用该函数的地址，则永远不会引用该函数自己的汇编代码。这种情况下，编译器不会实际输出该函数的汇编代码，除非您指定了命令行选项 `-fkeep-inline-functions`。一些调用会由于各种原因而无法进行合并（特别是，无法合并函数的定义之前的调用，也无法合并定义中的递归调用）。如果存在无法合并的调用，则该函数将像正常情况下一样编译为汇编代码。如果程序引用了该函数的地址，则该函数也必须像正常情况下一样进行编译，因为无法对这种情况进行内联。只有函数声明为 `static`，并且函数定义在所有函数使用之前，编译器才会消除内联函数。

当某个 `inline` 函数不为 `static` 时，则编译器必须假定可能存在来自其他源文件的调用。由于全局符号只能在所有程序中定义一次，所以绝对不能在其他源文件中定义该函数，导致无法合并其中的调用。因此，非 `static` 内联函数总是以通常的方式独立地编译。

如果在函数定义中同时指定了 inline 和 extern，则该定义仅用于内联。在任何情况下不会独立编译该函数，即使显式地引用了它的地址。此类地址会变为外部引用，如同仅声明了函数但未定义它。

inline 和 extern 的这种组合具有类似于宏的效果。将某个函数定义与这些关键字一起放入头文件，并将该定义（缺少 inline 和 extern）的另一个副本放入库文件。头文件中的定义将导致对函数的大多数调用被内联。如果仍然还有对该函数的任何调用，它们将引用库中的单个副本。

内联通常是一项建议，可能会被忽略。

13.7 存储模型

编译器支持几种存储模型。根据您使用的特定 dsPIC DSC 器件和存储器使用类型，可以使用命令行选项来选择适用于应用程序的最佳存储模型。

表 13-1： 存储模型命令行选项

选项	存储器定义	说明
-msmall-data	最高 8 KB 的数据存储器。**这是默认设置。**	允许使用 PIC18 类似指令来访问数据存储器。
-msmall-scalar	最高 8 KB 的数据存储器。**这是默认设置。**	允许使用 PIC18 类似指令来访问数据存储器中的标量。
-mlarge-data	大于 8 KB 的数据存储器。	使用间接寻址进行数据引用。
-msmall-code	最高 32K 字的程序存储器。**这是默认设置。**	函数指针不会经过跳转表。函数调用使用 RCALL 指令。
-mlarge-code	大于 32K 字的程序存储器。	函数指针可能会经过跳转表。函数调用使用 CALL 指令。
-mconst-in-data	常量定位在数据存储器中。	值由启动代码从程序存储器复制。
-mconst-in-code	常量定位在程序存储器中。**这是默认设置。**	值通过程序空间可视性 （PSV）数据窗口访问。
-mconst-in-auxflash	常量位于辅助闪存中	值通过程序空间可视性窗口访问。

命令行选项全局应用于被编译的模块。各个变量和函数可以声明为 near、far 或 eds，以更好地控制代码生成。关于设置各个变量或函数属性的信息，请参见**第 8.12 节 “变量属性”**和**第 13.2.1 节 “函数说明符”**。

© 2015 Microchip Technology Inc.

13.7.1 near 或 far 代码

相近的函数（处于彼此的 32 K 字半径内）可以比不相近的函数更高效地互相调用。如果已知应用程序中的所有函数都是相近的，则可以在编译每个模块时使用默认的 `-msmall-code` 命令行选项，指示编译器使用效率更高形式的函数调用。

如果该默认选项不适用，则可用以下替代方法：

1. 可以使用 `-msmall-code` 命令行选项编译应用程序的某些模块。在这种情况下，只有这些模块中的函数调用会使用效率更高的形式的函数调用。
2. 如果使用了 `-msmall-code` 命令行选项，则可以通过将各个函数标上 `far` 属性来指示编译器对于各个函数使用长形式的函数调用。
3. 可以通过将函数的声明和定义标上 `near` 属性，指示编译器使用效率更高形式的函数调用来调用各个函数，而不是使用命令行选项，命令行选项的作用域为函数范围。
4. 使用命名段将可局部引用的代码组合在一起，或使该代码保持在公共翻译单元中。

如果调用方之一无法使用效率更高形式的函数调用到达声明为 `near` 的函数，链接器将产生一条错误消息。

13.8 函数调用约定

在调用函数时：

- 寄存器 W0-W7 由调用方保存。如果随后从函数调用返回之后需要这些值，则调用函数必须在函数调用之前保存这些值。堆栈是保留这些值的适合位置。
- 寄存器 W8-W14 由被调用方保存。被调用函数必须保存这些寄存器中它会修改的任何寄存器。
- 寄存器 W0-W4 用于函数返回值。
- 寄存器 W0-W7 用于参数传递。
- 如果使用了 `-mconst-in-code`（`auto_psv`）存储模型，则应保存 DBRPAG/PSVPAG。

表 13-2： 需要的寄存器

数据类型	需要的工作寄存器数
`char`	1
`int`	1
`short`	1
`pointer`	1（eds 指针需要 2 个）
`long`	2（邻接——对齐到偶编号寄存器）
`float`	2（邻接——对齐到偶编号寄存器）
`double*`	2（邻接——对齐到偶编号寄存器）
`long double`	4（邻接——对齐到编号为 4 的倍数的寄存器）
`structure`	结构中每 2 个字节 1 个寄存器
`_Fract`	1
`long _Fract`	2（邻接——对齐到偶编号寄存器）
`_Accum`	3（邻接——对齐到编号为 4 的倍数的寄存器）

* 如果使用了 `-fno-short-double`，则 `double` 等效于 `long double`。

参数放在可用的第一个对齐的邻接寄存器。调用函数必须保存参数（如需要）。结构没有任何对齐限制；如果有足够的寄存器来存放整个结构，结构参数将占用寄存器。函数结果存储在连续的寄存器中，从 W0 开始。

13.8.1 函数参数

前 8 个工作寄存器（W0-W7）用于存放函数参数。参数按从左到右的顺序分配到寄存器中，并且参数分配到第一个适当对齐的可用寄存器。

在以下示例中，所有参数均通过寄存器中传递，虽然不是按它们在声明中出现的顺序。这种格式使编译器可以最高效地使用可用的参数寄存器。

© 2015 Microchip Technology Inc.

例 13-1：　　函数调用模型

```
void
params0(short p0, long p1, int p2, char p3, float p4, void *p5)
{
        /*
        ** W0           p0
        ** W1           p2
        ** W3:W2        p1
        ** W4           p3
        ** W5           p5
        ** W7:W6        p4
        */
        ...
}
```

下一个示例说明了如何将结构传递给函数。如果整个结构可以装入可用的寄存器中，则通过寄存器传递结构；否则，结构参数会被放在堆栈中。

例 13-2：　　函数调用模型：传递结构

```
typedef struct bar {
  int i;
  double d;
} bar;

void
params1(int i, bar b) {
        /*
        ** W0          i
        ** W1          b.i
        ** W5:W2       b.d
        */

}
```

对应于可变长度参数列表的省略号（...）的参数不会被分配到寄存器中。未分配到寄存器的所有参数都会被压入堆栈，按从右到左的顺序。

在下一个示例中，结构参数由于长度太大而无法放入寄存器中。但是，这并不会阻止下一个参数使用寄存器。

例 13-3： 函数调用模型：基于堆栈的参数

```
typedef struct bar {
  double d,e;
} bar;

void
params2(int i, bar b, int j) {
        /*
        ** W0          i
        ** stack        b
        ** W1           j
        */
}
```

访问被放在堆栈中的参数取决于是否创建了帧指针。通常，编译器会产生帧指针（除非被指示不要这样做），并通过帧指针寄存器（W14）访问基于堆栈的参数。在前面的示例中，将从 W14-22 访问 b。帧指针偏移量为负 22，计算它的方式是 （见图 10-4）：减去用于上一个 FP 的 2 个字节，用于返回地址的 4 个字节，之后是用于 b 的 16 个字节。

不使用帧指针时，汇编编程人员必须知道自进入过程以来使用了多少堆栈空间。如果未使用更多的堆栈空间，则计算方式类似于例 13-3。b 将通过 W15-20 访问； 4 个字节用于返回地址，再过 16 个字节即可访问 b 的起始位置。

13.8.2 返回值

对于 8 位或 16 位标量，函数返回值在 W0 中返回；对于 32 位标量为 W1:W0 ；对于 64 位标量为 W3:W2:W1:W0。聚合通过 W0 间接返回，它由函数调用方设置为包含聚合值的地址。

13.8.3 在函数调用之间保留寄存器

编译器会安排在普通函数调用之间保留寄存器 W8-W15。寄存器 W0-W7 可用作临时寄存器。对于中断函数，编译器会安排保存所有必需的寄存器，即 W0-W15 和 RCOUNT。

© 2015 Microchip Technology Inc.

第 14 章 中断

14.1 简介

中断处理对于大多数单片机应用来说都是很重要的一个方面。中断用来使软件操作与实时发生的事件同步。当发生中断时，软件的正常执行流程被打断，调用专门的函数来处理事件。当中断处理结束时，恢复先前的现场信息并继续正常执行流程。

本章对中断处理进行概述。本文档中讨论了以下几个方面：

- **中断操作**——概述中断如何工作。
- **编写中断服务程序**——您可以将一个或多个 C 函数指定为在发生中断时调用的中断服务程序 （ISR）。通常为了获得最佳性能，将冗长的计算或需要库调用的操作放在主应用程序中。这种策略可以优化性能，并最大程度降低在快速发生中断事件时丢失信息的可能性。
- **指定中断向量**——16 位器件使用中断向量来在发生中断时转移应用程序控制权。中断向量是程序存储器中指定 ISR 地址的专用位置。要使用中断，应用程序必须在这些位置包含有效的函数地址。
- **中断服务程序现场保存**——为了在与中断之前的相同条件状态下处理从中断返回到代码，必须保存特定寄存器的现场信息。
- **嵌套中断**——调用中断时和执行第一条 ISR 指令时之间的时间为中断响应延时。
- **允许 / 禁止中断**——如何确定中断优先级。通过两种方式允许和禁止中断源：全局允许和单独允许。
- **ISR 注意事项**——与主干代码共享存储器、 PSV 使用与 ISR，以及在 ISR 内调用函数。

14.2 中断操作

该编译器包含了允许完全从 C 代码中处理中断的特性。中断函数通常称为 ISR。

16 位器件允许从许多中断源产生中断。大多数中断源具有它们自己的专用中断向量，整理在中断向量表 （IVT）中。每个向量包含一个地址，在该地址处可找到中断服务程序的入口点。一些中断表向量单元用于陷阱，这些陷阱是处理器件错误操作 （如地址错误）的不可屏蔽中断。

在一些器件上，提供了备用中断向量表（AIVT），它允许指定独立的中断向量。需要时可以使能该表，从而强制从 AIVT 而不是 IVT 读取 ISR 地址。

中断具有与其关联的优先级。可以对每个中断源独立地调整它。当多个具有相同优先级的中断同时在等待处理时，每个中断源的固有优先级或自然顺序优先级将发生作用。自然顺序优先级通常与 IVT 中的中断向量顺序相同。

编译器为 C 或行内汇编代码中的中断处理提供了完全支持。

中断代码指的是由于发生中断而执行的任何代码。中断代码在执行相应的中断返回指令时执行完毕。

这与主干代码形成对比，后者是一个独立的应用程序，通常是在复位后执行的程序的主要部分。

© 2015 Microchip Technology Inc.

14.3 编写中断服务程序

遵循本节介绍的准则，仅使用 C 语法结构就可以编写所有应用程序代码，包括中断服务程序。

所有 ISR 代码都会被放入名称以 .isr 开头的命名段中。具有 section 属性的函数会在给定名称前加上 .isr。使用 -ffunction-sections 编译的代码也在段名前加上 .isr。详情请参见**第 5.7.6.2 节 “-ffunction-sections 选项”**。

如果您创建了自己的链接描述文件，并且该文件的时间早于 MPLAB C30 v3.30 项目，则需要按照 MPLAB XC16 安装目录的 docs 子目录中的 Readme_XC16.html 文件修改您的链接描述文件。

14.3.1 编写 ISR 的准则

编写 ISR 建议遵循以下准则：

- 声明 ISR 时不使用参数，并使用 void 返回类型 （强制性）
- 不要让主干代码调用 ISR （强制性）
- 不要让 ISR 调用其他函数 （建议）

16 位器件 ISR 与所有其他 C 函数的相似之处是可以具有局部变量和访问全局变量。但是，ISR 需要声明为没有参数和返回值。这是必需的，因为在响应硬件中断或陷阱时将以与主干 C 程序异步的方式调用 ISR （也即，它不是以正常方式调用的，所以参数和返回值不适用）。

ISR 应仅通过硬件中断或陷阱调用，而不是从其他 C 函数调用。ISR 使用从中断返回（RETFIE）指令来从中断函数退出，而不是使用普通的 RETURN 指令。使用 RETFIE 指令退出上下文可能会破坏处理器资源，如状态寄存器。

最后，ISR 应避免调用其他函数。这是建议的，因为存在响应延时问题。更多信息，请参见**第 14.6 节 “嵌套中断”**。

14.3.2 编写 ISR 的语法

要将一个 C 函数声明为中断处理程序，需要使用 interrupt 属性标记函数 （关于 __attribute__ 关键字的说明，请参见**第 13.2.2 节 “函数属性”**）。interrupt 属性的语法为：

```
__attribute__((interrupt [(
                          [ save(symbol-list)]
                          [, irq(irqid)]
                          [, altirq(altirqid)]
                          [, preprologue(asm)]
                          )]
             ))
```

interrupt 属性名称和参数名称可以写为在名称前后加上双下划线字符的形式。因此，interrupt 和 __interrupt__ 是等效的，正如 save 和 __save__。

可选的 save 参数指定要在进入和退出 ISR 时保存和恢复的一个或多个变量的列表。名称的列表写在括号内，用逗号分隔名称。

对于在 ISR 中可能被修改的全局变量，如果不希望值被导出，则应当安排保存这些变量。会被 ISR 访问的全局变量应使用 volatile 进行限定。

可选的 irq 参数使您可以在特定中断处放置中断向量，可选的 altirq 参数使您可以在指定备用中断处放置中断向量。每个参数需要一个带圆括号的中断 ID 编号。（关于中断 ID 的列表，请参见**第 14.4 节 “指定中断向量”**）。

可选的preprologue参数使您可以在紧接编译器生成的函数序言之前向生成代码中插入汇编语言语句。

使用 interrupt 属性时，请指定 auto_psv 或 no_auto_psv。如果未指定，将会产生警告并采用 auto_psv。

14.3.3 编写 ISR 代码

以下原型将函数 isr0 声明为中断处理程序：

```
void __attribute__((interrupt(auto_psv))) isr0(void);
```

正如该原型所示，中断函数必须不带参数也不返回值。编译器会安排保存所有要保护的工作寄存器，如果需要还会保存状态寄存器和重复计数寄存器。其他变量可以通过将它们指定为 interrupt 属性的参数来保存。例如，要让编译器自动保存和恢复变量 var1 和 var2，请使用以下原型：

```
void __attribute__((interrupt(auto_psv, save(var1, var2))))
isr0(void);
```

要请求编译器使用快速现场保存（使用push.s和pop.s指令），请为函数标上shadow属性 （见**第 13.2.1 节 “函数说明符”**）。例如：

```
void __attribute__((interrupt(auto_psv, shadow))) isr0(void);
```

14.3.4 使用宏来声明简单 ISR

如果中断处理程序不需要 interrupt 属性的任何可选参数，则可以使用一种简化的语法。特定于器件的头文件中定义了以下宏：

```
#define _ISR __attribute__((interrupt))
#define _ISRFAST __attribute__((interrupt, shadow))
```

例如，要将某个中断处理程序声明为外部中断 0：

```
#include <xc.h>
void _ISR _INT0Interrupt(void);
```

要将 SPI1 中断的中断处理程序声明为使用快速现场保存：

```
#include <xc.h>
void _ISRFAST _SPI1Interrupt(void);
```


© 2015 Microchip Technology Inc.

14.4 指定中断向量

所有 16 位器件都具有一个主中断向量表。一些 16 位器件具有固定的备用中断向量表，一些则没有备用中断向量表，而一些器件则具有可禁止和移动的备用中断向量表。

> **注：** 器件复位不通过中断向量表来处理。实际上，在器件复位时，程序计数器会被清零。这会导致处理器在地址 0 处开始执行。按照约定，链接描述文件会在该地址构造一条将控制转移给 C 运行时启动模块的 GOTO 指令。

当 INTCON2 寄存器中的 ALTIVT 位置 1 时，将使用备用中断向量名称。对于具有可禁止和移动的备用中断向量表的器件， AIVT 支持通过配置字进行配置，特别是：

- AIVTDIS 用于使能向量表
- BSLIM 用于定位向量表

在这些器件上，工具链会检查这些配置字的值，以确定是否分配空间和填充备用向量表的值。只需为器件指定这些配置字的相应值：

```
#pragma config AIVTDIS = OFF
#pragma config BSLIM = 0x1FFD
```

然后，以正常方式定义备用向量，即：

```
void __attribute__((interrupt)) _AltT1Interrupt(void) {}
```

每个异常向量占用一个程序字。关于按器件系列列出的中断向量表，请参见**第 14.4.2 节“中断向量表”**。

14.4.1 中断向量使用

要设置中断，必须将一个函数的地址放置在向量表中的一个相应地址处，并且该函数必须保护它所使用的任何系统资源。它必须使用 RETFIE 处理器指令返回到前台任务。中断函数可以使用 C 语言编写。将某个 C 函数指定为中断处理程序时，编译器会安排保护编译器使用的所有系统资源，并使用适当的指令从函数返回。编译器可选择安排使用中断函数的地址填充中断向量表。

要安排让编译器填充中断向量，使之指向中断函数，请按照向量表中指示的形式命名函数（**第 14.4.2 节“中断向量表”**）。例如，如果定义了以下函数，将会自动填充堆栈错误向量：

```
void __attribute__((interrupt(auto_psv))) _StackError(void);
```

请注意使用前导下划线。类似地，如果定义了以下函数，将会自动填充备用堆栈错误向量：

```
void __attribute__((interrupt(auto_psv)) _AltStackError(void);
```

同样，请注意使用前导下划线。

对于没有特定处理程序的所有中断向量，将会安装默认的中断处理程序。默认中断处理程序由链接器提供，它只会简单地复位器件。应用程序也可以通过声明名为 _DefaultInterrupt 的中断函数来提供默认中断处理程序。

每个表中的最后 9 个中断向量没有预定义的硬件函数。可以通过使用向量表中指示的名称（**第 14.4.2 节 “中断向量表”**）来填充这些中断的向量，或者也可以使用更适合于应用程序的名称，而通过使用 `interrupt` 属性的 `irq` 或 `altirq` 参数仍然填充相应的向量。例如，要指定一个函数应使用主中断向量 52，请使用以下语句：

```
void __attribute__((interrupt(auto_psv, irq(52)))) MyIRQ(void);
```

类似地，要指定一个函数应使用备用中断向量 53，请使用以下语句：

```
void __attribute__((interrupt(auto_psv, altirq(52)))) MyAltIRQ(void);
```

`irq`/`altirq` 编号可以为中断请求编号 45 至 53 之一。如果使用了 `interrupt` 属性的 `irq` 参数，则编译器会创建外部符号名称 `__Interrupt`*n*，其中的 n 是向量编号。因此，C 标识符 `_Interrupt45` 至 `_Interrupt53` 是由编译器保留的。按相同的方式，如果使用了 `interrupt` 属性的 `altirq` 参数，则编译器会创建外部符号名称 `__AltInterrupt`*n*，其中的 n 是向量编号。因此， C 标识符 `_AltInterrupt45` 至 `_AltInterrupt53` 是由编译器保留的。

14.4.2 中断向量表

关于按器件系列列出的中断向量表：

- 在 MPLAB X IDE 中，对于较新版本的编译器，打开 Dashboard 窗口，并单击 **Compiler Help**（编译器帮助）按钮。
- 在命令行上，请参见 MPLAB XC16 C 编译器安装目录的 `docs` 子目录（**第 4.2 节 “MPLAB X IDE 和工具安装”**）。打开 `XC16MasterIndex` 文件，并单击 “Interrupt Vector Tables Reference”（中断向量表参考）链接。

© 2015 Microchip Technology Inc.

14.5 中断服务程序现场保存

由于中断本身的性质，中断可以在不可预知的时间发生。因此，中断代码必须能够使用发生中断时的相同机器状态继续工作。

为了正确处理从中断返回，ISR 函数的设置（序言）代码会在堆栈中自动保存编译器管理的工作寄存器和特殊功能寄存器，以便之后在 ISR 结束时恢复这些寄存器。您可以使用 `interrupt` 属性的可选 `save` 参数来指定要保存和恢复的额外变量和 SFR。

14.5.1 汇编代码和 ISR

在一些应用程序中，可能需要紧接在编译器生成的函数序言之前向 ISR 中插入汇编语句。例如，可能需要在进入中断服务程序时立即递增一个信号量。可以通过以下操作实现：

```
void __attribute__((interrupt(auto_psv(preprologue
  ("inc _semaphore"))))) isr0(void);
```

现场切换会导致中断代码执行中的响应延时，如**第 14.8.3 节 “中断响应延时”**所述。

14.5.2 `context` 属性

可以对中断服务程序应用`context`属性，向编译器指示该ISR以特定中断优先级（IPL）执行，该优先级也已被分配备用寄存器集。关于如何正确配置器件来使用备用寄存器集的详细信息，请参见器件数据手册或系列参考手册。该功能使用配置位进行设置。

使用该属性时，重要的是中断的优先级与已分配的现场的优先级匹配。更改中断服务程序的优先级可能会导致运行时损坏。

使用示例：

```
// Priority Level 7 routines will use context 1
#pragma config CTXT1 = 7
// T1 Interrupt uses its own context
void __attribute__((interrupt, context)) _T1Interrupt(void);
main() {
   // Timer 1 is configured to use priority level 7
   IPC0bits.T1IP = 7;
```

14.6 嵌套中断

16 位器件支持嵌套中断。由于在 ISR 中，处理器资源保存在堆栈中，所以嵌套 ISR 的代码编写方式与非嵌套的相同。嵌套中断通过清零 INTCON1 寄存器中的位 NSTDIS（嵌套中断禁止）来允许。请注意，这是默认条件，因为 16 位器件在退出复位时会允许嵌套中断。每个中断源在中断优先级控制寄存器 （IPC*n*）中分配优先级。

如果中断源的优先级大于当前的 CPU 优先级，则会跳转到中断。

14.7 允许 / 禁止中断

每个中断源可单独允许或禁止。每个 IRQ 的中断允许位分配在中断允许控制寄存器（IEC*n*）中。将某个中断允许位置 1 会允许相应的中断；将中断允许位清零会禁止相应的中断。当器件退出复位时，所有中断允许位都会被清零。此外，处理器还具有可以对所有中断禁止指定指令周期数的禁止中断指令 （DISI）。

注： 陷阱 （如地址错误陷阱）不能被禁止。只能禁止 IRQ。

可以通过使用以下方式在 C 程序中使用 DISI 指令：

```
__builtin_disi
```

例如：

```
__builtin_disi(16);
```

会在它出现在源程序中的位置点发出指定 DISI 指令。以这种方式使用 DISI 的缺点是，C 语言编程人员无法总是确定 C 编译器如何将 C 源代码转换为机器指令，所以可能难以确定 DISI 指令的周期数。可以通过使用 DISI 指令包围需要防止被中断的代码来解决此问题，第一条指令将周期数设置为最大值，第二条指令将周期数设置为零。例如，

```
__builtin_disi(0x3FFF); /* disable interrupts */
/* ... protected C code ... */
__builtin_disi(0x0000); /* enable interrupts */
```

一种替代方法是通过直接写入 DISICNT 寄存器来允许中断。只有在发出 DISI 指令后，并且 DISICNT 寄存器的内容不为零时，才能修改 DISICNT 寄存器。

```
__builtin_disi(0x3FFF); /* disable interrupts */
/* ... protected C code ... */
DISICNT = 0x0000; /* enable interrupts */
```

对于一些应用程序，可能还需要禁止级别 7 中断。它们只能通过修改 COROCON IPL 字段来禁止。所提供的支持文件包含了一些有用的预处理器宏函数，可帮助您安全地修改 IPL 值。这些宏包括：

```
SET_CPU_IPL(ipl)
SET_AND_SAVE_CPU_IPL(save_to, ipl)
RESTORE_CPU_IPL(saved_to)
```

例如，您可能希望对一段代码进行保护，使之不被中断。以下代码会调整当前的 IPL 设置，以及将 IPL 恢复为其先前的值。

```
void foo(void) {
  int current_cpu_ipl;

  SET_AND_SAVE_CPU_IPL(current_cpu_ipl, 7);  /* disable interrupts */
   /* protected code here */
  RESTORE_CPU_IPL(current_cpu_ipl);
}
```


© 2015 Microchip Technology Inc.

14.8 ISR 注意事项

以下几节介绍了如何确保您的中断代码按预期工作。

14.8.1 与主干代码共享存储器

在 main 或低优先级 ISR 和高优先级 ISR 内修改同一变量时需要小心。如果低优先级函数创建了访问同一变量的多指令读 - 修改 - 写序列，则当允许较高优先级的中断时，这些较高优先级的中断可能中断该多指令序列，产生意外的结果。因此，嵌入式系统必须实现 “原子” 操作，以确保插入其间的高优先级 ISR 不会写入低优先级 ISR 刚读取但尚未完成其写操作的变量。

原子操作是指不能进一步分解为各组成部分的操作——它不能被中断。不是所有 C 表达式都会转换为原子操作。在 dsPIC DSC 器件上，这些表达方式主要分为以下类别：32 位表达式、浮点算术、除法、对多位位域的操作，以及定点运算。其他因素将决定是否生成原子操作，如存储模型设置、优化级别和资源可用性。也即，C 并不保证操作的原子性。

假设具有以下一般表达式：

```
foo = bar op baz;
```

基于器件的架构，操作符 （op）可能是也可能不是原子的。在任何情况下，编译器可能无法在所有情况下都生成原子操作，这取决于一些因素，可能包括以下：

- 相应原子机器指令的可用性
- 资源可用性——特殊寄存器或其他约束
- 优化级别，以及影响数据 / 代码定位的其他选项

在不了解架构的情况下，假定一般表达式需要两次读操作（每个操作数一次）和一次写操作来存储结果是合理的。存在中断序列时，可能会出现一些困难，具体取决于特定的应用程序。

14.8.1.1 开发问题

以下是应考虑的问题的一些示例：

例 14-1： bar 必须与 baz 匹配

当要求 bar 与 baz 匹配时 （即，相互同步更新），如果 bar 或 baz 可能在较高优先级中断表达式中更新，则有可能存在危险。以下是一些示例的流程序列：

1. 安全：
   读 bar
   读 baz
   执行操作
   将结果写回 foo
2. 不安全：
   读 bar
   中断修改 baz
   读 baz
   执行操作
   将结果写回 foo
3. 安全：
   读 bar
   读 baz
   中断修改 bar 或 baz
   执行操作
   将结果写回 foo

第一个示例是安全的，因为所有中断都处于表达式边界之外。第二个是不安全的，因为应用程序要求 bar 和 baz 相互同步更新。第三个可能是安全的； foo 可能会具有旧值，但该值将与表达式开始时可用的数据一致。

例 14-2： foo、 bar 和 baz 的类型

另一种变化情况依赖于 foo、 bar 和 baz 的类型。操作 “读 bar”、“读 baz”或 “将结果写回 foo”可能不是原子的，这取决于目标处理器的架构。例如， dsPIC DSC 器件可以在 1 条 （原子）指令中读取或写入 8 位、 16 位或 32 位的量。但是， 32 位的量可能需要两条指令，这取决于指令选择 （这继而取决于优化和存储模型设置）。假定类型为 long，而编译器无法选择原子操作来访问数据。那么，访问会变为：

读 lsw bar
读 msw bar
读 lsw baz
读 msw baz
执行操作 （对 lsw 和 msw）
执行操作
将 lsw 结果写回 foo
将 msw 结果写回 foo

现在，更新 bar 或 baz 导致意外数据的可能性会更大。

© 2015 Microchip Technology Inc.

例 14-3：　　位域

第三种需要关心的问题是位域。 C 语言允许在位级别分配存储器，但未定义任何位操作。在纯粹意义上说，对于位的任何操作都会被当作是对于位域的基础类型的操作，通常需要一些操作来从 bar 和 baz 中提取位域，或将位域插入 foo。需要注意的重要考虑因素是（同样取决于指令架构、优化级别和存储设置），如果某个程序会写入 foo 所在位域的任何部分，该程序被中断可能会破坏数据。这在操作数之一也作为目标的情况下尤为明显。

dsPIC DSC 指令集可以按原子方式对 1 位进行操作。编译器可能会根据优化级别、存储设置和资源可用性选择这些指令。

例 14-4：　　寄存器中缓存的存储器值

最后，编译器可能会选择在寄存器中缓存存储器值。它们通常被称为寄存器变量，特别容易被中断破坏，即使在涉及变量的操作并未被中断时。请确保将在 ISR 和可中断函数之间共享的存储器资源指定为 volatile。这会向编译器指示，该存储单元可能会在连续代码序列之外被更新。这并不能防止非原子操作的影响，但仍然很重要。

14.8.1.2　开发解决方案

以下是一些可用于消除潜在危险的策略：

- 设计不会发生冲突事件的软件系统。不在 ISR 与其他函数之间共享存储器。使 ISR 尽可能简单，并将真正的工作转移到主代码中。
- 共享存储器时保持小心，如果可能，避免共享包含多个位的位域。
- 如同保护关键代码段一样，保护共享存储器的非原子更新，防止执行期间发生中断。以下宏可以用于该目的：

```
#define INTERRUPT_PROTECT(x) {              \
   char saved_ipl;                          \
                                            \
   SET_AND_SAVE_CPU_IPL(saved_ipl,7);       \
   x;                                       \
   RESTORE_CPU_IPL(saved_ipl); } (void) 0;
```

  该宏通过将当前优先级增大为 7 来禁止中断，执行所需的语句，然后恢复先前的优先级。

14.8.1.3 应用示例

以下示例突出了本节介绍的一些要点：

```
void __attribute__((interrupt))
     HigherPriorityInterrupt(void) {
      /* User Code Here */
      LATGbits.LATG15 = 1;  /* Set LATG bit 15 */
      IPC0bits.INT0IP = 2;  /* Set Interrupt 0
                               priority (multiple
                               bits involved) to 2 */
       }

int main(void) {
    /* More User Code */
    LATGbits.LATG10 ^= 1;  /* Potential HAZARD -
                              First reads LATG into a W reg,
                              implements XOR operation,
                              then writes result to LATG */

    LATG = 0x1238;         /* No problem, this is a write
                              only assignment operation */

    LATGbits.LATG5 = 1;    /* No problem likely,
                              this is an assignment of a
                              single bit and will use a single
                              instruction bit set operation */

    LATGbits.LATG2 = 0;    /* No problem likely,
                              single instruction bit clear
                              operation probably used */

    LATG += 0x0001;        /* Potential HAZARD -
                              First reads LATG into a W reg,
                              implements add operation,
                              then writes result to LATG */

    IPC0bits.T1IP = 5;     /* HAZARD -
                              Assigning a multiple bitfield
                              can generate a multiple
                              instruction sequence */

}
```

可以使用上面提供的 `INTERRUPT_PROTECT` 宏来保护语句，防止执行期间发生中断。对于该示例：

```
INTERRUPT_PROTECT(LATGbits.LATG15 ^= 1); /* Not interruptible by
                                            level 1-7 interrupt
                                            requests and safe
                                            at any optimization
                                            level */
```


© 2015 Microchip Technology Inc.

14.8.2 中断服务程序中的 PSV 使用

编译器 v3.0 引入了受管 psv 指针和 CodeGuard 安全性 psv 常量段，这意味着 ISR 不能对于 PSVPAG 的设置作出任何假定。对于具有引用 auto_psv 常量段的 ISR 的现有应用程序，这是一个移植问题。在先前版本的编译器中，ISR 可以假定在程序启动期间设置了 PSVPAG 的正确值（除非编程人员显式地更改它）。

为了帮助减轻这一问题，将引入两个新的函数属性：auto_psv 和 no_auto_psv。如果在使用 constants-in-code 存储模型的情况下 ISR 引用了 const 变量或字符串字面值，则应在函数定义中添加 auto_psv 属性。该属性会导致编译器保存 PSVPAG 的先前内容，并将其设置为段 .const。在退出时，将恢复 PSVPAG 的先前值。例如：

```
void __attribute__((interrupt, auto_psv)) _T1Interrupt()
{
  /* This function can reference const variables and
  string literals with the constants-in-code memory model. */
}
```

no_auto_psv 属性用于指示 ISR 不引用 auto_psv 常量段。如果这两个属性均未指定，编译器将采用 auto_psv，并插入必要的指令来确保在运行时正确工作。此外还会发出一条警告诊断消息，提醒用户注意移植问题，以及通过指定的 no_auto_psv 属性降低中断响应延时的可能性。

14.8.3 中断响应延时

有两个因素会影响在中断源发生和执行 ISR 代码第一条指令之间的周期数。这些因素包括：

- **处理器中断处理**——处理器识别中断并转移到中断向量第一个地址所需的时间量。对于特定处理器和所使用的中断源，要确定该值，请参见处理器数据手册。
- **ISR 代码**——虽然中断函数可以调用其他函数，无论它们是用户定义的函数、库函数还是用于实现 C 操作的隐式调用函数，但编译器通常无法知道被调用函数使用了哪些资源。因此，编译器将保存所有工作寄存器和 RCOUNT，即使它们并不全都在 ISR 自身中显式地使用。与调用关联的响应延时会增加，所以无助于产生快速的响应速度。

注：

© 2015 Microchip Technology Inc.

第 15 章 main、运行时启动和复位

15.1 简介

在创建 C 代码时，需要一些元素来确保程序正确工作：必须存在 main 函数；需要启动代码来初始化和清零变量，以及设置寄存器和处理器；需要处理复位条件。本节介绍了以下主题：

- main 函数
- 运行时启动和初始化

15.2 main 函数

标识符 main 是一个特殊的标识符。它必须用作要在程序中执行的第一个函数的名称。在程序中必须总是有且仅有一个名为 main() 的函数。但是，与 main() 关联的代码并不是复位后最先执行的代码。编译器提供的额外代码 （称为运行时启动代码）将先执行，并负责将控制转移到 main() 函数。

应用于 main() 的原型如下所示。

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

15.3 运行时启动和初始化

C 程序要求先初始化一些特定的对象，并且处理器处于某种特定状态，之后才会开始执行其 main() 函数。执行这些任务是*运行时启动代码*的工作，特别是（没有特定的顺序）：

- 初始化在定义时赋值的全局变量
- 初始化堆栈
- 清零未初始化的全局变量
- 对寄存器或处理器状态进行常规设置

两个 C 运行时启动模块包含在 libpic30-*omf*.a 归档 / 库中。两个启动模块的入口点为 __reset。链接描述文件会在程序存储器的地址 0 构造一条 GOTO __reset 指令，在器件复位时转移控制。

默认情况下将链接主启动模块，并执行以下任务：

1. 使用由链接器或定制链接描述文件所提供的值，初始化堆栈指针 （W15）和堆栈指针限制寄存器 （SPLIM）。更多信息，请参见**第 6.4 节 “堆栈”**。
2. 如果定义了 .const 段，则会视情况通过初始化 PSV 页和 CORCON 寄存器将它映射到程序空间可视性窗口中，前提是使用了 const-in-code 存储模型或将变量显式地分配到 space(auto_psv)。

3. 读取数据初始化模板，这会使所有未初始化的对象清零，并且使所有已初始化的对象使用从程序存储器读取的值进行初始化。数据初始化模板由链接器创建。

注： 永不清零或初始化持久性数据。

4. 如果应用程序定义了 `user_init` 函数 （见**第 13.2.2 节 “函数属性”**），则会调用它们。执行顺序取决于链接顺序。
5. 不带参数调用函数 `main()`。
6. 如果 `main()` 返回，处理器将复位。

指定 `-Wl`, `--no-data-init` 选项时会链接备用启动模块。它将执行相同操作，除了会忽略步骤 （3）。备用启动模块比主启动模块小，如果无需数据初始化，可选择备用启动模块来节省程序存储器空间。

在 `<xc16 install directory>\src\ibpic30.zip` 中提供了两个模块的压缩源代码 （使用 dsPIC DSC 汇编语言编写）。必要时可修改启动模块。例如，如果应用程序需要带参数调用 `main`，可以更改条件汇编伪指令来提供此支持。

© 2015 Microchip Technology Inc.

第 16 章 混合使用 C 代码和汇编代码

16.1 简介

本章介绍如何一起使用汇编语言和 C 语言模块。它给出了在汇编代码中使用 C 变量和函数的示例，以及在 C 中使用汇编语言变量和函数的示例。

讨论的内容包括：

- **混合使用汇编语言与 C 语言变量及函数**——可以汇编独立的汇编语言模块，然后与已编译的 C 模块进行链接。
- **使用行内汇编语言**——可以直接在 C 代码中嵌入汇编语言指令。行内汇编语言同时支持简单（无参数）汇编语言语句及扩展（有参数）语句（这种情况下，可以用汇编指令操作数的形式访问 C 变量）。
- **预定义的汇编宏**——提供了在 C 代码中使用的预定义汇编代码宏的列表。

16.2 混合使用汇编语言与 C 语言变量及函数

以下准则说明了如何将独立的汇编语言模块与 C 语言模块进行接口。

- 遵循**第 12.2 节“寄存器变量”**中所述的寄存器约定。特别是，使用寄存器 W0-W7 来进行参数传递。汇编语言函数将在这些寄存器中接收参数，并应在这些寄存器中向被调用函数传递参数。
- 中断处理期间调用的函数必须保存寄存器 W8-W15。也即，在修改这些寄存器中的值之前必须先保存它们，并在返回调用函数之前恢复。寄存器 W0-W7 可以在无需恢复它们的值的情况下使用。
- 中断函数必须保存所有寄存器。不同于普通的函数调用，中断可能在程序执行过程中的任意点发生。返回到正常程序时，所有寄存器都必须与发生中断之前相同。
- 在独立的汇编文件中声明且将由任意 C 源文件引用的变量或函数应使用汇编器伪指令 `.global` 声明为全局。外部符号前应至少具有一个下划线。 C 函数 `main` 在汇编代码中命名为 `_main` ；相反，汇编符号 `_do_something` 将在 C 代码中以 `do_something` 的形式引用。在汇编文件中使用的未声明符号将被视为外部定义。

以下示例说明了如何在汇编语言和 C 语言中使用变量和函数，无论它们最初在何处定义。

文件 `ex1.c` 定义了要在汇编语言文件中使用的 `foo` 和 `cVariable`。 C 文件还说明了如何调用汇编函数 `asmFunction` 和如何访问汇编代码定义的变量 `asmVariable`。

本节中的示例：

- 混合使用 C 代码和汇编代码
- 在 C 代码中调用汇编函数

例 16-1：　　混合使用 C 代码和汇编代码

```
/*
** file: ex1.c
*/
extern unsigned int asmVariable;
extern void asmFunction(void);
unsigned int cVariable;
void foo(void)
{
  asmFunction();
  asmVariable = 0x1234;
}
```

文件 ex2.s 定义了需要在链接的应用程序中使用的 asmFunction 和 asmVariable。汇编文件还说明了如何调用 C 函数 foo 和如何访问 C 代码定义的变量 cVariable。

```
;
;  file: ex2.s
;
  .text
  .global _asmFunction
_asmFunction:
  mov #0,w0
  mov w0,_cVariable
  return

  .global _main
_main:
  call _foo
  return

  .bss
  .global _asmVariable
  .align  2
_asmVariable:.space 2
 .end
```

在 C 文件 ex1.c 中，对汇编文件中声明的符号的外部引用使用标准 extern 关键字声明；请注意 asmFunction 或汇编源代码中的 _asmFunction 是 void 类型的函数，并相应地进行声明。

在汇编文件 ex1.s 中，符号 _asmFunction、_main 和 _asmVariable 通过使用 .global 汇编伪指令而设为全局可见，可以由任何其他源文件访问。符号 _main 仅进行引用，并未声明；因此，汇编器将它当作一个外部引用。

以下编译器示例说明了如何调用带两个参数的汇编函数。 call1.c 中的 C 函数 main 调用 call2.s 中具有两个参数的 asmFunction。

© 2015 Microchip Technology Inc.

例 16-2：　　在 C 代码中调用汇编函数

```
/*
** file: call1.c
*/
extern int asmFunction(int, int);
int x;
void
main(void)
{
  x = asmFunction(0x100, 0x200);
}
```

汇编语言函数对它的两个参数进行相加，并返回结果。

```
;
;  file: call2.s
;
  .global _asmFunction
_asmFunction:
  add w0,w1,w0
  return
 .end
```

第 13.8.2 节 “返回值”详细介绍了 C 代码中的参数传递。在前面的示例中，两个整型参数通过 W0 和 W1 寄存器传递。整型返回结果通过寄存器 W0 传递。更复杂的参数列表可能需要不同的寄存器，在手工编写的汇编代码中应小心地遵循这些准则。

16.3 使用行内汇编语言

在 C 函数内，可以使用 asm 语句将一行汇编语言代码插入编译器将生成的汇编语言中。行内汇编代码具有两种形式：简单和扩展。

在**简单**形式中，汇编指令使用以下语法编写：

```
asm ("instruction");
```

其中，*instruction* 是有效的汇编语言构造。如果在 ANSI C 程序中编写行内汇编代码，则编写 __asm__ 而不是 asm。

> **注：** 只能向简单形式的行内汇编代码传递单个字符串。

在使用 asm 的**扩展**汇编指令中，使用 C 表达式指定指令的操作数。扩展形式语法为：

```
asm("template" [ :[ "constraint"(output-operand) [ , ... ] ]
                  [ : [ "constraint"(input-operand) [ , ... ] ]
                       [ "clobber" [ , ... ] ]
                ]
              ]);
```

您必须指定一个汇编指令 *template*，再加上每个操作数的操作数 *constraint* 字符串。*template* 指定指令助记符，以及（可选）操作数的占位符。*constraint* 字符串指定操作数约束，例如，操作数必须处于寄存器内（通常情况），或操作数必须为立即数。

表 16-1 和表 16-2 分别列出了编译器支持的约束字母和修饰符。

表 16-1：编译器支持的约束字母

字母	约束
a	声明 WREG
b	除法支持寄存器 W1
c	乘法支持寄存器 W2
d	通用数据寄存器 W1-W14
e	非除法支持寄存器 W2-W14
g	不属于通用寄存器的寄存器除外，允许任何寄存器、存储器或立即数整型操作数。
i	允许立即数整型操作数（具有常量值的操作数）。这包括值仅在汇编时才知道的符号常量。
r	允许寄存器操作数，前提是它是通用寄存器。
v	AWB 寄存器 W13
w	累加器寄存器 A-B
x	x 预取寄存器 W8-W9
y	y 预取寄存器 W10-W11
z	MAC 预取寄存器 W4-W7
0, 1, … , 9	允许与指定操作数编号匹配的操作数。如果在同一个备选集中，数字与字母一起使用，则数字应在最后。默认情况下，%*n* 表示操作数（*n*）的第一个寄存器。要访问第二、第三或第四个寄存器，请使用修饰符字母。
C	偶 - 奇编号寄存器对

© 2015 Microchip Technology Inc.

表 16-1：编译器支持的约束字母 （续）

字母	约束
D	偶编号寄存器
T	near 或 far 数据操作数。
U	near 数据操作数。

表 16-2：编译器支持的约束修饰符

修饰符	约束
=	表示该操作数对于该指令是只写的：先前的值会被丢弃，替换为输出数据。
+	表示指令既会读取也会写入该操作数。
&	表示该操作数是一个 *earlyclobber* 操作数，即在使用输入操作数完成指令之前就会被修改。因此，该操作数不能位于用作输入操作数的寄存器中，也不能作为任何存储器地址的一部分。
d	操作数编号 *n* 的第 2 个寄存器，即， %d*n*。
q	操作数编号 *n* 的第 4 个寄存器，即， %q*n*。
t	操作数编号 *n* 的第 3 个寄存器，即， %t*n*。

本节中的示例：

- 传递 C 变量
- 破坏寄存器内容
- 使用多条汇编指令
- 使用 “&”来防止输入寄存器内容破坏
- 匹配操作数
- 指定操作数
- 可变 asm 语句
- 处理长度大于 int 的值

例 16-3： 传递 C 变量

以下示例说明了如何使用 swap 指令 （编译器通常不使用该指令）：

```
asm ("swap %0" : "+r"(var));
```

此处的 var 是操作数的 C 表达式，它既是输入操作数也是输出操作数。该操作数被约束为类型 r，它表示一个寄存器操作数。 +r 中的 + 表示操作数既是输入操作数也是输出操作数。

每个操作数通过一个操作数约束字符串 （后跟包含在括号中的 C 表达式）进行描述。一个冒号将汇编语言模板与第一个输出操作数分隔，另一个冒号将最后一个输出操作数与第一个输入操作数分隔 （如果有）。使用逗号来分隔各输出操作数和分隔各输入操作数。

如果没有输出操作数，但是有输入操作数；那么，必须有两个连续的冒号包围输出操作数本应所在的位置。编译器要求输出操作数表达式必须为左值。输入操作数不需要为左值。编译器无法检查操作数是否具有对应于要执行的指令的合理数据类型。它不会分析汇编指令模板，不知道它的含义，也不知道它是否是有效的汇编语言输入。扩展 asm 功能最常用于编译器本身不知道存在的机器指令。如果无法对输出表达式进行直接寻址（例如，它是一个位域），则约束必须允许使用寄存器。在这种情况下，编译器将使用寄存器作为 asm 的输出，然后将该寄存器存储到输出中。如果输出操作数是只写的，编译器将假定在执行该指令之前这些操作数中的值是死值，不需要生成这些值。

例 16-4： 破坏寄存器内容

一些指令会破坏特定硬寄存器的内容。为了描述这种情况，在输入操作数后书写第三个冒号，后面跟随内容已被破坏的硬寄存器的名称（以使用逗号进行分隔的字符串的形式给出）。此处给出了一个示例：

```
asm volatile ("mul.b %0"
: /* no outputs */
: "U" (nvar)
: "w2");
```

在此例中，操作数 nvar 是一个声明在 near 数据空间中的字符变量，这通过 “U”约束指定。如果汇编指令会改变标志 （状态代码）寄存器，则将 “cc”添加到内容已被破坏的寄存器的列表中。如果汇编指令会以不可预知的方式修改存储器，则将“memory”添加到内容已被破坏的寄存器的列表中。这会导致编译器不保留在汇编指令之间在寄存器中缓存的存储器值。

© 2015 Microchip Technology Inc.

例 16-5：　　使用多条汇编指令

您可以将多条汇编指令一起放在单个 asm 模板中，使用换行符（书写为 \n）分隔。确保输入操作数和输出操作数的地址不使用任何内容已被破坏的寄存器，以便您可以任意多次地读取和写入内容已被破坏的寄存器。以下是一个在模板中放入多条指令的示例；它假定子程序 _foo 在 W0 和 W1 寄存器中接受参数：

```
asm ("mov %0,w0\nmov %1,W1\ncall _foo"
: /* no outputs */
: "g" (a), "g" (b)
: "W0", "W1");
```

在该示例中，约束字符串 “g” 表示一个通用操作数。

例 16-6：　　使用 “&” 来防止输入寄存器内容破坏

除非输出操作数具有 & 约束修饰符，否则编译器可能会将它分配在与一个无关输入操作数相同的寄存器中，基于输入在输出产生之前使用的假定。如果汇编代码实际上包含多条指令，这种假定可能为假。在这种情况下，对每个不能与输入操作数重叠的输出操作数使用 &。例如，假设具有以下函数：

```
int
exprbad(int a, int b)
{
        int c;

        __asm__("add %1,%2,%0\n sl %0,%1,%0"
                : "=r"(c) : "r"(a), "r"(b));

        return(c);
}
```

其目的是计算 (a + b) << a 的值。但是，按照这种编写方式，计算出的值可能是也可能不是这个值。正确的代码编写需要向编译器指示操作数 c 会在 asm 指令结束使用输入操作数之前被修改，如下所示：

```
int
exprgood(int a, int b)
{
        int c;

        __asm__("add %1,%2,%0\n sl %0,%1,%0"
                : "=&r"(c) : "r"(a), "r"(b));

        return(c);
}
```

例 16-7： 匹配操作数

当汇编指令具有一个读写操作数，或一个只有一部分位需要更改的操作数时，逻辑上必须将其功能拆分为两个独立的操作数：一个输入操作数，一个只写输出操作数。它们之间的关系通过指示在指令执行时它们需要处于同一存储单元的约束来表示。您可以对两个操作数使用相同的 C 表达式，也可以使用不同的表达式。例如，以下是一条 add 指令，使用 bar 作为其只读源操作数， foo 作为其读写目标：

```
asm ("add %2,%1,%0"
: "=r" (foo)
: "0" (foo), "r" (bar));
```

操作数 1 的约束 “0” 指示它必须占用与操作数 0 相同的存储单元。只有输入操作数允许在约束中使用一个数字，并且该数字必须指代一个输出操作数。只有在约束中使用一个数字才能确保一个操作数会与另一个操作数处于同一地址。foo 同时是两个操作数的值这一事实并不足以确保它们会在生成的汇编代码中处于同一地址。以下形式可能不起作用：

```
asm ("add %2,%1,%0"
: "=r" (foo)
: "r" (foo), "r" (bar));
```

各种优化或重新装入可能导致操作数 0 和 1 处于不同的寄存器中。例如，编译器可能在一个寄存器中找到 foo 值的副本，并将它用于操作数 1，但在一个不同的寄存器中生成输出操作数 0（之后将它复制到 foo 自己的地址中）。

例 16-8： 指定操作数

此外，还可以使用可在汇编代码模板中引用的符号名称来指定输入和输出操作数。这些名称在约束字符串前的方括号内指定，可以在汇编代码模板内使用 %[*name*] 形式引用，而不是使用百分比符号后跟操作数数字的形式。使用指定操作数，上面的示例可以编写为如下形式：

```
asm ("add %[foo],%[bar],%[foo]"
: [foo] "=r" (foo)
: "0" (foo), [bar] "r" (bar));
```

例 16-9： 可变 ASM 语句

您可以通过在 asm 后书写关键字 volatile，阻止 asm 指令被删除、明显移动或合并。例如：

```
#define disi(n) \
asm volatile ("disi #%0" \
: /* no outputs */ \
: "i" (n))
```

在此例中，约束字母 “i” 表示 disi 指令所需的立即操作数。

例 16-10： 处理长度大于 INT 的值

可以使用约束字母和修饰符来标识各个实体，用它来代替某个特定操作数，例如以下语句中的 %0：

```
asm("mov %1, %0" : "r"(foo) : "r"(bar));
```


© 2015 Microchip Technology Inc.

该示例指示将 foo 中存储的值传送到 bar。除非 foo 或 bar 的长度大于 int，否则该代码示例将可执行该任务。

默认情况下， %0 表示操作数 （0）的第一个寄存器。要访问第二、第三或第四个寄存器，请使用表中指定的修饰符字母。

16.4 预定义的汇编宏

在包含 <xc.h> 之后，就会定义一些用于在 C 代码中插入汇编代码的宏。这些宏包括：Nop()、ClrWdt()、Sleep() 和 Idle()。后两个宏分别插入带有参数 #0 和 #1 的 PWRSAV 指令。

注：

© 2015 Microchip Technology Inc.

第 17 章 库程序

17.1 简介

在源代码中引用许多库函数或程序之后，它们（及其任何相关变量）将自动链接到程序中。使用来自一个库文件的某个函数并不会包含来自该库的所有其他函数。只有所使用的库函数会被链接到程序输出中并占用存储器。

库和预编译的目标文件存储在编译器的安装目录结构中。

对于从库中使用的任何函数或符号，程序都需要声明。它们包含在标准 C 头（.h）文件中。头文件不是库文件，这两种文件类型不应混为一谈。库文件包含预编译的代码，通常为函数和变量定义；头文件提供库文件中的函数、变量和类型，以及其他预处理器宏的声明（而不是定义）。

编译器的安装目录下的 include 目录是编译器存储标准 C 库系统头文件的位置。安装会自动查找它捆绑的包含文件。

一些库需要在您的项目中手动包含，或需要使用特殊的选项。关于特定库的问题，请参见《16 位语言工具函数库》（DS51456D_CN）。

自动查找的库包括：

- 标准 C 库
- dsPIC30 支持库
- 标准 IEEE 浮点库
- 定点库
- 器件外设库

例 17-1： 使用数学库

```
#include <math.h>    // declare function prototype for sqrt

void main (void)
{
  double i;

  // sqrt referenced; sqrt will be linked in from library file
  i = sqrt(23.5);
}
```

注：

© 2015 Microchip Technology Inc.

第 18 章 优化

18.1 简介

不同的 MPLAB XC16 C 编译器许可证级别支持不同的优化级别。最初下载的编译器从免费许可证级别开始。使能评估许可证将允许对编译器进行 60 天的评估。可以随时购买标准或专业许可证。

版本	成本	说明
专业 （PRO）	有	实现了最高的优化和性能级别。
标准 （STD）	有	实现了丰富的优化级别和高性能级别。
免费	无	对代码优化实现了最多的限制。
评估 （EVAL）	无	允许使用专业版 60 天，之后恢复为免费版。

18.2 设置优化级别

以下列出了不同编译器版本的优化，按 O 级别和 GCC 优化列出：

- 专业：-O3、 -Os 和 -mpa （还允许使用免费版和标准版的优化）
- 标准：-O2 （还允许使用免费版的优化）
- 免费：-O0 和 -O1

可以设置从无优化到完全优化的不同优化，具体取决于您的编译器版本。在调试代码时，您可能希望不对代码进行优化，以确保预期的程序流。

关于用于设置优化的编译器选项的详细信息，请参见**第 5.7.6 节“用于控制优化的选项”**。

18.3 优化功能汇总

每个版本的可用优化级别等同于下表中指定的功能。

表 18-1： 优化版本支持的功能

免费	标准	专业
• defer pop • delayed branch • omit frame pointer • guess branch prob • cprop registers • forward propagate • if conversion • if conversion2 • ipa pure const • ipa reference • merge constants • split wide types • tree ccp • tree dce • tree dom • tree dse • tree ter • tree sra • tree copyrename • tree fre • tree copy prop • tree sink • tree ch	所有免费版优化加上： • indirect inlining • thread jumps • crossjumping • optimize sibling calls • cse follow jumps • gcse • expensive optimizations • cse after loop • caller saves • peephole2 • schedule insns • schedule insns after reload • regmove • strict aliasing • strict overflow • reorder blocks • reorder functions • tree vrp • tree builtin call dce • tree pre • tree switch conversion • ipa cp • ipa sra	所有免费版和标准版优化加上： • predictive commoning • inline functions • unswitch loops • gcse after reload • tree vectorize • ipa cp clone • Whole-program optimizations

© 2015 Microchip Technology Inc.

第 19 章 预处理

19.1 简介

在编译之前，需要对所有 C 源文件进行预处理。 -E 选项可用于执行预处理，然后停止编译。请参见**第 5.7.2 节 “用于控制输出类型的选项”**。

如果汇编文件的文件扩展名为 .S 而不是 .s，则也需要对这些文件进行预处理 （见**第 5.2.3 节 “输入文件类型”**）。

本章讨论的内容包括：

- C 语言注释
- 预处理伪指令
- 预定义的宏名称
- pragma 伪指令与属性

19.2 C 语言注释

MPLAB XC16 C 编译器支持标准 C 注释，以及 C++ 样式的注释。下表对这两种类型进行了说明。

注释语法	说明	示例
/* */	标准 C 代码注释。用于一行或多行。	/* This is line 1 This is line 2 */
//	C++ 代码注释。仅用于一行。	// This is line 1 // This is line 2

19.3 预处理伪指令

除了标准伪指令之外，该编译器还可接受几条专用的预处理器伪指令。表 19-1 列出了所有这些伪指令。

表 19-1： 预处理器伪指令

伪指令	含义	示例
#define	定义预处理器宏	#define SIZE 5 #define FLAG #define add(a,b) ((a)+(b))
#elif	#else #if 的缩写	请参见 #ifdef
#else	根据条件包含源代码行	请参见 #if
#endif	终止条件性的源代码包含	请参见 #if
#error	产生一条错误消息	#error Size too big

#if	如果常量表达式为 true，则包含源代码行	#if SIZE < 10
		  c = process(10)
		#else
		  skip();
		#endif

#ifdef	如果预处理器符号已定义，则包含源代码行	#ifdef FLAG
		  do_loop();
		#elif SIZE == 5
		  skip_loop();
		#endif

#ifndef	如果预处理器符号未定义，则包含源代码行	#ifndef FLAG
		  jump();
		#endif

#include	在源代码中包含文本文件	#include <stdio.h> #include "project.h"
#line	指定列表的行号和文件名	#line 3 final
#pragma	特定于编译器的选项	请参见**第 19.5 节“pragma 伪指令与属性”**
#undef	取消定义预处理器符号	#undef FLAG
#warning	产生一条警告消息	#warning Length not set

带参数的宏扩展可以使用 # 字符将参数转换为字符串，以及使用 ## 序列来连接参数。如果要连接两个表达式，可以考虑使用两个宏，以处理其中任一表达式需要替换自身的情况，例如

```
#define paste1(a,b)  a##b
#define paste(a,b) paste1(a,b)
```

使您可以使用 paste 宏来连接两个自身可能需要进一步扩展的表达式。请记住，对某个宏标识符进行扩展之后，如果在连接之后出现它，将不会再次对它进行扩展。

关于预处理伪指令的实现定义的行为，请参见**第 A.14 节 “预处理伪指令”**。

© 2015 Microchip Technology Inc.

19.4 预定义的宏名称

编译器预定义了几个可通过源代码中的条件伪指令测试的宏。**附录 F“弃用功能”**列出了已被弃用的常量。

19.4.1 编译器版本宏

编译器会定义常量 __XC16_VERSION__，为版本标识符赋予一个数值。使用它可以利用编译器的新功能，同时与旧版本保持向后兼容。

该值基于当前发行版的主要和次要版本号。例如，发行版 1.00 的 __XC16_VERSION__ 定义为 1000。该宏可以与标准预处理器比较语句配合使用，根据条件来包含 / 排除各种代码构造。

通过在命令行中添加 --version 或通过检查发行版随附的 README.html 文件，可以找到 __XC16_VERSION__ 的当前定义。

19.4.2 输出类型和器件宏

使用 -ansi 命令行选项会定义以下符号。

表 19-2： 使用 -ANSI 时定义的宏

符号——前导双下划线	符号——前后双下划线	说明
__XC16	__XC16__	如果定义，则表示正在使用 16 位编译器。
__C30	__C30__	
__dsPICC30	__dsPIC30__	
__XC16ELF	__XC16ELF__	如果定义，则编译器产生 ELF 输出。
__C30ELF	__C30ELF__	
__dsPIC30ELF	__C30ELF__	
__XC16COFF	__XC16COFF__	如果定义，则编译器产生 COFF 输出。
__C30COFF	__C30COFF__	
__dsPIC30COFF	__dsPIC30COFF__	

不选择 -ansi 时会定义以下符号。

表 19-3： 不使用 -ANSI 时定义的宏

符号	说明
XC16	正在使用 16 位编译器。
C30	
dsPIC30	

此外，编译器会基于使用 -mcpu= 设置的目标器件定义一个符号。例如，-mcpu=30F6014，这会定义符号 __dsPIC30F6014__。

19.4.3 器件特性宏

如果使能了器件特性，则会定义以下符号。

表 19-4： 器件特性宏 / 符号

符号	说明
__HAS_DSP__	器件具有 DSP 引擎。
__HAS_EEDATA__	器件具有 EEDATA 存储器。
__HAS_DMA__	器件具有 DMA 控制器。 它是一个通用宏，如果存在任何 DMA 控制器则设置。它并不表示存在 DMA 存储器。要确定是否有任何 DMA 存储器，请使用 __DMA_BASE 或 __DMA_LENGTH 列表常量，这些常量将在器件头文件中定义。
__HAS_DMAV2__	器件具有 DMA V2 控制器。 如果存在 DMA V2 控制器，则设置该宏。它并不表示存在 DMA 存储器。要确定是否有任何 DMA 存储器，请使用 __DMA_BASE 或 __DMA_LENGTH 列表常量，这些常量将在器件头文件中定义。
__HAS_CODEGUARD__	器件具有 Codeguard™ 安全功能。
__HAS_PMP__	器件具有并行主端口。
__HAS_PMPV2__	器件具有并行主端口 V2。
__HAS_PMP_ENHANCED__	器件具有增强型并行主端口。
__HAS_EDS__	器件具有扩展数据空间。
__HAS_5VOLTS__	器件是 5V 器件。

19.4.4 其他宏

以下符号定义其他特性。

表 19-5： 其他宏 / 符号

符号	说明
__FILE__	以 C 字符串表示的当前文件名。
__LINE__	以 C 字符串表示的当前行号。
__DATE__	以 C 字符串表示的当前日期。

© 2015 Microchip Technology Inc.

19.5 PRAGMA 伪指令与属性

MPLAB XC16 C 编译器使用非 ANSI 属性而不是 pragma 伪指令或限定符来在存储器中定位变量和函数。作为比较， PIC18 MCU C 编译器 （也称为 MPLAB C18）对于段（code、romdata、udata、idata）、中断（高优先级和低优先级）和变量位置（存储区、段）使用 pragma 伪指令。先前的用于 PIC18 MCU 的 HI-TECH C 编译器和更新的 MPLAB XC8 编译器使用限定符或 pragma 伪指令来执行相同操作。

如果您习惯于使用 PIC18 编译器，本节将说明如何改用 XC16 属性。关于属性的更多信息，请参见**第 8.12 节 “变量属性”**和**第 13.2.1 节 “函数说明符”**。

表 19-6： C18 PRAGMA 伪指令与属性

pragma 伪指令 （MPLAB C18）	属性 （MPLAB XC16）
`#pragma udata [name]`	`__attribute__ ((section ("name")))`
`#pragma idata [name]`	`__attribute__ ((section ("name")))`
`#pragma romdata [name]`	`__attribute__ ((space (auto_psv)))`
`#pragma code [name]`	`__attribute__ ((section ("name"), space (prog)))`
`#pragma interruptlow`	`__attribute__ ((interrupt))`
`#pragma interrupt`	`__attribute__ ((interrupt, shadow))`
`#pragma varlocate bank`	NA*
`#pragma varlocate name`	NA*

*16 位器件不具有存储器分区 （bank）。

表 19-7： PICC18 PRAGMA 伪指令和限定符与属性

PICC18	属性 （MPLAB XC16）
`#pragma psect old=new`	`____attribute__ ((section ("name")))`
`const`	`const` 或 `__attribute__ ((space (auto_psv)))`
`interrupt low_priority`	`__attribute__ ((interrupt))`
`interrupt`	`__attribute__ ((interrupt, shadow))`

例 19-1： 在数据存储器的用户段中指定未初始化变量

PICC18	`#pragma psect oldbss=mybss` `int gi;`
C18	`#pragma udata mybss` `int gi;`
XC16	`int __attribute__((__section__(".mybss"))) gi;`

其中， *oldbss* 是变量通常会被放入的 psect （段）的名称。

表 19-8： 将变量 MABONGA 定位到数据存储器中的地址 0X100 处

PICC18	`int Mabonga @ 0x100;`
C18	`#pragma idata myDataSection=0x100;` ` int Mabonga = 1;`
XC16	`int __attribute__((address(0x100))) Mabonga = 1;`

表 19-9： 指定要放入程序存储器的变量

PICC18	`const char my_const_array[10] = {0,1,2,3,4,5,6,7,8,9};`

C18	`#pragma romdata const_table`
	`const rom char my_const_array[10] =`
	`  {0,1,2,3,4,5,6,7,8,9};`

XC16	`const` 或
	`__attribute__((space(auto_psv)))`
	`  char my_const_array[10] = {0,1,2,3,4,5,6,7,8,9};`

表 19-10： 将函数 PRINTSTRING 定位到程序存储器中的地址 0X8000 处

PICC18	`int PrintString(const char *s)@ 0x8000 {...}`
C18	`#pragma code myTextSection=0x8000;` `int PrintString(const char *s){...}`
XC16	`int __attribute__((address(0x8000))) PrintString` `  (const char *s) {...}`

表 19-11： 编译器自动保存和恢复变量 VAR1 和 VAR2

PICC18	无等效语句

C18	`#pragma interrupt isr0 save=var1, var2`
	`  void isr0(void)`
	`  {`
	`  /* perform interrupt function here */`
	`  }`

XC16	`void __attribute__((__interrupt__(__save__(var1,var2))))`
	`  isr0(void)`
	`  {`
	`  /* perform interrupt function here */`
	`  }`

© 2015 Microchip Technology Inc.

第 20 章 链接程序

20.1 简介

除非请求编译器在生成中间文件之后停止，否则编译器将会自动调用链接器。

链接器将使用从命令行驱动程序获取的选项运行。这些选项指定器件的存储器，以及应如何将对象放置在存储器中。将会使用特定于器件的链接描述文件。

链接器操作可以使用驱动程序控制，更多信息请参见**第 5.7.9 节 “用于链接的选项”**。

链接器会创建一个映射文件，其中将详细说明分配的存储器和代码中的一些对象。映射文件是查找存储器信息的最佳场所。关于该文件中的详细信息的说明，请参见《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

20.2 默认存储空间

编译器会定义几个特殊用途的存储空间，以匹配 16 位器件的架构特性。静态和外部变量可以通过使用 space 属性分配在特殊用途存储空间，如**第 8.12 节“变量属性”**所述。

`data`

通用数据空间。通用数据空间中的变量可以使用普通 C 语句访问。这是默认分配方式。

DD **`xmemory`**——仅对于 dsPIC30F 和 dsPIC33EP/F 器件

X 数据地址空间。X 数据空间中的变量可以使用普通 C 语句访问。X 数据地址空间对于面向 DSP 的库和 / 或汇编语言指令具有特殊的意义。

DD **`ymemory`**——仅对于 dsPIC30F 和 dsPIC33EP/F 器件

Y 数据地址空间。Y 数据空间中的变量可以使用普通 C 语句访问。Y 数据地址空间对于面向 DSP 的库和 / 或汇编语言指令具有特殊的意义。

`prog`

通用程序空间，通常保留用于可执行代码。该程序空间中的变量无法使用普通 C 语句访问。它们必须由编程人员显式地访问，通常使用表访问行内汇编指令、使用程序空间可视性窗口，或通过使用 `__prog__` 进行限定。

auto_psv

程序空间中的编译器管理区域，指定用于程序空间可视性窗口访问。可以使用普通 C 语句读取 （但不能写入）该空间中的变量，并且最多分配 32K 的总空间。

psv

程序空间，指定用于程序空间可视性窗口访问。PSV 空间中的变量不由编译器管理，不能使用普通 C 语句访问。它们必须由编程人员显式地访问，通常使用表访问行内汇编指令，或使用程序空间可视性窗口。可以通过单次设置 PSVPAG 寄存器或通过使用 __psv__ 进行限定来访问 PSV 空间中的变量。

DD **eedata**——仅对于支持 EEDATA 的器件

数据 EEPROM 空间，位于程序存储器中高地址处的 16 位宽的非易失性存储器区域。eedata 空间中的变量无法使用普通 C 语句访问。它们必须由编程人员显式地访问，通常使用表访问行内汇编指令，或使用程序空间可视性窗口。对于支持 EEDATA 的器件，会定义 __HAS_EEDATA__ 列表常量。

DD **dma**——仅对于支持 DMA 的器件

DPSRAM DMA 存储器。 DMA 存储器中的变量可以使用普通 C 语句和通过 DMA 外设访问。对于支持 DMA 的器件，会定义 __HAS_DMA__ 列表常量。如果器件支持 DMA，但没有特殊 DPSRAM 可用，链接器将无法分配空间，并输出一个错误。

© 2015 Microchip Technology Inc.

20.3 替换库符号

MPLAB XC16 C 编译器附带了一个库管理器，它使您可以将库文件进行解包，并使用您自己的修改版本来替换模块。请参见《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。但是，您可以在无需这种操作的情况下简便地替换链接到程序中的库模块。

如果向项目中添加一个源文件，该源文件包含与库程序同名的程序的定义，那么库程序将被您的程序替换。

在尝试解析一个符号（如函数名或变量名）时，编译器会先扫描所有源代码模块来查找定义。只有无法在这些文件中解析该符号时，它才会搜索库文件。

如果在源文件中定义了符号，则编译器永远不会在库中实际搜索该符号，即使库文件中存在该符号也不会产生任何错误。以下情况下则不是如此：如果某个符号在源文件中定义了两次，并且定义存在冲突，则可能会产生错误。

另一种方法是在声明符号时使用 weak 属性。弱符号可能会被全局定义取代。当向对外部符号的引用应用 weak 时，链接时可以不需要该符号。

weak 属性可应用于函数和变量。可以将代码编写为，只有在从其他模块链接了函数的情况下，才使用该函数。确定是否使用该功能会成为一个链接时的决定，而不是编译时的决定。

关于 weak 属性的更多信息，请参见**第 8.12 节 “变量属性”**。

20.4 链接器定义的符号

16 位链接器定义了几个可以在 C 代码开发中使用的符号。关于更多信息，请参见《MPLAB® XC16 汇编器、链接器和实用程序用户指南》（DS52106A_CN）。

在程序存储器中定义了一个有用的地址符号 _PROGRAM_END，用于标记 CODE 或 PSV 段使用的最高地址。在接受程序存储器中对象地址的内置函数调用中，应使用地址操作符 （&）引用该符号。应用程序可以使用该符号作为校验和计算的结束点。

例如：

```
unsigned int end_page, end_offset;
 _prog_addressT big_addr;

end_page   =  __builtin_tblpage(&_PROGRAM_END);
end_offset = __builtin_tbloffset(&_PROGRAM_END);

_init_prog_address(big_addr, _PROGRAM_END);
```

20.5 默认链接描述文件

命令行总是需要一个链接描述文件。但是，如果 MPLAB IDE 项目中没有指定链接描述文件，IDE 将使用编译器附带的器件链接描述文件（*device*.gld）作为默认链接描述文件。这个特定于器件的文件包含诸如以下的信息：

- 存储区定义
- 程序、数据和调试段映射
- 中断和备用中断向量表映射
- SFR 地址赋值
- 各种外设的基址

默认情况下，链接描述文件位于：

<*install-dir*>\support*DeviceFamily*\gld

其中， *DeviceFamily* 是 16 位器件系列，例如 dsPIC30F。

要在您的项目中使用定制链接描述文件，只需将该文件添加到命令行上或项目的“Linker Files”文件夹中。

© 2015 Microchip Technology Inc.

附录 A 实现定义的行为

A.1 简介

本章介绍 MPLAB XC16 C 编译器的实现定义的行为。C 语言的 ISO 标准要求供应商用文档描述语言的 “实现定义的”特性的具体细节。

讨论的内容包括：

- 翻译
- 环境
- 标识符
- 字符
- 整型
- 浮点型
- 数组和指针
- 寄存器
- 结构、联合、枚举和位域
- 限定符
- 声明符
- 语句
- 预处理伪指令
- 库函数
- 信号
- 流和文件
- TMPFILE
- errno
- 存储器
- abort
- exit
- getenv
- system
- strerror

A.2 翻译

ANSI C 标准第 G.3.1 节涵盖了翻译的实现定义的行为。

空白字符（换行符除外）的每个非空序列是保留还是替换为一个空格字符？（ISO 5.1.1.2）

将它替换为一个空格字符。

诊断消息如何标识？（ISO 5.1.1.3）

诊断消息的标识方式是，在它们的前面加上对应于消息的源文件名和行号，并使用冒号字符（“:”）分隔。

是否存在不同类别的消息？（ISO 5.1.1.3）

是。

如果是，它们是什么？（ISO 5.1.1.3）

错误，它们会禁止生成输出文件；以及警告，它们不会禁止生成输出文件。

每种类别的消息的翻译器返回状态代码是什么？（ISO 5.1.1.3）

错误的返回状态代码为 1；对于警告为 0。

是否可以控制诊断的级别？（ISO 5.1.1.3）

是。

如果是的话，控制采取什么形式？（ISO 5.1.1.3）

可以使用编译器命令行选项来请求或禁止产生警告消息。

A.3 环境

ANSI C 标准第 G.3.2 节涵盖了环境的实现定义的行为。

独立程序可以使用哪些库设施？（ISO 5.1.2.1）

所有标准 C 库工具均可用，前提是针对环境定制一小组函数，如“Run Time Libraries”（运行时库）部分所述。

描述独立环境中的程序终止。（ISO 5.1.2.1）

如果函数 main 返回或调用了函数 exit，则在一个无限循环中执行 HALT 指令。该行为是可定制的。

描述传递给函数 `main` 的实参（形参）？（ISO 5.1.2.2.1）

不向 `main` 传递任何参数。

以下哪一种是有效的交互式设备：（ISO 5.1.2.3）

异步终端	否
配对的显示器和键盘	否
程序间连接	否
其他，请描述？	无

© 2015 Microchip Technology Inc.

A.4 标识符

ANSI C 标准第 G.3.3 节涵盖了标识符的实现定义的行为。

不带外部链接的标识符中，在 31 个字符之外有多少个字符是有效的？（ISO 6.1.2）

所有字符均有效。

带外部链接的标识符中，在 6 个字符之外有多少个字符是有效的？（ISO 6.1.2）

所有字符均有效。

带外部链接的标识符中是否区分大小写？（ISO 6.1.2）

是。

A.5 字符

ANSI C 标准第 G.3.4 节涵盖了字符的实现定义的行为。

详细说明标准未明确指定的任何源代码字符和执行字符？（ISO 5.2.1）

无。

列出对于所列序列生成的转义序列值。（ISO 5.2.2）

表 A-1： 转义序列字符和值

序列	值
\a	7
\b	8
\f	12
\n	10
\r	13
\t	9
\v	11

执行字符集中的字符有多少个位？（ISO 5.2.4.2）

8.

源代码字符集的成员（在字符和字符串字面值中）与执行字符集的成员如何映射？（ISO 6.1.3.4）

恒等函数。

普通 char 的等效类型是什么？（ISO 6.2.1.1）

两种中的任意一种（由用户定义）。默认设置为 `signed char`。可以使用编译器命令行选项将默认设置设为 `unsigned char`。

A.6 整型

ANSI C 标准第 G.3.5 节涵盖了整型的实现定义的行为。

下表说明了各种整型类型的存储长度和范围：（ISO 6.1.2.5）

表 A-2： 整型类型

名称	长度 （位）	范围
char	8	-128 … 127
signed char	8	-128 127
unsigned char	8	0 … 255
short	16	-32768 … 32767
signed short	16	-32768 32767
unsigned short	16	0 65535
int	16	-32768 32767
signed int	16	-32768 32767
unsigned int	16	0 65535
long	32	-2147483648 2147438647
signed long	32	-2147483648 2147438647
unsigned long	32	0 4294867295

在值无法表示的情况下，整型转换为较短的有符号整型的结果是什么，或者无符号整型转换为同等长度的有符号整型的结果是什么？（ISO 6.2.1.2）

有效位数会减少。不会发出任何错误。

对有符号整型的按位操作的结果是什么？（ISO 6.3）

移位操作符会保留符号。其他操作符会将操作数当作无符号整型处理。

整型除法的余数的符号是什么？（ISO 6.3.5）

+

负值有符号整型的右移操作的结果是什么？（ISO 6.3.7）

保留符号。

© 2015 Microchip Technology Inc.

A.7 浮点型

ANSI C 标准第 G.3.6 节涵盖了浮点型的实现定义的行为。

浮点型常量的比例换算值（位于其对应类型的可表示值范围内）是最接近的可表示值、紧邻最接近可表示值的较大可表示值，还是紧邻最接近可表示值的最小可表示值？（ISO 6.1.3.1）

最接近的可表示值。

下表说明了各种浮点数类型的存储长度和范围：（ISO 6.1.2.5）

表 A-3： 浮点型

名称	长度（位）	范围
`float`	32	1.175494e-38 … 3.40282346e+38
`double*`	32	1.175494e-38 3.40282346e+38
`long double`	64	2.22507385e-308 … 1.79769313e+308

* 如果使用了 `-fno-short-double`，则 `double` 等效于 `long double`。

当整型数转换为不能精确表示原始值的浮点型数时的截断方向是什么？（ISO 6.2.1.3）

向下。

当浮点型数转换为较短的浮点型数时的截断或舍入方向是什么？（ISO 6.2.1.4）

向下。

A.8 数组和指针

ANSI C 标准第 G.3.7 节涵盖了数组和指针的实现定义的行为。

存放数组最大大小所需的整型的类型是什么，也即，大小操作符 `size_t` 的类型是什么？（ISO 6.3.3.4、ISO 7.1.1）

`unsigned int`。

要转换为整型类型的指针所需的整型的大小是多少？（ISO 6.3.4）

16 位。

将指针强制转换为整型或反之的结果是什么？（ISO 6.3.4）

映射是恒等函数。

两个指针指向同一数组中的成员，存放两个指针之差所需的整型的类型 `ptrdiff_t` 是什么？（ISO 6.3.6、ISO 7.1.1）

`unsigned int`。

A.9 寄存器

ANSI C 标准第 G.3.8 节涵盖了寄存器的实现定义的行为。

存储类说明符 `register` 对于对象在寄存器中的存储会产生什么程度的实际影响？（ISO 6.5.1）

如果禁止优化，则会尝试遵从 `register` 存储类；否则，它会被忽略。

A.10 结构、联合、枚举和位域

ANSI C 标准的第 A.6.3.9 节和第 G.3.9 节涵盖了结构、联合、枚举和位域的实现定义的行为。

访问联合对象的成员时，使用不同类型的成员的结果是什么？（ISO 6.3.2.3）

不应用任何转换。

描述结构成员的填充和对齐？（ISO 6.5.2.1）

字符按字节对齐。所有其他对象按字对齐。

普通 `int` 位域的等效类型是什么？（ISO 6.5.2.1）

用户定义的。默认情况下，为 `signed int` 位域。可以使用命令行选项设为 `unsigned int` 位域。

位域在 int 中的分配顺序是什么？（ISO 6.5.2.1）

位从最低位到最高位分配。

位域可以跨越存储单元边界吗？（ISO 6.5.2.1）

是。

选择使用哪种整型类型来表示枚举类型的值？（ISO 6.5.2.2）

`int`。

A.11 限定符

ANSI C 标准第 G.3.10 节涵盖了限定符的实现定义的行为。

描述什么操作构成对 volatile 限定类型的对象的访问？（ISO 6.5.3）

如果在表达式中指定了某个对象，则说明它已被访问。

A.12 声明符

ANSI C 标准第 G.3.11 节涵盖了声明符的实现定义的行为。

可修改算术、结构或联合类型的声明符的最大数量是什么？（ISO 6.5.4）

无限制。

© 2015 Microchip Technology Inc.

A.13 语句

ANSI C 标准第 G.3.12 节涵盖了语句的实现定义的行为。

switch 语句中 case 分支值的最大数量是多少？（ISO 6.6.4.2）

无限制。

A.14 预处理伪指令

ANSI C 标准第 G.3.13 节涵盖了预处理伪指令的实现定义的行为。

控制条件包含的常量表达式中单字符字符常量的值是否与执行字符集中相同字符常量的值匹配？（ISO 6.8.1）

是。

此类字符常量是否可以具有负值？（ISO 6.8.1）

是。

用什么方法来查找可包含的源文件？（ISO 6.8.2）

预处理器搜索当前目录，紧接着搜索使用命令行选项指定的目录。

如何标识头文件，或如何指定位置？（ISO 6.8.2）

头文件在 `#include` 伪指令中标识，包含在 < 和 > 定界符之间，或在 " 和 " 定界符之间。位置使用命令行选项指定。

可包含源文件是否支持加引号的名称？（ISO 6.8.2）

是。

带定界符的字符序列和外部源文件名之间的映射是什么？（ISO 6.8.2）

恒等函数。

描述每个可识别的 #pragma 伪指令的行为。（ISO 6.8.6）

表 A-4： #PRAGMA 伪指令行为

Pragma 伪指令	行为
`#pragma code section-name`	命名代码段。
`#pragma code`	将代码段的名称复位为其默认值（即 `.text`）。
`#pragma config`	设置配置位或寄存器。
`#pragma idata section-name`	命名初始化数据段。
`#pragma idata`	将初始化数据段的名称复位为其默认值（即，`.data`）。
`#pragma udata section-name`	命名未初始化数据段。
`#pragma udata`	将未初始化数据段的名称复位为其默认值（即，`.bss`）。
`#pragma interrupt function-name`	将 function-name 指定为中断函数。

在翻译的日期和时间不可用时，__DATE__ 和 __TIME__ 的定义分别是什么？（ISO 6.8.8）

不适用。在这些函数不可用的环境中不支持该编译器。

A.15 库函数

ANSI C 标准第 G.3.14 节涵盖了库函数的实现定义的行为。

NULL 宏扩展为的空指针常量是什么？（ISO 7.1.5）

0.

如何识别由 assert 函数打印的诊断信息，该函数的终止行为是什么？（ISO 7.2）

assert 函数会打印文件名、行号和测试表达式，用冒号字符（“:”）分隔。然后，它会调用 abort 函数。

由 isalnum、isalpha、iscntrl、islower、isprint 和 isupper 函数测试的字符集是什么？（ISO 7.3.1）

表 A-5： IS 函数测试的字符

函数	测试的字符
isalnum	字母或数字之一：isalpha 或 isdigit。
isalpha	字母之一：islower 或 isupper。
iscntrl	5 个标准运动控制字符、退格键和报警字符之一：\f、\n、\r、\t、\v、\b 和 \a。
islower	字母 “a” 至 “z” 之一。
isprint	图形字符或空格字符：isalnum 或 ispunct 或 space。
isupper	字母 “A” 至 “Z” 之一。
ispunct	字符之一：!"#%&'();<=>?[\]*+,-./:^

数学函数在域错误后返回的值是什么？（ISO 7.5.1）

NaN。

在发生下溢范围错误时，数学函数是否将整型表达式 errno 设置为宏 ERANGE 的值？（ISO 7.5.1）

是。

当 fmod 函数的第二个参数为零时，是收到域错误还是返回零？（ISO 7.5.6.4）

域错误。

© 2015 Microchip Technology Inc.

A.16 信号

信号函数的信号集是什么？（ISO 7.7.1.1）

表 A-6： 信号函数

名称	说明
SIGABRT	异常终止。
SIGINT	接收到交互式注意信号。
SIGILL	检测到无效的函数映像。
SIGFPE	错误的算术运算。
SIGSEGV	对存储的访问无效。
SIGTERM	向程序发送了终止请求。

描述信号函数识别的每个信号的参数和用法。（ISO 7.7.1.1）

由应用程序定义。

描述对于信号函数识别的每个信号的默认处理和在程序启动时的处理？（ISO 7.7.1.1）

无。

如果在调用信号处理函数之前未执行 signal (sig,SIG_DFL) 的等效函数，则执行哪种信号阻塞？（ISO 7.7.1.1）

无。

如果指定给信号函数的处理程序接收到 SIGILL 信号，是否复位默认处理？（ISO 7.7.1.1）

否。

A.17 流和文件

文本流的最后一行是否需要一个终止换行符？（ISO 7.9.2）

否。

写到文本流中、紧接在换行符前面的空格字符在读回流时是否出现？（ISO 7.9.2）

是。

可以在写入二进制流的数据后附加多少个空字符？（ISO 7.9.2）

无。

附加模式流的文件位置说明符最初位于文件开始还是末尾？（ISO 7.9.3）

开始。

对文本流的写操作是否导致关联的文件在该点之后被截断？（ISO 7.9.3）

由应用程序定义。

描述文件缓冲的特性。（ISO 7.9.3）

完全缓冲。

零长度文件是否可以实际存在？（ISO 7.9.3）

是。

构造有效文件名的规则是什么？（ISO 7.9.3）

由应用程序定义。

同一文件是否可以打开多次？（ISO 7.9.3）

由应用程序定义。

remove 函数对已打开文件会有什么影响？（ISO 7.9.4.1）

由应用程序定义。

如果在调用 rename 函数之前，已存在具有新名称的文件，会有什么影响？（ISO 7.9.4.2）

由应用程序定义。

`fprintf` 函数中的 `%p` 转换的输出形式是什么？（ISO 7.9.6.1）

十六进制表示形式。

`fscanf` 函数中的 `%p` 转换采用什么输入形式？（ISO 7.9.6.2）

十六进制表示形式。

© 2015 Microchip Technology Inc.

A.18 TMPFILE

如果程序异常终止，是否删除已打开的临时文件？（ISO 7.9.4.3）

是。

A.19 ERRNO

在发生失败时，`fgetpos` 或 `ftell` 函数将宏 `errno` 设置为什么值？（ISO 7.9.9.1 和 ISO 7.9.9.4）

由应用程序定义。

由 `perror` 函数生成的消息的格式是什么？（ISO 7.9.10.4）

`perror` 的参数后面跟随一个冒号，后面再跟随 `errno` 值的文本说明。

A.20 存储器

如果请求的大小为零，`calloc`、`malloc` 或 `realloc` 函数的行为是什么？（ISO 7.10.3）

分配长度为零的块。

A.21 ABORT

调用 `abort` 函数时，已打开文件和临时文件会发生什么？（ISO 7.10.4.1）

不会发生任何事。

A.22 EXIT

如果参数值为零、EXIT_SUCCESS 或 EXIT_FAILURE 之外的值，exit 函数返回的状态是什么？（ISO 7.10.4.3）

参数的值。

A.23 GETENV

对环境名称有什么限制？（ISO 7.10.4.4）

由应用程序定义。

描述用于更改通过调用 `getenv` 函数获得的环境列表的方法。（ISO 7.10.4.4）

由应用程序定义。

A.24 SYSTEM

描述传递给 system 函数的字符串的格式。（ISO 7.10.4.5）

由应用程序定义。

system 函数执行哪种执行模式？（ISO 7.10.4.5）

由应用程序定义。

A.25 STRERROR

描述 `strerror` 函数的错误消息输出的格式。*（ISO 7.11.6.2）*

普通字符串。

列出对 `strerror` *函数的调用返回的错误消息字符串的内容。（ISO 7.11.6.2）*

表 A-7： 错误消息字符串

Errno	消息
0	无错误
EDOM	域错误
ERANGE	范围错误
EFPOS	文件定位错误
EFOPEN	文件打开错误
nnn	错误号 nnn

© 2015 Microchip Technology Inc.

附录 B 嵌入式编译器兼容性模式

B.1 简介

全部 3 种 MPLAB XC C 编译器都可以置为兼容性模式。在该模式下，它们将在语法上与其他非 Microchip 嵌入式编译器供应商所使用的非标准 C 语言扩展兼容。这种兼容性使针对其他编译器编写的 C 源代码在使用 MPLAB XC 编译器时只需最低限度的修改。

由于其他编译器可能针对的是差别很大的器件架构，所以非标准扩展的语义可能与 MPLAB XC 编译器中不同。本文档将说明何时可能需要复核原始 C 代码。

以下主题将介绍 MPLAB C 编译器提供的兼容性功能：

- 在兼容性模式下编译
- 语法兼容性
- 数据类型
- 操作符
- 扩展关键字
- 固有函数
- pragma 伪指令

B.2 在兼容性模式下编译

有一个选项用于使能特定于供应商的语法兼容性。使用 MPLAB XC8 时，该选项为 `--ext=vendor`；使用MPLAB XC16或MPLAB XC32时，该选项为 `-mext=vendor`。参数 *vendor* 是用于表示语法的键名。关于可用于 MPLAB XC 编译器的所有键名的列表，请参见表 B-1。

表 B-1： 供应商键名

供应商键名	语法	XC8 支持	XC16 支持	XC32 支持
cci	通用 C 接口	是	是	是
iar	IAR C/C++ Compiler for ARM	是	是	是

通用 C 接口是一种对所有 Microchip MPLAB XC 编译器通用的语言标准。**第 2 章 “通用 C 接口”**已经介绍了与该语法关联的非标准扩展，这里不再重复。

B.3 语法兼容性

该语法兼容性功能的目标是，使其他 C 编译器的源代码移植为本机 MPLAB XC 编译器语法时的移植过程变得方便。

针对 Microchip 器件进行编译时，许多非标准扩展是不需要的，对于它们 MPLAB XC 编译器未提供等效的扩展。 MPLAB XC 编译器将只是简单地忽略这些扩展，虽然通常会产生一条警告消息，以确保您了解不同的编译器行为。您应确认您的项目在禁止这些功能时仍然可以正确工作。

其他非标准扩展与 Microchip 器件不兼容。如果未从源代码中删除这些扩展， MPLAB XC 编译器将产生错误。您应复核删除扩展的结果，并决定是否需要对项目中的其他源代码进行更改。

表 B-2 列出了本指南的表格中使用的各种兼容性级别。

表 B-2： 支持级别指示符

级别	说明
支持	语法在指定兼容性模式下是可接受的，并且其含义将仿效它用于原始编译器时的含义。
支持 （无参数）	对于 pragma 伪指令，在指定的兼容性模式下支持基本 pragma 伪指令，但参数会被忽略。
本机支持	语法等效于已被 MPLAB XC 编译器接受的语法，并且语义是兼容的。您可以在不使能供应商兼容性模式的情况下使用该功能。
忽略	语法在指定的兼容性模式下是可接受的，但不需要或不执行所隐含的操作。该扩展会被忽略，编译器将发出警告。
错误	指定兼容性模式下不接受该语法。将会发出错误，编译将会终止。

请注意，即使 MPLAB XC 编译器支持某个 C 功能，在针对 Microchip 器件进行编译时，地址、寄存器名称、汇编指令或任何其他特定于器件的参数也很可能不是有效的。请总是配合您所使用目标 Microchip 器件的数据手册复核使用这些项的代码。

© 2015 Microchip Technology Inc.

B.4 数据类型

一些编译器允许使用布尔类型 `bool` 以及关联的值 `true` 和 `false`，如 C99 ANSI 标准所规定。如表 B-3 所示，在兼容模式[1] 下，该类型和这些值可由所有 MPLAB XC 编译器使用。

如 ANSI 标准所规定，在用于 MPLAB XC 编译器时，必须包含 `<stdbool.h>` 头文件，该功能才能按预期工作。

表 B-3： 对 C99 BOOL 类型的支持

IAR 兼容性模式			
类型	**XC8**	**XC16**	**XC32**
`bool`	支持	支持	支持

不要将布尔类型 `bool` 与 MPLAB XC8 实现的整型类型 `bit` 混淆。

B.5 操作符

@操作符可用于其他编译器，用以指示对象所需的存储地址。如表 B-4 所示，各 MPLAB C 编译器中只有 MPLAB XC8 支持该语法。

对于其他器件指定的任何地址很可能在新架构上是不正确的。请配合目标 Microchip 器件数据手册来复核地址。

在兼容模式下，对 MPLAB XC8 使用 @ 将正常工作，但会产生警告。要防止再次出现该警告，请改为通过 MPLAB C `__at()` 说明符使用经过复核的地址。

对于 MPLAB XC16/32，可以考虑使用 `address` 属性。

表 B-4： 对非标准操作符的支持

IAR 兼容性模式			
操作符	**XC8**	**XC16**	**XC32**
@	本机支持	错误	错误

1. 不是所有 Microchip MPLAB XC 编译器都采用了所有 C99 功能。

B.6 扩展关键字

非标准扩展通常会指定如何定义或访问对象。通常使用关键字来指示该功能。表 B-5 列出了对应于其他编译器的非标准 C 关键字，以及 MPLAB XC 编译器提供的兼容性级别。表的注释中提供了关于某些扩展的更多信息。

表 B-5： 对非标准关键字的支持

IAR 兼容性模式			
关键字	**XC8**	**XC16**	**XC32**
__section_begin	忽略	支持	支持
__section_end	忽略	支持	支持
__section_size	忽略	支持	支持
__segment_begin	忽略	支持	支持
__segment_end	忽略	支持	支持
__segment_size	忽略	支持	支持
__sfb	忽略	支持	支持
__sfe	忽略	支持	支持
__sfs	忽略	支持	支持
__asm 或 asm(1)	支持 (2)	本机支持	本机支持
__arm	忽略	忽略	忽略
__big_endian	错误	错误	错误
__fiq	支持	错误	错误
__intrinsic	忽略	忽略	忽略
__interwork	忽略	忽略	忽略
__irq	支持	错误	错误
__little_endian(3)	忽略	忽略	忽略
__nested	忽略	忽略	忽略
__no_init	支持	支持	支持
__noreturn	忽略	支持	支持
__ramfunc	忽略	忽略	支持 (4)
__packed	忽略 (5)	支持	支持
__root	忽略	支持	支持
__swi	忽略	忽略	忽略
__task	忽略	支持	支持
__weak	忽略	支持	支持
__thumb	忽略	忽略	忽略
__farfunc	忽略	忽略	忽略
__huge	忽略	忽略	忽略
__nearfunc	忽略	忽略	忽略
__inline	支持	本机支持	本机支持

注 1： 通过该构造指定的所有汇编代码都是特定于器件的，移植到任何 Microchip 器件时都需要复核。

2： MPLAB XC8 本机支持关键字 asm，但该编译器在 IAR 兼容性模式下仅支持 __asm 关键字。

3： 这是所有 MPLAB XC 编译器使用的默认 （也是唯一的）尾数法。

4： 用于 MPLAB XC32 时，它必须与 __longcall__ 宏配合使用才会完全兼容。

5： 虽然该关键字会被忽略，但默认情况下，在使用 MPLAB XC8 时所有结构都会压缩，所以不会损失任何功能。

© 2015 Microchip Technology Inc.

B.7 固有函数

固有函数可以用于在源代码中执行常见的任务。表 B-6 列出了 MPLAB XC 编译器对于由其他编译器提供的固有函数的支持。

表 B-6： 对非标准固有函数的支持

IAR 兼容性模式			
函数	**XC8**	**XC16**	**XC32**
__disable_fiq[1]	支持	忽略	忽略
__disable_interrupt	支持	支持	支持
__disable_irq[1]	支持	忽略	忽略
__enable_fiq[1]	支持	忽略	忽略
__enable_interrupt	支持	支持	支持
__enable_irq[1]	支持	忽略	忽略
__get_interrupt_state	忽略	支持	支持
__set_interrupt_state	忽略	支持	支持

注 1： 这些固有函数映射到一些宏，这些宏禁止或使能 8 位 PIC® 器件上的全局中断允许位。为使受支持的函数正常工作，必须包含头文件 <xc.h>。

B.8 PRAGMA 伪指令

编译器可以使用 pragma 伪指令来控制代码生成。所有编译器都会忽略未知的 pragma 伪指令，但 MPLAB XC 编译器在兼容性模式下也实现由其他编译器实现的许多 pragma 伪指令。表 B-7 列出了 pragma 伪指令和在使用每种 MPLAB XC 编译器时的支持级别。

其中许多 pragma 伪指令都带有参数。即使 MPLAB XC 编译器支持某个 pragma 伪指令，这种支持可能并不适用于 pragma 伪指令的所有参数。下表说明了这一点。

表 B-7： 对非标准 PRAGMA 伪指令的支持

IAR 兼容性模式			
Pragma 伪指令	**XC8**	**XC16**	**XC32**
bitfields	忽略	忽略	忽略
data_alignment	忽略	支持	支持
diag_default	忽略	忽略	忽略
diag_error	忽略	忽略	忽略
diag_remark	忽略	忽略	忽略
diag_suppress	忽略	忽略	忽略
diag_warning	忽略	忽略	忽略
include_alias	忽略	忽略	忽略
inline	支持 （无参数）	支持 （无参数）	支持 （无参数）
language	忽略	忽略	忽略
location	忽略	支持	支持
message	支持	本机支持	本机支持

表 B-7： 对非标准 PRAGMA 伪指令的支持 （续）

IAR 兼容性模式			
Pragma 伪指令	**XC8**	**XC16**	**XC32**
object_attribute	忽略	忽略	忽略
optimize	忽略	本机支持	本机支持
pack	忽略	本机支持	本机支持
__printf_args	支持	支持	支持
required	忽略	支持	支持
rtmodel	忽略	忽略	忽略
__scanf__args	忽略	支持	支持
section	忽略	支持	支持
segment	忽略	支持	支持
swi_number	忽略	忽略	忽略
type_attribute	忽略	忽略	忽略
weak	忽略	本机支持	本机支持

© 2015 Microchip Technology Inc.

附录 C 诊断

C.1 简介

本附录列出了 MPLAB XC16 C 编译器生成的最常见诊断消息。

编译器可以产生两种诊断消息：错误和警告。每种诊断消息具有不同的用途。

- 错误消息报告致使您的程序无法编译的问题。当问题比较明显时，编译器在报告错误时还会给出源文件名和行号。
- 警告消息报告您的代码中可能表明存在问题的其他异常条件，虽然编译可以（也确实）继续进行。警告消息还会报告源文件名和行号，但其中带有文本“`warning:`”，以与错误消息区别开来。

  警告可能指示应检查的危险点，以确保您的程序按指示执行。警告可能指示使用了编译器的过时功能或非标准功能。许多警告只有在您通过 `-W` 选项之一请求产生它们时才会发出 （例如， `-Wall` 会请求产生一系列有用的警告）。

在极少数情况下，编译器可能会发出内部错误消息报告。这表明编译器本身检测到了应向 Microchip 支持人员报告的故障。前言给出了与支持人员联系的详细信息。

C.2 错误

符号

\x used with no following HEX digits （使用 \x 时后面没有跟随十六进制数字）

转义序列 \x 后应跟随十六进制数字。

‘&’ constraint used with no register class （使用 “&”约束时没有给出寄存器类）

asm 语句是无效的。

‘%’ constraint used with last operand （对最后一个操作数使用了 “%”约束）

asm 语句是无效的。

#elif after #else （#else 后存在 #elif）

在预处理器条件语句中， #else 子句必须出现在任何 #elif 子句之后。

#elif without #if （有 #elif 却没有 #if）

在预处理器条件语句中，在使用 #elif 之前必须使用 #if。

#else after #else （#else 后存在 #else）

在预处理器条件语句中， #else 子句只能出现一次。

#else without #if （有 #else 却没有 #if）

在预处理器条件语句中，在使用 #else 之前必须使用 #if。

#endif without #if （有 #endif 却没有 #if）

在预处理器条件语句中，在使用 #endif 之前必须使用 #if。

#error '*message*' （# 错误 “消息”）

该错误出现在 #error 伪指令的响应中。

#if with no expression （#if 不带有表达式）

需要一个求值结果为常量算术值的表达式。

#include expects “FILENAME” or <FILENAME> （#include 需要 “FILENAME” 或 <FILENAME>）

#include 的文件名缺失或不完整。它必须用引号或尖括号括住。

'#' is not followed by a macro parameter （“#”后没有跟随宏参数）

字符串长度操作符 “#”后面必须跟随宏参数名。

'#*keyword*' expects “FILENAME” or <FILENAME> （“#keyword”需要 “FILENAME” 或 <FILENAME>）

指定的 “*#keyword*”需要一个带引号或括号的文件名作为参数。

'#' is not followed by a macro parameter （“#”后没有跟随宏参数）

操作符 “#”后面应跟随宏参数名。

'##' cannot appear at either end of a macro expansion （“##”不能出现在宏扩展的任意一端）

连接操作符 “##”不能出现在宏扩展的开头或结尾。

A

a parameter list with an ellipsis can't match an empty parameter name list declaration （带省略号的参数列表不能匹配空参数名称列表声明）

函数的声明和定义必须一致。

“*symbol*” after #line is not a positive integer （#line 后的符号 “*symbol*” 不是一个正整数）

#line 需要一个源代码行号，它必须为正数。

aggregate value used where a complex was expected （在需要复数值的位置使用了聚合值）

不要在需要复数值的位置使用聚合值。

aggregate value used where a float was expected （在需要浮点值的位置使用了聚合值）

不要在需要浮点值的位置使用聚合值。

aggregate value used where an integer was expected （在需要整型值的位置使用了聚合值）

不要在需要整型值的位置使用聚合值。

alias arg not a string （别名参数不是字符串）

alias 属性的参数必须为一个字符串，该字符串用于指定一个目标，当前标识符为该目标的别名。

alignment may not be specified for '*identifier*' （不能为标识符 “*identifier*” 指定对齐量）

aligned 属性只能用于变量。

'__alignof' applied to a bit-field （“__alignof” 被应用于一个位域）

“`__alignof`”操作符不能应用于位域。

alternate interrupt vector is not a constant （备用中断向量不是常量）

中断向量编号必须是一个整型常量。

alternate interrupt vector number *n* is not valid （备用中断向量编号 *n* 无效）

需要一个有效的中断向量编号。

© 2015 Microchip Technology Inc.

ambiguous abbreviation argument （存在歧义的缩写参数）

指定的命令行缩写存在歧义。

an argument type that has a default promotion can't match an empty parameter name list declaration. （具有默认提升的参数类型不能匹配空参数名称列表声明。）

函数的声明和定义必须一致。

args to be formatted is not ... （要格式化的参数不是 ...）

format 属性的 first-to-check 索引参数指定了一个不是声明为 "…" 的参数。

argument '*identifier*' doesn't match prototype（参数 "*identifier*" 与原型不匹配）

函数参数类型应与函数的原型匹配。

argument of 'asm' is not a constant string （"asm" 的参数不是常量字符串）

"asm" 的参数必须为常量字符串。

argument to '-B' is missing （缺少 "-B" 的参数）

缺少目录名称。

argument to '-l' is missing （缺少 "-l" 的参数）

缺少库名称。

argument to '-specs' is missing （缺少 "-specs" 的参数）

缺少说明文件的名称。

argument to '-x' is missing （缺少 "-x" 的参数）

缺少语言名称。

argument to '-Xlinker' is missing （缺少 "-Xlinker" 的参数）

缺少要传递给链接器的参数。

arithmetic on pointer to an incomplete type （对指向不完整类型的指针执行算术运算）

不允许对指向不完整类型的指针执行算术运算。

array index in non-array initializer （在非数组初始值设定项中使用数组索引）

不要在非数组初始值设定项中使用数组索引。

array size missing in '*identifier*' （"*identifier*" 中缺少数组大小）

缺少数组大小。

array subscript is not an integer （数组下标不是一个整数）

数组下标必须是整数。

'asm' operand constraint incompatible with operand size （"asm" 操作数约束与操作数大小不兼容）

asm 语句是无效的。

'asm' operand requires impossible reload（"asm" 操作数要求不可能的重新装入）

asm 语句是无效的。

asm template is not a string constant （asm 模板不是字符串常量）

asm 模板必须为字符串常量。

assertion without predicate （断言没有谓词）

#assert 或 #unassert 后面必须跟随一个谓词，它必须是单个标识符。

'*attribute*' attribute applies only to functions （"*attribute*" 属性仅适用于函数）

属性 "*attribute*" 只能应用于函数。

B

bit-field ‘*identifier*’ has invalid type （位域 “*identifier*” 具有无效的类型）

位域必须为枚举或整型类型。

bit-field ‘*identifier*’ width not an integer constant （位域 “*identifier*” 的宽度不是一个整型常量）

位域宽度必须为整型常量。

both long and short specified for ‘*identifier*’（同时为 “*identifier*” 指定了 long 和 short）

变量不能既是 long 类型又是 short 类型。

both signed and unsigned specified for ‘*identifier*’ （同时为 “*identifier*” 指定了 signed 和 unsigned）

变量不能既为有符号又为无符号。

braced-group within expression allowed only inside a function（只有在函数内才允许具有处于表达式内的花括号组合）

在函数外具有处于表达式内的花括号组合是非法的。

break statement not within loop or switch （break 语句不处于循环或 switch 内）

break 语句只能在循环或 switch 内使用。

__builtin_longjmp second argument must be 1（builtin_longjmp 第二个参数必须为 1）

`__builtin_longjmp` 要求它的第二个参数为 1。

C

called object is not a function （被调用的对象不是一个函数）

在 C 中只有函数才可以进行调用。

cannot convert to a pointer type （无法转换为指针类型）

无法将表达式转换为指针类型。

cannot put object with volatile field into register（不能将具有可变位域的对象放入寄存器）

将具有可变位域的对象放入寄存器是非法的。

cannot reload integer constant operand in ‘asm’（不能在 “asm” 中重新装入整型常量操作数）

asm 语句是无效的。

cannot specify both near and far attributes （不能同时指定 near 和 far 属性）

属性 near 和 far 是互斥的，对于函数或变量只能使用一个。

cannot take address of bit-field ‘*identifier*’ （无法获取位域 “*identifier*” 的地址）

试图获取位域的地址是非法的。

can’t open ‘*file*’ for writing （无法打开文件 “*file*” 进行写入）

系统无法打开指定的 “*file*” 。可能原因是没有足够的磁盘空间来打开文件、目录不存在，或目标目录没有写权限。

can’t set ‘*attribute*’ attribute after definition（不能在定义之后设置 “*attribute*” 属性）

“*attribute*” 属性必须在定义符号时使用。

case label does not reduce to an integer constant（case 标号不能化简为一个整型常量）

case 标号必须为编译时整型常量。

case label not within a switch statement （case 标号不处于 switch 语句内）

case 标号必须处于 switch 语句内。

cast specifies array type （强制类型转换指定了数组类型）

不允许强制类型转换指定数组类型。

© 2015 Microchip Technology Inc.

cast specifies function type （强制类型转换指定了函数类型）

不允许强制类型转换指定函数类型。

cast to union type from type not present in union （从联合中不存在的类型强制转换为联合类型）

在强制转换为联合类型时，请从联合中存在的类型进行转换。

char-array initialized from wide string （char 数组使用宽字符串初始化）

不应使用宽字符串初始化 char 数组。请使用普通字符串。

***file: compiler* compiler not installed on this system（该系统上未安装编译器“*file: compiler*”）**

仅分发了 C 编译器；不支持其他高级语言。

complex invalid for ‘*identifier*’ （复数对于 “*identifier*” 无效）

复数限定符只能应用于整型和浮点类型。

conflicting types for ‘*identifier*’ （“*identifier*” 存在冲突的类型）

该标识符存在多个不一致的声明。

continue statement not within loop （continue 语句不处于循环内）

continue 语句只能在循环内使用。

conversion to non-scalar type requested （请求转换为非标量类型）

类型转换必须转换为标量 （非聚合）类型。

D

data type of ‘*name*’ isn’t suitable for a register（“*name*” 的数据类型不适合于寄存器）

数据类型无法装入请求的寄存器。

declaration for parameter ‘*identifier*’ but no such parameter （声明了参数 “*identifier*”，但不存在此参数）

只能声明处于参数列表中的参数。

declaration of ‘*identifier*’ as array of functions （“*identifier*” 声明为函数的数组）

函数数组是非法的。

declaration of ‘*identifier*’ as array of voids （“*identifier*” 声明为 void 的数组）

void 的数组是非法的。

‘*identifier*’ declared as function returning a function （“*identifier*” 被声明为返回一个函数的函数）

函数不能返回函数。

‘*identifier*’ declared as function returning an array （“*identifier*” 被声明为返回一个数组的函数）

函数不能返回数组。

decrement of pointer to unknown structure （递减指向未知结构的指针）

不要递减指向未知结构的指针。

‘default’ label not within a switch statement（“default”标号不处于switch语句内）

default case 标号必须处于 switch 语句内。

‘*symbol*’ defined both normally and as an alias （同时以正常方式和别名方式定义了符号 “*symbol*”）

如果 “symbol” 已定义，则不能将它用作另一个符号的别名。）

‘defined’ cannot be used as a macro name （“defined” 不能用作宏的名称）

宏名称不能名为 “defined”。

dereferencing pointer to incomplete type （解引用指向不完整类型的指针）

进行解引用的指针不能为指向不完整类型的指针。

division by zero in #if （在 #if 中被零除）

被 0 除是不可计算的。

duplicate case value （重复的 case 值）

case 值必须是唯一的。

duplicate label '*identifier*' （重复的标号 "*identifier*"）

标号必须在其作用域内唯一。

duplicate macro parameter '*symbol*' （重复的宏参数 "*symbol*"）

"symbol" 在参数列表中多次使用。

duplicate member '*identifier*' （重复的成员 "*identifier*"）

结构不能具有重复的成员。

duplicate (or overlapping) case value （重复 （或重叠）的 case 值）

case 范围不能具有重复或重叠的值。错误消息 "这是与该值重叠的第一个条目" 将提供第一处出现重复或重叠值的位置。 case 范围是编译器对于 ANSI 标准的扩展。

E

elements of array '*identifier*' have incomplete type （数组元素 "*identifier*" 具有不完整的类型）

数组元素应具有完整的类型。

empty character constant （空的字符常量）

空的字符常量是非法的。

empty file name in '#*keyword*' （"#*keyword*" 中存在空文件名）

指定为指定 #keyword 的参数的文件名是空的。

empty index range in initializer （初始值设定项中存在空的索引范围）

不要在初始值设定项中使用空的索引范围

empty scalar initializer （空的标量初始值设定项）

标量初始值设定项不能为空。

enumerator value for '*identifier*' not integer constant（"*identifier*" 的枚举值不是整型常量）

枚举值必须为整型常量。

error closing '*file*' （关闭文件 "*file*" 时出错）

系统无法关闭指定的 "*file*"。可能原因是没有足够的磁盘空间来写入该文件或该文件太大。

error writing to '*file*' （写入文件 "*file*" 时出错）

系统无法写入指定的 "*file*"。可能原因是没有足够的磁盘空间来写入该文件或该文件太大。

excess elements in char array initializer （char 数组初始值设定项中存在过多的元素）

列表中的元素数量多于初始值设定项指定的元素数量。

excess elements in struct initializer （结构初始值设定项中存在过多的元素）

不要在结构初始值设定项中使用过多的元素。

expression statement has incomplete type （表达式语句具有不完整的类型）

表达式的类型是不完整的。

extra brace group at end of initializer （初始值设定项的末尾存在额外的花括号组合）

不要在初始值设定项的末尾放置额外的花括号组合。

© 2015 Microchip Technology Inc.

extraneous argument to '*option*' option （选项 "*option*" 存在无关的参数）

传递给指定命令行选项的参数太多。

F

'*identifier*' fails to be a typedef or built in type（"*identifier*" 未能设为某种 typedef 类型或内置类型）

数据类型必须为某种 typedef 类型或内置类型。

field '*identifier*' declared as a function （位域 "*identifier*" 被声明为函数）

位域不能声明为函数。

field '*identifier*' has incomplete type （位域 "*identifier*" 具有不完整的类型）

位域必须具有完整的类型。

first argument to __builtin_choose_expr not a constant（__builtin_choose_expr 的第一个参数不是常量）

第一个参数必须是可以在编译时确定的常量表达式。

flexible array member in otherwise empty struct （结构中除灵活数组成员之外为空）

灵活数组成员必须是具有多个命名成员的结构的最后一个元素。

flexible array member in union （联合中存在灵活数组成员）

联合中不能使用灵活数组成员。

flexible array member not at end of struct （灵活数组成员不是位于结构末尾）

灵活数组成员必须是结构的最后一个元素。

'for' loop initial declaration used outside C99 mode（在 C99 模式外使用 "for" 循环初始声明）

在 C99 模式外，"for" 循环初始声明是无效的。

format string arg follows the args to be formatted （格式字符串参数跟随在要格式化的参数之后）

format 属性的参数不一致。格式字符串参数索引必须小于要检查的第一个参数的索引。

format string arg not a string type （格式字符串参数不是字符串类型）

format 属性的格式字符串索引参数指定了一个不是字符串类型的参数。

format string has invalid operand number （格式字符串具有无效的操作数数字）

format 属性的操作数数字参数必须是一个编译时常量。

function definition declared 'register' （函数定义声明了 "register"）

函数定义不能声明 "register"。

function definition declared 'typedef' （函数定义声明了 "typedef"）

函数定义不能声明 "typedef"。

function does not return string type （函数不返回字符串类型）

format_arg 属性只能用于返回值是字符串类型的函数。

function '*identifier*' is initialized like a variable （函数 "identifier" 以类似于变量的方式初始化）

以类似于变量的方式初始化函数是非法的。

function return type cannot be function （函数的返回类型不能是函数）

函数的返回类型不能是函数。

G

global register variable follows a function definition（全局寄存器变量跟随在函数定义后）

全局寄存器变量应在函数定义之前。

global register variable has initial value （全局寄存器变量具有初始值）

不要为全局寄存器变量指定初始值。

global register variable '*identifier*' used in nested function （在嵌套函数中使用了全局寄存器变量 "*identifier*"）

不要在嵌套函数中使用全局寄存器变量。

H

'*identifier*' has an incomplete type （"*identifier*" 具有不完整的类型）

对于指定的 "*identifier*"，具有不完整的类型是非法的。

'*identifier*' has both 'extern' and initializer（"*identifier*" 同时具有 "extern" 和初始值设定）

不能初始化声明为 "extern" 的变量。

hexadecimal floating constants require an exponent（十六进制浮点常量需要一个指数）

十六进制浮点常量必须具有指数。

I

implicit declaration of function '*identifier*'（隐式声明的函数 "*identifier*"）

在前面没有原型声明或函数定义的情况下使用了该函数标识符。

impossible register constraint in 'asm'（"asm" 中存在不可能的寄存器约束）

asm 语句是无效的。

incompatible type for argument n of '*identifier*'（"*identifier*" 的参数 n 的类型不兼容）

在 C 代码中调用函数时，请确保实参类型与形参类型匹配。

incompatible type for argument *n* of indirect function call（间接函数调用的参数 n 的类型不兼容）

在 C 代码中调用函数时，请确保实参类型与形参类型匹配。

incompatible types in *operation* （操作 "*operation*" 中的类型不兼容）

operation 中使用的类型必须兼容。

incomplete '*name*' option （"*name*" 选项不完整）

命令行参数 *name* 的选项是不完整的。

inconsistent operand constraints in an 'asm'（"asm" 中存在不一致的操作数约束）

asm 语句是无效的。

increment of pointer to unknown structure （递增指向未知结构的指针）

不要递增指向未知结构的指针。

initializer element is not computable at load time（初始值设定元素不是在装入时可计算的）

初始值设定项元素必须在装入时是可计算的。

initializer element is not constant （初始值设定元素不是常量）

初始值设定元素必须为常量。

initializer fails to determine size of '*identifier*'（初始值设定无法确定 "identifier" 的大小）

数组初始值设定无法确定其大小。

© 2015 Microchip Technology Inc.

initializer for static variable is not constant （静态变量的初始值设定项不是常量）

静态变量的初始值设定项必须是常量。

initializer for static variable uses complicated arithmetic （静态变量的初始值设定使用了复杂的算术运算）

静态变量的初始值设定不应使用复杂的算术运算。

input operand constraint contains '*constraint*' （输入操作数约束包含了约束 *"constraint"*）

指定的约束对于输入操作数是无效的。

int-array initialized from non-wide string （int 数组使用非宽字符串初始化）

不应使用非宽字符串初始化 int 数组。

interrupt functions must not take parameters （中断函数必须不带参数）

中断函数不能接受参数。必须使用 *void* 来显式指示参数列表为空。

interrupt functions must return void （中断函数必须返回 void）

中断函数必须具有 *void* 返回类型。不允许其他返回类型。

interrupt modifier '*name*' unknown （中断修饰符 *"name"* 未知）

编译器预期的是使用 *"irq"*、*"altirq"* 或 *"save"* 作为 interrupt 属性修饰符。

interrupt modifier syntax error （中断修饰符的语法错误）

interrupt 属性修饰符存在语法错误。

interrupt pragma must have file scope（中断 pragma 伪指令必须具有文件作用域）

#pragma interrupt 必须具有文件作用域。

interrupt save modifier syntax error （中断 save 修饰符的语法错误）

interrupt 属性的 "save" 修饰符存在语法错误。

interrupt vector is not a constant （中断向量不是常量）

中断向量编号必须是一个整型常量。

interrupt vector number *n* is not valid （中断向量编号 *n* 无效）

需要一个有效的中断向量编号。

invalid #ident directive （无效的 #ident 伪指令）

#ident 后应跟随加引号的字符串。

invalid arg to '__builtin_frame_address'（传递给 "__builtin_frame_address" 的参数无效）

参数应为函数调用方的级别（其中，0 将得到当前函数的帧地址，1 将得到当前函数的调用方的帧地址，如此类推），并且是一个整型立即数。

invalid arg to '__builtin_return_address'（传递给 "__builtin_return_address" 的参数无效）

级别参数必须是整型立即数。

invalid argument for '*name*' （*"name"* 的参数无效）

编译器预期的是使用 "data" 或 "prog" 作为 space 属性参数。

invalid character '*character*' in #if （#if 中存在无效字符 *"character"*）

当 #if 后出现不可打印的字符 （如控制字符）时，会出现该消息。

invalid initial value for member '*name*' （成员 *"name"* 的初始值无效）

位域 "name" 只能使用整数初始化。

invalid initializer （无效的初始值设定项）

不要使用无效的初始值设定项。

Invalid location qualifier: '*symbol*' （无效的位置限定符：“*symbol*”）

预期的是使用“*sfr*”或“*gpr*”（它们在 dsPIC DSC 器件上会被忽略）作为位置限定符。

invalid operands to binary '*operator*' （二元操作符 “*operator*”的操作数无效）

指定的二元操作符的操作数无效。

Invalid option '*option*' （无效的选项 “*option*”）

指定的命令行选项是无效的。

Invalid option '*symbol*' to interrupt pragma（中断 pragma 伪指令的选项“*symbol*”无效）

预期的是使用 shadow 和 / 或 save 作为中断 pragma 伪指令的选项。

Invalid option to interrupt pragma （中断 pragma 伪指令的选项无效）

pragma 伪指令末尾存在垃圾项。

Invalid or missing function name from interrupt pragma （中断 pragma 伪指令的函数名称无效或缺失）

中断 pragma 伪指令需要被调用的函数的名称。

Invalid or missing section name （段名无效或缺失）

段名必须以字母或下划线（“_”）开头，后面跟随字母、下划线和 / 或数字的序列。名称“*access*”、“*shared*”和“*overlay*”具有特殊的含义。

invalid preprocessing directive #'*directive*' （无效的预处理伪指令 “#*directive*”）

不是一个有效的预处理伪指令。请检查拼写。

invalid preprologue argument （preprologue 参数无效）

preprologue 选项预期的是使用一条或多条汇编语句作为其括在双引号中的参数。

invalid register name for '*name*' （“*name*”的寄存器名称无效）

文件作用域变量 “*name*”被声明为寄存器变量，但使用了非法的寄存器名称。

invalid register name '*name*' for register variable （寄存器变量的寄存器名称 “*name*”无效）

指定的 *name* 不是寄存器的名称。

invalid save variable in interrupt pragma（中断 pragma 伪指令中存在无效的 save 变量）

预期的是要保存的一个或多个符号。

invalid storage class for function '*identifier*' （函数 “*identifier*”的存储类无效）

函数不能具有 “register”存储类。

invalid suffix '*suffix*' on integer constant （整型常量具有无效后缀 “*suffix*”）

整型常量只能使用字母 “u”、“U”、“l”和 “L”作为后缀。

invalid suffix on floating constant （浮点常量的后缀无效）

浮点常量后缀只能为 “f”、“F”、“l”或 “L”。如果存在两个 “L”，它们必须相邻，并且大小写相同。

invalid type argument of '*operator*' （操作符 “*operator*”的参数的类型无效）

operator 的参数的类型无效。

invalid type modifier within pointer declarator （指针声明符中的修饰符的类型无效）

只有 `const` 或 `volatile` 可以用作指针声明符内的类型修饰符。

invalid use of array with unspecified bounds （无效地使用未指定边界的数组）

必须以有效的方式使用未指定边界的数组。

invalid use of incomplete typedef '*typedef*'（无效地使用不完整的 typedef “*typedef*”）

以无效的方式使用指定的 *typedef*；这是不允许的。

© 2015 Microchip Technology Inc.

invalid use of undefined type ‘*type identifier*’（无效地使用未定义类型“*type identifier*”）

以无效的方式使用指定的 *type* ； 这是不允许的。

invalid use of void expression （无效地使用 void 表达式）

不能使用 void 表达式。

“*name*” is not a valid filename （“*name*” 不是有效的文件名）

#line 需要一个有效的文件名。

‘*filename*’ is too large （文件 “*filename*” 太大）

指定文件太大，无法处理该文件。它可能大于 4 GB，预处理器拒绝处理如此大的文件。要求文件的大小小于 4 GB。

ISO C forbids data definition with no type or storage class （ISO C 禁止没有类型或存储类的数据定义）

在 ISO C 中，要求数据定义具有类型说明符或存储类说明符。

ISO C requires a named argument before ‘...’（ISO C 要求在 “...” 之前具有命名参数）

ISO C 要求在 “...” 之前具有命名参数。

L

label *label* referenced outside of any function（标号 “*label*” 在任意函数外部被引用）

只能在函数内引用标号。

label ‘*label*’ used but not defined （使用了标号 “*label*”，但它是未定义的）

使用了指定的 *label*，但它是未定义的。

language ‘*name*’ not recognized （无法识别语言 “*name*”）

允许的语言包括：c assembler none。

***filename*: linker input file unused because linking not done （*filename:* 链接器输入文件未被使用，因为链接未完成）**

在命令行上指定了指定文件 *filename*，它被作为一个链接器输入文件接受（因为无法将它识别为其他文件类型）。但是，链接步骤未运行。因此，该文件被忽略。

long long long is too long for GCC （long long long 对于 GCC 来说太长）

编译器支持不长于 `long long` 的整型。

long or short specified with char for ‘*identifier*’（对于 “*identifier*”，指定 char 时一起指定了 long 或 short）

long 和 short 限定符不能用于 char 类型。

long or short specified with floating type for ‘*identifier*’（对于 “*identifier*”，指定浮点类型时一起指定了 long 或 short）

long 和 short 限定符不能用于浮点类型。

long, short, signed or unsigned invalid for ‘*identifier*’（对于 “*identifier*”，long、short、 signed 或 unsigned 是无效的）

long、 short 和 signed 限定符只能用于整型类型。

M

macro names must be identifiers （宏名称必须为标识符）

宏名称必须以字母或下划线开头，后面跟随字母、数字或下划线。

macro parameters must be comma-separated （宏参数必须使用逗号分隔）

参数列表中的参数之间必须使用逗号分隔。

macro ‘*name*’ passed *m* arguments, but takes just *n*（向宏 “*name*” 传递了 *m* 个参数，但它只接受 *n* 个）

传递给宏 “name” 的参数太多。

macro '*name*' requires *m* arguments, but only *n* given （宏 "*name*" 需要 *m* 个参数，但只提供了 n 个）

传递给宏 "name" 的参数不足。

matching constraint not valid in output operand （输出操作数中的匹配约束无效）

asm 语句是无效的。

'*symbol*' may not appear in macro parameter list （"*symbol*" 不能出现在宏参数列表中）

不允许使用 "symbol" 作为一个参数。

Missing '=' for 'save' in interrupt pragma（中断 pragma 伪指令中的 "save" 缺少 "="）

save 参数在列出的变量之前需要一个等号。例如，`#pragma interrupt isr0 save=var1,var2`

missing '('after predicate （谓词后缺少 "("）

#assert 或 #unassert 需要在答案前后加上括号。例如：`#assert PREDICATE (ANSWER)`

missing '(' in expression （表达式中缺少 "("）

括号不匹配，需要一个左括号。

"defined" 后缺少 ")"

需要一个右括号。

missing ')' in expression （表达式中缺少 ")"）

括号不匹配，需要一个右括号。

missing ')' in macro parameter list （宏参数列表中缺少 ")"）

宏要求参数处于括号内，并以逗号分隔。

missing ')' to complete answer （缺少 ")" 来使答案完整）

#assert 或 #unassert 需要在答案前后加上括号。

missing argument to '*option*' option （选项 "*option*" 缺少参数）

指定的命令行选项需要一个参数。

missing binary operator before token '*token*'（标记 "*token*" 前缺少二元操作符）

"token" 前需要一个操作符。

missing terminating '*character*' character （缺少终止字符 "*character*"）

缺少终止字符，例如单引号 '、双引号 " 或右尖括号 >。

missing terminating > character （缺少终止字符 >）

#include 伪指令中需要使用终止的 >。

more than *n* operands in 'asm' （"asm" 中的操作数超过 *n* 个）

asm 语句是无效的。

multiple default labels in one switch （一个 switch 中存在多个 default 标号）

每个 switch 只能具有单个 default 标号。

multiple parameters named '*identifier*' （多个参数命名为 "*identifier*"）

参数名称必须是唯一的。

multiple storage classes in declaration of '*identifier*' （"*identifier*" 的声明中存在多个存储类。）

每个声明应具有单个存储类。

© 2015 Microchip Technology Inc.

N

negative width in bit-field ‘*identifier*’（位域“*identifier*”具有负宽度）

位域宽度不能为负值。

nested function ‘name’ *declared* ‘extern’（嵌套函数“name”*声明了*“extern”）

嵌套函数不能声明“extern”。

nested redefinition of ‘*identifier*’（嵌套重新定义“*identifier*”）

嵌套的重新定义是非法的。

no data type for mode ‘mode’（没有对应于模式“mode”的数据类型）

为 mode 属性指定的参数 mode 是一个可识别的 GCC 机器模式，但编译器中未实现该模式。

no include path in which to find ‘*name*’（没有可找到“*name*”的包含路径）

无法找到包含文件“*name*”。

no macro name given in #‘*directive*’ directive（“#*directive*”伪指令中未给出宏名称）

#define、 #undef、 #ifdef 或 #ifndef 伪指令后必须跟随宏名称。

nonconstant array index in initializer（初始值设定项中存在非常量数组索引）

初始值设定项中只能使用常量数组索引。

non-prototype definition here（无原型定义在此处）

如果函数原型跟随在没有原型的定义后面，而两者之间的参数数量不一致，则该消息会标识无原型定义的行号。

number of arguments doesn’t match prototype（参数数量与原型不匹配）

函数参数的数量必须与函数的原型匹配。

O

operand constraint contains incorrectly positioned ‘+’ or ‘=’.（操作数约束包含放置位置错误的“+”或“=”。）

asm 语句是无效的。

operand constraints for ‘asm’ differ in number of alternatives（“asm”的操作数约束的备选项的数量不同）

asm 语句是无效的。

operator “defined” requires an identifier（操作符“defined”需要一个标识符）

“defined”需要一个标识符。

operator ‘*symbol*’ has no right operand（操作符“symbol”没有右操作数）

预处理器操作符“symbol”需要一个处于右侧的操作数。

output number *n* not directly addressable（输出数字 n 不可直接寻址）

asm 语句是无效的。

output operand constraint lacks ‘=’（输出操作数约束缺少“=”）

asm 语句是无效的。

output operand is constant in ‘asm’（“asm”中的输出操作数是常量）

asm 语句是无效的。

overflow in enumeration values（枚举值发生溢出）

枚举值必须处于“int”的范围内。

P

parameter '*identifier*' declared void （参数 "*identifier*" 声明为 void）

参数不能声明为 void。

parameter '*identifier*' has incomplete type（参数 "*identifier*" 具有不完整的类型）

参数必须具有完整的类型。

parameter '*identifier*' has just a forward declaration（参数 "*identifier*" 只有一个前置声明）

参数必须具有完整的类型；前置声明是不够的。

parameter '*identifier*' is initialized （参数 "*identifier*" 被初始化）

初始化参数是非法的。

parameter name missing （缺少参数名称）

该宏需要一个参数名称。请检查是否存在两个逗号之间没有名称的情况。

parameter name missing from parameter list （参数列表中缺少参数名称）

参数列表中必须包含参数名称。

parameter name omitted （参数名称被省略）

参数名称不可以省略。

param types given both in param list and separately （同时在参数列表中给出了和单独给出了参数类型）

参数类型应在参数列表中给出或单独给出，但不能同时以两种方式给出。

parse error （解析错误）

无法解析源代码行；它包含错误。

pointer value used where a complex value was expected（在需要复数值的位置使用了指针值）

不要在需要复数值的位置使用指针值。

pointer value used where a floating point value was expected（在需要浮点值的位置使用了指针值）

不要在需要浮点值的位置使用指针值。

pointers are not permitted as case values （指针不允许用作 case 值）

case 值必须为整型值常量或常量表达式。

predicate must be an identifier （谓词必须是一个标识符）

#assert 或 #unassert 要求使用单个标识符作为谓词。）

predicate's answer is empty （谓词的答案为空）

#assert 或 #unassert 具有一个谓词和括号，但在括号内没有答案，答案是必需的。

previous declaration of '*identifier*' （先前声明了 "*identifier*"）

该消息会标识与标识符当前声明冲突的先前声明的位置。

***identifier* previously declared here （"*identifier*" 先前在此处声明）**

该消息会标识与标识符当前声明冲突的先前声明的位置。

***identifier* previously defined here （"*identifier*" 先前在此处定义）**

该消息会标识与标识符当前定义冲突的先前定义的位置。

prototype declaration （原型声明）

标识声明了函数原型的位置的行号。与其他错误消息配合使用。

© 2015 Microchip Technology Inc.

R

redeclaration of '*identifier*' （重新声明了 *"identifier"*）

多次声明了 *identifier*。

redeclaration of '*enum identifier*' （重新声明了枚举标识符 *"enum identifier"*）

枚举不能重新声明。

'*identifier*' redeclared as different kind of symbol （*"identifier"* 被重新声明作为不同种类的符号）

identifier 存在多个不一致的声明。

redefinition of '*identifier*' （重新定义了 *"identifier"*）

多次定义了 *identifier*。

redefinition of '*struct identifier*' （重新定义了结构标识符 *"struct identifier"*）

不能重新定义结构。

redefinition of '*union identifier*' （重新定义了联合标识符 *"union identifier"*）

不能重新定义联合。

register name given for non-register variable '*name*' （为非寄存器变量 *"name"* 提供了寄存器名称）

试图将一个寄存器映射到一个未标记为寄存器的变量。

register name not specified for '*name*' （未为 *"name"* 指定寄存器名称）

文件作用域变量 *"name"* 被声明为寄存器变量，但未提供寄存器。

register specified for '*name*' isn't suitable for data type（为 *"name"* 指定的寄存器不适合于数据类型）

对齐或其他限制阻止使用所请求的寄存器。

request for member '*identifier*' in something not a structure or union （请求不属于结构或联合的对象中的成员 *"identifier"*）

只有结构或联合具有成员。引用任何其他对象的成员是非法的，因为其他任何对象都不具有成员。

requested alignment is not a constant （请求的对齐量不是一个常量）

aligned 属性的参数必须是一个编译时常量。

requested alignment is not a power of 2 （请求的对齐量不是 2 的幂）

aligned 属性的参数必须是 2 的幂。

requested alignment is too large （请求的对齐量太大）

请求的对齐大小大于链接器允许的大小。大小必须小于等于 4096 且为 2 的幂。

return type is an incomplete type （返回类型是不完整的类型）

返回类型必须是完整的。

S

save variable '*name*' index not constant （保存的变量 *"name"* 的索引不是常量）

数组 "name" 的下标不是一个常量整数。

save variable '*name*' is not word aligned （保存的变量 *"name"* 不是按字对齐）

保存的对象必须按字对齐

save variable '*name*' size is not even （保存的变量 *"name"* 的大小不是偶数）

保存的对象必须在偶边界对齐。

save variable '*name*' size is not known （保存的变量 *"name"* 的大小未知）

保存的对象必须具有已知的大小。

section attribute cannot be specified for local variables（不能为局部变量指定 section 属性）

局部变量总是被分配在寄存器或堆栈中。因此试图将局部变量放入命名段是非法的。

section attribute not allowed for *identifier*（section 属性不允许用于“*identifier*”）

section 属性只能用于函数或变量。

section of *identifier* conflicts with previous declaration（“*identifier*”的段与先前声明冲突）

如果同一 *identifier* 的多个声明指定了 section 属性，则该属性的值必须一致。

sfr address ‘*address*’ is not valid （sfr 地址 “*address*”是无效的）

地址必须小于 0x2000 才有效。

sfr address is not a constant （sfr 地址不是一个常量）

sfr 地址必须是一个常量。

‘size of’ applied to a bit-field （“sizeof”被应用于一个位域）

“sizeof”不能被应用于一个位域。

size of array ‘*identifier*’ has non-integer type （数组 “*identifier*”的大小具有非整型类型）

数组大小说明符必须是整型类型。

size of array ‘*identifier*’ is negative （数组 “*identifier*”的大小为负值）

数组大小不能为负值。

size of array ‘*identifier*’ is too large （数组 “*identifier*”的大小太大）

指定的数组太大。

size of variable ‘*variable*’ is too large （变量 “*variable*”的大小太大）

变量的最大大小可以为 32768 字节。

storage class specified for parameter ‘*identifier*’（为参数 “*identifier*”指定了存储类）

不能为参数指定存储类。

storage size of ‘*identifier*’ isn’t constant （“*identifier*”的存储长度不是常量）

存储大小必须为编译时常量。

storage size of ‘*identifier*’ isn’t known （“*identifier*”的存储长度未知）

未完全指定 *identifier* 的长度。

stray ‘*character*’ in program （程序中存在杂散字符 “*character*”）

不要在源程序中放入杂散的 “*character*”字符。

strftime formats cannot format arguments （strftime 格式无法格式化参数）

在使用属性 format 时，如果 archetype 参数为 strftime，则该属性的第三个参数（它指定要根据格式字符串进行匹配的第一个参数）应为 0。 strftime 样式的函数不具有要根据格式字符串进行匹配的输入值。

structure has no member named ‘*identifier*’（结构没有名为 “*identifier*”的成员）

引用了名为 “*identifier*”的结构成员；但引用的结构不包含此成员。这是不允许的。

subscripted value is neither array nor pointer（带下标的值既不是数组也不是指针）

只有数组或指针可以带下标。

switch quantity not an integer （switch 量不是整数）

switch 量必须是整数。

© 2015 Microchip Technology Inc.

symbol '*symbol*' not defined （符号 "*symbol*" 未定义）

符号 "symbol" 需要先声明，之后才能在 pragma 伪指令中使用。

syntax error （语法错误）

指定行上存在语法错误。

syntax error ':' without preceding '?'（语法错误 ":" 前面没有 "?"）

在 "?:" 操作符中，":" 前面必须具有 "?"。

T

the only valid combination is 'long double'（唯一的有效组合为 "long double"）

long 限定符是可与 double 类型一起使用的唯一限定符。

this built-in requires a frame pointer （这个内置函数需要帧指针）

`__builtin_return_address` 需要帧指针。请不要使用 -fomit-frame-pointer 选项。

this is a previous declaration （这是先前的声明）

如果某个标号重复，则该消息会标识先前声明的行号。

too few arguments to function （函数的参数太少）

在 C 代码中调用一个函数时，指定的参数数量不得少于函数需要的数量。也不应指定太多的参数。

too few arguments to function '*identifier*'（函数 "*identifier*" 的参数太少）

在 C 代码中调用一个函数时，指定的参数数量不得少于函数需要的数量。也不应指定太多的参数。

too many alternatives in 'asm'（"asm" 中的备选项太多）

asm 语句是无效的。

too many arguments to function （函数的参数太多）

在 C 代码中调用一个函数时，指定的参数数量不得多于函数需要的数量。也不应指定太少的参数。

too many arguments to function '*identifier*'（函数 "*identifier*" 的参数太多）

在 C 代码中调用一个函数时，指定的参数数量不得多于函数需要的数量。也不应指定太少的参数。

too many decimal points in number （数字中的小数点太多）

只需要一个小数点。

top-level declaration of '*identifier*' specifies 'auto'（"*identifier*" 的最高层级的声明指定了 "auto"）

自动变量只能在函数内声明。

two or more data types in declaration of '*identifier*'（"*identifier*" 的声明中存在两种或更多的数据类型）

每个标识符只能具有一种数据类型。

two types specified in one empty declaration （一个空声明中指定了两种类型）

指定的类型不应多于一种。

type of formal parameter *n* is incomplete （形参 n 的类型是不完整的）

请为所指示的参数指定一个完整的类型。

type mismatch in conditional expression （条件表达式中的类型不匹配）

条件表达式中的类型必须匹配。

typedef '*identifier*' is initialized （typedef "*identifier*" 被初始化）

初始化 typedef 是非法的。请改为使用 __typeof__。

U

'*identifier*' undeclared (first use in this function)（“*identifier*”未声明（在该函数中第一次使用））

必须声明指定的标识符。

'*identifier*' undeclared here (not in a function)（此处的“*identifier*”未声明（不是在函数中））

必须声明指定的标识符。

union has no member named '*identifier*'（联合没有名为“*identifier*”的成员）

引用了名为“*identifier*”的联合成员；但引用的联合不包含此成员。这是不允许的。

unknown field '*identifier*' specified in initializer（在初始值设定中指定了未知位域“*identifier*”）

不要在初始值设定中使用未知位域。

unknown machine mode '*mode*'（未知的机器模式“*mode*”）

为 mode 属性指定的参数 *mode* 不是可识别的机器模式。

unknown register name '*name*' in 'asm'（“asm”中存在未知的寄存器名称“*name*”）

asm 语句是无效的。

unrecognized format specifier（无法识别的格式说明符）

format 属性的参数是无效的。

unrecognized option '*-option*'（无法识别的选项“*-option*”）

指定的命令行选项是无法识别的。

unrecognized option '*option*'（无法识别的选项“*option*”）

“*option*”不是一个已知的选项。

'*identifier*' used prior to declaration（“*identifier*”在声明之前使用）

该标识符在其声明之前使用。

unterminated #'*name*'（未终止的“#*name*”）

需要使用 #endif 来终止 #if、 #ifdef 或 #ifndef 条件语句。

unterminated argument list invoking macro '*name*'（调用宏“*name*”的参数列表未终止）

在对函数宏进行求值时，在完成宏扩展之前遇到了文件末尾。

unterminated comment（未终止的注释）

在扫描注释终止符时到达文件末尾。

V

'va_start' used in function with fixed args（在具有固定参数的函数中使用了“va_start”）

只有在具有可变参数列表的函数中才能使用“va_start”。

variable '*identifier*' has initializer but incomplete type（变量“*identifier*”具有初始值设定，但具有不完整的类型）

对于具有不完整类型的变量进行初始化是非法的。

variable or field '*identifier*' declared void（变量或位域“*identifier*”声明为 void）

变量和位域都不能声明为 void。

不能初始化可变大小的对象

初始化一个可变大小的对象是非法的。

virtual memory exhausted（虚拟存储器已耗尽）

剩余的存储器不足，无法写入错误消息。

© 2015 Microchip Technology Inc.

void expression between '(' and ')' （“(”和 “)”之间存在 void 类型的表达式）

在括号之间预期的是一个常量表达式，却发现一个 void 类型的表达式。

'void' in parameter list must be the entire list （参数列表中的 “void”必须为整个列表）

如果 “void”显示为一个参数列表中的参数，则不能存在其他参数。

void value not ignored as it ought to be （void 值本应被忽略，但未被忽略）

不应在表达式中使用 void 函数的值。

W

warning: -pipe ignored because -save-temps specified （警告：由于指定了 -save-temps， -pipe 被忽略）

-pipe 选项不能与 -save-temps 选项一起使用。

warning: -pipe ignored because -time specified （警告：由于指定了 -time， -pipe 被忽略）

-pipe 选项不能与 -time 选项一起使用。

warning: '-x *spec*' after last input file has no effect （警告：最后一个输入文件后的 *“-x spec”* 没有任何作用）

“-x*”* 命令行选项只会影响命令行中在它之后指定的那些文件；如果没有这样的文件，那么该选项没有任何作用。

weak declaration of '*name*' must be public （*“name”* 的弱声明必须是公共的）

弱符号必须是外部可见的。

weak declaration of '*name*' must precede definition （*“name”* 的弱声明必须位于定义之前）

“name” 先被定义，然后再声明为 weak。

wrong number of arguments specified for *attribute* attribute （为 *“attribute”* 属性指定的参数数量错误）

为名为 *“attribute”* 的属性提供的参数数量太少或太多。

wrong type argument to bit-complement （位补码操作的参数的类型错误）

不要对该操作符使用错误的参数类型。

wrong type argument to decrement （递减操作的参数的类型错误）

不要对该操作符使用错误的参数类型。

wrong type argument to increment （递增操作的参数的类型错误）

不要对该操作符使用错误的参数类型。

wrong type argument to unary exclamation mark （一元感叹号操作符的参数的类型错误）

不要对该操作符使用错误的参数类型。

wrong type argument to unary minus （一元减号操作符的参数的类型错误）

不要对该操作符使用错误的参数类型。

wrong type argument to unary plus （一元加号操作符的参数的类型错误）

不要对该操作符使用错误的参数类型。

Z

zero width for bit-field '*identifier*' （位域 *“identifier”* 具有零宽度）

位域宽度不能为零。

C.3 警告

符号

'/*' within comment （注释内存在 "/ *"）

在注释中发现注释标记。

'$' character(s) in identifier or number （标识符或数字中存在 "$" 字符）

标识符名称中的美元符号是对标准的扩展。

#'*directive*' is a GCC extension （伪指令 "*directive*" 是一项 GCC 扩展）

#warning、#include_next、#ident、#import、#assert 和 #unassert 伪指令是 GCC 扩展，不是 ISO C89 的扩展。

#import is obsolete, use an #ifndef wrapper in the header file （#import 已过时，请在头文件中使用 #ifndef 包装器）

#import 伪指令已过时。 #import 过去用于包含一个尚未被包含的文件。请改为使用 #ifndef 伪指令。

#include_next in primary source file （#include_next 位于主源文件中）

在搜索找到当前文件的目录之后， `#include_next` 会开始搜索头文件目录的列表。在此例中，由于没有先前头文件，所以它在主源文件中开始。

#pragma pack (pop) encountered without matching #pragma pack (push, <n>)（遇到 #pragma pack (pop)，但不存在匹配的 #pragma pack (push, <n>)）

pack(pop) pragma 伪指令必须与 pack(push) pragma 伪指令配对使用，后者在源文件中必须处于前者之前。

#pragma pack (pop, *identifier*) encountered without matching #pragma pack (push, *identifier*, <n>)（遇到 #pragma pack (pop, identifier)，但不存在匹配的 #pragma pack (push, identifier, <n>)）

pack(pop) pragma 伪指令必须与 pack(push) pragma 伪指令配对使用，后者在源文件中必须处于前者之前。

#warning:*message* （#warning：消息）

伪指令 #warning 会导致预处理器发出警告，并且继续预处理。#warning 后的标记用作警告消息。

A

absolute address specification ignored （绝对地址说明被忽略）

忽略 #pragma 语句中的代码段的绝对地址说明，因为它在该编译器中不受支持。地址必须在链接描述文件中指定，可以使用关键字 `__attribute__` 定义代码段。

address of register variable '*name*' requested（请求了寄存器变量 "name" 的地址）

寄存器说明符阻止获取一个变量的地址。

alignment must be a small power of two, not n（对齐量必须为 2 的较小幂，而不是 n）

pack pragma 伪指令的 alignment 参数必须为 2 的较小幂。

anonymous enum declared inside parameter list（在参数列表内声明了匿名枚举）

在函数参数列表内声明了匿名枚举。在参数列表外声明枚举通常是更好的编程做法，因为在参数列表内定义它们时，它们不会成为完整的类型。

anonymous struct declared inside parameter list（在参数列表内声明了匿名结构）

在函数参数列表内声明了匿名结构。在参数列表外声明结构通常是更好的编程做法，因为在参数列表内定义它们时，它们不会成为完整的类型。

© 2015 Microchip Technology Inc.

anonymous union declared inside parameter list（在参数列表内声明了匿名联合）

在函数参数列表内声明了匿名联合。在参数列表外声明联合通常是更好的编程做法，因为在参数列表内定义它们时，它们不会成为完整的类型。

anonymous variadic macros were introduced in C99（匿名可变参数宏是在 C99 中引入的）

接受可变参数数量的宏是一项 C99 特性。

argument ‘*identifier*’ might be clobbered by ‘longjmp’ or ‘vfork’（参数 “*identifier*” 可能会被 “longjmp” 或 “vfork” 破坏）

参数可能会被 longjmp 调用更改。只有在优化编译时才可能出现这些警告。

array ‘*identifier*’ assumed to have one element （数组 “*identifier*” 被假定为具有一个元素）

指定数组的长度未显式指明。在没有相应信息的情况下，编译器假定它具有一个元素。

array subscript has type ‘char’ （数组下标具有 “char” 类型）

数组下标具有 “*char*” 类型。

array type has incomplete element type （数组类型具有不完整的元素类型）

数组类型不应具有不完整的元素类型。

asm operand *n* probably doesn’t match constraints（asm 操作数 *n* 可能与约束不匹配）

指定的扩展 asm 操作数可能与它的约束不匹配。

assignment of read-only member ‘*name*’ （对只读成员 “*name*” 赋值）

成员 “name” 被声明为 const，无法通过赋值修改。

assignment of read-only variable ‘*name*’ （对只读变量 “*name*” 赋值）

“name” 被声明为 const，无法通过赋值修改。

‘*identifier*’ attribute directive ignored （“*identifier*” 属性伪指令被忽略）

指定的属性不是已知的或受支持的属性，因此被忽略。

‘*identifier*’ attribute does not apply to types （“*identifier*” 属性不适用于类型）

指定属性不能用于类型。它被忽略。

‘*identifier*’ attribute ignored （“*identifier*” 属性被忽略）

指定属性在给定的上下文中没有意义，因此被忽略。

‘*attribute*’ attribute only applies to function types（“*attribute*” 属性仅适用于函数类型）

指定属性只能应用于函数的返回类型，不能应用于其他声明。

B

backslash and newline separated by space （反斜杠和换行符以空格分隔）

在处理转义序列时，发现反斜杠和换行符以空格分隔。

backslash-newline at end of file （反斜杠和换行符处于文件末尾）

在处理转义序列时，在文件的末尾发现反斜杠和换行符。

bit-field ‘*identifier*’ type invalid in ISO C （位域 “identifier” 类型在 ISO C 中是无效的）

对指定标识符使用的类型在 ISO C 中是无效的。

braces around scalar initializer （标量初始值设定两边存在花括号）

初始值设定两边提供了一组多余的花括号。

built-in function ‘*identifier*’ declared as non-function （内置函数 “*identifier*” 被声明为非函数）

指定的函数与某个内置函数同名，但被声明为非函数对象。

C

C++ style comments are not allowed in ISO C89（ISO C89 中不允许 C++ 样式的注释）

请使用 C 样式的注释 “/*” 和 “* /” 而不是 C++ 样式的注释 “//”。

call-clobbered register used for global register variable （对全局寄存器变量使用了会被调用破坏的寄存器）

选择一个通常会被函数调用保存和恢复的寄存器（W8-W13），从而使库程序不会破坏它。

cannot inline function ‘main’ （不能内联函数 “main”）

函数 “main” 使用 *inline* 属性进行声明。不支持这样做，因为 main 必须从单独编译的 C 启动代码中调用。

can’t inline call to ‘*identifier*’ called from here （无法对从此处调用的 “*identifier*” 调用进行内联）

编译器无法内联对指定函数的调用。

case value ‘*n*’ not in enumerated type （case 值 “*n*” 不在枚举类型中）

switch 语句的控制表达式是一个枚举类型，但 case 表达式具有值 n，该值并不对应于任何枚举值。

case value ‘*value*’ not in enumerated type ‘*name*’ （case 值 “*value*” 不在枚举类型 “*name*” 中）

“*value*” 是一个额外的 switch case，不属于枚举类型 “*name*” 的元素。

cast does not match function type （强制类型转换与函数类型不匹配）

某个函数的返回类型被强制转换为一个与该函数的类型不匹配的类型。

cast from pointer to integer of different size （从指针强制转换为不同大小的整型）

指针被强制转换为一个不是 16 位宽的整型。

cast increases required alignment of target type（强制类型转换会增大目标类型所需的对齐量）

使用 `-Wcast-align` 命令行选项进行编译时，编译器会检查强制类型转换是否会增大目标类型所需的对齐量。例如，如果指向 char 类型的指针被强制转换为指向 int 的指针，由于 char 类型的对齐量（字节对齐）小于 int 的对齐要求（字对齐），将会给出该警告消息。

character constant too long （字符常量太长）

字符常量不能太长。

comma at end of enumerator list （枚举数列表的末尾存在逗号）

枚举数列表的末尾存在多余的逗号。

comma operator in operand of #if （#if 的操作数中存在逗号操作符）

#if 伪指令中不需要逗号操作符。

comparing floating point with == or != is unsafe （使用 == 或 != 比较浮点值是不安全的）

浮点值可能是无限精度实数的近似值。请使用关系操作符来确定两个值是否具有重叠的范围，而不是测试相等性。

comparison between pointer and integer （在指针和整型之间进行比较）

指针类型与一个整型类型进行比较。

© 2015 Microchip Technology Inc.

comparison between signed and unsigned （在有符号和无符号值之间进行比较）

比较的操作数之一是有符号的，而另一个是无符号的。有符号操作数会被当做无符号值，这可能是不正确的。

comparison is always *n* （比较总是为 *n*）

比较仅涉及常量表达式，所以编译器可以求得比较的运行时结果。结果总是为 *n*。

comparison is always *n* due to width of bit-field（由于位域的宽度，比较总是为 *n*）

由于位域的宽度，涉及位域的比较总是求值为 *n*。

comparison is always false due to limited range of data type （由于数据类型的范围有限，比较结果总是为 false）

由于数据类型的范围，比较在运行时总是求值为 false。

comparison is always true due to limited range of data type （由于数据类型的范围有限，比较结果总是为 true）

由于数据类型的范围，比较在运行时总是求值为 true。

comparison of promoted ~unsigned with constant（将经过提升的无符号值（~unsigned）与常量进行比较）

比较的操作数之一是经过提升的无符号值 （~unsigned），而另一个为常量。

comparison of promoted ~unsigned with unsigned（将经过提升的无符号值（~unsigned）与无符号值进行比较）

比较的操作数之一是经过提升的无符号值 （~unsigned），而另一个为无符号值。

comparison of unsigned expression >= 0 is always true （无符号表达式 >=0 的比较总是为 true）

比较表达式将一个无符号值与零进行比较。由于无符号值不能小于零，比较在运行时总是求值为 true。

comparison of unsigned expression < 0 is always false（无符号表达式 < 0 的比较总是为 false）

比较表达式将一个无符号值与零进行比较。由于无符号值不能小于零，比较在运行时总是求值为 false。

comparisons like X<=Y<=Z do not have their mathematical meaning （类似于 X<=Y<=Z 的比较没有数学意义）

C 表达式并不一定与相应的数学表达式具有相同的含义。特别是， C 表达式 X<=Y<=Z 并不等效于数学表达式 X ≤ Y ≤ Z。

conflicting types for built-in function '*identifier*' （类型与内置函数 "*identifier*" 冲突）

指定的函数与某个内置函数同名，但声明为冲突的类型。

const declaration for '*identifier*' follows non-const（"*identifier*" 的 const 声明前具有非 const 声明）

指定标识符在先前声明为非 const 之后又声明为 const。

control reaches end of non-void function （控制到达非 void 函数的末尾）

非 void 函数的所有退出路径都应返回一个适当的值。编译器检测到非 void 函数终止但没有显式返回值的情况。因此，返回值可能是不可预知的。

conversion lacks type at end of format （转换在格式末尾缺少类型）

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现格式字符串中的某个格式字段缺少类型说明符。

concatenation of string literals with __FUNCTION__ is deprecated（使用 __FUNCTION__ 串联字符串字面值的方式已被弃用）

__FUNCTION__ 将使用与 __func__ 相同的方式处理（由 ISO 标准 C99 定义）。__func__ 是一个变量，而不是字符串字面值，所以它不与其他字符串进行连接。

conflicting types for '*identifier*'（“*identifier*”存在冲突的类型）

指定的标识符具有多个不一致的声明。

D

data definition has no type or storage class（数据定义没有任何类型或存储类）

检测到缺少类型和存储类的数据定义。

data qualifier '*qualifier*' ignored（数据限定符“*qualifier*”被忽略）

该编译器中不使用数据限定符（包括“access”、“shared”和“overlay”），但提供它们是为了与用于 PIC18 MCU 的 MPLAB C 编译器保持兼容性。

declaration of '*identifier*' has 'extern' and is initialized（“*identifier*”的声明具有“extern”并被初始化）

不应初始化 extern 对象。

declaration of '*identifier*' shadows a parameter（“*identifier*”的声明会遮蔽某个参数）

指定的 *identifier* 声明会遮蔽某个参数，使该参数不可访问。

declaration of '*identifier*' shadows a symbol from the parameter list（“*identifier*”的声明会遮蔽参数列表中的某个符号）

指定的标识符声明会遮蔽参数列表中的某个符号，使该符号不可访问。

declaration of '*identifier*' shadows global declaration（“*identifier*”的声明会遮蔽全局声明）

指定的 *identifier* 声明会遮蔽某个全局声明，使它不可访问。

'*identifier*' declared inline after being called（“*identifier*”在被调用后声明为 inline）

指定函数在被调用之后声明为 inline。

'*identifier*' declared inline after its definition（“*identifier*”在其定义后被声明为 inline）

指定函数在定义之后声明为 inline。

'*identifier*' declared 'static' but never defined（“*identifier*”声明为“static”，但从未定义）

指定函数被声明为 static，但从未定义。

decrement of read-only member '*name*'（对只读成员“*name*”进行递减）

成员“name”被声明为 const，无法通过递减修改。

decrement of read-only variable '*name*'（对只读变量“*name*”进行递减）

“name”被声明为 const，无法通过递减修改。

'*identifier*' defined but not used（定义但未使用“*identifier*”）

定义了指定函数，但从未使用。

deprecated use of label at end of compound statement（在复合语句末尾使用标号的方式已被弃用）

标号不应处于语句的末尾。它应跟随一条语句。

dereferencing 'void *' pointer（解引用“void *”指针）

解引用“void*”指针是不正确的。请先将它强制转换为相应类型的指针，然后再对该指针进行解引用。

division by zero（被零除）

检测到编译时被零除。

© 2015 Microchip Technology Inc.

duplicate ‘const’（重复的 “const”）

“const”限定符应仅应用于声明一次。

duplicate ‘restrict’（重复的 “restrict”）

“restrict”限定符应仅应用于声明一次。

duplicate ‘volatile’（重复的 “volatile”）

“volatile”限定符应仅应用于声明一次。

E

embedded ‘\0’ in format （格式中包含嵌入的 “\0”）

检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现格式字符串包含嵌入的“\0”（零），它会导致格式字符串处理提前终止。

empty body in an else-statement （else 语句中的主体为空）

else 语句为空。

empty body in an if-statement （if 语句中的主体为空）

if 语句为空。

empty declaration （空声明）

声明不包含任何要声明的名称。

empty range specified （指定了空范围）

case 范围中的值的范围为空，也即，下限表达式的值大于上限表达式的值。回忆一下，case 范围的语法为 `case` *`low`* ... *`high`*:。

‘enum *identifier*’ declared inside parameter list（参数列表内声明了枚举 “ *identifier*”）

在函数参数列表内声明了指定的枚举。在参数列表外声明枚举通常是更好的编程做法，因为在参数列表内定义它们时，它们不会成为完整的类型。

enum defined inside parms （参数内定义了枚举）

函数参数列表内定义了枚举。

enumeration value ‘*identifier*’ not handled in switch （枚举值 “*identifier*” 在 switch 中未处理）

switch 语句的控制表达式是一个枚举类型，但并不是所有枚举值都具有 case 表达式。

enumeration values exceed range of largest integer（枚举值超出最大整型的范围）

枚举值以整型来表示。编译器检测到一个枚举范围无法使用任何编译器整型格式表示，包括最大的整型格式。

excess elements in array initializer （数组初始值设定项中存在过多的元素）

初始值设定项列表中的元素数量多于数组声明的元素数量。

excess elements in scalar initializer （标量初始值设定项中存在过多的元素）

标量变量应只有一个初始值设定项。

excess elements in struct initializer （结构初始值设定项中存在过多的成员）

初始值设定项列表中的成员数量多于结构声明的成员数量。

excess elements in union initializer （联合初始值设定项中存在过多的成员）

初始值设定项列表中的成员数量多于联合声明的成员数量。

extra semicolon in struct or union specified （指定的结构或联合中存在额外的分号）

结构类型或联合类型包含额外的分号。

extra tokens at end of #‘*directive*’ directive （“*#directive*”伪指令末尾存在额外的标记）

编译器在包含 “#directive”伪指令的源代码行中检测到额外文本。

F

-ffunction-sections may affect debugging on some targets （-ffunction-sections 可能会影响某些目标上的调试）

如果同时指定了-g选项和 `-ffunction-sections`选项，则可能会遇到调试方面的问题。

first argument of ‘*identifier*’ should be ‘int’ （“*identifier*”的第一个参数应为 “int”）

要求指定标识符的第一个参数的声明为 int 类型。

floating constant exceeds range of ‘double’（浮点常量超出 “double”的范围）

浮点常量太大或太小 （量值），无法以 “double”表示。

floating constant exceeds range of ‘float’（浮点常量超出 “float”的范围）

浮点常量太大或太小 （量值），无法以 “float”表示。

floating constant exceeds range of ‘long double’（浮点常量超出 “long double”的范围）

浮点常量太大或太小 （量值），无法以 “long double”表示。

floating point overflow in expression （表达式中发生浮点溢出。）

在叠算浮点常量表达式时，编译器发现表达式发生溢出，也即，它不能用 float 表示。

‘*type1*’ format, ‘*type2*’ arg (arg ‘*num*’) （格式为 “*type1*”，参数为 “*type2*”（参数 “num”））

格式的类型为 “type1”，但传递的参数的类型为 “type2”。有问题的参数为 “num”参数。

format argument is not a pointer (arg *n*) （格式参数不是一个指针 （参数 *n*））

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现指定的参数号 *n* 不是指针，而格式说明符指示它应该是。

format argument is not a pointer to a pointer (arg *n*) （格式参数不是一个指向指针的指针 （参数 *n*））

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现指定的参数号 *n* 不是指针，而格式说明符指示它应该是。

fprefetch-loop-arrays not supported for this target （该目标不支持 fprefetch-loop-arrays）

该目标不支持生成预取存储器的指令。

function call has aggregate value （函数调用具有聚合值）

函数的返回值是一个聚合。

function declaration isn’t a prototype （函数声明不是一个原型）

使用 `-Wstrict-prototypes` 命令行选项编译时，编译器会确保为所有函数指定了函数原型。在此例中，遇到了一个函数定义，但前面没有函数原型。

function declared ‘noreturn’ has a ‘return’ statement （声明为 “noreturn”的函数具有 “return”语句）

一个函数声明了 noreturn 属性 （指示该函数不返回），但该函数包含 return 语句。这是不一致的。

© 2015 Microchip Technology Inc.

function might be possible candidate for attribute ‘noreturn’ （函数可能可以使用属性 “noreturn”）

编译器检测到该函数不返回。如果该函数使用 “noreturn” 属性进行声明，则编译器可能能够生成更好的代码。

function returns address of local variable （函数返回局部变量的地址）

函数不应返回局部变量的地址，因为当函数返回时，局部变量会被释放存储空间。

function returns an aggregate （函数返回一个聚合）

函数的返回值是一个聚合。

function ‘*name*’ redeclared as inline （函数 “*name*” 重新声明为 inline）
previous declaration of function ‘*name*’ with attribute noinline（函数 “*name*” 的先前声明带有属性 noinline）

函数 “name” 使用关键字 “inline” 进行了第二次声明，现在会允许考虑对该函数进行内联。

function ‘*name*’ redeclared with attribute noinline （函数 “*name*” 使用属性 noinline 重新声明）
previous declaration of function ‘*name*’ was inline （函数 “*name*” 的先前声明为 inline）

函数 “name” 使用 noinline 属性进行了第二次声明，使它现在不适合于进行内联。

function ‘*identifier*’ was previously declared within a block（先前在一个块内声明了函数 “*identifier*”）

指定的函数先前在某个块中具有显式声明，但它在当前行上具有隐式声明。

G

GCC does not yet properly implement ‘[*]’ array declarators（GCC 尚未适当地实现 “[*]” 数组声明符）

该编译器当前不支持可变长度的数组。

H

hex escape sequence out of range （十六进制转义序列超出范围）

十六进制序列必须小于十六进制的 100 （十进制的 256）。

I

ignoring asm-specifier for non-static local variable ‘*identifier*’ （忽略非静态局部变量 “*identifier*” 的 asm 说明符）

将 asm 说明符用于普通的非寄存器局部变量时，它会被忽略。

ignoring invalid multibyte character （忽略无效的多字节字符）

在解析一个多字节字符时，编译器确定它是无效的。无效的字符被忽略。

ignoring option ‘*option*’ due to invalid debug level specification （由于指定的调试级别无效，忽略选项 “*option*”）

对不属于有效调试级别的调试级别使用了调试选项。

ignoring #pragma *identifier* （忽略 #pragma *identifier*）

该编译器不支持指定的 pragma 伪指令，它被忽略。

imaginary constants are a GCC extention （虚常量是一项 GCC 扩展）

ISO C 不允许虚数常量。

implicit declaration of function ‘*identifier*’ （隐式声明的函数 “*identifier*”）

指定函数没有先前的显式声明（定义或函数原型），所以编译器对其返回类型和参数作出了假定。

increment of read-only member ‘*name*’（对只读成员 “*name*”进行递增）

成员 “name”被声明为 const，无法通过递增修改。

increment of read-only variable ‘*name*’（对只读变量 “*name*”进行递增）

“name”被声明为 const，无法通过递增修改。

initialization of a flexible array member（灵活数组成员进行初始化）

灵活数组成员旨在进行动态分配而不是静态分配。

‘*identifier*’ initialized and declared ‘extern’（“identifier”进行了初始化，并被声明为 “extern”）

不应初始化 extern 对象。

initializer element is not constant（初始值设定项元素不是常量）

初始值设定项元素应为常量。

inline function ‘*name*’ given attribute noinline（对内联函数 “*name*”使用了属性 noinline）

函数 “name”已声明为 inline，但 noinline 属性阻止考虑对该函数进行内联。

inlining failed in call to ‘*identifier*’ called from here（对从此处调用 “*identifier*”的调用进行内联失败）

编译器无法内联对指定函数的调用。

integer constant is so large that it is unsigned（整型常量太大，所以将它设为无符号）

在源代码中出现的某个整型常量值并没有显式的 unsigned 修饰符，但该数值无法以 signed int 类型表示；因此，编译器自动将它视为 unsigned int 类型。

integer constant is too large for ‘*type*’ type（整型常量值对于 “*type*”类型来说太大）

对于 unsigned long int，整型常量不应超出 2^32 - 1，对于 long long int 不应超出 2^63 - 1，对于 unsigned long long int 不应超出 2^64 - 1。

integer overflow in expression（表达式中发生整型溢出）

在叠算整型常量表达式时，编译器发现表达式发生溢出，也即，它不能用 int 表示。

invalid application of ‘sizeof’ to a function type（对函数类型无效地应用 “sizeof”）

建议不要对函数类型应用 sizeof 操作符。

invalid application of ‘sizeof’ to a void type（对 void 类型无效地应用 “sizeof”）

sizeof 操作符不应应用于 void 类型。

invalid digit ‘digit’ in octal constant（八进制常量中存在无效数字 “digit”）

所有数字都必须处于所使用的基数的范围内。例如，对于于八进制基数，只能使用数字 0 至 7。

invalid second arg to __builtin_prefetch; using zero（__builtin_prefetch 的第二个参数无效；使用零）

第二个参数必须是 0 或 1。

invalid storage class for function ‘*name*’（函数 “*name*”的存储类无效）

不应对在最高层级上定义的函数使用 “auto”存储类。如果函数不是在最高层级上定义的，则不应使用 “static”存储类。

invalid third arg to __builtin_prefetch; using zero（__builtin_prefetch 的第三个参数无效；使用零）

第三个参数必须是 0、1、2 或 3。

‘*identifier*’ is an unrecognized format function type（“*identifier*”是一种无法识别的格式函数类型）

用于 format 属性的指定 *identifier* 不是可识别的格式函数类型 `printf`、`scanf` 或 `strftime` 之一。

© 2015 Microchip Technology Inc.

'*identifier*' is narrower than values of its type（*"identifier"* 的长度小于其类型的值）

结构的某个位域成员的类型为枚举，但位域的宽度不足以表示所有枚举值。

'*storage class*' is not at beginning of declaration （存储类 *"storage class"* 不是处于声明的开头）

指定的存储类不是处于声明的开头。存储类需要处于声明的最前面。

ISO C does not allow extra ';' outside of a function（ISO C 不允许在函数外具有额外的 ";"）

在函数外发现额外的 ";"。 ISO C 不允许这样。

ISO C does not support '++' and '--' on complex types （ISO C 不支持对复数类型执行 "++" 和 "--"

ISO C 不支持对复数类型使用递增操作符和递减操作符。

ISO C does not support '~' for complex conjugation（ISO C 不支持对复共轭执行 "~"）

在 ISO C 中，按位取反操作符不能用于复共轭。

ISO C does not support complex integer types （ISO C 不支持复整型类型）

ISO C 不支持复整型类型，例如， `__complex__` short int。

ISO C does not support plain 'complex' meaning 'double complex' （ISO C 不支持普通 "complex" 表示 "double complex"）

ISO C不支持不带另一个修饰符情况下使用`__complex__`等效于"complex double"。

ISO C does not support the '*char*' '*kind of format*' format（ISO C 不支持 *"char"* "kind of format" 格式）

对于指定的格式类型 "kind of format"， ISO C 不支持使用说明字符 "char"。

ISO C doesn't support unnamed structs/unions（ISO C不支持未命名的结构/联合）

在 ISO C 中，所有结构和 / 或联合都必须命名。

ISO C forbids an empty source file （ISO C 禁止空的源文件）

该文件不包含任何函数或数据。 ISO C 不允许这样。

ISO C forbids empty initializer braces （ISO C 禁止空的初始值设定项花括号）

ISO C 要求花括号内具有初始值设定项值。

ISO C forbids nested functions （ISO C 禁止嵌套的函数）

在一个函数内定义了另一个函数。

ISO C forbids omitting the middle term of a ?: expression （ISO C 禁止省略 ?: 表达式的中间项）

条件表达式需要 "?" 和 ":" 之间的中间项或表达式。

ISO C forbids qualified void function return type （ISO C 禁止限定 void 函数返回类型）

不能对 void 函数返回类型使用限定符。

ISO C forbids range expressions in switch statements （ISO C 禁止在 switch 语句中使用范围表达式）

在 ISO C 中，不允许在单个 case 标号中指定一个连续值的范围。

ISO C forbids subscripting 'register' array（ISO C 禁止对 "register" 数组使用下标）

ISO C 不允许对 "register" 数组使用下标。

ISO C forbids taking the address of a label （ISO C 禁止获取标号的地址）

ISO C 不允许获取标号的地址。

ISO C forbids zero-size array '*name*' （ISO C 禁止大小为零的数组 *"name"*）

"name" 的数组大小必须大于零。

ISO C restricts enumerator values to range of 'int' （ISO C 将枚举数的值限制在“int”范围内）

枚举数的值的范围不能超出 int 类型的范围。

ISO C89 forbids compound literals （ISO C89 禁止复合字面值）

复合字面值在 ISO C89 中是无效的。

ISO C89 forbids mixed declarations and code （ISO C89 禁止混合的声明和代码）

在编写任何代码之前，应先进行声明。不应将它与代码混在一起。

ISO C90 does not support '[*]' array declarators（ISO C90 不支持“[*]”数组声明符）

ISO C90 不支持可变长度数组。

ISO C90 does not support complex types （ISO C90 不支持复数类型）

ISO C90 不支持复数类型，例如， `__complex__ float x`。

ISO C90 does not support flexible array members（ISO C90 不支持灵活数组成员）

灵活数组成员是 C99 中的一项新特性。 ISO C90 不支持它。

ISO C90 does not support 'long long' （ISO C90 不支持 “long long”）

ISO C90 不支持 `long long` 类型。

ISO C90 does not support 'static' or type qualifiers in parameter array declarators （ISO C90 不支持在参数数组声明符中使用 “static”或类型限定符。）

使用数组作为函数的参数时，ISO C90 不允许数组声明符使用“static”或类型限定符。

ISO C90 does not support the '*char*' '*function*' format （ISO C90 不支持 “*char*” “*function*”格式）

对于指定的函数格式， ISO C 不支持使用说明字符 “char”。

ISO C90 does not support the '*modifier*' '*function*' length modifier （ISO C90 不支持 “*modifier*”“*function*”长度修饰符）

不支持使用指定的修饰符作为给定函数的长度修饰符。

ISO C90 forbids variable-size array '*name*' （ISO C90 禁止可变大小的数组 “*name*”）

在 ISO C90 中，数组中的元素数量必须使用整型常量表达式指定。

L

label '*identifier*' defined but not used （定义了标号 “*identifier*”，但未使用）

指定的标号已定义，但未引用。

large integer implicitly truncated to unsigned type（大整型被隐式地截断为无符号类型）

在源代码中出现的某个整型常量值并没有显式的 unsigned 修饰符，但该数值无法以 signed int 类型表示；因此，编译器自动将它视为 unsigned int 类型。

left-hand operand of comma expression has no effect （逗号表达式的左侧操作数不起作用）

比较的操作数之一是经过提升的无符号值，而另一个为无符号值。

left shift count >= width of type （左移位数 >= 类型的宽度）

移位位数应小于要进行移位的类型中的位数。否则，移位是无意义的，并且结果是未定义的。

left shift count is negative （左移位数为负值）

移位位数应为正值。左移数为负值并不表示右移；它是无意义的。

© 2015 Microchip Technology Inc.

library function ‘*identifier*’ declared as non-function（库函数“*identifier*”被声明为非函数）

指定的函数与某个库函数同名，但被声明为非函数对象。

line number out of range （行号超出范围）

在 C89 中， #line 伪指令的行号限制为 32767，在 C99 中为 2147483647。

‘*identifier*’ locally external but globally static（“*identifier*”声明为局部外部对象，但又声明为全局静态对象）

指定的 *identifier* 既是局部外部对象，但又是全局静态对象。这是可疑的。

location qualifier ‘*qualifier*’ ignored （位置限定符 “*qualifier*”被忽略）

该编译器不使用位置限定符（包括“grp”和“sfr”），但提供它们是为了与用于 PIC18 MCU 的 MPLAB C 编译器保持兼容性。

‘long’ switch expression not converted to ‘int’ in ISO C（在 ISO C 中，“long”类型的 switch 表达式不会被转换为 “int”类型）

ISO C 不会将 “long”类型的 switch 表达式转换为 “int”类型。

M

‘main’ is usually a function （“main”通常是一个函数）

标识符 main 通常用作应用程序主入口点的名称。编译器检测到它被以某种其他方式使用，例如，作为一个变量的名称。

‘*operation*’ makes integer from pointer without a cast（操作“*operation*”基于指针产生整型，而未使用强制类型转换）

一个指针被隐式转换为一个整型。

‘*operation*’ makes pointer from integer without a cast（操作“*operation*”基于整型产生指针，而未使用强制类型转换）

一个整型被隐式转换为一个指针。

malformed ‘#pragma pack-ignored’ （“#pragma pack-ignored”格式不正确）

pack pragma 伪指令的语法不正确。

malformed ‘#pragma pack(pop[,id])-ignored’（“#pragma pack(pop[,id])-ignored”格式不正确）

pack pragma 伪指令的语法不正确。

malformed ‘#pragma pack(push[,id],<n>)-ignored’ （“#pragma pack(push[,id],<n>)-ignored”格式不正确）

pack pragma 伪指令的语法不正确。

malformed ‘#pragma weak-ignored’ （“#pragma weak-ignored”格式不正确）

weak pragma 伪指令的语法不正确。

‘*identifier*’ might be used uninitialized in this function （该函数中可能在未初始化的情况下使用了 “identifier”）

编译器检测到函数中的一条控制路径可能在初始化指定标识符之前使用它。

missing braces around initializer （初始值设定项两边缺少花括号）

初始值设定项两边缺少一组必需的花括号。

missing initializer （缺少初始值设定项）

缺少初始值设定项。

modification by ‘asm’ of read-only variable ‘*identifier*’ （“asm”修改了只读变量“*identifier*”）

某个 const 变量在 “asm”语句中处于赋值的左侧。

multi-character *character* constant （多字符的字符常量）

一个字符常量包含多个字符。

N

negative integer implicitly converted to unsigned type （负整型被隐式地转换为无符号类型）

源代码中出现了一个负整型常量值，但该数值无法以 signed int 类型表示；因此，编译器自动将它视为 unsigned int 类型。

nested extern declaration of '*identifier*' （“*identifier*”嵌套的 extern 声明）

指定的 *identifier* 存在嵌套的 extern 定义。

no newline at end of file （文件末尾没有换行符）

源文件的最后一行未使用换行符终止。

no previous declaration for '*identifier*' （不存在“*identifier*”的先前声明）

使用 `-Wmissing-declarations` 命令行选项编译时，编译器会确保函数先声明后定义。在此例中，遇到了一个函数定义，但前面没有函数声明。

no previous prototype for '*identifier*' （不存在“*identifier*”的先前原型）

使用 `-Wmissing-prototypes` 命令行选项编译时，编译器会确保为所有函数指定了函数原型。在此例中，遇到了一个函数定义，但前面没有函数原型。

no semicolon at end of struct or union （结构或联合的末尾没有分号）

结构或联合声明的末尾缺少一个分号。

non-ISO-standard escape sequence, '*seq*' （非 ISO 标准的转义序列“*seq*”）

“seq”为 “\e”或 “\E”，它是 ISO 标准的扩展。该序列可以在字符串或字符常量中使用，表示 ASCII 字符 <ESC>。

non-static declaration for '*identifier*' follows static （“*identifier*”的非静态声明前具有静态声明）

指定标识符在先前声明为静态之后又声明为非静态。

'noreturn' function does return （“noreturn”函数发生返回）

使用 *noreturn* 属性声明的函数发生返回。这是不一致的。

'noreturn' function returns non-void value （“noreturn”函数返回非 void 值）

使用 *noreturn* 属性声明的函数返回非 void 值。这是不一致的。

null format string （格式字符串为空）

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现缺少格式字符串。

O

octal escape sequence out of range （八进制转义序列超出范围）

八进制序列必须小于八进制的 400（十进制的 256）。

output constraint '*constraint*' for operand *n* is not at the beginning（操作数 *n* 的输出约束“*constraint*”不是处于开头）

扩展 asm 中的输出约束应位于开头。

overflow in constant expression （常量表达式中发生溢出）

常量表达式超出了其类型的可表示值范围。

overflow in implicit constant conversion （隐式常量转换中发生溢出）

隐式常量转换产生的数值无法以 signed int 类型表示；因此，编译器会自动将它视为 unsigned int 类型。

© 2015 Microchip Technology Inc.

P

parameter has incomplete type （参数具有不完整的类型）

函数参数具有不完整的类型。

parameter names (without types) in function declaration （函数声明中只有参数名称 （无类型））

函数声明列出了参数的名称，但未给出它们的类型。

parameter points to incomplete type （参数指向不完整的类型）

函数参数指向不完整的类型。

parameter '*identifier*' points to incomplete type （参数 "*identifier*" 指向不完整的类型）

指定的函数参数指向不完整的类型。

passing arg '*number*' of '*name*' as complex rather than floating due to prototype （由于原型的原因，以复数值而不是浮点值的形式传递 "*name*" 的参数 "*number*"）

原型将参数 "number" 声明为复数，但使用了一个浮点值，所以编译器将其转换为一个复数值，以便与原型一致。

passing arg '*number*' of '*name*' as complex rather than integer due to prototype （由于原型的原因，以复数值而不是整型值的形式传递 "*name*" 的参数 "*number*"）

原型将参数 "number" 声明为复数，但使用了一个整型值，所以编译器将其转换为一个复数值，以便与原型一致。

passing arg '*number*' of '*name*' as floating rather than complex due to prototype （由于原型的原因，以浮点值而不是复数值的形式传递 "*name*" 的参数 "*number*"）

原型将参数 "number" 声明为浮点型，但使用了一个复数值，所以编译器将其转换为一个浮点值，以便与原型一致。

passing arg '*number*' of '*name*' as 'float' rather than 'double' due to prototype （由于原型的原因，以单精度浮点值而不是双精度浮点值的形式传递 "*name*" 的参数 "*number*"）

原型将参数 "number" 声明为单精度浮点型，但使用了一个双精度浮点值，所以编译器将其转换为一个单精度浮点值，以便与原型一致。

passing arg '*number*' of '*name*' as floating rather than integer due to prototype （由于原型的原因，以浮点值而不是整型值的形式传递 "*name*" 的参数 "*number*"）

原型将参数 "number" 声明为浮点型，但使用了一个整型值，所以编译器将其转换为一个浮点值，以便与原型一致。

passing arg '*number*' of '*name*' as integer rather than complex due to prototype （由于原型的原因，以整型值而不是复数值的形式传递 "*name*" 的参数 "*number*"）

原型将参数 "number" 声明为整型，但使用了一个复数值，所以编译器将其转换为一个整型值，以便与原型一致。

passing arg '*number*' of '*name*' as integer rather than floating due to prototype （由于原型的原因，以整型值而不是浮点值的形式传递 "*name*" 的参数 "*number*"）

原型将参数 "number" 声明为整型，但使用了一个浮点值，所以编译器转换为一个整型值，以便与原型一致。

pointer of type 'void *' used in arithmetic（在算术运算中使用了 "void*" 类型的指针）

"void" 类型的指针没有大小，不应在算术运算中使用。

pointer to a function used in arithmetic （算术运算中使用了指向函数的指针）

算术运算中不应使用指向函数的指针。

previous declaration of '*identifier*' （先前声明了 "*identifier*"）

该警告消息与另一条警告消息一起出现。前面的消息标识可疑代码的位置。该消息标识 *identifier* 的第一个声明或定义。

previous implicit declaration of '*identifier*' （先前隐式声明了 "*identifier*"）

该警告消息与 "类型与先前的隐式声明不匹配" 警告消息一起出现。它定位与标识符的显式声明冲突的隐式声明。

R

“*name*” reasserted （重新断言了 “*name*”）

“*name*”的答案重复。

“*name*” redefined （重新定义了 “*name*”）

“name”先前已定义，现在又重新定义。

redefinition of ‘*identifier*’ （重新定义了 “*identifier*”）

指定的标识符具有多个不兼容的定义。

redundant redeclaration of ‘*identifier*’ in same scope（在同一作用域内多余地重新声明了 “*identifier*”）

指定的标识符在同一作用域内重新声明。这是多余的。

register used for two global register variables（寄存器被用于两个全局寄存器变量）

两个全局寄存器变量被定义为使用同一个寄存器。

repeated ‘*flag*’ flag in format （格式中存在重复的 “*flag*” 标志）

在检查对 *strftime* 的调用的参数列表时，编译器发现某个标志在格式字符串中重复。

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现 { ,+,#,0,-} 中的标志之一在格式字符串中重复。

return-type defaults to ‘int’ （返回类型默认设为 “int”）

在没有显式函数返回类型声明的情况下，编译器假定函数返回一个 int 类型。

return type of ‘*name*’ is not ‘int’ （“*name*” 的返回类型不是 “int”）

编译器预期 “*name*” 的返回类型为 “int”。

‘return’ with a value, in function returning void（在返回 void 的函数中，“return” 带有一个值）

函数被声明为 void，但返回了一个值。

‘return’ with no value, in function returning non-void （在返回非 void 的函数中，“return” 不带有值）

声明为返回非 void 值的函数包含不带返回值的 return 语句。这是不一致的。

right shift count >= width of type （右移位数 >= 类型的宽度）

移位位数应小于要进行移位的类型中的位数。否则，移位是无意义的，并且结果是未定义的。

right shift count is negative （右移位数为负值）

移位位数应为正值。右移位数为负值并不表示左移；它是无意义的。

S

second argument of ‘*identifier*’ should be ‘char **’ （“*identifier*” 的第二个参数应为 “char **”）

要求指定标识符的第二个参数的类型为 “char **”。

second parameter of ‘va_start’ not last named argument （“va_start” 的第二个参数不是最后一个指定参数）

“va_start” 的第二个参数必须是最后一个指定参数。

shadowing built-in function ‘*identifier*’ （遮蔽内置函数 “*identifier*”）

指定函数与一个内置函数同名，因而遮蔽了该内置函数。

shadowing library function ‘*identifier*’ （遮蔽库函数 “*identifier*”）

指定函数与一个库函数同名，因而遮蔽了该库函数。

© 2015 Microchip Technology Inc.

shift count >= width of type （移位位数 >= 类型的宽度）

移位位数应小于要进行移位的类型中的位数。否则，移位是无意义的，并且结果是未定义的。

shift count is negative （移位位数为负值）

移位位数应为正值。左移位数为负值并不意味着右移，右移位数为负值也不意味着左移；它们是无意义的。

size of '*name*' is larger than *n* bytes （“*name*”的大小大于 “*n*”个字节）

使用 `-Wlarger-than-len` 时，如果 “name” 的大小大于所定义的 *len* 个字节，则会产生以上警告。

size of '*identifier*' is n bytes （“*identifier*”的大小为 “*n*”个字节）

指定标识符的大小（*n* 字节）大于使用 `-Wlarger-than-`*len* 命令行选项指定的大小。

size of return value of '*name*' is larger than *n* bytes （“*name*”的返回值的大小大于 “*n*”个字节）

使用 `-Wlarger-than-len` 时，如果 “name” 的返回值的大小大于所定义的 *len* 个字节，则会产生以上警告。

size of return value of '*identifier*' is n bytes （“*identifier*”的返回值的大小为 n 个字节）

指定函数的返回值的大小为 *n* 个字节，它大于使用 `-Wlarger-than-`*len* 命令行选项指定的大小。

spurious trailing '%' in format （格式中存在谬误的尾随 “%”）

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现格式字符串中存在谬误的尾随 “%” 字符。

statement with no effect （不起作用的语句）

语句没有任何作用。

static declaration for '*identifier*' follows non-static （“identifier”的静态声明前具有非静态声明）

指定标识符在先前声明为非静态之后又声明为静态。

string length '*n*' is greater than the length '*n*' ISO C*n* compilers are required to support （字符串长度 “*n*” 大于 ISO Cn 编译器需要支持的长度 “n”）

ISO C89 的最大字符串长度为 509。 ISO C99 的最大字符串长度为 4095。

'struct *identifier*' declared inside parameter list （参数列表内声明了结构标识符 “*identifier*”）

在函数参数列表内声明了指定的结构。在参数列表外声明结构通常是更好的编程做法，因为在参数列表内定义它们时，它们不会成为完整的类型。

struct has no members （结构没有任何成员）

结构为空，它没有任何成员。

structure defined inside parms （参数内定义了结构）

函数参数列表内定义了联合。

style of line directive is a GCC extension （line 伪指令的样式是一项 GCC 扩展）

对于传统 C，请使用 “#line linenum” 格式

subscript has type 'char' （下标具有 “char” 类型）

数组下标具有 “*char*” 类型。

suggest explicit braces to avoid ambiguous 'else' （建议明确加上花括号以避免存在歧义的 “else”）

嵌套的 if 语句具有存在歧义的 else 子句。建议使用花括号来消除歧义。

suggest hiding #*directive* from traditional C with an indented #（建议通过使用缩进的 # 来对传统 C 隐藏伪指令 “#*directive*”）

指定的伪指令不是传统 C 的伪指令，可以通过缩进 # 进行“隐藏”。除非伪指令的 # 处于第一列，否则它会被忽略。

suggest not using #elif in traditional C （建议不要在传统 C 代码中使用 #elif）

在传统 K&R C 中不应使用 #elif。

suggest parentheses around assignment used as truth value（建议在用作真值的赋值前后加上括号）

将赋值用作真值时，应将它们包围在括号内，使源程序的读者可以明确了解这种意图。

suggest parentheses around + or - inside shift（建议在移位内，在 + 或 - 前后加上圆括号）
suggest parentheses around && within || （建议在 || 内，在 && 前后加上圆括号）
suggest parentheses around arithmetic in operand of | （建议在 | 的操作数中，在算术运算前后加上圆括号）
suggest parentheses around comparison in operand of | （建议在 | 的操作数中，在比较前后加上圆括号）
suggest parentheses around arithmetic in operand of ^ （建议在 ^ 的操作数中，在算术运算前后加上圆括号）
suggest parentheses around comparison in operand of ^（建议在 ^ 的操作数中，在比较前后加上圆括号）
suggest parentheses around + or - in operand of & （建议在 & 的操作数中，在 + 或 - 前后加上圆括号）
suggest parentheses around comparison in operand of &（建议在 & 的操作数中，在比较前后加上圆括号）

虽然 C 中很好地定义了操作符优先级，但如果表达式的读者必须仅依靠优先级规则，而没有显式括号的帮助，有时读者可能需要花费一些额外时间来理解表达式中的操作数的求值顺序。一个典型示例就是在移位内使用 “+” 或 “-” 操作符。如果使用圆括号来清楚地表达编程人员的意图，许多读者可以免去不必要的精力消耗，即使表达式的意图对于编程人员和编译器都很明确。

T

‘*identifier*’ takes only zero or two arguments （“*identifier*”仅接受零个或两个参数）

仅预期零个或两个参数。

the meaning of ‘\a’ is different in traditional C（“\a”的含义在传统 C 中是不同的）

当使用 `-wtraditional` 选项时，转义序列 “\a” 不会被识别为元序列：它的值仅仅为 “a”。在非传统编译中，“\a” 代表 ASCII BEL 字符。

the meaning of ‘\x’ is different in traditional C（“\x”的含义在传统 C 中是不同的）

当使用 `-wtraditional` 选项时，转义序列 “\x” 不会被识别为元序列：它的值仅仅为 “x”。在非传统编译中，“\x” 会引入一个十六进制转义序列。

third argument of ‘*identifier*’ should probably be ‘char **’（“*identifier*”的第三个参数可能应为 “char **”）

要求指定标识符的第三个参数的类型为 “char **”。

this function may return with or without a value（该函数可能返回值也可能不返回值）

非 void 函数的所有退出路径都应返回一个适当的值。编译器检测到非 void 函数终止，但有时具有显式返回值有时没有显式返回值的情况。因此，返回值可能是不可预知的。

this target machine does not have delayed branches （该目标机器不具有延迟转移）

不支持 `-fdelayed-branch` 选项。

too few arguments for format （格式的参数太少）

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现实际参数的数量少于格式字符串需要的数量。

© 2015 Microchip Technology Inc.

too many arguments for format （格式的参数太多）

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现实际参数的数量多于格式字符串需要的数量。

traditional C ignores #'*directive*' with the # indented（传统 C 会忽略将 # 进行缩进的 “*#directive*”伪指令）

传统上，除非伪指令的 # 处于第一列，否则它会被忽略。

traditional C rejects initialization of unions （传统 C 拒绝对联合进行初始化）

在传统 C 中不能初始化联合。

traditional C rejects the 'ul' suffix （传统 C 拒绝 “ul”后缀）

后缀 “u”在传统 C 中是无效的。

traditional C rejects the unary plus operator （传统 C 拒绝一元加号操作符）

一元加号加操作符在传统 C 中是无效的。

trigraph ??*char* converted to *char* （三字符组 ??*char* 被转换为 *char*）

三字符组是一个三字符序列，可以用于表示键盘中可能缺少的符号。三字符组序列的转换如下：

??(= [	??) =]	??< = {	??> = }	??= = #	??/ = \	??' = ^	??! = |	??- = ~

trigraph ??*char* ignored （三字符组 ??*char* 忽略）

三字符组序列被忽略。 *char* 可以为 (、)、 <、 >、 =、 /、 '、 ! 或 -。

type defaults to 'int' in declaration of '*identifier*'（在 “*identifier*” 的声明中，类型默认设为 “int”）

在没有指定 *identifier* 的显式类型声明时，编译器会假定其类型为 int。

type mismatch with previous external decl （类型与先前的外部声明不匹配）
previous external decl of '*identifier*' （“*identifier*” 存在先前的外部声明）

指定标识符的类型与先前声明不匹配。

type mismatch with previous implicit declaration（类型与先前的隐式声明不匹配）

显式声明与先前的隐式声明冲突。

type of '*identifier*' defaults to 'int' （“*identifier*” 的类型默认设为 “int”）

在没有显式类型声明时，编译器假定 *identifier* 的类型为 int。

type qualifiers ignored on function return type （函数返回类型的类型限定符被忽略）

用于函数返回类型的类型限定符被忽略。

U

undefining 'defined' （取消定义 “defined”）

“defined”不能用作宏的名称，不应取消定义。

undefining '*name*' （取消定义 “*name*”）

#undef 伪指令被用于一个先前定义的宏名称 “name”。

union cannot be made transparent （联合无法设置为透明）

`transparent_union` 属性被应用于一个联合，但指定变量不符合该属性的要求。

'union *identifier*' declared inside parameter list（参数列表内声明了联合 “*identifier*”）

在函数参数列表内声明了指定的联合。在参数列表外声明联合通常是更好的编程做法，因为在参数列表内定义它们时，它们不会成为完整的类型。

union defined inside parms （参数内定义了联合）

函数参数列表内定义了联合。

union has no members （联合没有任何成员）

联合为空，它没有任何成员。

unknown conversion type character ‘*character*’ in format（格式中存在未知的转换类型字符 “*character*”）

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现格式字符串中有一个转换字符是无效的 （无法识别）。

unknown conversion type character 0x*number* in format（格式中存在未知的转换类型字符 0x*number*）

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现格式字符串中有一个转换字符是无效的 （无法识别）。

unknown escape sequence ‘*sequence*’ （未知的转义序列 “*sequence*”）

“sequence”不是一个有效的转义代码。转义代码必须以 “\” 开头，并使用以下字符之一：n、t、b、r、f、b、\、'、"、或 ?，或者它必须是以八进制或十六进制表示的数字序列。以八进制表示时，数字序列必须小于八进制的 400。以十六进制表示时，数字序列必须以 “x” 开头，并小于十六进制的 100。

unnamed struct/union that defines no instances （未定义任何实例的未命名结构 / 联合）

结构 / 联合为空且没有名称。

unreachable code at beginning of *identifier*（“*identifier*” 开头具有无法到达的代码）

指定函数的开头具有无法到达的代码。

unrecognized gcc debugging option:*char* （无法识别的 gcc 调试选项：*char*）

“char” 不是 -d*letters* 调试选项的有效字母。

unused parameter ‘*identifier*’ （未用参数 “*identifier*”）

函数中未使用指定的函数参数。

unused variable ‘*name*’ （未用变量 “*name*”）

指定变量已声明，但未被使用。

use of ‘*’ and ‘flag’ together in format （在格式中将 “*” 和 “flag” 一起使用）

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现格式字符串中同时出现了 “*” 和 “flag”。

use of C99 long long integer constants （使用了 C99 long long 类型的整型常量）

在 ISO C89 中，不允许将整型常量声明为 long long 类型。

use of ‘*length*’ length modifier with ‘*type*’ type character（对 “type” 类型字符使用了 “*length*” 长度修饰符）

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现对指定的 *type* 错误地使用了指定的长度。

‘*name*’ used but never defined （使用了 “*name*”，但从未定义）

使用了指定函数，但从未定义。

‘*name*’ used with ‘*spec*’ ‘*function*’ format （“*name*” 被用于 “*spec*” “*function*” 格式）

“name” 对于指定函数的格式中的转换说明 “spec” 是无效的。

useless keyword or type name in empty declaration （空声明中存在无用的关键字或类型名称）

空声明中包含无用的关键字或类型名称。

© 2015 Microchip Technology Inc.

V

__VA_ARGS__ can only appear in the expansion of a C99 variadic macro（__VA_ARGS__ 只能出现在 C99 可变参数宏的扩展中）

预定义宏 __VA_ARGS 应在使用省略号的宏定义的替换部分中使用。

value computed is not used （计算的值未被使用）

计算的值未被使用。

variable '*name*' declared 'inline'（变量 "*name*" 被声明为 "inline"）

关键字 "inline" 应仅用于函数。

variable '%s' might be clobbered by 'longjmp' or 'vfork'（变量 "%s" 可能会被 "longjmp" 或 "vfork" 破坏）

非可变自动变量可能被对 longjmp 的调用更改。只有在优化编译时才可能出现这些警告。

volatile register variables don't work as you might wish （可变寄存器变量可能不会按您希望的方式工作）

将变量作为参数传递可能会将变量传递到指定用于参数传递的寄存器（如果不是 w0-w7）之外的其他寄存器（w0-w7）。或者，编译器可能会发出不适合于指定寄存器并可能需要临时将值传送到另一个位置的指令。只有指定寄存器会被异步修改（即，通过 ISR 修改）时，才会产生这些问题。

W

-Wformat-extra-args ignored without -Wformat（没有 -Wformat 的情况下，-Wformat-extra-args 被忽略）

要使用 `-Wformat-extra-args`，必须指定 `-Wformat`。

-Wformat-nonliteral ignored without -Wformat（没有 -Wformat 的情况下，-Wformat-nonliteral 被忽略）

要使用 `-Wformat-nonliteral`，必须指定 `-Wformat`。

-Wformat-security ignored without -Wformat（没有 -Wformat 的情况下，-Wformat-security 被忽略）

要使用 `-Wformat-security`，必须指定 `-Wformat`。

-Wformat-y2k ignored without -Wformat（没有 -Wformat 的情况下，-Wformat-y2k 被忽略）

必须指定 `-Wformat` 才能使用。

-Wid-clash-LEN is no longer supported （不再支持 -Wid-clash-LEN）

不再支持选项 `-Wid-clash-LEN`。

-Wmissing-format-attribute ignored without -Wformat（没有 -Wformat 的情况下，-Wmissing-format-attribute 被忽略）

要使用 `-Wmissing-format-attribute`，必须指定 `-Wformat`。

-Wuninitialized is not supported without -O（没有 -O 的情况下，不支持 -Wuninitialized）

只有开启优化的情况下才能使用 `-Wuninitialized` 选项。

'*identifier*' was declared 'extern' and later 'static'（"*identifier*" 先被声明为 "extern"，之后又被声明为 "static"）

指定标识符先前被声明为 "extern"，现在又被声明为 static。

'*identifier*' was declared implicitly 'extern' and later 'static'（"*identifier*" 先被隐式声明为 "extern"，之后又被声明为 "static"）

指定标识符先前被隐式声明为 "extern"，现在又被声明为 static。

'*identifier*' was previously implicitly declared to return 'int'（"*identifier*" 先前被隐式声明为返回 "int"）

与先前的隐式声明不匹配。

'*identifier*' was used with no declaration before its definition （在定义前没有声明的情况下使用了 *"identifier"*）

使用 `-Wmissing-declarations` 命令行选项编译时，编译器会确保函数先声明后定义。在此例中，遇到了一个函数定义，但前面没有函数声明。

'*identifier*' was used with no prototype before its definition（在定义前没有原型的情况下使用了 *"identifier"*）

使用 `-Wmissing-prototypes` 命令行选项编译时，编译器会确保为所有函数指定了函数原型。在此例中，遇到了一个函数调用，但前面没有被调用函数的函数原型。

writing into constant object (arg *n*) （写入常量对象 （参数 *n*））

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现指定的参数号 *n* 是一个 const 对象，而格式说明符指示对它进行写入。

Z

zero-length *identifier* format string （长度为零的 *identifier* 格式字符串）

在检查对 *printf* 和 *scanf* 等函数的调用的参数列表时，编译器发现格式字符串为空（""）。

© 2015 Microchip Technology Inc.

附录 D GNU 自由文档许可证

版本 1.3， 2008 年 11 月 3 日

Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
<http://fsf.org/>
任何人都可以复制和分发本许可证文档的逐字拷贝，但不允许对它进行修改。

D.1 前导

本许可证的目的在于使手册、教材或其他功能性的有用文档的性质自由：确保任何人都确实可以自由地复制和再分发这些文档，无论是否进行修改，或是否用于商业用途。此外，本许可证保留了作者和发行者由于他们的工作而得到信誉的方式，同时不会被认为应为他人所做的修改承担责任。

本许可证是一种 “copyleft”，这表示文档的衍生作品必须也是同样自由的。它是针对自由软件设计的 copyleft 许可证 GNU 通用公共许可证的补充。

我们设计本许可证的目的是将它用于自由软件的使用手册，因为自由软件需要自由的文档：一个自由的程序附带的手册应当具有与该软件同等的自由。但本许可证的适用范围不仅限于软件手册，它可以用于任何文本作品，无论它的主题或它是否作为一本印刷书籍被发行。建议将本许可证主要用于以提供说明或参考为目的的作品。

D.2 适用范围和定义

本许可证适用于采用任何媒介发布的任何手册或其他作品，其中包含了由版权所有者做出的声明，指明可以基于本许可证的条款分发它。此声明授予全球范围内的、免版税的且没有期限的许可，可在本文所陈述条件下使用该作品。以下的“文档”均指代此类手册或作品。公众中的任何成员都是被许可人，并且称作为“您”。如果您以一种需要在版权法下取得允许的方式复制、修改或分发作品，就表示您接受了本许可证。

文档的“修改版本”指的是任何包含文档或文档一部分内容的作品，无论是逐字复制还是经过修改，以及 / 或者翻译为其他语言。

“附属章节”是文档的具名附录或前序部分，专门用来处理文档的发行者或作者与文档整体主题（或相关内容）的关系，并且不包含任何可直接归入该整体主题的内容。（因而，如果文档的部分内容是作为数学教科书，那么附属章节就不可以用来解释任何数学。）其关系可以是与主题或相关内容的历史关联，或者是与其相关的法律、商业、哲学、伦理或政治立场。

“不可变章节”是在指明文档以本许可证发布的声明中，指出了标题（由于它们是不可变章节的标题）的一些特定附属章节。如果某个章节不符合上述的附属章节定义，就不允许将它指定为不可变章节。文档可以不包含不可变章节。如果文档未标识任何不可变章节，则没有不可变章节。

“封皮文字”是在指明文档以本许可证发布的声明中，列为封面文字或封底文字的一些简短文字段落。封面文字最多 5 个单词，封底文字最多 25 个单词。

文档的“透明”拷贝指的是一份机器可读的拷贝，它以一种一般公众可以获取其规范的格式来表示，适合于直接使用通用文本编辑器、（对于由像素构成的图像）通用绘图程序或（对于绘画）可广泛获取的绘画编辑器来修改文档，也适用于输入到文本格式处理程序或自动转换为适合于输入到文本格式处理程序的各种格式。一份用其他透明文件格式表示的拷贝，如果其标记或缺少标记构成读者进行后续修改的障碍，则此拷贝是不透明的。如果用于文本，不论数量多少，图像格式都是不透明的。不透明的拷贝被称为“非透明”。

适于作为透明拷贝的格式的示例包括：没有标记的纯 ASCII 文本、 Texinfo 输入格式、LaTeX 输入格式、使用可公开获取其 DTD 的 SGML 或 XML、符合标准的简单 HTML、可以手工修改的 PostScript 或 PDF。透明图像格式的示例包括 PNG、XCF 和 JPG。非透明格式包括：只能使用专有字处理程序读取和编辑的专有格式、其 DTD 和 / 或处理工具不可广泛获取的 SGML 或 XML、机器生成的 HTML、一些字处理程序仅为输出目的生成的 PostScript 或 PDF。

对于印刷书籍，“标题页”指的是标题页本身，以及需要用来容纳本许可证要求出现在标题页的易读内容之类的随后页面。对于没有此类标题页的作品形式，“标题页”指的是文本主体开始之前作品标题最显著位置的文字。

“发行者”指的是将文档拷贝分发给公众的任何人或实体。

“标题为 XYZ”的章节指的是文档的一个具名子单元，其标题恰为 XYZ 或是在译为其他语言的文字后的括号内包含 XYZ。（此处的 XYZ 代表的是下面提及的特定章节名称，例如“致谢”、“贡献”、“背书”或“历史”。）当您修改文档时，为此类章节“保留标题”指的是，它保留一个符合该定义的“标题为 XYZ”的章节。

文档可以在指明本许可证适用于文档的声明后包含免责声明。这些免责声明被认为是通过引用包含在本许可证中，但它们仅仅用于免除担保：这些免责声明可能具有的任何其他暗示都是无效的，并且也对本许可证的含义没有影响。

© 2015 Microchip Technology Inc.

D.3 逐字复制

您可以采用任何媒介复制和分发文档，无论是否用于商业用途，其条件为在所有拷贝中复制本许可证、版权声明和指明本许可证适用于文档的声明，并且不在本许可证的条件中增加任何其他条件。您不能使用技术手段来阻碍或控制您制作或分发的拷贝的阅读或再次复制。不过您可以在拷贝的交易中得到报酬。如果您分发的拷贝数量足够大，则必须遵循下面第三条中的条件。

您也可以在上述的相同条件下借出拷贝，并且您可以公开地陈列拷贝。

D.4 大量复制

如果您发行文档的印刷拷贝（或采用其他媒介的通常具有印刷封皮的拷贝）多于100份，而文档的许可声明中要求有封皮文字，则这些拷贝的封皮必须附上清楚且易读的文字：封面文字在封面上，封底文字在封底上。封面和封底上还必须清楚且易读地标明您是这些拷贝的发行者。封面必须同等显著、可见地完整展现标题的所有文字。您可以在封皮上加入其他内容。如果复制仅限于更改封皮，只要它们保留文档的标题并满足这些条件，可以在其他方面被视为逐字的复制。

如果需要加上的文字对于封面或封底来说内容过多，无法符合易读的原则，您应该在实际封皮上列出前面的文字 （在合理的前提下尽量多），将剩余文字放在相邻的页面上。

如果您发行或分发超过100份的文档的非透明拷贝，您必须随每个非透明拷贝包含一份机器可读的透明拷贝，或是在每个非透明拷贝中给出一个计算机网络地址，通过这个地址，使用计算机网络的公众可以使用公开标准的网络协议下载一份文档的完整透明拷贝，此拷贝中没有增加额外的内容。如果您选择后者，当您开始大量分发非透明拷贝时，必须采取合理的审慎步骤，确保该透明拷贝在发布一开始就在其所指定位置可供获取，直到您最后一次向公众分发（直接或通过代理商与零售商）该版本的非透明拷贝之后的至少一年之内。

在再分发任何大量拷贝之前，请您（但不是必须）与文档的作者联系，以便可以得到文档的更新版本。

D.5 修改

您可以在上述第二条和第三条的条件下，复制与分发文档的修改版本，其条件为严格地在本许可证下发布修改版本，且修改版本补足了文档的角色，从而将修改版本的分发和修改权利授予任何拥有其拷贝的人。此外，您必须在修改版本中做到如下几点：

a) 在标题页（以及封皮，如果有）上使用与原文档和以前的各个版本（如果有的话，应被列在文档的版本历史章节中）不同的标题。如果某个先前版本的原始发行者允许，您可以使用与该版本相同的标题。

b) 在标题页上列出承担修改版本中修改的作者责任的一个或多个人或实体作为作者，并列出至少五位原文档的主要作者（如果不足五位就列出全部主要作者），除非他们免除了您这个要求。

c) 在标题页陈述修改版本的发行者的名称作为发行者。

d) 保留文档的所有版权声明。

e) 为您的修改增加一个与其他版权声明相邻的适当的版权声明。

f) 紧接在版权声明后面，以下面的许可证附录所示的形式，包含一个允许公众在本许可证条款下使用修改版本的许可声明。

g) 在该许可声明中保留原文档的许可声明中给出的全部不可变章节和必需封皮文字。

h) 包含一份未作任何修改的本许可证的拷贝。

i) 保留标题为“历史”的章节，保留它的标题，并在其中增加一项，至少按标题页上给出的内容陈述修改版本的标题、年份、新作者和发行者。如果文档中没有标题为“历史”的章节，则创建这一章节，并按其标题页上给出的内容陈述原文档的标题、年份、作者与发行者，然后增加一项按前面句子所陈述的情形来描述修改版本。

j) 如果原文档中给出了供公众获取文档透明拷贝的网址，则保留网址，对于文档所基于的先前版本中给出的网址，也是如此。可以将它们放入“历史”章节。您可以省略在文档本身之前已经至少发行了四年的作品的网络地址，或是如果它所参照的那个版本的原始发行者给予允许的情形下也可以省略它。

k) 对于任何标题为“致谢”或“贡献”的章节，保留章节的标题，并在章节中保留其全部内容和对每个贡献者的致谢与列出的贡献的语气。

l) 保留文档的所有不可变章节，不改变它们的文字和标题。章节编号或等效内容不被视为章节标题的一部分。

m) 删除所有标题为“背书”的章节。修改版本中不能包含这样的章节。

n) 不要将任何现有章节的标题重新命名为“背书”或与任何不可变章节相冲突的标题。保留所有免责声明。

如果修改版本包含新的符合附属章节定义的前序章节或附录，并且其中不包含任何复制自原文档的内容，则您可以按照自己的意愿将其中部分或全部章节指定为不可变章节。如果要这样做，则将它们的标题加入修改版本的许可声明的不可变章节列表中。这些标题必须能与任何其他章节标题加以区别。

© 2015 Microchip Technology Inc.

您可以增加标题为“背书”的章节，其条件为它只包含各个当事人对于您的修改版本的背书——例如同侪评议的说明或说明文本已被某个机构认可为一个标准的权威定义。

您可以增加一个最多五个字的段落作为封面文字和一个最多 25 个字的段落作为封底文字，将它们增加到修改版本的封皮文字列表的末尾。任何一个实体（或任何一个实体所做出的安排）只能增加一段封面文字或封底文字。如果文档已经包含了同一封皮的封面文字——先前由您或由您代表的相同实体所作出的安排而加入，则您不可以另增一个；但是您可以在先前发行者的明确允许下替换掉旧文字。

文档的作者和发行者并不通过本许可证准许使用他们的名字进行宣传，或者声称或暗示对于任何修改版本的背书。

D.6 组合文档

您可以按照上述第四条对于修改版本规定的条款，将文档与其他在本许可证下发行的文档组合起来，其条件为，您要在组合作品中包括所有原始文档的所有不可变章节，不做修改，同时在组合作品的许可声明中将它们全部列为不可变章节，并且您要保留它们所有的免责声明。

组合作品只需包含本许可证的一份拷贝，并且多个相同的不可变章节可由单个拷贝代替。如果有多个名称相同但内容不同的不可变章节，则使得每个此类章节的标题唯一，方法是在标题后面用圆括号加上该章节的原始作者或发行者的名称（如果已知，否则加上一个唯一编号）在组合作品许可声明的不可变章节的列表中，对其章节标题做出相同的调整。

在组合作品中，必须组合不同原始文档中所有标题为“历史”的章节，形成一个标题为“历史”的章节；同样地，组合标题为“致谢”和“贡献”的章节。必须删除所有标题为“背书”的章节。

D.7 文档的合集

您可以制作含有文档以及其他以本许可证发布的文档的合集，并且将本许可证在不同文档中的各个拷贝替换为包含在合集中的单个拷贝，其条件为您要在所有其他方面遵循本许可证对于逐字复制每个文档的规定。

您可以从一个这样的合集中提取一个单独的文档，并在本许可证下单独分发它，其条件为您向该提取出的文档中加入本许可证的一份拷贝，并在所有其他方面遵循本许可证关于逐字拷贝该文档的规定。

D.8 独立作品的聚集

文档或其衍生作品和其他分离且独立的文档或作品编纂在一卷存储或分发媒介上，如果经由编纂产生的版权对编纂作品使用者的法律权利限制没有超出原来的独立作品的许可范围，则称为文档的聚集品。当文档被包含在一个聚集品中时，本许可证不适用于聚集品中不属于该文档衍生作品的其他作品。

如果第三条中的封皮文字要求适用于文档的这些拷贝，那么如果文档在聚集品中所占的比重小于一半，则可以将文档的封皮文字放在聚集品内包装文档的封皮上，或是电子形式文档中的等效电子封皮上。否则，它必须出现在包装整个聚集品的印刷封皮上。

D.9 翻译

翻译被视为是一种修改，所以您可以在第四条的条款下分发文档的译文。用译文更换不可变章节需要取得版权所有者的特别允许，但是您可以在保留这些不可变章节的原始版本的同时附加上部份或所有不可变章节的译文。您可以包括本许可证、文档中的所有许可声明和任何免责声明的译文，其条件为您也必须包括本许可证的原始英文版本，以及这些声明与免责声明的原始版本。当译文与本许可证、声明或免责声明的原始版本不一致时，以原始版本为准。

如果文档中某个章节的标题为“致谢”、“贡献”、“历史”，第四条的“保留标题”要求通常需要更改实际标题。

D.10 终止

除非明确表示在本许可证的条件下提供文档，否则您不可以复制、修改、附加许可或分发文档。任何其他的复制、修改、附加许可或分发文档的意图都是无效的，并且将会自动终止您在本许可证下获得的权利。

但是，如果您终止所有违反本许可证的行为，则您从特定版权所有者那里获得的许可会被 (a) 暂时恢复，直到该版权所有者明确并最终地终止您的许可证；或者，如果版权所有者在您停止违反许可证后的 60 天内没有通过合理的方式通知您违反许可证，则会被 (b) 永久性恢复。

此外，如果版权所有者通过一些合理的方式通知您违反了许可证规定（对于任意作品），而这是您第一次从版权所有者那里收到违反许可证的通知，并且您在收到通知后的 30 天内终止了这种行为，那么此版权所有者会永久地恢复您的许可证。

按照本条规定终止您的权利并不会终止按照本许可证从您那里得到您的拷贝和权利的当事人的许可证。如果您的权利被终止而没有被永久性地恢复，则获得部分或全部相同的材料的拷贝并不会赋予您使用它的任何权利。

D.11 本许可证的未来修订版

自由软件基金会可能会不时发布 GNU 自由文档许可证的新修订版本。这些新版本会秉持与当前版本类似的精神，但在细节上可能会有不同，以应对新的问题或相关的事。请参见 http://www.gnu.org/copyleft/。

本许可证的每个版本都会被指定一个可供区别的版本号。如果文档指定本许可证的特定编号版本或以后的任何版本适用于它，您可以选择遵循指定版本或自由软件基金会已发布的任何更新版本（非草稿）的条款和条件。如果文档未指定本许可证的版本号，那么

© 2015 Microchip Technology Inc.

您可以选择遵循自由软件基金会曾经发布的任何版本（非草稿）。如果文档指定代理人可以决定能使用本许可证的哪些未来版本，则代理人公开声明接受一个版本将永久地授予您为文档选择该版本的权利。

D.12 重新许可

“大规模多作者协作网站”（或“MMC 网站”）指的是任何发布受版权保护的作品并提供任何人均可编辑这些作品的卓越设施的万维网服务器。任何人都可以编辑的公众维基就是此类服务器的一个示例。该网站包含的“大规模多作者协作”（或“MMC”）指的是任何一组在 MMC 网站上发布的受版权保护的作品。

“CC-BY-SA”指的是由Creative Commons Corporation组织发布的Creative Commons Attribution-Share Alike 3.0 许可证，以及由同一组织发布的该许可证的未来 copyleft 版本；它是一家非盈利性组织，主要业务地点位于加利福尼亚州的旧金山。

“合并”指的是以整体或作为另一个文档的一部分发布或重新发布一个文档。

如果某个MMC在本许可证下获得许可，并且如果所有作品最初在本许可证下在该MMC以外发布，后来以整体或部分合并到 MMC 中， (1) 没有封皮文本或不可变章节，并且 (2) 在 2008 年 11 月 1 日之前合并，则该 MMC 符合重新许可的条件。

MMC 网站的运营者在 2009 年 8 月 1 日之前的任何时间可以按照 CC-BY-SA 在同一网站中重新发布网站上包含的 MMC，其条件为该 MMC 符合重新许可的条件。

注：

© 2015 Microchip Technology Inc.

附录 E ASCII 字符集

表 E-1： ASCII 字符集

低位字符 \ 高位字符	0	1	2	3	4	5	6	7
0	NUL	DLE	空格	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	Bell	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

注：

© 2015 Microchip Technology Inc.

附录 F 弃用功能

F.1 简介

下面介绍的功能被认为是过时的，已被替换为更高级的功能。对于所提到的语言工具版本，依赖于弃用功能的项目将可以正常工作。使用弃用功能会导致产生警告；建议编程人员修改其项目，以消除对于弃用功能的任何依赖。在语言工具的未来版本中，可能会完全去除对这些功能的支持。

涵盖的弃用功能包括：

- 预定义的常量
- 指定寄存器中的变量
- 更改非自动变量分配

F.2 预定义的常量

编译器会定义以下预处理符号。

符号	随 -ansi 命令行选项定义？
dsPIC30	否
__dsPIC30	是
__dsPIC30__	是

特定于 ELF 的编译器版本会定义以下预处理符号。

符号	随 -ansi 命令行选项定义？
dsPIC30ELF	否
__dsPIC30ELF	是
__dsPIC30ELF__	是

特定于 COFF 的编译器版本会定义以下预处理符号。

符号	随 -ansi 命令行选项定义？
dsPIC30COFF	否
__dsPIC30COFF	是
__dsPIC30COFF__	是

关于最新信息，请参见**第 19.4 节 “预定义的宏名称”**。

F.3 指定寄存器中的变量

编译器允许您将一些全局变量放入指定的硬件寄存器。

> **注：** 使用太多寄存器（特别是寄存器 W0）可能会削弱 16 位编译器进行编译的能力。建议不要将寄存器放入固定寄存器中。

您还可以指定普通寄存器变量应分配到的寄存器。

- 全局寄存器变量会在整个程序中保留寄存器。诸如编程语言解释器之类的程序具有一些经常访问的全局变量，对于这些程序，该功能可能会很有用。
- 处于特定寄存器中的局部寄存器变量不会保留寄存器。编译器的数据流分析能够确定指定寄存器何时包含有效的值，何时可用于其他用途。显现为未被使用时，对局部寄存器变量的存储操作可能会被删除。对局部寄存器变量的引用可能会被删除、移动或简化。

如果希望将汇编指令的一个输出直接写入特定寄存器，则有时可以将这些局部变量与扩展行内汇编代码方便地配合使用（见**第 16 章 “混合使用 C 代码和汇编代码”**）。（这样将可以实现目的，前提是您指定的寄存器符合在行内汇编语句中为该操作数指定的约束）。

F.3.1 定义全局寄存器变量

您可以定义全局寄存器变量，如下：

```
register int *foo asm ("w8");
```

此处的 w8 是应使用的寄存器的名称。选择一个通常会被函数调用保存和恢复的寄存器（W8-W13），从而使库程序不会破坏它。

在某个寄存器中定义全局寄存器变量会使该寄存器至少在当前编译中完全保留用于该用途。在当前编译中，该寄存器将不会被分配用于函数中的任何其他用途。这些函数将不会保存和恢复该寄存器。对该寄存器的存储操作永远不会被删除，即使它们显现为已无效，但引用可能会被删除、移动或简化。

从信号处理程序中或从多个控制线程中访问全局寄存器变量是不安全的，因为系统库程序可能会临时使用该寄存器来处理其他事项（除非您特别针对手中的任务重新编译它们）。

一个函数使用了某个全局寄存器变量，它通过一个函数 `lose` 来调用另一个此类函数 `foo`，而 `lose` 函数是在不知道该变量的情况下编译的（即，它处于未声明该变量的源文件中），这样的做法是不安全的。这是因为 `lose` 可能会保存该寄存器并将某个其他值放入其中。例如，不能预期某个全局寄存器变量在传递给 `qsort` 的比较函数中是可用的，因为 `qsort` 可能会在该寄存器中放入某个值。通过使用相同的全局寄存器变量定义来重新编译 `qsort`，可以避免这种问题。

如果要重新编译实际上不使用您的全局寄存器变量的 `qsort` 或其他源文件，从而使它们不会将该寄存器用于任何其他用途，则指定编译器命令行选项 `-ffixed-`*`reg`* 就足够了。不需要在它们的源代码中实际添加一个全局寄存器声明。

© 2015 Microchip Technology Inc.

如果某个函数会更改一个全局寄存器变量的值，而另一个函数是在没有该变量的情况下编译的，则无法从后面这个函数中安全地调用前面那个函数，因为它可能会破坏调用方预期在返回时获取的值。因此，如果某个函数是进入使用了全局寄存器变量的程序部分的入口点，则该函数必须显式地保存和恢复属于其调用方的值。

库函数 `longjmp` 会将每个全局寄存器变量恢复为在 `setjmp` 时具有的值。

所有全局寄存器变量声明必须位于所有函数定义之前。如果此类声明出现在函数定义之后，寄存器可能会在前面的函数中被用于其他用途。

全局寄存器变量可能没有初始值，因为可执行文件无法通过任何方式提供寄存器的初始内容。

F.3.2 为局部变量指定寄存器

您可以使用指定寄存器来定义局部寄存器变量，如下：

```
register int *foo asm ("w8");
```

此处的 `w8` 是应使用的寄存器的名称。请注意，该语法与用于定义全局寄存器变量的语法是相同的，但对于局部变量，它将出现在函数内。

定义此类寄存器变量并不会保留寄存器；在流程控制确定该变量的值无效时，它仍可用于其他用途。使用该功能时，留给编译器编译某些函数的可用寄存器可能会太少。

该选项并不能确保编译器生成的代码在任何时候都将该变量保留在您指定的寄存器中。您不能在 `asm` 语句中编写显式引用该寄存器、并假定它总是引用该变量的代码。

显现为未被使用时，对局部寄存器变量的赋值操作可能会被删除。对局部寄存器变量的引用可能会被删除、移动或简化。

F.4 更改非自动变量分配

定位数据的另一种方式是将变量放入一个用户定义的段，并在定制链接描述文件中指定该段的起始地址。其实现方式如下：

1. 在 C 源代码中修改数据声明，指定一个用户定义的段。
2. 将这个用户定义的段添加到定制链接描述文件中，指定该段的起始地址。

例如，要将变量 Mabonga 定位到数据存储器中的地址 0x1000 处，需要先在 C 源代码中如下声明变量：

```
int __attribute__((__section__(".myDataSection"))) Mabonga = 1;
```

section 属性指定该变量应放入名为 .myDataSection 的段，而不是默认的 .data 段。它并不指定将用户定义的段定位到何处。同样，这必须在定制链接描述文件中完成，如下所示。使用特定于器件的链接描述文件作为基础，添加以下段定义：

```
.myDataSection 0x1000 :
  {
        *(.myDataSection);
  } >data
```

该定义指定输出文件应包含名为 .myDataSection、从地址 0x1000 开始的段，并包含名为 .myDataSection 的所有输入段。该示例中，由于该段中存在单个变量 Mabonga，所以该变量会被定位到数据存储器中的地址 0x1000 处。

© 2015 Microchip Technology Inc.

附录 G 内置函数

G.1 简介

本附录列出了特定于 MPLAB XC16 C 编译器的内置函数。

内置函数使C语言程序员可以访问目前只能通过行内汇编访问但却十分有用、适用于一系列广泛应用程序的汇编操作符或机器指令。内置函数在 C 源文件中编写，语法类似于函数调用，但它们被编译为直接实现函数的汇编代码，不涉及函数调用或库程序。

有一些原因使得提供内置函数优于要求程序员使用行内汇编。这些原因包括：

1. 提供用于特定目的的内置函数可以简化编码。
2. 使用行内汇编时，某些优化会被禁止。内置函数则不会如此。
3. 对于使用专用寄存器的机器指令，在编写行内汇编代码时，需要特别细心处理，避免寄存器分配错误。内置函数让这个过程变得简单，因为您不需要关心每条机器指令的特殊寄存器要求。

内置函数列表

__builtin_addab
__builtin_add
__builtin_btg
__builtin_clr
__builtin_clr_prefetch
__builtin_write_CRYOTP
__builtin_dataflashoffset
__builtin_disable_interrupts
__builtin_disi
__builtin_divf
__builtin_divmodsd
__builtin_divmodud
__builtin_divsd
__builtin_divud
__builtin_dmapage
__builtin_dmaoffset
__builtin_ed
__builtin_edac
__builtin_edspage
__builtin_edsoffset
__builtin_enable_interrupts
__builtin_fbcl
__builtin_get_isr_state
__builtin_lac
__builtin_mac
__builtin_modsd
__builtin_modud
__builtin_movsac
__builtin_mpy
__builtin_mpyn
__builtin_msc
__builtin_mulss
__builtin_mulsu
__builtin_mulus
__builtin_muluu
__builtin_nop
__builtin_psvpage
__builtin_psvoffset
__builtin_readsfr
__builtin_return_address
__builtin_sac
__builtin_sacr
__builtin_section_begin
__builtin_section_end
__builtin_section_size
__builtin_set_isr_state
__builtin_sftac
__builtin_software_breakpoint
__builtin_subab
__builtin_tbladdress
__builtin_tblpage
__builtin_tbloffset
__builtin_tblrdh
__builtin_tblrdl
__builtin_tblwth
__builtin_tblwtl
__builtin_write_NVM
__builtin_write_NVM_secure
__builtin_write_PWMSFR
__builtin_write_RTCWEN
__builtin_write_OSCCONL
__builtin_write_OSCCONH

© 2015 Microchip Technology Inc.

G.2 内置函数说明

本节介绍编译器内置函数的程序员接口。由于这些函数是“内置”的，所以不存在与之关联的头文件。类似地，不存在与内置函数关联的命令行开关——它们总是可用。所选择的内置函数名称使它们属于编译器的命名空间（均具有前缀 __builtin_），所以它们不会与程序员命名空间中的函数或变量名称发生冲突。

__builtin_addab

说明： 将累加器 A 和 B 相加，并将结果回写到指定的累加器。例如：

```
volatile register int result asm("A");
volatile register int B asm("A");

result = __builtin_addab(result,B);
```

将生成：

```
add A
```

原型：

```
int __builtin_addab(int Accum_a, int Accum_b);
```

参数：

Accum_a 要相加的第一个累加器。
Accum_b 要相加的第二个累加器。

返回值： 将加法结果返回到累加器。

汇编操作符 / 机器指令： add

错误消息 如果结果不是累加器寄存器，则将显示一条错误消息。

__builtin_add

说明： 将 value 与 result 指定的累加器相加，并按立即数指定的移位量进行移位。例如：

```
volatile register int result asm("A");
int value;
result = __builtin_add(result,value,0);
```

如果 value 存放在 w0 中，会生成以下代码：

```
add w0, #0, A
```

原型：

```
int __builtin_add(int Accum,int value,
  const int shift);
```

参数：

Accum 要相加的累加器。
value 要与累加器值相加的整型数。
shift 所产生的累加器值的移位量。

返回值： 将移位后的加法结果返回到累加器。

汇编操作符 / 机器指令： add

错误消息 如果出现以下情况，将显示一条错误消息：

- 结果不是累加器寄存器
- 参数 0 不是累加器
- 移位值不是处于范围内的立即数

__builtin_btg

说明： 该函数将生成一条 btg 机器指令。
一些示例包括：

```
int i;   /* near by default */
int l __attribute__((far));

struct foo {
  int bit1:1;
} barbits;

int bar;

void some_bittoggles() {
  register int j asm("w9");
  int k;

  k = i;

  __builtin_btg(&i,1);
  __builtin_btg(&j,3);
  __builtin_btg(&k,4);
  __builtin_btg(&l,11);

  return j+k;
}
```

请注意，获取处于寄存器中的变量的地址时，编译器会产生警告，并导致寄存器被保存到堆栈中（从而可以获取它的地址）；建议不要使用这种形式。该警告仅适用于由编程人员显式放置在寄存器中的变量。

原型： `void __builtin_btg(unsigned int *, unsigned int 0x`*n*`);`

参数：
`*` 指向某个位要进行翻转的数据项的指针。
0x*n* 介于 0 到 15 范围内的立即数。

返回值： 返回一条 btg 机器指令。

汇编操作符 / 机器指令： `btg`

错误消息 如果参数值不处于范围内，将显示一条错误消息。

__builtin_clr

说明： 清零指定的累加器。例如：

```
volatile register int result asm("A");
result = __builtin_clr();
```

将生成：

```
clr A
```

原型： `int __builtin_clr(void);`

参数： 无

返回值： 将清零值结果返回到累加器。

汇编操作符 / 机器指令： `clr`

错误消息 如果结果不是累加器寄存器，则将显示一条错误消息。

© 2015 Microchip Technology Inc.

__builtin_clr_prefetch

说明：

清零累加器并为之后的 MAC 操作预取数据。
xptr 可以为空，表示不需要执行 X 预取，这种情况下 *xincr* 和 *xval* 的值会被忽略，但它们是必需的。
yptr 可以为空，表示不需要执行 Y 预取，这种情况下 *yincr* 和 *yval* 的值会被忽略，但它们是必需的。
xval 和 *yval* 指定 C 变量的地址，预取值将存储在上述变量中。
xincr 和 *yincr* 可以为立即数：-6、 -4、 -2、 0、 2、 4、 6 或整型值。
如果 *AWB* 非空，则另一个累加器会被回写到所引用的变量中。
例如：

```
  volatile register int result asm("A");
  volatile register int B asm("B");
  int x_memory_buffer[256]
  __attribute__((space(xmemory)));
  int y_memory_buffer[256]
  __attribute__((space(ymemory)));
  int *xmemory;
  int *ymemory;
  int awb;
  int xVal, yVal;

  xmemory = x_memory_buffer;
  ymemory = y_memory_buffer;
  result = __builtin_clr(&xmemory, &xVal, 2,
              &ymemory, &yVal, 2, &awb, B);
```

可能生成：

```
  clr A, [w8]+=2, w4, [w10]+=2, w5, w13
```

编译器可能需要溢出 （spill） w13，以确保可以对它进行回写。建议用户将该寄存器声明为用于该目的。
在该指令之后：

- 结果将被清零
- xVal 将包含 x_memory_buffer[0]
- yVal 将包含 y_memory_buffer[0]
- xmemory 和 ymemory 将递增 2，准备好进行下一个 mac 操作

原型：

```
int __builtin_clr_prefetch(
  int **xptr, int *xval, int xincr,
  int **yptr, int *yval, int yincr, int *AWB,
  int AWB_accum);
```

参数：

xptr	指向 x 预取的整型指针。
xval	x 预取的整型值。
xincr	x 预取的整型递增值。
yptr	指向 y 预取的整型指针。
yval	y 预取的整型值。
yincr	y 预取的整型递增值。
AWB	累加器回写目标。
AWB_accum	要被回写的累加器。

返回值：

将清零值结果返回到累加器。

汇编操作符 / 机器指令：

```
clr
```

错误消息

如果出现以下情况，将显示一条错误消息：

- 结果不是累加器寄存器
- *xval* 是一个空值，但 *xptr* 不为空
- *yval* 是一个空值，但 *yptr* 不为空
- *AWB_accum* 不是累加器，并且 *AWB* 不为空

__builtin_write_CRYOTP

说明：	通过发出正确的解锁序列并将 CRYWR 位置 1 来启动对加密 OTP 的写操作。 为了正确工作，可能需要禁止中断。 该内置函数可以用作器件数据手册或系列参考手册中介绍的复杂序列的一部分。 更多信息，请参见上述文档。
原型：	`void __builtin_write_CRYOTP(void);`
参数：	无。
返回值：	无。

汇编操作符 / 机器指令：	`mov #0x55, Wn`
	`mov Wn, _CRYKEY`
	`mov #0xAA, Wn`
	`mov Wn, _CRYKEY`
	`bset _CRYCON, #0`
	`nop`
	`nop`

错误消息	无。

__builtin_dataflashoffset

说明：	禁止指定的中断。
原型：	`int __builtin_dataflashoffset(unsigned int &var);`
参数：	*&var* = 指向 dataflash 的指针的地址。
返回值：	以整型表示的偏移值。
汇编操作符 / 机器指令：	
错误消息	无。

__builtin_disable_interrupts

说明：	禁止指定的中断。
原型：	`void __builtin_disable_interrupts(void);`
参数：	无。
返回值：	无。
汇编操作符 / 机器指令：	
错误消息	无。

© 2015 Microchip Technology Inc.

__builtin_disi

说明： 将所有中断禁止指定的指令周期数。请参见**第 14.7 节 “允许 / 禁止中断”**。
会在它出现在源程序中的位置点发出指定 DISI 指令。`disi #<disi_count>`

原型：
```
void __builtin_disi(int disi_count);
```

参数： *disi_count* 指令周期数。必须为介于 0 和 16383 之间的整型立即数。

返回值： N/A

汇编操作符 / 机器指令： `disi.f`

__builtin_divf

说明： 计算商 *num*/*den*。如果 *den* 为零，会发生数学错误异常。函数参数是无符号的，函数的结果也是。

原型：
```
unsigned int __builtin_divf(unsigned int num,
 unsigned int den);
```

参数： *num* 分子
den 分母

返回值： 返回商 *num*/*den* 的无符号整型值。

汇编操作符 / 机器指令： `div.f`

__builtin_divmodsd

说明： 发出 16 位架构的本机有符号除法支持，其限制与 《16 位 MCU 和 DSC 程序员参考手册》（DS70157E_CN）中介绍的相同。 值得注意的是，如果商无法装入 16 位结果中，结果（包括余数）将是非预期的。这种形式的内置函数会同时捕捉商和余数。

原型：
```
signed int __builtin_divmodsd(
 signed long dividend, signed int divisor,
 signed int *remainder);
```

参数： *dividend* 被除数
divisor 除数
remainder 指向余数的指针

返回值： 商和余数。

汇编操作符 / 机器指令： `divmodsd`

错误消息 无。

__builtin_divmodud

说明： 发出 16 位架构的本机无符号除法支持，其限制与 《16 位 MCU 和 DSC 程序员参考手册》（DS70157E_CN）中介绍的相同。 值得注意的是，如果商无法装入 16 位结果中，结果（包括余数）将是非预期的。这种形式的内置函数会同时捕捉商和余数。

原型：
```
unsigned int __builtin_divmodud(
 unsigned long dividend, unsigned int divisor,
 unsigned int *remainder);
```

参数：

dividend	被除数
divisor	除数
remainder	指向余数的指针

返回值： 商和余数。

汇编操作符 / 机器指令： divmodud

错误消息 无。

__builtin_divsd

说明： 计算商 *num*/*den*。如果 *den* 为零，会发生数学错误异常。函数参数是有符号的，函数的结果也是。可以使用命令行选项 -Wconversions 来检测意外的符号转换。

原型：
```
int __builtin_divsd(const long num, const int den);
```

参数：

num	分子
den	分母

返回值： 返回商 *num*/*den* 的有符号整型值。

汇编操作符 / 机器指令： div.sd

__builtin_divud

说明： 计算商 *num*/*den*。如果 *den* 为零，会发生数学错误异常。函数参数是无符号的，函数的结果也是。可以使用命令行选项 -Wconversions 来检测意外的符号转换。

原型：
```
unsigned int __builtin_divud(const unsigned
  long num, const unsigned int den);
```

参数：

num	分子
den	分母

返回值： 返回商 *num*/*den* 的无符号整型值。

汇编操作符 / 机器指令 ： div.ud

© 2015 Microchip Technology Inc.

__builtin_dmapage

说明： 获取 DMA 存储器内的某个符号的页号。
例如：

```
unsigned int result;
char buffer[256] __attribute__((space(dma)));

result = __builtin_dmapage(&buffer);
```

可能生成：

```
mov #dmapage(buffer), w0
```

原型： `unsigned int __builtin_dmapage(const void *p);`

参数： *p* 指向 DMA 地址值的指针

返回值： 返回位于 DMA 存储器中的变量的页号。

汇编操作符 / 机器指令： dmapage

错误消息 如果参数不是全局符号的地址，将显示一条错误消息。

__builtin_dmaoffset

说明： 获取 DMA 存储器内的某个符号的偏移量。
例如：

```
unsigned int result;
char buffer[256] __attribute__((space(dma)));

result = __builtin_dmaoffset(&buffer);
```

可能生成：

```
mov #dmaoffset(buffer), w0
```

原型： `unsigned int __builtin_dmaoffset(const void *p);`

参数： *p* 指向 DMA 地址值的指针

返回值： 返回位于 DMA 存储器中的变量的偏移量。

汇编操作符 / 机器指令： dmaoffset

错误消息 如果参数不是全局符号的地址，将显示一条错误消息。

__builtin_ed

说明： 计算 sqr 的平方值，并将它作为结果返回。此外，预取用于之后的平方运算的数据，方法是计算 **xptr - **yptr，并将结果存储在 *distance 中。
xincr 和 yincr 可以为立即数：-6、-4、-2、0、2、4、6 或整型值。
例如：

```
volatile register int result asm("A");
int *xmemory, *ymemory;
int distance;

result = __builtin_ed(distance,
                      &xmemory, 2,
                      &ymemory, 2,
                      &distance);
```

可能生成：

```
        ed w4*w4, A, [w8]+=2, [W10]+=2, w4
```

原型：

```
int __builtin_ed(int sqr, int **xptr, int xincr,
  int **yptr, int yincr, int *distance);
```

__builtin_ed

参数：

sqr	整型平方值。
xptr	指向 x 预取的整型指针。
xincr	x 预取的整型递增值。
yptr	指向 y 预取的整型指针。
yincr	y 预取的整型递增值。
distance	指向 distance 的整型指针。

返回值： 将平方结果返回到累加器。

汇编操作符 / 机器指令： ed

错误消息 如果出现以下情况，将显示一条错误消息：

- 结果不是累加器寄存器
- *xptr* 为空
- *yptr* 为空
- *distance* 为空

__builtin_edac

说明： 计算 sqr 的平方值，并与指定的累加器寄存器相加，将它作为结果返回。此外，预取用于之后的平方运算的数据，方法是计算 **xptr - **yptr，并将结果存储在 *distance 中。xincr 和 yincr 可以为立即数：-6、 -4、 -2、 0、 2、 4、 6 或整型值。

例如：

```
volatile register int result asm("A");
int *xmemory, *ymemory;
int distance;

result = __builtin_ed(result, distance,
                      &xmemory, 2,
                      &ymemory, 2,
                      &distance);
```

可能生成：

```
edac w4*w4, A, [w8]+=2, [W10]+=2, w4
```

原型：

```
int __builtin_edac(int Accum, int sqr,
  int **xptr, int xincr, int **yptr, int yincr,
  int *distance);
```

参数：

Accum	要相加的累加器。
sqr	整型平方值。
xptr	指向 x 预取的整型指针。
xincr	x 预取的整型递增值。
yptr	指向 y 预取的整型指针。
yincr	y 预取的整型递增值。
distance	指向 distance 的整型指针。

返回值： 将平方结果返回到指定累加器。

汇编操作符 / 机器指令： edac

错误消息 如果出现以下情况，将显示一条错误消息：

- 结果不是累加器寄存器
- *Accum* 不是累加器寄存器
- *xptr* 为空
- *yptr* 为空
- *distance* 为空

© 2015 Microchip Technology Inc.

__builtin_edspage

说明： 返回地址作为参数给出的对象的 eds 页号。参数 *p* 必须是处于扩展数据空间 （EDS）中的对象的地址；否则会产生错误消息，并且编译会失败。请参见**第 2.3.1 节 “指定变量的属性”**中的 space 属性。

原型：
```
unsigned int __builtin_edspage(const void *p);
```

参数： *p* 对象地址

返回值： 返回地址作为参数给出的对象的 eds 页号。

汇编操作符 / 机器指令：
```
edspage
```

错误消息 如果该函数的使用不正确，则会产生以下错误消息：
“Argument to __builtin_edspage() is not the address of an object in extended data space”（__builtin_edspage() 的参数不是处于扩展数据空间中的对象的地址）。
参数必须为显式对象地址。
例如，如果 *obj* 是一个处于可执行或只读段中的对象，则以下语法是有效的：
```
unsigned page = __builtin_edspage(&obj);
```

__builtin_edsoffset

说明： 返回地址作为参数给出的对象的 eds 页偏移量。参数 *p* 必须是处于扩展数据空间 （EDS）中的对象的地址；否则会产生错误消息，并且编译会失败。请参见**第 2.3.1 节 “指定变量的属性”**中的 space 属性。

原型：
```
unsigned int __builtin_edsoffset(const void *p);
```

参数： *p* 对象地址

返回值： 返回地址作为参数给出的对象的 eds 页偏移量。

汇编操作符 / 机器指令：
```
edsoffset
```

错误消息 如果该函数的使用不正确，则会产生以下错误消息：
“Argument to __builtin_edsoffset() is not the address of an object in extended data space”（__builtin_edsoffset() 的参数不是处于扩展数据空间中的对象的地址）。
参数必须为显式对象地址。
例如，如果 *obj* 是一个处于可执行或只读段中的对象，则以下语法是有效的：
```
unsigned page = __builtin_edsoffset(&obj);
```

__builtin_enable_interrupts

说明： 允许指定的中断。

原型：
```
void __builtin_enable_interrupts(void);
```

参数： 无。

返回值： 无。

汇编操作符 / 机器指令：

错误消息 无。

__builtin_fbcl

说明： 在 *value* 中从左边开始查找第一个位变化。它对于定点数据的动态比例换算很有用。例如：

```
int result, value;
result = __builtin_fbcl(value);
```

可能生成：

```
fbcl w4, w5
```

原型： `int __builtin_fbcl(int value);`

参数： *value* 要检查变化的整型数。

返回值： 返回一个进行符号扩展的立即数，表示要左移的位数。

汇编操作符 / 机器指令： `fbcl`

错误消息 无。

__builtin_get_isr_state

说明： 确定当前的 CPU 中断状态。

原型： `unsigned int __builtin_get_isr_state(void);`

参数： 无。

返回值： 返回一个指定当前 CPU 中断状态的整型值。

汇编操作符 / 机器指令： `get_isr_state`

错误消息 无。

__builtin_lac

说明： 将值移位 *shift*（介于 -8 和 7 之间的立即数）位，并返回要存储到累加器寄存器中的值。例如：

```
volatile register int result asm("A");
int value;
result = __builtin_lac(value,3);
```

可能生成：

```
lac w4, #3, A
```

原型： `int __builtin_lac(int value, int shift);`

参数： *value* 要进行移位的整型数。
shift 表示移位量的立即数。

返回值： 将移位后的结果返回到累加器。

汇编操作符 / 机器指令： `lac`

错误消息 如果出现以下情况，将显示一条错误消息：

- 结果不是累加器寄存器
- 移位值不是处于范围内的立即数

© 2015 Microchip Technology Inc.

__builtin_mac

说明：

计算 *a* x *b*，并与累加器相加；此外，为之后的 MAC 操作预取数据。
xptr 可以为空，表示不需要执行 X 预取，这种情况下 *xincr* 和 *xval* 的值会被忽略，但它们是必需的。
yptr 可以为空，表示不需要执行 Y 预取，这种情况下 *yincr* 和 *yval* 的值会被忽略，但它们是必需的。
xval 和 *yval* 指定 C 变量的地址，预取值将存储在上述变量中。
xincr 和 *yincr* 可以为立即数：-6、-4、-2、0、2、4、6 或整型值。
如果 *AWB* 非空，则另一个累加器会被回写到所引用的变量中。
例如：

```
volatile register int result asm("A");
volatile register int B asm("B");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_mac(result, xVal, yVal,
                       &xmemory, &xVal, 2,
                       &ymemory, &yVal, 2, 0, B);
```

可能生成：

```
mac w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

原型：

```
int __builtin_mac(int Accum, int a, int b,
  int **xptr, int *xval, int xincr,
  int **yptr, int *yval, int yincr, int *AWB,
  int AWB_accum);
```

参数：

Accum	要相加的累加器。
a	整型被乘数。
b	整型乘数。
xptr	指向 x 预取的整型指针。
xval	指向 x 预取的值的整型指针。
xincr	x 预取的整型递增值。
yptr	指向 y 预取的整型指针。
yval	指向 y 预取的值的整型指针。
yincr	y 预取的整型递增值。
AWB	累加器回写目标。
AWB_accum	要被回写的累加器。

返回值：

返回累加器加上 a x b 的结果得到的值

汇编操作符 / 机器指令：

mac

错误消息

如果出现以下情况，将显示一条错误消息：

- 结果不是累加器寄存器
- *Accum* 不是累加器寄存器
- *xval* 是一个空值，但 *xptr* 不为空
- *yval* 是一个空值，但 *yptr* 不为空
- *AWB_accum* 不是累加器寄存器，并且 *AWB* 不为空

__builtin_modsd

说明：	发出 16 位架构的本机有符号除法支持，其限制与 《16 位 MCU 和 DSC 程序员参考手册》（DS70157E_CN）中介绍的相同。 值得注意的是，如果商无法装入 16 位结果中，结果（包括余数）将是非预期的。这种形式的内置函数只会捕捉余数。
原型：	`signed int __builtin_modsd(signed long` *`dividend`*`,` ` signed int` *`divisor`*`);`
参数：	*`dividend`* 被除数 *`divisor`* 除数
返回值：	余数。
汇编操作符 / 机器指令：	`modsd`
错误消息	无。

__builtin_modud

说明：	发出 16 位架构的本机无符号除法支持，其限制与 《16 位 MCU 和 DSC 程序员参考手册》（DS70157E_CN）中介绍的相同。 值得注意的是，如果商无法装入 16 位结果中，结果（包括余数）将是非预期的。这种形式的内置函数只会捕捉余数。
原型：	`unsigned int __builtin_modud(unsigned long` *`dividend`*`,` ` unsigned int` *`divisor`*`);`
参数：	*`dividend`* 被除数 *`divisor`* 除数
返回值：	余数。
汇编操作符 / 机器指令：	`modud`
错误消息	无。

© 2015 Microchip Technology Inc.

__builtin_movsac

说明：

不进行任何计算，但为之后的 MAC 操作预取数据。
xptr 可以为空，表示不需要执行 X 预取，这种情况下 *xincr* 和 *xval* 的值会被忽略，但它们是必需的。
yptr 可以为空，表示不需要执行 Y 预取，这种情况下 *yincr* 和 *yval* 的值会被忽略，但它们是必需的。
xval 和 *yval* 指定 C 变量的地址，预取值将存储在上述变量中。
xincr 和 *yincr* 可以为立即数：-6、-4、-2、0、2、4、6 或整型值。
如果 *AWB* 非空，则另一个累加器会被回写到所引用的变量中。
例如：

```
volatile register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_movsac(&xmemory, &xVal, 2,
                          &ymemory, &yVal, 2, 0, 0);
```

可能生成：

```
        movsac A, [w8]+=2, w4, [w10]+=2, w5
```

原型：

```
int __builtin_movsac(
  int **xptr, int *xval, int xincr,
  int **yptr, int *yval, int yincr, int *AWB
  int AWB_accum);
```

参数：

xptr	指向 x 预取的整型指针。
xval	指向 x 预取的值的整型指针。
xincr	x 预取的整型递增值。
yptr	指向 y 预取的整型指针。
yval	指向 y 预取的值的整型指针。
yincr	y 预取的整型递增值。
AWB	累加器回写目标。
AWB_accum	要被回写的累加器。

返回值：

返回预取数据。

汇编操作符 / 机器指令：

movsac

错误消息

如果出现以下情况，将显示一条错误消息：

- 结果不是累加器寄存器
- *xval* 是一个空值，但 *xptr* 不为空
- *yval* 是一个空值，但 *yptr* 不为空
- *AWB_accum* 不是累加器寄存器，并且 *AWB* 不为空

__builtin_mpy

说明：

计算 *a* x *b* ；此外，为之后的 MAC 操作预取数据。
xptr 可以为空，表示不需要执行 X 预取，这种情况下 *xincr* 和 *xval* 的值会被忽略，但它们是必需的。
yptr 可以为空，表示不需要执行 Y 预取，这种情况下 *yincr* 和 *yval* 的值会被忽略，但它们是必需的。
xval 和 *yval* 指定 C 变量的地址，预取值将存储在上述变量中。
xincr 和 *yincr* 可以为立即数：-6、 -4、 -2、 0、 2、 4、 6 或整型值。
例如：

```
volatile register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_mpy(xVal, yVal,
                       &xmemory, &xVal, 2,
                       &ymemory, &yVal, 2);
```

可能生成：

```
mpy w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

原型：

```
int __builtin_mpy(int a, int b,
  int **xptr, int *xval, int xincr,
  int **yptr, int *yval, int yincr);
```

参数：

a	整型被乘数。
b	整型乘数。
xptr	指向 x 预取的整型指针。`
xval	指向 x 预取的值的整型指针。
xincr	x 预取的整型递增值。
yptr	指向 y 预取的整型指针。
yval	指向 y 预取的值的整型指针。
yincr	y 预取的整型递增值。
AWB	指向所选累加器的整型指针。

返回值：

返回 *a* x *b* 的值。

汇编操作符 / 机器指令：

mpy

错误消息

如果出现以下情况，将显示一条错误消息：

- 结果不是累加器寄存器
- *xval* 是一个空值，但 *xptr* 不为空
- *yval* 是一个空值，但 *yptr* 不为空

© 2015 Microchip Technology Inc.

__builtin_mpyn

说明：

计算 *-a* x *b* ；此外，为之后的 MAC 操作预取数据。
xptr 可以为空，表示不需要执行 X 预取，这种情况下 *xincr* 和 *xval* 的值会被忽略，但它们是必需的。
yptr 可以为空，表示不需要执行 Y 预取，这种情况下 *yincr* 和 *yval* 的值会被忽略，但它们是必需的。
xval 和 *yval* 指定 C 变量的地址，预取值将存储在上述变量中。
xincr 和 *yincr* 可以为立即数：-6、 -4、 -2、 0、 2、 4、 6 或整型值。
例如：

```
volatile register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_mpy(xVal, yVal,
                       &xmemory, &xVal, 2,
                       &ymemory, &yVal, 2);
```

可能生成：

```
mpy.n w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

原型：

```
int __builtin_mpyn(int a, int b,
  int **xptr, int *xval, int xincr,
  int **yptr, int *yval, int yincr);
```

参数：

a	整型被乘数。
b	整型乘数。
xptr	指向 x 预取的整型指针。
xval	指向 x 预取的值的整型指针。
xincr	x 预取的整型递增值。
yptr	指向 y 预取的整型指针。
yval	指向 y 预取的值的整型指针。
yincr	y 预取的整型递增值。
AWB	指向所选累加器的整型指针。

返回值：

返回 *-a* x *b* 的值。

汇编操作符 / 机器指令：

mpyn

错误消息

如果出现以下情况，将显示一条错误消息：

- 结果不是累加器寄存器
- *xval* 是一个空值，但 *xptr* 不为空
- *yval* 是一个空值，但 *yptr* 不为空

__builtin_msc

说明：

计算 *a* x *b*，并从累加器中减去；此外，为之后的 MAC 操作预取数据。
xptr 可以为空，表示不需要执行 X 预取，这种情况下 *xincr* 和 *xval* 的值会被忽略，但它们是必需的。
yptr 可以为空，表示不需要执行 Y 预取，这种情况下 *yincr* 和 *yval* 的值会被忽略，但它们是必需的。
xval 和 *yval* 指定 C 变量的地址，预取值将存储在上述变量中。
xincr 和 *yincr* 可以为立即数：-6、-4、-2、0、2、4、6 或整型值。
如果 *AWB* 非空，则另一个累加器会被回写到所引用的变量中。
例如：

```
volatile register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_msc(result, xVal, yVal,
                       &xmemory, &xVal, 2,
                       &ymemory, &yVal, 2, 0, 0);
```

可能生成：

```
msc w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

原型：

```
int __builtin_msc(int Accum, int a, int b,
  int **xptr, int *xval, int xincr,
  int **yptr, int *yval, int yincr, int *AWB,
  int AWB_accum);
```

参数：

Accum	要相减的累加器。
a	整型被乘数。
b	整型乘数。
xptr	指向 x 预取的整型指针。
xval	指向 x 预取的值的整型指针。
xincr	x 预取的整型递增值。
yptr	指向 y 预取的整型指针。
yval	指向 y 预取的值的整型指针。
yincr	y 预取的整型递增值。
AWB	累加器回写目标。
AWB_accum	要被回写的累加器。

返回值：

返回累加器减去 *a* x *b* 的结果得到的值。

汇编操作符 / 机器指令：

```
msc
```

错误消息

如果出现以下情况，将显示一条错误消息：

- 结果不是累加器寄存器
- *Accum* 不是累加器寄存器
- *xval* 是一个空值，但 *xptr* 不为空
- *yval* 是一个空值，但 *yptr* 不为空
- *AWB_accum* 不是累加器寄存器，并且 *AWB* 不为空

© 2015 Microchip Technology Inc.

__builtin_mulss

说明： 计算 *p0* x *p1* 的积。函数参数为有符号整型，函数结果为有符号长整型。可以使用命令行选项 -Wconversions 来检测意外的符号转换。

例如：

```
volatile register int a asm("A");
signed long result;
const signed int p0, p1;
const unsigned int p2, p3;
result = __builtin_mulss(p0,p1);
a = __builtin_mulss(p0,p1);
```

原型： `signed long __builtin_mulss(const signed int p0, const signed int p1);`

参数：
p0 被乘数
p1 乘数

返回值： 返回 *p0* x *p1* 的积的有符号长整型值。值可以返回到有符号长整型的变量中，也可以直接返回到累加器寄存器中。

汇编操作符 / 机器指令： `mul.ss`

__builtin_mulsu

说明： 计算 *p0* x *p1* 的积。函数参数为具有混合符号的整型，函数结果为有符号长整型。可以使用命令行选项 -Wconversions 来检测意外的符号转换。该函数支持指令的所有寻址模式，包括操作数 *p1* 的立即数寻址模式。

例如：

```
volatile register int a asm("A");
signed long result;
const signed int p0, p1;
const unsigned int p2, p3;
result = __builtin_mulsu(p0,p2);
a = __builtin_mulsu(p0,p2);
```

原型： `signed long __builtin_mulsu(const signed int p0, const unsigned int p1);`

参数：
p0 被乘数
p1 乘数

返回值： 返回 *p0* x *p1* 的积的有符号长整型值。值可以返回到有符号长整型的变量中，也可以直接返回到累加器寄存器中。

汇编操作符 / 机器指令： `mul.su`

__builtin_mulus

说明： 计算 *p0* x *p1* 的积。函数参数为具有混合符号的整型，函数结果为有符号长整型。可以使用命令行选项 -Wconversions 来检测意外的符号转换。该函数支持指令的所有寻址模式。
例如：

```
volatile register int a asm("A");
signed long result;
const signed int p0, p1;
const unsigned int p2, p3;
result = __builtin_mulus(p2,p0);
a = __builtin_mulus(p2,p0);
```

原型： `signed long __builtin_mulus(const unsigned int p0, const signed int p1);`

参数：
p0 被乘数
p1 乘数

返回值： 返回 *p0* x *p1* 的积的有符号长整型值。值可以返回到有符号长整型的变量中，也可以直接返回到累加器寄存器中。

汇编操作符 / 机器指令： mul.us

__builtin_muluu

说明： 计算 *p0* x *p1* 的积。函数参数为无符号整型，函数结果为无符号长整型。可以使用命令行选项 -Wconversions 来检测意外的符号转换。该函数支持指令的所有寻址模式，包括操作数 *p1* 的立即数寻址模式。
例如：

```
volatile register int a asm("A");
unsigned long result;
const signed int p0, p1;
const unsigned int p2, p3;
result = __builtin_muluu(p2,p3);
a = __builtin_muluu(p2,p3);
```

原型： `unsigned long __builtin_muluu(const unsigned int p0, const unsigned int p1);`

参数：
p0 被乘数
p1 乘数

返回值： 返回 *p0* x *p1* 的积的有符号长整型值。值可以返回到无符号长整型的变量中，也可以直接返回到累加器寄存器中。

汇编操作符 / 机器指令： mul.uu

__builtin_nop

说明： 生成一条 nop 指令。

原型： `void __builtin_nop(void);`

参数： 无。

返回值： 返回无操作 （nop）。

汇编操作符 / 机器指令： nop

© 2015 Microchip Technology Inc.

__builtin_psvpage

说明：	返回地址作为参数给出的对象的 psv 页号。参数 *p* 必须是处于 EEdata、 PSV 或可执行存储空间中的对象的地址；否则会产生错误消息，并且编译会失败。请参见**第 2.3.1 节 “指定变量的属性”**中的 space 属性。
原型：	`unsigned int __builtin_psvpage(const void *p);`
参数：	*p* 对象地址
返回值：	返回地址作为参数给出的对象的 psv 页号。
汇编操作符 / 机器指令：	`psvpage`
错误消息	如果该函数的使用不正确，则会产生以下错误消息： “Argument to `__builtin_psvpage()` is not the address of an object in code, psv, or eedata section”（`__builtin_psvpage()` 的参数不是处于代码、psv 或 eedata 段中的对象的地址）。 参数必须为显式对象地址。 例如，如果 *obj* 是一个处于可执行或只读段中的对象，则以下语法是有效的： `unsigned page = __builtin_psvpage(&obj);`

__builtin_psvoffset

说明：	返回地址作为参数给出的对象的 psv 页偏移量。参数 *p* 必须是处于 EEdata、 PSV 或可执行存储空间中的对象的地址；否则会产生错误消息，并且编译会失败。请参见**第 2.3.1 节 “指定变量的属性”**中的 space 属性。
原型：	`unsigned int __builtin_psvoffset(const void *p);`
参数：	*p* 对象地址
返回值：	返回地址作为参数给出的对象的 psv 页偏移量。
汇编操作符 / 机器指令：	`psvoffset`
错误消息	如果该函数的使用不正确，则会产生以下错误消息： “Argument to `__builtin_psvoffset()` is not the address of an object in code, psv, or eedata section”（`__builtin_psvoffset()` 的参数不是处于代码、psv 或 eedata 段中的对象的地址）。 参数必须为显式对象地址。 例如，如果 *obj* 是一个处于可执行或只读段中的对象，则以下语法是有效的： `unsigned page = __builtin_psvoffset(&obj);`

__builtin_readsfr

说明：	读取 SFR。
原型：	`unsigned int __builtin_readsfr(const void *p);`
参数：	*p* 对象地址
返回值：	返回 SFR。
汇编操作符 / 机器指令：	`readsfr`
错误消息	如果对象地址不在 SFR 存储空间范围内，将产生错误消息。 SFR 存储空间范围请查阅您所用器件的数据手册。

__builtin_return_address

说明： 返回当前函数或其调用方之一的返回地址。对于 *level* 参数，值为 0 将产生当前函数的返回地址，值为 1 将产生当前函数的调用方的返回地址，如此类推。当 level 超出当前堆栈深度时，将返回 0。该函数应仅使用非零参数，用于调试目的。

原型： `int __builtin_return_address (const int level);`

参数： *level* 要在调用堆栈中扫描的帧数。

返回值： 返回当前函数或其调用方之一的返回地址。

汇编操作符 / 机器指令： `return_address`

__builtin_sac

说明： 将值移位 *shift*（介于 -8 和 7 之间的立即数）位，并返回值。

例如：

```
volatile register int value asm("A");
int result;

result = __builtin_sac(value,3);
```

可能生成：

```
sac A, #3, w0
```

原型： `int __builtin_sac(int value, int shift);`

参数： *value* 要进行移位的整型数。
shift 表示移位量的立即数。

返回值： 将移位后的结果返回到累加器。

汇编操作符 / 机器指令： `sac`

错误消息 如果出现以下情况，将显示一条错误消息：

- 结果不是累加器寄存器
- 移位值不是处于范围内的立即数

__builtin_sacr

说明： 将值移位 *shift*（介于 -8 和 7 之间的立即数）位，并返回使用由 CORCONbits.RND 控制位决定的舍入模式进行舍入的值。

例如：

```
volatile register int value asm("A");
int result;

result = __builtin_sac(value,3);
```

可能生成：

```
sac.r A, #3, w0
```

原型： `int __builtin_sacr(int value, int shift);`

参数： *value* 要进行移位的整型数。
shift 表示移位量的立即数。

返回值： 将移位后的结果返回到 CORCON 寄存器。

汇编操作符 / 机器指令： `sacr`

错误消息 如果出现以下情况，将显示一条错误消息：

- 结果不是累加器寄存器
- 移位值不是处于范围内的立即数

© 2015 Microchip Technology Inc.

__builtin_section_begin

说明： 获取关于段起始地址的运行时信息。

原型： `unsigned long __builtin_section_begin("section_name");`

参数： *section_name* 段的名称

返回值： 返回指定段的起始地址。

汇编操作符 / 机器指令： section_begin

错误消息 如果无法找到指定的段，将显示一条错误消息。

__builtin_section_end

说明： 获取关于段结束地址的运行时信息。

原型： `unsigned long __builtin_section_end("section_name");`

参数： *section_name* 段的名称

返回值： 返回指定段的结束地址。

汇编操作符 / 机器指令： section_end

错误消息 如果无法找到指定的段，将显示一条错误消息。

__builtin_section_size

说明： 获取关于段大小的运行时信息。

原型： `unsigned long __builtin_section_size("section_name");`

参数： *section_name* 段的名称

返回值： 返回指定段的大小。

汇编操作符 / 机器指令： section_size

错误消息 如果无法找到指定的段，将显示一条错误消息。

__builtin_set_isr_state

说明： 设置当前 CPU 中断状态。

原型： `void __builtin_get_isr_state(unsigned int state);`

参数： *state* 指定当前 CPU 中断状态的整型值。

返回值： 无。

汇编操作符 / 机器指令： set_isr_state

错误消息 无。

__builtin_sftac

说明： 将累加器移位 *shift* 位。有效移位范围为 -16 至 16。
例如：
```
volatile register int result asm("A");
int i;

result = __builtin_sftac(result,i);
```
可能生成：
```
sftac A, w0
```

原型： `int __builtin_sftac(int Accum, int shift);`

参数： *Accum* 要移位的累加器。
shift 要移位的位数。

返回值： 将移位后的结果返回到累加器。

汇编操作符 / 机器指令： `sftac`

错误消息 如果出现以下情况，将显示一条错误消息：
- 结果不是累加器寄存器
- *Accum* 不是累加器寄存器
- 移位值不是处于范围内的立即数

__builtin_software_breakpoint

说明： 提供软件断点。如果连接了调试器， IDE 会暂停。如果未连接调试器，则器件会复位。

原型： `void __builtin_software_breakpoint(void);`

参数： 无。

返回值： 无。

汇编操作符 / 机器指令： `software_breakpoint`

错误消息 无。

__builtin_subab

说明： 将累加器 A 和 B 相减，并将结果回写到指定的累加器。例如：
```
volatile register int result asm("A");
volatile register int B asm("B");
result = __builtin_subab(result,B);
```
将生成：
```
sub A
```

原型： `int __builtin_subab(int Accum_a, int Accum_b);`

参数： *Accum*_a 要从中减去值的累加器。
Accum_b 要减去的累加器。

返回值： 将减法结果返回到累加器。

汇编操作符 / 机器指令： `sub`

错误消息 如果结果不是累加器寄存器，则将显示一条错误消息。

© 2015 Microchip Technology Inc.

__builtin_tbladdress

说明： 返回一个表示对象在程序存储器中的地址的值。参数 *p* 必须是处于 EEdata、 PSV 或可执行存储空间中的对象的地址；否则会产生错误消息，并且编译会失败。请参见**第 2.3.1 节 “指定变量的属性”**中的 space 属性。

原型：
```
unsigned long __builtin_tbladdress(const void *p);
```

参数： *p* 对象地址

返回值： 返回一个表示对象在程序存储器中的地址的 unsigned long 值。

汇编操作符 / 机器指令：
```
tbladdress
```

错误消息 如果该函数的使用不正确，则会产生以下错误消息：
“Argument to __builtin_tbladdress() is not the address of an object in code, psv, or eedata section”（__builtin_tbladdress() 的参数不是处于代码、psv 或 eedata 段中的对象的地址）。
参数必须为显式对象地址。
例如，如果 *obj* 是一个处于可执行或只读段中的对象，则以下语法是有效的：
```
unsigned long page = __builtin_tbladdress(&obj);
```

__builtin_tblpage

说明： 返回地址作为参数给出的对象的表页号。参数 *p* 必须是处于 EEdata、 PSV 或可执行存储空间中的对象的地址；否则会产生错误消息，并且编译会失败。请参见**第 2.3.1 节 “指定变量的属性”**中的 space 属性。

原型：
```
unsigned int __builtin_tblpage(const void *p);
```

参数： *p* 对象地址

返回值： 返回地址作为参数给出的对象的表页号。

汇编操作符 / 机器指令：
```
tblpage
```

错误消息 如果该函数的使用不正确，则会产生以下错误消息：
“Argument to __builtin_tblpage() is not the address of an object in code, psv, or eedata section”（__builtin_tblpage() 的参数不是处于代码、psv 或 eedata 段中的对象的地址）。
参数必须为显式对象地址。
例如，如果 *obj* 是一个处于可执行或只读段中的对象，则以下语法是有效的：
```
unsigned page = __builtin_tblpage(&obj);
```

__builtin_tbloffset

说明：	返回地址作为参数给出的对象的表页偏移量。参数 *p* 必须是处于 EEdata、 PSV 或可执行存储空间中的对象的地址；否则会产生错误消息，并且编译会失败。请参见**第 2.3.1 节 “指定变量的属性”**中的 space 属性。
原型：	`unsigned int __builtin_tbloffset(const void *p);`
参数：	*p* 对象地址
返回值：	返回地址作为参数给出的对象的表页偏移量。
汇编操作符 / 机器指令：	`tbloffset`
错误消息	如果该函数的使用不正确，则会产生以下错误消息： “Argument to `__builtin_tbloffset()` is not the address of an object in code, psv, or eedata section”（`__builtin_tbloffset()` 的参数不是处于代码、psv 或 eedata 段中的对象的地址）。 参数必须为显式对象地址。 例如，如果 *obj* 是一个处于可执行或只读段中的对象，则以下语法是有效的： `unsigned page = __builtin_tbloffset(&obj);`

__builtin_tblrdh

说明：	发出 `tblrdh.w` 指令来从闪存或 EEDATA 存储器中读取一个字。必须将 TBLPAG 设置为指向相应的页。要实现该目的，可以使用 `__builtin_tbloffset()` 和 `__builtin_tblpage()`。 关于读取和写入程序闪存的完整详细信息，请参见器件数据手册或系列参考手册。
原型：	`unsigned int __builtin_tblrdh(unsigned int offset);`
参数：	*offset* 所需的存储器偏移量
返回值：	闪存或 EEDATA 存储器中的存储器地址的内容。
汇编操作符 / 机器指令：	`tblrdh`
错误消息	无。

__builtin_tblrdl

说明：	发出 `tblrdl.w` 指令来从闪存或 EEDATA 存储器中读取一个字。必须将 TBLPAG 设置为指向相应的页。要实现该目的，可以使用 `__builtin_tbloffset()` 和 `__builtin_tblpage()`。 关于读取和写入程序闪存的完整详细信息，请参见器件数据手册或系列参考手册。
原型：	`unsigned int __builtin_tblrdl(unsigned int offset);`
参数：	*offset* 所需的存储器偏移量
返回值：	闪存或 EEDATA 存储器中的存储器地址的内容。
汇编操作符 / 机器指令：	`tblrdl`
错误消息	无。

© 2015 Microchip Technology Inc.

__builtin_tblwth

说明： 发出 tblwth.w 指令来向闪存或 EEDATA 存储器中写入一个字。必须将 TBLPAG 设置为指向相应的页。要实现该目的，可以使用 __builtin_tbloffset() 和 __builtin_tblpage()。关于读取和写入程序闪存的完整详细信息，请参见器件数据手册或系列参考手册。

原型：
```
void __builtin_tblwth(unsigned int offset
 unsigned int data);
```

参数：
offset 所需的存储器偏移量
data 要写入的数据

返回值： 无。

汇编操作符 / 机器指令：
```
tblwth
```

错误消息 无。

__builtin_tblwtl

说明： 发出 tblrdl.w 指令来向闪存或 EEDATA 存储器中写入一个字。必须将 TBLPAG 设置为指向相应的页。要实现该目的，可以使用 __builtin_tbloffset() 和 __builtin_tblpage()。关于读取和写入程序闪存的完整详细信息，请参见器件数据手册或系列参考手册。

原型：
```
void __builtin_tblwtl(unsigned int offset
 unsigned int data);
```

参数：
offset 所需的存储器偏移量
data 要写入的数据

返回值： 无。

汇编操作符 / 机器指令：
```
tblwtl
```

错误消息 无。

__builtin_write_NVM

说明： 通过发出正确解锁序列并使能 NVMCON 寄存器的写入位来使能闪存的写操作。
为了正确工作，可能需要禁止中断。
该内置函数可以用作器件数据手册或系列参考手册中介绍的复杂序列的一部分。
更多信息，请参见上述文档。

原型：
```
void __builtin_write_NVM(void);
```

参数： 无。

返回值： 无。

汇编操作符 / 机器指令：
```
mov #0x55, Wn
mov Wn, _NVMKEY
mov #0xAA, Wn
mov Wn, _NVMKEY
bset _NVMCON, #15
nop
nop
```

错误消息 无。

__builtin_write_NVM_secure

说明： 通过发出由两个密钥指定的解锁序列并使能 NVMCON 寄存器的写入位来使能闪存的写操作。完成之后，两个密钥会被清零。
为了正确工作，可能需要禁止中断。
该内置函数可以用作器件数据手册或系列参考手册中介绍的复杂序列的一部分。
更多信息，请参见上述文档。

原型：
```
void __builtin_write_NVM_secure(unsigned int key1,
  unsigned int key2);
```

参数：
key1 NVM 解锁序列中的第一个密钥
key2 NVM 解锁序列中的第二个密钥

返回值： 无。

汇编操作符 / 机器指令： 取决于密钥的位置：
```
mov W0, Wn
mov Wn, _NVMKEY
mov W1, Wn
mov Wn, _NVMKEY
bset _NVMCON, #15
nop
nop
```

错误消息 无。

__builtin_write_PWMSFR

说明： 将 PWM 解锁序列写入 *PWM_KEY* 指向的 SFR，然后将 *value* 写入 *PWM_sfr* 指向的 SFR。

原型：
```
void __builtin_write_PWMSFR(volatile unsigned int *PWM_sfr,
unsigned int value, volatile unsigned int *PWM_KEY);
```

参数：
PWM_sfr 要写入的寄存器
value 要写入的值
PWM_KEY 硬件解锁密钥位置

返回值： 无。

汇编操作符 / 机器指令：
```
mov #<it>PWM_KEY</it>, w3
mov #<it>value</it>, w2
mov #0x4321, w1
mov #0xABCD, w0
mov w1,[w3]
mov w0,[w3]
mov w2,[w3]
```

错误消息 无。

示例
例 1：
```
__builtin_write_PWMSFR(&PWM1CON1, 0x123, &PWM1KEY);
```

例 2：
```
__builtin_write_PWMSFR(&P1FLTACON, 0x123, &PWMKEY);
```

PWM_KEY 的选择可能取决于架构。

© 2015 Microchip Technology Inc.

__builtin_write_RTCWEN

说明： 用于写入 RTCC 定时器，先实现解锁序列：向 NVMKEY 写入正确的解锁值，然后将 RCFGCAL SFR 的 RTCWREN 位置 1。
为了正确工作，可能需要禁止中断。
该内置函数可以用作器件数据手册或系列参考手册中介绍的复杂序列的一部分。
更多信息，请参见上述文档。

原型：
```
void __builtin_write_RTCWEN(void);
```

参数： 无。

返回值： 无。

汇编操作符 / 机器指令：
```
mov #0x55, Wn
mov Wn, _NVMKEY
mov #0xAA, Wn
mov Wn, _NVMKEY
bset _RCFGCAL, #13
nop
nop
```

错误消息 无。

__builtin_write_OSCCONL

说明： 解锁并将其参数写入 OSCCONL。
为了正确工作，可能需要禁止中断。
该内置函数可以用作器件数据手册或系列参考手册中介绍的复杂序列的一部分。
更多信息，请参见上述文档。

原型：
```
void __builtin_write_OSCCONL(unsigned char value);
```

参数： *value* 要写入的字符

返回值： 无。

汇编操作符 / 机器指令 *：
```
mov #0x46, w0
mov #0x57, w1
mov #_OSCCON, w2
mov.b w0, [w2]
mov.b w1, [w2]
mov.b value, [w2]
```

错误消息 无。

* 确切的序列可能会有所不同。

__builtin_write_OSCCONH

说明：	解锁并将其参数写入 OSCCONH。 为了正确工作，可能需要禁止中断。 该内置函数可以用作器件数据手册或系列参考手册中介绍的复杂序列的一部分。 更多信息，请参见上述文档。
原型：	`void __builtin_write_OSCCONH(unsigned char value);`
参数：	*value* 要写入的字符
返回值：	无。

汇编操作符 / 机器指令 *：	`mov #0x78, w0`
	`mov #0x9A, w1`
	`mov #_OSCCON+1, w2`
	`mov.b w0, [w2]`
	`mov.b w1, [w2]`
	`mov.b value, [w2]`

错误消息	无。

* 确切的序列可能会有所不同。

© 2015 Microchip Technology Inc.

附录 H 文档版本历史

版本 A （2012 年 4 月）

本文档的初始版本。

版本 B （2012 年 7 月）

- 增加了**第 2 章 “通用 C 接口”**。
- 更新了表 5-16 “软件开发工具数据流”。
- 表 5-16 “链接选项”现在包含了 -fill 选项。
- 在**第 8.11.4 节 “__pack_upper_byte 类型限定符”**和**第 10.10 节 “压缩存储在闪存中的数据”**中增加了 -pack_upper_byte 限定符信息。
- 在**第 13.8 节 “函数调用约定”**下增加了关于保留 DBRPAG/PSVPAG 的列表项
- 修正了**第 14.4 节 “指定中断向量”**中的代码语法。
- 修正了**第 18 章 “优化”**下的评估版说明。
- 在**附录 G “内置函数”**中，为 SFR 寄存器添加了 “volatile”。
- 在**附录 G “内置函数”**中增加了内置函数 __builtin_write_CRYOTP 和 __builtin_write_NVM_secure。

版本 C （2013 年 9 月）

- 将 “用于 PIC24 MCU 和 dsPIC DSC 的 MPLAB 汇编器 / 链接器”（以及各种变化形式）重命名为 “MPLAB XC16 汇编器 / 链接器”。
- 将可执行输出从 `.out` 更改为 `.elf`。
- 更新了**第 1.4 节 “编译器和其他开发工具”**中的 MDB 信息。
- 增加了**第 4 章 “XC16 工具链和 MPLAB X IDE”**和**第 4 章 “XC16 工具链和 MPLAB X IDE”**。
- 在**第 5.7 节 “驱动程序选项说明”**下增加了选项：`-menable-fixed` 和 `-fsigned-bitfields`。
- 在**第 6.5 节 “配置位访问”**下增加了关于使用 #pragma 伪指令的信息。
- 增加了定点算术支持：
  - **第 9 章 “定点算术支持”**
  - **第 8.4 节 “浮点型数据类型”**
  - **第 12.2 节 “寄存器变量”**(`_Sat`、 `_Fract` 和 `_Accum`)
  - **第 13.2.2 节 “函数属性”**（`round`）
  - **第 13.8 节 “函数调用约定”**（`_Fract` 和 `_Accum`）
- 更新了**第 8.6.2 节 “结构中的位域”**下的位域。
- 在**第 13.2.2 节 “函数属性”**中增加了以下属性：`naked` 和 `keep`。
- 在**第 14.3 节 “编写中断服务程序”**下增加了关于 ISR 段命名的信息。此外，从本手册中删除了中断向量信息，并将其移至编译器安装目录的 `docs` 子目录中，如**第 14.4 节 “指定中断向量”**所述。

- 在**第 18 章 “优化”**中增加了优化详细信息。
- 更新了**第 19.4.2 节 “输出类型和器件宏”**。
- 在**第 A.10 节 “结构、联合、枚举和位域”**中增加了关于位域的信息，在**第 A.14 节 “预处理伪指令”**中增加了关于 `#pragma config` 的信息。
- 在**附录 G “内置函数”**中增加了以下内置函数：
  - __builtin_disable_interrupts
  - __builtin_enable_interrupts
  - __builtin_get_isr_state
  - __builtin_set_isr_state
  - __builtin_section_begin
  - __builtin_section_end
  - __builtin_section_size
- 增加了**附录 B “嵌入式编译器兼容性模式”**。

版本 D （2014 年 8 月）

- 增加了**第 3 章 “操作指南”**。
- 删除了第 4 章 “XC16 工具链和 MPLAB IDE v8”。
- **前言**——进行了更新，增加了 “操作指南”章节的引用，删除了 “XC16 工具链和 MPLAB IDE v8”章节的引用。
- **整篇文档**——删除了提及的 MPLAB IDE v8.xx 文字，必要的情况除外。
- **第 4.2 节 “MPLAB X IDE 和工具安装”**——更新了许可证信息。
- **第 4.5 节 “项目设置”**——更新了工具链选项。

版本 E （2014 年 12 月）

- **第 2.5.10 节 “中断函数”**——更正了一个函数。
- **第 4.2 节 “MPLAB X IDE 和工具安装”**——更新了许可信息。
- **第 4.5 节 “项目设置”**——更新了 MPLAB X IDE 中的编译器选项。
- **第 5.4.1.2 节 “用户定义的库”**——增加了关于内容的信息。
- **第 5.7.1 节 “特定于 16 位器件的选项”**——增加了用于双分区器件的 `--partition` 选项。
- **第 5.7.4 节 “用于控制警告和错误的选项”**——拆分为几个小节。摘取来自表中的信息，编排为介绍 `-W` 的小节。
- **第 5.7.6 节 “用于控制优化的选项”**——拆分为几个小节。增加了一些信息，编排为介绍 `-ffunction-sections` 选项的小节。
  增加了来自**第 14.3 节 “编写中断服务程序”**的交叉引用。
- **第 6.3.2 节 “编译时存储器信息”**——增加了章节。
  在 `space` 属性中增加了 `dataflash` 参数。
- **第 10.3.1 节 “自动和非自动变量与局部和全局变量”**——根据第 10.3 节最后两段内容创建的章节。
- **第 13.2.2 节 “函数属性”**和**第 14.5.2 节 “context 属性”**——增加了关于 context 属性的信息。
- **第 13.8 节 “函数调用约定”**——更新了表中关于 EDS 指针要求的信息。
- **第 14.3.3 节 “编写 ISR 代码”**、**第 14.4.1 节 “中断向量使用”**和**第 14.5.1 节 “汇编代码和 ISR”**——更新了代码片段。

© 2015 Microchip Technology Inc.

- **第 14.4 节 “指定中断向量”**——增加了关于可移动备用中断表的信息，并将剩余的文字拆分为两个小节。
- **第 16.3 节 “使用行内汇编语言”**——增加了编译器约束字母。
- **第 19.4.3 节 “器件特性宏”**——对 __HAS_DMA__ 宏进行了说明。
- **附录 G “内置函数”**——增加了 __builtin_write_NVM_secure 和 __builtin_software_breakpoints；更新了 __builtin_enable_interrupts 和 __builtin_disable_interrupts。

支持

简介

关于支持问题，请参考本章提供的内容。

- myMicrochip 个性化通知客户服务
- Microchip 网站
- Microchip 论坛
- 客户支持
- 联系 Microchip Technology

myMICROCHIP 个性化通知客户服务

myMicrochip：http://www.microchip.com/pcn

Microchip 的个人通知客户服务有助于客户了解关于所关注 Microchip 产品的最新信息。注册客户可在某个产品系列或开发工具发生变更、更新、发布新版本或勘误表时，收到电子邮件通知。

请访问 myMicrochip 开始注册过程，并选择接收个性化通知的首选项。页面上提供了 FAQ 和注册详细信息，可以通过选择上面的链接打开该页面。

在选择首选项时，选择 “Development Systems”（开发系统）将会在列表中填入可用的开发工具。下面列出了主要的工具类别：

- **编译器**——Microchip C 编译器、汇编器、链接器及其他语言工具的最新信息。包括所有 MPLAB C 编译器；所有 MPLAB 汇编器 （包括 MPASM™ 汇编器）；所有 MPLAB 链接器 （包括 MPLINK™ 目标链接器）；以及所有 MPLAB 库管理器（包括 MPLIB™ 目标库管理器）。
- **仿真器**——Microchip 在线仿真器的最新信息，包括 MPLAB REAL ICE™ 在线仿真器。
- **在线调试器**——Microchip 在线调试器的最新信息，包括 PICkit™2、 PICkit 3 和 MPLAB ICD 3 在线调试器。
- **MPLAB® IDE**——关于开发系统工具的 Windows® 集成开发环境 Microchip MPLAB IDE 的最新信息。主要针对 MPLAB IDE、 MPLAB IDE 项目管理器、MPLAB 编辑器、 MPLAB SIM 模拟器以及一般编辑和调试功能。
- **编程器**——Microchip 编程器的最新信息。包括器件 （生产）编程器 MPLAB REAL ICE 在线仿真器、 MPLAB ICD 3 在线调试器和 MPLAB PM3 以及开发（非生产）编程器 PICkit 2 和 3。
- **入门 / 演示板**——它们包括 MPLAB 入门工具包电路板、 PICDEM 演示板以及各种其他评估板。

MICROCHIP 网站

网站：http://www.microchip.com

Microchip 网站为客户提供在线支持。客户可通过该网站方便地获取文件和信息。只要使用常用的互联网浏览器即可访问。网站提供以下信息：

- **产品支持**——数据手册和勘误表、应用笔记和示例程序、设计资源、用户指南以及硬件支持文档、最新的软件版本以及归档软件
- **一般技术支持**——常见问题解答 （FAQ）、技术支持请求、在线讨论组以及 Microchip 顾问计划成员名单
- **Microchip 业务**——产品选型和订购指南、最新 Microchip 新闻稿、研讨会和活动安排表、 Microchip 销售办事处、代理商以及工厂代表列表

MICROCHIP 论坛

论坛：http://www.microchip.com/forums

Microchip 网上论坛为客户提供更多的在线支持。当前提供的论坛包括：

- 开发工具
- 8 位 PIC MCU
- 16 位 PIC MCU
- 32 位 PIC MCU

客户支持

Microchip 产品的用户可通过以下渠道获得帮助：

- 代理商或代表
- 当地销售办事处
- 应用工程师 （FAE）
- 技术支持

客户应联系其代理商、代表或应用工程师（FAE）寻求支持。当地销售办事处也可为客户提供帮助。本文档后附有销售办事处的联系方式。请访问我们的网站获取完整、最新的销售办事处列表。

技术支持：http://www.microchip.com/support

文档错误或意见可通过电子邮件发送至 CTRC@microchip.com。

联系 MICROCHIP TECHNOLOGY

您可以通过以下号码致电或发送传真到 Microchip 总部：

电话：(480) 792-7200

传真：(480) 792-7277

© 2015 Microchip Technology Inc.

注：

注：

© 2015 Microchip Technology Inc.

术语表

A

ANSI

美国国家标准学会，是美国负责制订和批准标准的组织。

ASCII

美国信息交换标准码是使用 7 个二进制位来表示每个字符的字符集编码。它包括大写和小写字母、数字、符号以及控制字符。

B

Build

编译并链接一个应用程序的所有源文件。

八进制 （Octal）

使用数字 0-7，以 8 为基数的计数体制。最右边的位表示 1 的倍数，右边第二位表示 8 的倍数，右边第三位表示 8^2 = 64 的倍数，以此类推。

本机数据大小 （Native Data Size）

对于本机跟踪，在 Watches （观察）窗口中使用的变量大小必须等于选定器件的数据存储器的大小：对于 PIC18 器件为字节，对于 16 位器件为字。

编译堆栈 （Compiled Stack）

一个由编译器管理的存储区，在其中为变量静态分配空间。当无法在目标器件上高效地实现软件或硬件堆栈机制时，它可以替代软件或硬件堆栈。

编译器 （Compiler）

将用高级语言编写的源文件翻译成机器码的程序。

标识符 （Identifier）

函数名或变量名。

表达式 （Expression）

被算术或逻辑操作符分隔开的常量和 / 或符号的组合。

C

C/C++

C 语言是具有表达式简练、现代控制流程和数据结构，以及操作符丰富等特点的通用编程语言。 C++ 是 C 语言的面向对象版本。

COFF

公共目标文件格式。这种格式的目标文件包含机器码、调试及其他信息。

CPU

*参见*中央处理单元。

操作符 （Operator）

在构成定义明确的表达式时使用的符号，例如加号“+”和减号“-”。每个操作符都具有指定的优先级，用于确定求值的顺序。

操作码 （Opcode）

操作码。*参见*助记符。

插件 （Plug-in）

MPLAB IDE/MPLAB X IDE 具有内置组件和插件模块，用于针对各种软件和硬件工具来配置系统。可在 Tools （工具）菜单下找到几个插件工具。

场景 （Scenario）

MPLAB SIM 软件模拟器用于激励控制的一种特定设置。

程序存储器 （Program Memory）

MPLAB IDE/MPLAB X IDE——器件中存储指令的存储器。亦指仿真器或软件模拟器中包含下载的目标应用固件的存储器。

16 位汇编器 / 编译器——器件中存储指令的存储器。

程序计数器 （Program Counter）

包含正在执行的指令的地址的存储单元。

程序计数器单元 （Program Counter Unit）

16 位汇编器——程序存储器布局的概念性表示。对于每个指令字，程序计数器将递增 2。在可执行段中，2 个程序计数器单元相当于 3 个字节。在只读段中，2 个程序计数器单元相当于 2 个字节。

持久性数据 （Persistent Data）

永不清除或初始化的数据。其目的是使应用程序可以在器件复位时保存数据。

重定位 （Relocation）

链接器执行的一个过程，在这个过程中，为可重定位段分配绝对地址，且可重定位段中的所有符号都更新为其新地址。

触发输出 （Trigger Output）

触发输出指可在任意地址或地址范围产生的仿真器输出信号，与跟踪和断点的设置无关。可设置任意个触发输出点。

次数计数器 （Pass Counter）

每次一个事件 （如执行特定地址处的一条指令）发生时都会递减 1 的计数器。当次数计数器的值为零时，事件满足。可将次数计数器分配给中断 （break）和跟踪逻辑，以及在 complex trigger （复杂触发）对话框中的任何连续事件。

存储类别 （Storage Class）

确定与所标识对象相关联存储区的存在时间。

© 2015 Microchip Technology Inc.

存储器模型 （Memory Model）

对于 C 编译器，指应用程序可使用的存储区的表示。对于 XC16 C 编译器，指一种描述，它指定指向程序存储器的指针的位数。

存储限定符 （Storage Qualifier）

指明所声明对象的特殊属性 （如 `const`）。

错误 / 错误文件 （Error/Error File）

错误报告使程序不能继续处理的问题。而且，当问题比较明显时，错误还尽可能标识出源文件名和行号。错误文件包含由语言工具生成的错误消息和诊断信息。

D

DWARF

使用任意记录格式调试。 DWARF 是用于 ELF 文件的调试信息格式。

带序列号的快速批量编程 （Serialized Quick Turn Programming）

序列化使您可以将序列号编程到器件编程器进行编程的每个单片机中。该编号可用作记录代码、密码或 ID 号。

单片机 （Microcontroller）

高度集成的芯片，它包括 CPU、 RAM、程序存储器、 I/O 端口和定时器。

单片机模式 （Microcontroller Mode）

PIC18 单片机的一种程序存储器配置。在单片机模式下，仅允许内部执行。因此，在单片机模式下仅可使用片内程序存储器。

导出 （Export）

以标准的格式将数据发送出 MPLAB IDE/MPLAB X IDE。

导入 （Import）

从外面的源 （如 hex 文件）将数据送入 MPLAB IDE/MPLAB X IDE。

地址 （Address）

标识存储器中位置的值。

递归 （Recursion）

已定义的函数或宏可调用自己的概念。当编写递归宏时要特别小心；当递归没有出口时容易陷入无限循环。

递归调用 （Recursive Call）

一个直接或间接调用自身的函数。

段

OCG psect 的 GCC 等效形式。被链接器视为一个整体的代码或数据块。

段属性 （Section Attribute）

赋予段的 GCC 特性 （例如， `access` 段）。

断点 （Breakpoint）

硬件断点：一个事件，执行这种事件会导致暂停。

软件断点：一个地址，固件会在这个地址处暂停执行。通常由特殊的 break 指令获得。

堆 （Heap）

用于动态存储器分配的存储区，其中的存储器块按运行时确定的任意顺序进行分配和释放。

堆栈，编译 （Stack, Compiled）

一个由编译器管理和分配的存储区，在其中为变量静态分配空间。当无法在目标器件上高效地实现软件堆栈机制时，它可以替代软件堆栈。它会阻止重入。

堆栈，软件 （Stack, Software）

应用程序用来存储返回地址、函数参数和局部变量的存储区。该存储区在运行时通过程序中的指令动态分配。它支持可重入函数调用。

堆栈，硬件 （Stack, Hardware）

PIC 单片机中调用函数时存储返回地址的存储区。

E

EEPROM

电可擦除的可编程只读存储器。一种可电擦除的特殊 PROM。一次写或擦除一个字节。EEPROM 即使电源关闭时也能保留内容。

ELF

可执行链接格式。这种格式的目标文件包含机器码。调试和其他信息使用 DWARF 指定。 ELF/DWARF 可提供优于 COFF 的优化代码调试。

Epilogue

编译器生成代码的一部分，负责释放堆栈空间、恢复寄存器，以及执行运行时模型中指定的任何其他特定于机器的要求。此代码在给定函数的任何用户代码之后、在函数返回之前执行。

EPROM

可擦除的可编程只读存储器。通常通过紫外线照射来擦除的可编程只读存储器。

二进制 （Binary）

使用数字 0 和 1，以 2 为基数的计数体制。最右边的位表示 1 的倍数，右边第二位表示 2 的倍数，右边第三位表示 $2^2 = 4$ 的倍数，以此类推。

F

Fixup

在链接器重定位之后，使用绝对地址替换目标文件符号引用的过程。

FNOP

强制空操作。强制 NOP 周期是双周期指令的第二个周期。由于 PIC 单片机的架构是流水线型，在执行当前指令的同时预取物理地址空间中的下一条指令。但是，如果当前指令改变了程序计数器，那么这条预取的指令就被忽略了，导致一个强制 NOP 周期。

Free-Standing

一种实现，它接受任何不使用复杂类型的严格符合程序，而且在这种实现中，对库条款（ANSI ‘89 标准条款 7）中规定的属性的使用，仅限于标准头文件：`<float.h>`、`<iso646.h>`、`<limits.h>`、`<stdarg.h>`、`<stdbool.h>`、`<stddef.h>` 和 `<stdint.h>`。

© 2015 Microchip Technology Inc.

仿真 / 仿真器 （Emulation/Emulator）

参见 ICE/ICD。

访问入口点 （Access Entry Point）

访问入口点提供了一种方法，可跨段将控制转移到某个可能未在链接时定义的函数。它们支持独立链接引导应用程序段和安全应用程序段。

非扩展模式 （Non-Extended Mode）（PIC18 MCU）

在非扩展模式下，编译器不会使用扩展指令和立即数变址寻址。

非实时 （Non Real Time）

指处理器执行到断点或单步执行指令，或 MPLAB IDE/MPLAB X IDE 运行于软件模拟器模式。

非易失性存储器 （Non-Volatile Storage）

电源关闭时保存其内容的存储器件。

符号 （Symbol）

符号是描述组成程序的各个部分的一种通用机制。这些部分包括函数名、变量名、段名、文件名和结构 / 枚举 / 联合标记名等。 MPLAB IDE/MPLAB X IDE 中的符号主要指变量名、函数名和汇编标号。链接后符号的值就是其在存储器中的值。

符号，绝对 （Symbol, Absolute）

表示一个立即数的值，例如通过汇编 .equ 伪指令指定的定义。

G

GPR

通用寄存器。器件数据存储器 （RAM）的一部分，作为一般用途。

概要文件 （Profile）

对于 MPLAB SIM 软件模拟器，寄存器已执行激励的汇总列表。

高级语言 （High Level Language）

编写程序的语言，它比汇编语言更不依赖于具体的处理器。

跟踪 （Trace）

记录程序执行的仿真器或软件模拟器功能。仿真器将程序执行记录到其跟踪缓冲区内，并可上载到 MPLAB IDE/MPLAB X IDE 的跟踪窗口中。

跟踪存储区 （Trace Memory）

跟踪存储区包含在仿真器内部。跟踪存储区有时称为跟踪缓冲区。

跟踪宏 （Trace Macro）

一个通过仿真器数据来提供跟踪信息的宏。由于该宏属于软件跟踪，所以必须将它添加到代码中、必须重新编译或重新汇编代码，并且必须使用该代码对目标器件进行编程，之后跟踪才会工作。

工具栏 （Tool Bar）

一行或一列图标，单击这些图标时将执行 MPLAB IDE/MPLAB X IDE 功能。

工作簿 （Workbook）

MPLAB SIM 软件模拟器用于生成 SCL 激励的一种设置。

观察变量 （Watch Variable）

调试会话期间可在 Watches 窗口中监控的变量。

归档 / 归档器 （Archive/Archiver）

归档 / 库是可重定位目标模块的集合。由将多个源文件编译 / 汇编为目标文件，然后使用归档器 / 库管理器将目标文件组合为一个归档 / 库文件生成。可将归档 / 库与目标模块和其他归档 / 库链接，生成可执行代码。

国际标准化组织 （International Organization for Standardization）

制订许多行业和技术 （包括计算和通信）方面的标准的一个组织。也称为 ISO。

过滤器 （Filter）

通过选择确定在跟踪显示或数据文件中包含 / 排除哪些数据。

H

Hex 代码 /Hex 文件 （Hex Code/Hex File）

Hex 代码是以十六进制格式代码存储的可执行指令。 Hex 代码包含在 hex 文件中。

宏 （Macro）

宏指令。以缩写形式表示指令序列的指令。

宏伪指令 （Macro Directive）

控制宏定义体中执行和数据分配的伪指令。

环回测试板 （Loop-Back Test Board）

用于测试 MPLAB REAL ICE 在线仿真器的功能。

环境 （Environment）

MPLAB PM3——包含关于如何对器件编程的文件的文件夹。该文件夹可以转移到 SD/MMC 卡。

汇编语言 / 汇编器 （Assembly/Assembler）

汇编语言是以符号形式描述二进制机器码的编程语言。汇编器是将汇编源代码翻译成机器码的语言工具。

I

ICE/ICD

在线仿真器 / 在线调试器：用于对目标器件进行调试和编程的硬件工具。仿真器具有比调试器更多的功能，例如跟踪。

在线仿真 / 在线调试：使用在线仿真器或调试器进行仿真或调试的行为。

-ICE/-ICD：带有在线仿真或调试电路的器件（MCU 或 DSC）。该器件总是安装在仿真头板上，并用于通过在线仿真器或调试器进行调试。

ICSP

在线串行编程。使用串行通信和最少数量的器件引脚对 Microchip 嵌入式器件进行编程的方法。

© 2015 Microchip Technology Inc.

IDE

集成开发环境，如 MPLAB IDE/MPLAB X IDE。

IEEE

电气电子工程师协会。

J

机盒 （Pod）

在线仿真器或调试器的外壳。其他名称还有 “Puck”（如果外壳为圆形）和 “Probe”（不要与逻辑探头相混淆）。

机器码 （Machine Code）

处理器实际读和解释的计算机程序的表示。二进制机器码程序由一系列机器指令（可能还包含数据）组成。特定处理器的所有可能指令的集合称为 “指令集”。

机器语言 （Machine Language）

特定中央处理单元的指令集，不需翻译即可供处理器使用。

基数 （Radix）

数制，十六进制或十进制，用于指定一个地址。

激励 （Stimulus ）

软件模拟器的输入（即为模拟对外部信号的响应而生成的数据）。通常数据采用文本文件中一系列动作的形式。激励可以是异步的，同步的（引脚），时钟激励和寄存器激励。

交叉引用文件 （Cross Reference File）

引用符号表的一个文件及引用符号的文件的列表。如果定义了符号，则在列出的第一个文件中包含符号的定义。其他文件包含对符号的引用。

校准存储区 （Calibration Memory）

用于保存 PIC 单片机内 RC 振荡器或其他外设校准值的特殊功能寄存器或通用寄存器。

节点 （Node）

MPLAB IDE/MPLAB X IDE 项目组成部分。

警告 （Warning）

MPLAB IDE/MPLAB X IDE——提醒出现了可能导致器件、软件文件或设备物理损坏的通知。

16 位汇编器 / 编译器——警告报告可能存在问题的条件，但并不暂停处理。在 MPLAB C30 中，警告消息报告源文件名和行号，但包含文本 “`warning:`”来将警告与错误消息区分开。

静态 RAM 或 SRAM （Static RAM or SRAM）

静态随机访问存储器。目标板上可读 / 写且不需要经常刷新的程序存储器。

局部标号 （Local Label）

局部标号是用 LOCAL 伪指令在一个宏内部定义的标号。这些标号特定于宏实例化的一个给定示例。也就是说，声明为 local 的符号和标号在遇到 ENDM 宏后不再可访问。

绝对变量 / 函数 （Absolute Variable/Function）

使用 OCG 编译器的 `@ address` 语法放置在绝对地址处的变量或函数。

绝对段 （Absolute Section）

具有链接器不能改变的固定 （绝对）地址的 GCC 编译器段。

K

看门狗定时器 （Watchdog Timer， WDT）

PIC 单片机中在一段可选择的时间长度后复位处理器的定时器。使用配置位来使能、禁止和设置 WDT。

可重定位 （Relocatable）

对象的地址没有分配到存储器中的固定地址。

可重定位段 （Relocatable Section）

16 位汇编器——地址非固定 （绝对）的段。链接器通过一个称为重定位的过程来为可重定位段分配地址。

可重入函数 （Reentrant）

可以有多个同时运行的实例的函数。在下面两种情况下可能发生函数重入：直接或间接递归调用函数；或者在由函数转入的中断处理过程中又执行此函数。

可执行代码 （Executable Code）

可装入来执行的软件。

控制伪指令 （Control Directive）

汇编语言代码中根据汇编时指定表达式的值包含或忽略代码的伪指令。

库 / 库管理器 （Library/Librarian）

*参见*归档 / 归档器。

快速操作存储区 （Access Memory）

仅限 PIC18——PIC18 器件中的一些特殊寄存器，对这些寄存器的访问与存储区选择寄存器 （BSR）的设置无关。

扩展单片机模式 （Extended Microcontroller Mode）

在扩展单片机模式下，既可使用片内程序存储器，也可使用外部存储器。如果程序存储器地址大于 PIC18 器件的内部存储空间，执行自动切换到外部存储器。

扩展模式 （Extended Mode）（PIC18 MCU）

在扩展模式下，编译器将使用扩展指令 （即 ADDFSR、 ADDULNK、 CALLW、 MOVSF、 MOVSS、 PUSHL、 SUBFSR 和 SUBULNK）以及立即数变址寻址。

L

LVDS

低电压差分信号传输。一种通过铜线进行高速（每秒千兆位）数据传输的低噪声、低功耗、低幅值方法。

对于标准 I/O 信号，数据存储依赖于实际电压大小。电压大小会受线路长度影响（线路越长，电阻就越高，这会使电压下降）。但对于 LVDS，数据存储仅通过正负电压值区分，而不是实际电压大小。因此，数据可以传输更长的线路距离，同时保持干净、一致的数据流。

来源：http://www.webopedia.com/TERM/L/LVDS.html。

链接描述文件 （Linker Script File）

链接描述文件是链接器的命令文件。它们定义链接选项并描述目标平台上的可用存储器。

© 2015 Microchip Technology Inc.

链接器 （Linker）

将目标文件和库文件组合起来生成可执行代码并解析一个模块对另外一个模块引用的语言工具。

列表伪指令 （Listing Directive）

列表伪指令是控制汇编列表文件格式的伪指令。它们允许指定标题、分页及其他列表控制。

列表文件 （Listing File）

列表文件是列出为每条 C 源语句生成的机器码、源文件中遇到的汇编指令、汇编器伪指令或宏的 ASCII 文本文件。

逻辑探头 （Logic Probe）

Microchip 的某些仿真器最多可连接 14 个逻辑探头。逻辑探头提供外部跟踪输入、触发输出信号、 +5V 和公共接地端。

M

Makefile

将用于 Make 项目的指令导出到一个文件中。使用该文件可以在 MPLAB IDE/MPLAB X IDE 之外使用 `make` 编译项目。

Make 项目 （Make Project）

重新编译应用程序的命令，仅编译自上次编译完成后更改了的源文件。

MCU

单片机。 microcontroller 的缩写形式。也写作 uC。

MPASM™ 汇编器 （MPASM™ Assembler）

Microchip Technology 用于 PIC 单片机、KeeLoq® 器件及 Microchip 存储器件的可重定位宏汇编器。

MPLAB ICD

Microchip 的在线调试器，与 MPLAB IDE/MPLAB X IDE 配合使用。*参见* ICE/ICD。

MPLAB IDE/MPLAB X IDE

Microchip 的集成开发环境。 MPLAB IDE/MPLAB X IDE 附带了编辑器、项目管理器和软件模拟器。

MPLAB PM3

Microchip 的器件编程器。用于对 PIC18 单片机和 dsPIC 数字信号控制器进行编程。可与 MPLAB IDE/MPLAB X IDE 配合使用或独立使用。代替 PRO MATE II。

MPLAB REAL ICE™ 在线仿真器 （MPLAB REAL ICE™ In-Circuit Emulator）

Microchip 的新一代在线仿真器，与 MPLAB IDE/MPLAB X IDE 配合使用。*参见* ICE/ICD。

MPLAB SIM

Microchip 的软件模拟器，与 MPLAB IDE/MPLAB X IDE 配合使用，支持 PIC MCU 和 dsPIC DSC 器件。

MPLIB™ 目标库管理器 （MPLIB™ Object Librarian）

Microchip 的库管理器，与 MPLAB IDE/MPLAB X IDE 配合使用。MPLIB 库管理器是用于将由 MPASM 汇编器（mpasm 或 mpasmwin v2.0）或 MPLAB C18 C 编译器生成的 COFF 目标模块组合成库文件的目标库管理器。

MPLINK™ 目标链接器 （MPLINK™ Object Linker）

MPLINK 链接器是 Microchip MPASM 汇编器和 Microchip C18 C 编译器的目标链接器。也可将 MPLINK 链接器与 Microchip MPLIB 库管理器配合使用。MPLINK 链接器设计为可与 MPLAB IDE/MPLAB X IDE 配合使用，尽管它也可独立于 MPLAB IDE/MPLAB X IDE 使用。

MRU

最近使用的。指可从 MPLAB IDE/MPLAB X IDE 主下拉菜单选择的文件和窗口。

命令行接口 （Command Line Interface）

仅基于文本输入和输出，在程序和其用户之间进行通信的一种方式。

模板 （Template）

为以后插入自己的文件中使用而创建的文本行。 MPLAB 编辑器将模板存储到模板文件中。

模块 （Module）

源文件在执行预处理器伪指令之后的预处理输出。也称为翻译单元。

目标 （Target）

指用户硬件。

目标板 （Target Board）

构成目标应用的电路和可编程器件。

目标处理器 （Target Processor）

目标应用板上的单片机。

目标代码 / 目标文件 （Object Code/Object File）

目标代码是由汇编器或编译器生成的机器码。目标文件是包含机器码，也可能包含调试信息的文件。它可以直接执行；或为可重定位的，需要与其他目标文件（如库文件）链接来生成完整的可执行程序。

目标文件伪指令 （Object File Directive）

仅当生成目标文件时使用的伪指令。

目标应用程序 （Target Application）

目标板上的软件。

© 2015 Microchip Technology Inc.

N

NOP

空操作。执行该指令时，除了程序计数器加 1 外没有任何其他影响。

内部链接 （Internal Linkage）

如果不能从定义函数或变量的模块外部访问它们，则这样的函数或变量具有内部链接。

匿名结构 （Anonymous Structure）

16 位 C 编译器——未命名的结构。

PIC18 C 编译器——属于 C 联合成员的未命名结构。匿名结构的成员可以像外围联合的成员一样进行访问。例如，在以下代码中， `hi` 和 `lo` 是联合 `caster` 中的匿名结构的成员。

```
union castaway{
 int intval;
 struct {
  char lo; //accessible as caster.lo
  char hi; //accessible as caster.hi
 };
} caster;
```

O

OTP

一次性编程。非窗口封装的 EPROM 器件。由于 EPROM 需要紫外线照射来擦除其存储内容，因此只有窗口片是可擦除的。

P

PC

个人计算机或程序计数器。

PC 主机 （PC Host）

运行有一个支持的 Windows 操作系统的任何 PC。

PIC MCU

PIC 单片机 （MCU）指 Microchip 的所有单片机系列。

PICkit 2 和 3 （PICkit 2 and 3）

Microchip 适用于开发阶段的器件编程器，具有通过 Debug Express 进行调试的功能。要了解每个工具支持哪些器件，请参见每个工具的自述文件。

Pragma 伪指令 （Pragma）

对特定编译器有意义的伪指令。通常一条 pragma 伪指令用于向编译器传达实现定义的信息。 MPLAB C30 使用属性来传达这种信息。

Prologue

编译器生成代码的一部分，负责分配堆栈空间、保存寄存器，以及执行运行时模型中指定的任何其他特定于机器的要求。此代码在给定函数的任何用户代码之前执行。

Psect

GCC 段的 OCG 等效形式， program section （程序段）的缩写。被链接器视为一个整体的代码或数据块。

PWM 信号 （PWM Signal）

脉宽调制信号。某些 PIC MCU 包含 PWM 外设。

跑表 （Stopwatch）

测量执行周期的计数器。

配置位 （Configuration Bit）

可编程来设置 PIC 单片机工作模式的专用位。配置位可或不可再编程。

片外存储器 （Off-Chip Memory）

片外存储器指 PIC18 器件的一种存储器选择，这种情况下存储器可位于目标板上，或所有程序存储器都由仿真器提供。从 *Options>Development Mode（选项> 开发模式）*访问 **Memory** 选项卡可打开 Off-Chip Memory Selection （片外存储器选择）对话框。

Q

弃用功能 （Deprecated Feature）

由于历史原因仍然支持但最终将逐步淘汰且不再使用的功能。

器件编程器 （Device Programmer）

用于对电可编程半导体器件 （如单片机）进行编程的工具。

***器件*的 MPLAB 入门工具包 （MPLAB Starter Kit for *Device*）**

Microchip的入门工具包包含了着手研究指定器件所需的一切。查看工作中的应用程序，然后对您所做的更改进行调试和编程。

***器件*的 MPLAB *语言工具* （MPLAB *Language Tool* for *Device*）**

Microchip 用于指定器件的 C 编译器、汇编器和链接器。根据应用中所使用的器件来选择语言工具类型。例如，如果为 PIC18 MCU 编写 C 代码，就选择 MPLAB XC16 C 编译器。

嵌套深度 （Nesting Depth）

宏可包含其他宏的最大深度。

清除 （Clean）

清除会删除活动项目的所有中间项目文件，例如目标文件、hex 文件和调试文件。编译项目时，将基于其他文件重新创建这些文件。

R

RAM

随机访问存储器 （数据存储器）。可以以任意顺序访问这种存储器中的信息。

ROM

只读存储器 （程序存储器）。不能修改的存储器。

Run

将仿真器从暂停状态释放，允许仿真器实时运行应用代码、实时改变 I/O 状态或实时响应 I/O 的命令。

软件模拟器

模仿器件操作的软件程序。

© 2015 Microchip Technology Inc.

S

Shell

MPASM 汇编器 shell 是宏汇编器的提示输入接口。有两个 MPASM 汇编器 shell：一个针对 DOS 版本，一个针对 Windows 版本。

Single Step

这一命令单步执行代码，一次执行一条指令。执行每条指令后，MPLAB IDE/MPLAB X IDE 更新寄存器窗口、观察变量及状态显示，使您可分析和调试指令。您也可以单步执行 C 编译器源代码，但是 MPLAB IDE/MPLAB X IDE 将执行由一行高级 C 语句生成的所有汇编指令，而不是执行单条汇编指令。

Skew

不同时间出现在处理器总线上与指令执行有关的信息。例如，执行前一条指令的过程中取指时，被执行的操作码出现在总线上；当实际执行该操作码时，源数据地址及其值以及目标数据地址出现在总线上。当执行下一条指令时，目标数据值出现在总线上。跟踪缓冲区一次捕捉总线上的这些信息。因此，跟踪缓冲区的一条记录将包含三条指令的执行信息。执行一条指令时，从一条信息到另一条信息的捕捉周期数称为 skew。

Skid

当使用硬件断点来暂停处理器时，在处理器暂停前可能再执行一条或多条指令。断点后执行的指令条数称为 skid。

SQTP

*参见*带序列号的快速批量编程。

Step Into

这一命令与 Single Step 相同。 Step Into （与 Step Over 相对）在 CALL 指令后，单步执行子程序。

Step Out

Step Out 使您可以单步跳出当前正在单步执行的子程序。该命令会执行该子程序中的剩余代码，然后在该子程序的返回地址处停止执行。

Step Over

Step Over 允许调试代码时不单步执行子程序。当 step over 一条 CALL 指令时，下一个断点将设置在 CALL 指令后的下一条指令处。如果由于某种原因，子程序陷入无限循环或不正确返回，下一个断点将永远执行不到。除了对 CALL 指令的处理外， Step Over 命令与 Single Step 相同。

三字符组合 （Trigraph）

三字符序列，都以 ?? 开头，由 ISO C 定义为单个字符的替代。

闪存 （Flash）

按块 （而不是按字节）写或擦除数据的一种 EEPROM。

上电复位仿真 （Power-on-Reset Emulation）

在应用刚上电时，将随机值写到数据 RAM 区中来模拟 RAM 中的未初始化值的软件随机过程。

上载 （Upload）

上载功能将数据从一个工具 （如仿真器或编程器）传送到主机 PC，或将数据从目标板传送到仿真器。

生产编程器 （Production Programmer）

生产编程器是一种编程工具，其中设计了可对器件进行快速编程的资源。它具有在各种电压下进行编程的能力并完全符合编程规范。在生产环境中，应用电路需要在组装线上传送，时间是极其重要的，所以尽可能快地对器件编程至关重要。

十六进制 （Hexadecimal）

使用数字 0-9 以及字母 A-F（或 a-f），以 16 为基数的计数体制。字母 A-F 表示值为 10 至 15（十进制）的十六进制数字。最右边的位表示 1 的倍数，右边第二位表示 16 的倍数，右边第三位表示 16^2 = 256 的倍数，以此类推。

实时 （Real Time）

当在线仿真器或调试器从暂停状态释放时，处理器以实时模式运行且与芯片的正常操作相同。在实时模式下，使能仿真器的实时跟踪缓冲区，并持续捕捉所有选择的周期，使能所有断点逻辑。在在线仿真器或调试器模式下，处理器实时运行，直到有效断点导致暂停，或者直到用户暂停执行。

在软件模拟器模式下，实时仅意味着单片机指令的执行速度与主机 CPU 可模拟的指令速度一样快。

事件 （Event）

对可能包含地址、数据、次数计数、外部输入、周期类型 （取指和读 / 写）及时间标记的总线周期的描述。事件用于描述触发、断点和中断。

舒缓 （Relaxation）

将某一指令转换为功能相同但大小较小的指令的过程。这对于节省代码长度非常有用。MPLAB ASM30 目前知道如何将 CALL 指令 relax（舒缓）为 RCALL 指令。当被调用的符号处于当前指令的 +/- 32k 指令字范围内时，将会执行该操作。

书签 （Bookmarks）

使用书签可以简便地定位文件中的特定行。

选择编辑器工具栏上的 Toggle Bookmarks （切换书签）可添加 / 删除书签。单击该工具栏上的其他图标可移动到下一个或上一个书签。

© 2015 Microchip Technology Inc.

属性 （Attribute）

C 程序中变量或函数的 GCC 特性，用于描述特定于机器的性质。

属性，段 （Attribute, Section）

段的 GCC 特性，例如 “可执行”、“只读” 或 “数据”，可以在汇编器 `.section` 伪指令中将这些特性指定为标志。

数据存储器 （Data Memory）

在 Microchip MCU 和 DSC 器件中，数据存储器 （RAM）由通用寄存器 （GPR）和特殊功能寄存器 （SFR）组成。某些器件还有 EEPROM 数据存储器。

数据监视与控制界面 （Data Monitor and Control Interface， DMCI）

数据监视与控制界面 （DMCI）是 MPLAB X IDE 中的一个工具。该界面可以对项目中的应用程序变量提供动态输入控制。应用程序生成的数据可以使用 4 个可动态分配图形窗口中的任意一个以图形方式进行查看。

数据伪指令 （Data Directive）

数据伪指令是指控制汇编器的程序或数据存储空间分配，并提供通过符号（即有意义的名称）引用数据项的方法的伪指令。

数字信号处理 / 数字信号处理器（Digital Signal Processing/Digital Signal Processor）

数字信号处理 （DSP）指数字信号以及已转换为数字形式 （经过采样的）的一般模拟信号（声音或图像）的计算机处理。数字信号处理器是设计为用于数字信号处理的微处理器。

数字信号控制器 （Digital Signal Controller）

数字信号控制器 （DSC）是具有数字信号处理能力的单片机 （即 Microchip 的 dsPIC DSC 器件）。

顺序断点 （Sequenced Breakpoints）

按顺序发生的断点。断点的执行顺序为从下到上；序列中的最后一个断点最先发生。

T

特殊功能寄存器 （Special Function Register， SFR）

数据存储器 （RAM）的一部分，专用于控制 I/O 处理器函数、 I/O 状态、定时器或其他模式及外设的寄存器。

条件汇编 （Conditional Assembly）

基于指定表达式在汇编时的值包含或忽略的汇编语言代码。

条件编译 （Conditional Compilation）

只有当预处理器伪指令指定的某个常量表达式为真时才编译程序段的操作。

调试 / 调试器 （Debug/Debugger）

参见 ICE/ICD。

调试信息 （Debugging Information）

编译器和汇编器选项，在选中时，它们将提供不同程度的信息来用于调试应用程序代码。关于选择调试选项的详细信息，请参见编译器或汇编器文档。

U

USB

通用串行总线。一种外部外设接口标准，用于通过电缆使用双向串行传输在计算机和外部外设之间进行通信。USB 1.0/1.1 支持 12 Mbps 的数据传输速率。USB 2.0 也称为高速 USB，它支持最高 480 Mbps 的数据传输速率。

V

Volatile

变量限定符，它可以防止编译器应用会影响变量在存储器中的访问方式的优化。

W

Watches 窗口 （Watches Window）

Watch 窗口包含一系列观察变量，这些变量在每次执行到断点时更新。

外部 RAM （External RAM）

芯片外部的读 / 写存储器。

外部标号 （External Label）

具有外部链接的标号。

外部符号 （External Symbol）

具有外部链接的标识符符号。这可能是一个引用或一个定义。

外部符号解析 （External Symbol Resolution）

链接器收集所有输入模块的外部符号定义来解析所有外部符号引用的过程。没有相应定义的任何外部符号引用都会导致报告链接器错误。

外部链接 （External Linkage）

如果可以在定义函数或变量的模块外部对函数或变量进行引用，则函数或变量具有外部链接。

外部输入线 （External Input Line）

用于根据外部信号设置事件的外部输入信号逻辑探针线 （TRIGIN）。

微处理器模式 （Microprocessor Mode）

PIC18 单片机的一种程序存储器配置。在微处理器模式下，不使用片内程序存储器。整个程序存储器映射到外部。

伪指令 （Directive）

源代码中控制语言工具操作的语句。

尾数法 （Endianness）

多字节对象中的字节存储顺序。

未初始化数据 （Uninitialized Data）

定义时未指定初始值的数据。在 C 中，

```
int myVar;
```

定义了将存放到未初始化数据段的一个变量。

© 2015 Microchip Technology Inc.

未分配段 （Unassigned Section）

在链接器命令文件中未分配到特定目标存储区的段。链接器必须找到用于分配未分配段的目标存储区。

文件寄存器 （File Register）

片内数据存储器，包括通用寄存器 （GPR）和特殊功能寄存器 （SFR）。

X

系统窗口控件 （System Window Control）

系统窗口控件位于窗口和某些对话框的左上角。单击这一控件通常会弹出包含Minimize（最小化）、 Maximize （最大化）和 Close （关闭）等项的菜单。

下载 （Download）

下载是将数据从主机发送到其他设备 （如仿真器、编程器或目标板）的过程。

限定符 （Qualifier）

次数计数器使用的地址或地址范围，或用作复杂触发中另一个操作之前的事件。

响应延时 （Latency）

事件与其得到响应之间的延迟时间。

向量 （Vector）

复位或中断发生时应用程序跳转到的存储地址。

项目 （Project）

项目包含编译应用程序所需的文件（源代码和链接描述文件等），以及它们与各种编译工具和编译选项的关联。

消息 （Message）

显示出来的文本，警告在语言工具的操作中可能存在的问题。消息不会停止操作。

小尾数法 （Little Endian）

多字节数据的数据存储顺序机制，在这种机制中，低字节存储在较低的地址中。

虚拟字节 （Phantom Byte）

dsPIC 架构中的未实现字节，在将 24 位指令字视为 32 位指令字时使用。虚拟字节出现在 dsPIC hex 文件中。

Y

样机系统 （Prototype System）

指代用户的目标应用或目标板的术语。

已初始化数据 （Initialized Data）

用初始值定义的数据。在 C 中，

```
int myVar=5;
```

定义了将存放到已初始化数据段中的一个变量。

已分配段 （Assigned Section）

在链接器命令文件中已分配到目标存储区的 GCC 编译器段。

异步 （Asynchronously）

不同时发生的多个事件。通常用来指可能在处理器执行过程中的任意时刻发生的中断。

异步激励 （Asynchronous Stimulus）

为模拟传递到被模拟器件的外部输入而生成的数据。

应用

可由 PIC® 单片机控制的一组软硬件。

优先顺序 （Precedence）

定义表达式中求值顺序的规则。

与断点 （ANDed Breakpoint）

为中断设置逻辑与条件，即只有断点 1 和断点 2 同时出现时，才会暂停程序。只有数据断点和程序存储器断点同时出现时，才会完成它。

原始数据 （Raw Data）

与一个段有关的代码或数据的二进制表示。

源代码 （Source Code）

编程人员编写计算机程序的形式。采用某种正式的编程语言编写的源代码可翻译成机器码或被解释程序执行。

源文件 （Source File）

包含源代码的 ASCII 文本文件。

运行时观察 （Runtime Watch）

Watches（观察）窗口，其中的变量会在应用程序运行时更改。要确定如何设置运行时观察，请参见相应的工具文档。并不是所有工具都支持运行时观察。

运行时模型 （Run-time Model）

描述目标架构资源的使用。

Z

暂停 （Halt）

停止程序执行。执行 Halt 与在断点处停止相同。

帧指针 （Frame Pointer）

引用堆栈中地址，并将基于堆栈的参数和基于堆栈的局部变量分隔开的指针。为访问当前函数的局部变量和其他值提供了方便。

只读存储器 （Read Only Memory）

存储器硬件，它允许快速访问其中永久存储的数据，但不允许添加或修改数据。

指令 （Instruction）

告知中央处理单元执行特定操作，并可能包含操作中要使用的数据的位序列。

指令集 （Instruction Set）

特定处理器理解的机器语言指令的集合。

© 2015 Microchip Technology Inc.

致命错误 （Fatal Error）

引起编译立即停止的错误。不产生其他消息。

中断 （Interrupt）

传递到 CPU 的信号，它使 CPU 暂停执行正在运行的应用程序，将控制权转交给中断服务程序 （ISR），以处理事件。在完成 ISR 时，将恢复应用程序的正常执行。

中断处理程序 （Interrupt Handler）

发生中断时处理特殊代码的子程序。

中断服务程序 （Interrupt Service Routine， ISR）

语言工具——处理中断的函数。

MPLAB IDE/MPLAB X IDE——当产生中断时进入的用户生成代码。代码在程序存储器中的位置通常取决于所产生中断的类型。

中断服务请求 （Interrupt Service Request， IRQ）

使处理器暂停正常的指令执行并开始执行中断处理程序的事件。某些处理器有几种中断请求事件，允许具有不同优先级的中断。

中断向量 （Interrupt Vector）

中断服务程序或中断处理程序的地址。

中央处理单元 （Central Processing Unit）

器件的一部分，负责取出要执行的正确指令，对指令进行译码，然后执行指令。如果有必要，它和算术逻辑单元（Arithmetic Logic Unit，ALU）一起工作来完成指令的执行。它控制程序存储器的地址总线、数据存储器的地址总线和对堆栈的访问。

助记符 （Mnemonic）

可直接翻译成机器码的文本指令。也称为操作码。

状态条 （Status Bar）

状态条位于 MPLAB IDE/MPLAB X IDE 窗口的底部，指示光标位置、开发模式和器件，以及有效工具条等当前信息。

字母数字字符 （Alphanumeric）

字母数字字符由字母字符和十进制数字 （0, 1, … 9）组成。

字母字符 （Alphabetic Character）

字母字符指属于阿拉伯字母表 （a, b, … z, A, B, … Z）中字母的字符。

左值 （L-value）

引用可被检查和 / 或修改的对象的表达式。左值表达式用在赋值的左侧。

注：

© 2015 Microchip Technology Inc.

索引

符号

.bss 155, 259
.c 72
.const 227
.data 155, 259
.gld 72
.s 72
.text 102, 203, 206, 259
#define 117
#ident 122
#if 109
#include 117, 118
#include 伪指令 92
#line 119
#pragma 106, 247, 259
__align 限定符 34
__bank 限定符 33
__builtin_add 327
__builtin_addab 327
__builtin_btg 328
__builtin_clr 328
__builtin_clr_prefect 329
__builtin_dataflashoffset 330
__builtin_disable_interrupts 330
__builtin_disi 331
__builtin_divf 331
__builtin_divmodsd 331
__builtin_divmodud 332
__builtin_divsd 332
__builtin_divud 332
__builtin_dmaoffset 333
__builtin_dmapage 333
__builtin_ed 333
__builtin_edac 334
__builtin_edsoffset 335
__builtin_edspage 335
__builtin_enable_interrupts 335
__builtin_fbcl 336
__builtin_get_isr_state 336
__builtin_lac 336
__builtin_mac 337
__builtin_modsd 338
__builtin_modud 338
__builtin_movsac 339
__builtin_mpy 340
__builtin_mpyn 341
__builtin_msc 342
__builtin_mulss 343
__builtin_mulsu 343
__builtin_mulus 344
__builtin_muluu 344
__builtin_nop 344
__builtin_psvoffset 345
__builtin_psvpage 345
__builtin_readsfr 345
__builtin_return_address 346
__builtin_sac 346
__builtin_sacr 346
__builtin_section_begin 347
__builtin_section_end 347
__builtin_section_size 347
__builtin_set_isr_state 347
__builtin_sftac 348
__builtin_software_breakpoint 348
__builtin_subab 348
__builtin_tbladdress 349
__builtin_tbloffset 350
__builtin_tblpage 349
__builtin_tblrdh 350
__builtin_tblrdl 350
__builtin_tblwth 351
__builtin_tblwtl 351
__builtin_write_CRYOTP 330
__builtin_write_NVM 351
__builtin_write_NVM_secure 352
__builtin_write_OSCCONH 354
__builtin_write_OSCCONL 353
__builtin_write_PWMSFR 352
__builtin_write_RTCWEN 353
__deprecate 限定符 39
__eeprom 限定符 35
__far 限定符 30
__interrupt 限定符 36
__near 限定符 31
__pack 限定符 38
__persistent 限定符 32
__section 限定符 39
__XC16_VERSION__ 245
__xdata 限定符 33
__ydata 限定符 33
_Accum 210
_delay 函数 61
_Fract 210
-B 91

数字

0b 二进制基数说明符 145

A

-A ... 117
-ansi ... 104, 119, 207
-aux-info ... 104
abort ... 203, 263
address 属性 ... 152, 198
alias 属性 ... 199
aligned 属性 ... 152
ANSI 89 扩展 ... 136
ANSI C 标准 ... 13, 18
  符合标准 ... 133
  实现定义的行为 ... 133
ANSI C，严格 ... 105
ANSI 标准库支持 ... 14
ASCII 字符 ... 137
  扩展 ... 146
ASCII 字符集 ... 319
asm ... 152, 232
asm C 语句 ... 42
auto_psv 空间 ... 101
按位取反操作符 ... 194
安装，参见编译器安装与激活

B

boot 属性 ... 152, 199
帮助！ ... 45
保护代码 ... 60
编写 ISR 代码 ... 216
编写 ISR 的准则 ... 215
编写 ISR 的语法 ... 215
编写中断服务程序 ... 215
编译
  增量编译 ... 96
编译多个文件 ... 95
编译器 ... 93
  概述 ... 13
  命令行 ... 89
  驱动程序 ... 14, 93
编译器安装与激活 ... 45
编译器生成的代码 ... 65
编译器说明 ... 13
编译器选择 ... 47
变量
  auto ... 170
  存储持续时间 ... 167
  静态 ... 168
  绝对 ... 29
  位置 ... 65
  最大大小 ... 64
变量属性 ... 151
标量 ... 191, 208
标识符 ... 255
  唯一长度 ... 23
标志，肯定和否定 ... 116, 122
标准 I/O 函数 ... 14

C

-C ... 117
-c ... 103, 120
C 标准库 ... 98
C 堆栈使用 ... 171
C 方言控制选项 ... 104
  -ansi ... 104
  -aux-info ... 104
  -ffreestanding ... 104
  -fno-asm ... 104
  -fno-builtin ... 104
  -fno-signed-bitfields ... 104
  -fno-unsigned-bitfields ... 104
  -fsigned-bitfields ... 104
  -fsigned-char ... 104
  -funsigned-bitfields ... 104
  -funsigned-char ... 104
  -menable-fixed ... 104
  -traditional ... 207
char ... 104, 105, 136, 153, 195, 210
char 数据类型 ... 24, 137
COFF ... 93
const 对象
  初始化 ... 147
const 属性 ... 200
const 限定符 ... 147, 177
context 属性 ... 201, 219
CORCON ... 129, 227
C，与汇编语言混合使用 ... 229
参数，参见函数和参数
参数，函数 ... 210
常量
  C 说明符 ... 145
  预定义的 ... 245, 321
  字符 ... 146
  字符串，参见字符串字面值 ... 146
程序存储器指针 ... 191, 208
持久性数据 ... 170, 228
处理器 ID ... 102
传统 C ... 109
存储持续时间 ... 167
存储空间 ... 168
存储模型 ... 14, 191, 208
  -mconst-in-auxflash ... 191, 208
  -mconst-in-code ... 191, 208
  -mconst-in-data ... 191, 208
  -mlarge-code ... 191, 208
  -mlarge-data ... 191, 208
  -msmall-code ... 191, 208
  -msmall-data ... 191, 208
  -msmall-scalar ... 191, 208
存储器 ... 263
  剩余 ... 66
  摘要 ... 66
存储器分配 ... 165
  非自动变量 ... 167
  函数代码 ... 206
  静态变量 ... 168
  数据存储器 ... 167
错误 ... 271
错误控制选项
  -pedantic-errors ... 105
  -Werror ... 108
  -Werror-implicit-function-declaration ... 105
错误消息 ... 67
  位置 ... 67

© 2015 Microchip Technology Inc.

D

-D 117, 118, 119
-dD 117
-dM 117
-dN 117
DWARF 102
deprecated 属性 108, 153, 201
double 123, 138, 195, 210
大代码模型 101
大数据模型 102
大小限制 64
    const 变量 177
代码长度，减少 101, 111
代码生成约定选项 122
    -fargument- noalias-global 122
    -fargument-alias 122
    -fargument-noalias 122
    -fcall-saved 122
    -fcall-used 122
    -ffixed 122
    -fno-ident 122
    -fno-short-double 123
    -fno-verbose-asm 123
    -fpack-struct 122
    -fpcc-struct-return 123
    -fshort-enums 123
    -fverbose-asm 123
低优先级中断 221
调度 113
定义全局寄存器变量 322
读物，推荐 12
段 76, 112
堆栈 219
    C 使用 171
    软件 170, 171
    溢出 68
    指针限制寄存器 （SPLIM） 127, 170, 227
    指针 （W15） 122, 127, 170, 171, 227
对齐方式 152, 155, 210, 258

E

-E 103, 117, 118, 119, 120
EEDATA 170
EEPROM，数据 170
ELF 93, 102
eds 属性 153
errno 263
exit 263
extern 108, 116, 208
二进制常量
    C 代码 145

F

--fill 120
-falign-functions 112
-falign-labels 112
-falign-loops 112
-fargument- noalias-global 122
-fargument-alias 122
-fargument-noalias 122
-fcaller-saves 112
-fcall-saved 122
-fcall-used 122
-fcse-follow-jumps 112
-fcse-skip-blocks 112
-fdata-sections 112
-fdefer-pop。*请参见* -fno-defer
-fexpensive-optimizations 112
-ffixed 122, 323
-ffreestanding 104
-ffunction-sections 112
-fgcse 112
-fgcse-lm 112
-fgcse-sm 113
-finline-functions 108, 111, 116, 207
-finline-limit 116
-finstrument-functions 202
-fkeep-inline-functions 116, 207
-fkeep-static-consts 116
-fno 116, 122
-fno-asm 104
-fno-builtin 104
-fno-defer-pop 113
-fno-function-cse 116
-fno-ident 122
-fno-inline 116
-fno-keep-static-consts 116
-fno-peephole 113
-fno-peephole2 113
-fno-short-double 123
-fno-show-column 117
-fno-signed-bitfields 104
-fno-verbose-asm 123
-fno-unsigned-bitfields 104
-fomit-frame-pointer 111, 116
-foptimize-register-move 113
-foptimize-sibling-calls 116
-fpack-struct 122
-fpcc-struct-return 123
-fregmove 113
-frename-registers 113
-frerun-cse-after-loop 113, 114
-frerun-loop-opt 113
-fschedule-insns 113
-fschedule-insns2 113
-fshort-enums 123
-fsigned-bitfields 104
-fsigned-char 104
-fstrength-reduce 113, 114
-fstrict-aliasing 111, 112, 114
-fsyntax-only 105
-fthread-jumps 111, 114
-fverbose-asm 123
-funroll-all-loops 112, 114
-funroll-loops 111, 112, 114
-funsigned-bitfields 104
-funsigned-char 104
F 常量后缀 146
far 属性 153, 192, 201, 209, 233
far 数据空间 192
fillupper 属性 153

float 123, 138, 153, 195, 210
format_arg 属性 201
format 属性 201
翻译 254
翻译单元 92
返回类型 106
返回值 212
非易失性 RAM 147
浮点类型，复数 144
浮点型 257
浮点型常量后缀 146
浮点型数据类型 138
符号 121
复数 144
    浮点型 144
    数据类型 144
    整型 144
复位 217, 219, 220

G

--gc-sections 120
-g 111
getenv 263
高优先级中断 221
公共子表达式消除 112, 113, 114, 200
归档器 93
过程抽象 102

H

--help 103
--heap 190
-H 117
hex 文件 94
函数
    参数 170, 210
    大小 65
    大小限制 206
    调用约定 210
    调用，保留寄存器 212
    静态 197
    绝对 29
    说明符 197
    位置 65
    指针 143, 191, 208
行内 232
宏 79, 117, 118, 119, 208
宏，数据存储器分配 169
后缀 LL 136
后缀 ULL 136
互联网地址， Microchip 359
环境 254
环境变量
    PIC30_C_INCLUDE_PATH 91
    PIC30_COMPILER_PATH 91
    PIC30_LIBRARY_ PATH 91
    TMPDIR 91
    XC16_C_INCLUDE_PATH 91
    XC16_COMPILER_PATH 91
    XC16_EXEC_PREFIX 91
    XC16_LIBRARY_ PATH 91
汇编代码
    编写 58
    与 C 混合使用 58
汇编列表文件 100
汇编器 93
汇编选项 119
    -Wa 119
汇编源文件 92
汇编，行内 232
汇编，与 C 语言混合使用 229
混合使用汇编语言与 C 变量及函数 229

I

-I 91, 117, 119
-I- 117, 119
-idirafter 118
-imacros 118, 119
-include 118, 119
-iprefix 118
-iwithprefix 118
-iwithprefixbefore 118
-isystem 118
IAR 兼容性 265,270
inline 116, 207
int 136, 153, 195, 210
interrupt 属性 202, 205, 216, 219, 247
ISR
    编写 215
    编写代码 216
    编写语法 215
    编写准则 215

J

激活，参见编译器安装与激活
基数说明符
    C 代码 145
寄存器 322, 323
    定义文件 126
    行为 258
    由函数使用 196
减少代码长度 101, 111
结构 140, 195, 210, 258
    位域 141
结构限定符 140
结构，位域 141
禁止警告 105
禁止中断 60
警告 290
警告和错误控制选项 105
    -fsyntax-only 105
    -pedantic 105
    -pedantic-errors 105
    -W 108
    -w 105
    -Waggregate-return 108
    -Wall 105
    -Wbad-function-cast 108
    -Wcast-align 108
    -Wcast-qual 108
    -Wchar-subscripts 105
    -Wcomment 105
    -Wconversion 108

© 2015 Microchip Technology Inc.

-Wdiv-by-zero 105
-Werror 108
-Werror-implicit-function-declaration 105
-Wformat 105
-Wimplicit 105
-Wimplicit-function-declaration 105
-Wimplicit-int 105
-Winline 108
-Wlarger-than- 108
-Wlong-long 108
-Wmain 105
-Wmissing-braces 105
-Wmissing-declarations 108
-Wmissing-format-attribute 108
-Wmissing-noreturn 108
-Wmissing-prototypes 108
-Wmultichar 105
-Wnested-externs 108
-Wno-long-long 108
-Wno-multichar 105
-Wno-sign-compare 109
-Wpadded 108
-Wparentheses 106
-Wpointer-arith 109
-Wredundant-decls 109
-Wreturn-type 106
-Wsequence-point 106
-Wshadow 109
-Wsign-compare 109
-Wstrict-prototypes 109
-Wswitch 106
-Wsystem-headers 106
-Wtraditional 109
-Wtrigraphs 106
-Wundef 109
-Wuninitialized 106
-Wunknown-pragmas 106
-Wunused 107
-Wunused-function 107
-Wunused-label 107
-Wunused-parameter 107
-Wunused-value 107
-Wunused-variable 107
-Wwrite-strings 109
警告消息 67
禁止 68
位置 67
警告，禁止 105
静态变量 168
静态函数 197
局部寄存器变量 322, 323
绝对变量 29
绝对函数 29

K

keep 属性 202
开发工具 15
看门狗定时器 369
客户支持 359
控制输出类型的选项 103
库 120, 239
ANSI 标准 14
创建 48
更换模块 251
功能 260
用户定义的 98
库管理器 93, 251
窥孔优化 113
扩展名 118
扩展字符集 146

L

-L 120
-l 120
-legacy-libc 120
LL，后缀 136
L 常量后缀 145
lib 目录 98
limits.h 头文件 136, 138
long 136, 153, 195, 210
long double 123, 138, 153, 195, 210
long _Fract 210
long long 108, 136, 153, 195
long long int 136
类型之间的转换 193
类型转换 108, 193
联合 258
限定符 140
联系 Microchip Technology 360
链接描述文件 126, 129
链接器 93, 120
链接描述文件 72
链接选项 120
--fill 120
--gc-sections 120
-L 120
-l 120
-legacy-libc 120
-nodefaultlibs 121
-nostdlib 121
-s 121
-u 121
-Wl 121
-Xlinker 121
临时变量 170
临时文件 263
流 262

M

-M 118
-MD 118
-MF 118
-mauxflash 102
-MG 118
-MM 118
-MMD 118
-MP 118
-MQ 119
-MT 119
-mconst-in-auxflash 101, 191, 208
-mconst-in-code 101, 191, 208
-mconst-in-data 101, 191, 208

-mcpu 102
-menable-fixed 104, 139
-merrata 101
-mfillupper 101
-mlarge-arrays 101
-mlarge-code 101, 191, 208
-mlarge-data 102, 191, 208
-mno-isr-warn 102
-mno-pa 102
-momf= 102
-mpa 102
-mpa= 102
-msmall-code 102, 191, 208, 209
-msmall-data 102, 191, 192, 208
-msmall-scalar 102, 191, 192, 208
-msmart-io 102
-mtext= 102
Mabonga 248, 324
main 函数 21, 227
make 文件 96
mode 属性 153
MPLAB X IDE 69
项目属性选项 48
myMicrochip 个性化通知客户服务 358
枚举 258
命令行编译器 89
命令行模拟器 14, 93
命令行选项 89, 101
模块 92
模拟器，命令行 14, 93
目标文件 76, 93, 112, 118, 120, 239
目录 79, 117, 118, 119
目录搜索选项 122
-B 91
-specs= 122

N

-nodefaultlibs 121
-nostdinc 117, 119
-nostdlib 121
NULL 宏 28
NULL 指针 143
naked 属性 202
near 和 far 代码 209
near 和 far 数据 192, 209
near 数据段 192
Near 数据空间 234
near 属性 154, 192, 202, 209, 233
no_instrument_function 属性 202
noload 属性 154, 202
noreturn 属性 108, 203
内联 108, 111, 116
内联函数 207
内置函数
__builtin_add 327
__builtin_addab 327
__builtin_btg 328
__builtin_clr 328
__builtin_clr_prefect 329
__builtin_dataflashoffset 330
__builtin_disable_interrupts 330
__builtin_disi 331
__builtin_divf 331
__builtin_divmodsd 331
__builtin_divmodud 332
__builtin_divsd 332
__builtin_divud 332
__builtin_dmaoffset 333
__builtin_dmapage 333
__builtin_ed 333
__builtin_edac 334
__builtin_edsoffset 335
__builtin_edspage 335
__builtin_enable_interrupts 335
__builtin_fbcl 336
__builtin_get_isr_state 336
__builtin_lac 336
__builtin_mac 337
__builtin_modsd 338
__builtin_modud 338
__builtin_movsac 339
__builtin_mpy 340
__builtin_mpyn 341
__builtin_msc 342
__builtin_mulss 343
__builtin_mulsu 343
__builtin_mulus 344
__builtin_muluu 344
__builtin_nop 344
__builtin_psvoffset 345
__builtin_psvpage 345
__builtin_readsfr 345
__builtin_return_address 346
__builtin_sac 346
__builtin_sacr 346
__builtin_section_begin 347
__builtin_section_end 347
__builtin_section_size 347
__builtin_set_isr_state 347
__builtin_sftac 348
__builtin_software_breakpoint 348
__builtin_subab 348
__builtin_tbladdress 349
__builtin_tbloffset 350
__builtin_tblpage 349
__builtin_tblrdh 350
__builtin_tblrdl 350
__builtin_tblwth 351
__builtin_tblwtl 351
__builtin_write_CRYOTP 330
__builtin_write_NVM 351
__builtin_write_NVM_secure 352
__builtin_write_OSCCONH 354
__builtin_write_OSCCONL 353
__builtin_write_PWMSFR 352
__builtin_write_RTCWEN 353

O

-O 111
-o 94, 103
-O0 111
-O1 111

© 2015 Microchip Technology Inc.

-O2 111
-O3 111
-Os 111

P

-P 119
--partition 102
-pedantic 105, 108
-pedantic-errors 105
PATH 93
packed 属性 122, 154
persistent 属性 155
pic30-ar 69
pic30-as 69
PIC30_C_INCLUDE_PATH 91
PIC30_COMPILER_PATH 91
pic30-gcc 89
pic30-ld 69
PIC30_LIBRARY_ PATH 91
printf 函数 61
Project Properties 73
    MPLAB XC16 C 编译器选项 76
    MPLAB XC16 汇编器选项 74
    MPLAB XC16 目标归档器选项 84
    MPLAB XC16 目标链接器选项 80
    XC16 工具套件全局选项 73
PSV 窗口 174, 188, 191, 208
PSV 使用 188

Q

-Q 111
启动
    和初始化 98
    模块 171
    模块，备用 98, 228
    模块，主 98, 227
器件支持 65
器件支持文件 125
前缀 118
嵌套中断 219
强制类型转换 107, 108, 194
驱动程序
    输入文件 90
驱动程序选项 48, 90
    CCI 43
    EXT 266
    PRE 243
全局寄存器变量 322

R

RAW 依赖性 113
reverse 属性 155
round 属性 203
软件堆栈 170, 171, 205

S

-S 103, 120
-s 121
-save-temps 111
-specs= 122
save(list) 属性 203
section 属性 155, 203, 206, 247, 324
secure 属性 156, 204
SFR 14, 94, 126, 129
sfr 属性 156
shadow 属性 205, 216, 247
short 136, 195, 210
signed char 136
signed int 136
signed long 136
signed long long 136
signed short 136
space 属性 156, 247
SPLIM 127, 170
strerror 264
struct 类型，参见结构
switch 106
system 263
三字符组合 106, 119
声明符 258
十六进制常量
    C 代码 145
实数 144
实现定义的行为 133, 253
使用行内汇编语言 232
示例 85
输出控制选项
    --help 103
    -c 103
    -E 103
    -o 103
    -S 103
    -v 103
    -x 103
输出文件
    名称 99
输入文件 90
书写中断向量 217
属性 14, 151, 198, 247
属性，变量 151, 152
    aligned 152
    boot 152
    deprecated 153
    eds 153
    far 153, 192, 209
    fillupper 153
    near 154, 192, 209
    noload 154
    packed 154
    persistent 155
    reverse 155
    section 155
    secure 156
    sfr 156
    space 156
    transparent_union 158
    unordered 158
    unsupported 158
    unused 158
    weak 158
    模式 153

属性，函数
address 198
alias 199
boot 199
const 200
context 201
deprecated 201
far 201
format 201
format_arg 201
interrupt 202, 216, 219
keep 202
naked 202
near 202
no_instrument_function 202
noload 202
noreturn 108, 203
round 203
save(list) 203
section 203, 206, 324
secure 204
shadow 205, 216
unsupported 205
unused 205
user_init 205
weak 205
数据存储空间 101, 102, 190
数据存储空间， near 154
数据存储器 167
数据存储器分配 169
数据类型 153
大小 23
复数 144
数据，压缩 186
数组
初始化 146
数组和指针 257

T

-T 126
-traditional 104, 207
-trigraphs 119
TBLRD 189
TMPDIR 91
transparent_union 属性 158
特定于 16 位的选项 101
特定于 dsPIC 的选项
-mauxflash 102
-mconst-in-auxflash 101
-mconst-in-code 101
-mconst-in-data 101
-mcpu 102
-merrata 101
-mfillupper 101
-mlarge-arrays 101
-mlarge-code 101
-mlarge-data 102
-mno-isr-warn 102
-mno-pa 102
-momf= 102
-mpa 102
-mpa= 102
-msmall-code 102
-msmall-data 102
-msmall-scalar 102
-msmart-io 102
-mtext= 102
--partition 102
特殊功能寄存器 94, 219
特殊功能寄存器，参见 SFR
替换库模块 251
调试信息 111
调试选项 111
-g 111
-Q 111
-save-temps 111
通用 C 接口，见 CCI
通用寄存器 232
头文件 21, 79, 91, 92, 117, 118, 119, 239
搜索路径 22

U

-U 117, 118, 119
-u 121
-undef 119
USB 377
unsigned char 136
unsigned int 136
unsigned long 136
unsigned long long 136
unsigned long long int 136
unsigned short 136
unsupported 属性 158, 205
unused 属性 107, 158, 205
user_init 属性 205
ULL，后缀 136
unordered 属性 158
U 常量后缀 145

V

-v 103
void 195
volatile 限定符 60, 147

W

-W 105, 107, 108, 109, 271
-w 105
-Wa 119
-Waggregate-return 108
-Wall 105, 106, 108, 109
-Wbad-function-cast 108
-Wcast-align 108
-Wcast-qual 108
-Wchar-subscripts 105
-Wcomment 105
-Wconversion 108
-Wdiv-by-zero 105
-Werror 108
-Werror-implicit-function-declaration 105
-Wformat 105, 108, 201
-Wimplicit 105
-Wimplicit-function-declaration 105

© 2015 Microchip Technology Inc.

-Wimplicit-int 105
-Winline 108, 207
-Wl 121
-Wlarger-than- 108
-Wlong-long 108
-Wmain 105
-Wmissing-braces 105
-Wmissing-declarations 108
-Wmissing-format-attribute 108
-Wmissing-noreturn 108
-Wmissing-prototypes 108
-Wmultichar 105
-Wnested-externs 108
-Wno- 105
-Wno-deprecated-declarations 108
-Wno-div-by-zero 105
-Wno-long-long 108
-Wno-multichar 105
-Wno-sign-compare 109, 110
-Wpadded 108
-Wparentheses 106
-Wpointer-arith 109
-Wredundant-decls 109
-Wreturn-type 106
-Wsequence-point 106
-Wshadow 109
-Wsign-compare 109
-Wstrict-prototypes 109
-Wswitch 106
-Wsystem-headers 106
-Wtraditional 109
-Wtrigraphs 106
-Wundef 109
-Wuninitialized 106
-Wunknown-pragmas 106
-Wunused 107
-Wunused-function 107
-Wunused-label 107
-Wunused-parameter 107
-Wunused-value 107
-Wunused-variable 107
-Wwrite-strings 109
W14 171
W15 171
Wall 110
Wunused 110
W 寄存器 210, 229
weak 属性 158, 205
外部符号 229
网站， Microchip 359
为局部变量指定寄存器 323
未命名结构成员 141
未命名位域 141
未用变量 107
    删除 147
未用存储器 66
未用函数参数 107
位域 26, 27, 104, 141, 223, 258
文档 13
    编排 9
    约定 11
文件 262
文件扩展名 92
文件类型
    输入 90
文件命名约定 90

X

-x 103
-Xlinker 121
XC16_C_INCLUDE_PATH 91
XC16_COMPILER_PATH 91
XC16_EXEC_PREFIX 91
XC16_LIBRARY_ PATH 91
XC16_VERSION 245
系统头文件 106, 118
下划线 215, 229
限定符 147-150, 258
    __align 34
    __bank 33
    __deprecate 39
    __eeprom 35
    __far 30
    __interrupt 36
    __near 31
    __pack 38
    __persistent 32
    __section 39
    __xdata 33
    __ydata 33
    auto 170
    const 147, 177
    volatile 60, 147
    和结构 140
    和自动变量 170
项目 71, 96
项目名称 99
响应延时 219
消息
    含义 67
小代码模型 14, 102
小数据模型 14, 102
信号 261
虚数 144
选项
    C 方言控制 104
    代码生成约定 122
    汇编 119
    警告和错误控制 105
    链接 120
    目录搜索 122
    特定于 16 位 101
    调试 111
    优化控制 111
    预处理器控制 117
循环移位操作符 61
循环优化 200
循环优化器 113
循环展开 77, 114

Y

压缩存储在闪存中的数据 186
延时程序 61
映射文件 100
用户定义的数据段 324
用户定义的 text 段 206, 324
优化 14
  导致损坏 60
  更快的代码 63
优化控制选项 111
  -falign-functions 112
  -falign-labels 112
  -falign-loops 112
  -fcaller-saves 112
  -fcse-follow-jumps 112
  -fcse-skip-blocks 112
  -fdata-sections 112
  -fexpensive-optimizations 112
  -ffunction-sections 112
  -fgcse 112
  -fgcse-lm 112
  -fgcse-sm 113
  -finline-functions 116
  -finline-limit 116
  -fkeep-inline-functions 116
  -fkeep-static-consts 116
  -fno-defer-pop 113
  -fno-function-cse 116
  -fno-inline 116
  -fno-peephole 113
  -fno-peephole2 113
  -fomit-frame-pointer 116
  -foptimize-register-move 113
  -foptimize-sibling-calls 116
  -fregmove 113
  -frename-registers 113
  -frerun-cse-after-loop 113
  -frerun-loop-opt 113
  -fschedule-insns 113
  -fschedule-insns2 113
  -fstrength-reduce 113
  -fstrict-aliasing 114
  -fthread-jumps 114
  -funroll-all-loops 114
  -funroll-loops 114
  -O 111
  -O0 111
  -O1 111
  -O2 111
  -O3 111
  -Os 111
优化，窥孔 113
优化，循环 113, 200
语法检查 105
语句 259
预处理 243
预处理器宏
  预定义 41
预处理器控制选项 117
  -A 117
  -C 117
  -D 117
  -dD 117
  -dM 117
  -dN 117
  -fno-show-column 117
  -H 117
  -I 117
  -I- 117
  -idirafter 118
  -imacros 118
  -include 118
  -iprefix 118
  -isystem 118
  -iwithprefix 118
  -iwithprefixbefore 118
  -M 118
  -MD 118
  -MF 118
  -MG 118
  -MM 118
  -MMD 118
  -MQ 119
  -MT 119
  -nostdinc 119
  -P 119
  -trigraphs 119
  -U 119
  -undef 119
预处理伪指令 259
预定义的常量 245, 321
源代码 72
源文件 92
原子操作 221
允许 / 禁止中断 220
运行时启动代码 227

Z

在函数调用之间保留寄存器 212
增量编译 96
展开循环 77, 114
帧指针 （W14） 78, 116, 122, 171
诊断 271
诊断文件 100
整型 232
  类型，复数 144
  行为 256
整型常量 145
整型后缀 145
整型提升 193
指定寄存器中的变量 322
只读变量 147
指针 109, 142-143, 195, 210
  比较 143
  定义 142
  堆栈 122
  赋予整型 143
  函数 143, 191, 208
  类型 142
  限定符 142
  帧 78, 116, 122

© 2015 Microchip Technology Inc.

中断
保护 223
处理 229
低优先级 221
服务程序现场保存 219
高优先级 221
功能 229
禁止 60
嵌套 219
响应延时 219
向量，书写 217
优先级 219
允许 / 禁止 220
中断向量表 218
中间文件 90
主干代码 214
注释 105, 117
转义序列 255
自动变量 106, 167, 170, 171
存储器分配 170-173
字符 255
字符常量
在 C 中 146
字符串字面值 146
存储单元 146
类型 146

注：

© 2015 Microchip Technology Inc.

全球销售及服务网点

美洲

公司总部 Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 1-480-792-7200
Fax: 1-480-792-7277
技术支持：
http://www.microchip.com/support
网址：www.microchip.com

亚特兰大 Atlanta
Duluth, GA
Tel: 1-678-957-9614
Fax: 1-678-957-1455

奥斯汀 Austin, TX
Tel: 1-512-257-3370

波士顿 Boston
Westborough, MA
Tel: 1-774-760-0087
Fax: 1-774-760-0088

芝加哥 Chicago
Itasca, IL
Tel: 1-630-285-0071
Fax: 1-630-285-0075

克里夫兰 Cleveland
Independence, OH
Tel: 1-216-447-0464
Fax: 1-216-447-0643

达拉斯 Dallas
Addison, TX
Tel: 1-972-818-7423
Fax: 1-972-818-2924

底特律 Detroit
Novi, MI
Tel: 1-248-848-4000

休斯敦 Houston, TX
Tel: 1-281-894-5983

印第安纳波利斯 Indianapolis
Noblesville, IN
Tel: 1-317-773-8323
Fax: 1-317-773-5453

洛杉矶 Los Angeles
Mission Viejo, CA
Tel: 1-949-462-9523
Fax: 1-949-462-9608

纽约 New York, NY
Tel: 1-631-435-6000

圣何塞 San Jose, CA
Tel: 1-408-735-9110

加拿大多伦多 Toronto
Tel: 1-905-673-0699
Fax: 1-905-673-6509

亚太地区

亚太总部 Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2943-5100
Fax: 852-2401-3431

中国 - 北京
Tel: 86-10-8569-7000
Fax: 86-10-8528-2104

中国 - 成都
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

中国 - 重庆
Tel: 86-23-8980-9588
Fax: 86-23-8980-9500

中国 - 东莞
Tel: 86-769-8702-9880

中国 - 杭州
Tel: 86-571-8792-8115
Fax: 86-571-8792-8116

中国 - 香港特别行政区
Tel: 852-2943-5100
Fax: 852-2401-3431

中国 - 南京
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

中国 - 青岛
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

中国 - 上海
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

中国 - 沈阳
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

中国 - 深圳
Tel: 86-755-8864-2200
Fax: 86-755-8203-1760

中国 - 武汉
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

中国 - 西安
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

中国 - 厦门
Tel: 86-592-238-8138
Fax: 86-592-238-8130

中国 - 珠海
Tel: 86-756-321-0040
Fax: 86-756-321-0049

亚太地区

台湾地区 - 高雄
Tel: 886-7-213-7828

台湾地区 - 台北
Tel: 886-2-2508-8600
Fax: 886-2-2508-0102

台湾地区 - 新竹
Tel: 886-3-5778-366
Fax: 886-3-5770-955

澳大利亚 Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

印度 India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4123

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

印度 India - Pune
Tel: 91-20-3019-1500

日本 Japan - Osaka
Tel: 81-6-6152-7160
Fax: 81-6-6152-9310

日本 Japan - Tokyo
Tel: 81-3-6880-3770
Fax: 81-3-6880-3771

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

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

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

马来西亚 Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

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

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

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

欧洲

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

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

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

德国 Germany - Dusseldorf
Tel: 49-2129-3766400

德国 Germany - Karlsruhe
Tel: 49-721-625370

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

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

意大利 Italy - Venice
Tel: 39-049-7625286

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

波兰 Poland - Warsaw
Tel: 48-22-3325737

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

瑞典 Sweden - Stockholm
Tel: 46-8-5090-4654

英国 UK - Wokingham
Tel: 44-118-921-5800
Fax: 44-118-921-5820

07/14/15

© 2015 Microchip Technology Inc.