

MPLAB® XC8 C 编译器
用户指南

© 2013 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。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、 KEELOQ、 KEELOQ 徽标、 MPLAB、 PIC、 PICmicro、 PICSTART、 PIC32 徽标、 rfPIC、 SST、 SST 徽标、 SuperFlash 和 UNI/O 均为 Microchip Technology Inc. 在美国和其他国家或地区的注册商标。

FilterLab、 Hampshire、 HI-TECH C、 Linear Active Thermistor、 MTP、 SEEVAL 和 The Embedded Control Solutions Company 均为 Microchip Technology Inc. 在美国的注册商标。

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

Analog-for-the-Digital Age、 Application Maestro、 BodyCom、 chipKIT、 chipKIT 徽标、 CodeGuard、 dsPICDEM、 dsPICDEM.net、 dsPICworks、 dsSPEAK、 ECAN、 ECONOMONITOR、 FanSense、 HI-TIDE、 In-Circuit Serial Programming、 ICSP、 Mindi、 MiWi、 MPASM、 MPF、 MPLAB Certified 徽标、 MPLIB、 MPLINK、 mTouch、 Omniscient Code Generation、 PICC、 PICC-18、 PICDEM、 PICDEM.net、 PICkit、 PICtail、 REAL ICE、 rfLAB、 Select Mode、 SQI、 Serial Quad I/O、 Total Endurance、 TSHARC、 UniWinDriver、 WiperLock、 ZENA 和 Z-Scale 均为 Microchip Technology Inc. 在美国和其他国家或地区的商标。

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

GestIC 和 ULPP 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。

© 2013, Microchip Technology Inc. 版权所有。

ISBN：978-1-62076-656-9

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。

© 2013 Microchip Technology Inc.

目录

前言 **7**

第 1 章 编译器概述

1.1 简介 11
1.2 编译器说明和文档 11
1.3 器件说明 12

第 2 章 通用 C 接口

2.1 简介 13
2.2 背景知识——对于可移植代码的追求 13
2.3 使用 CCI 16
2.4 ANSI 标准细化 17
2.5 ANSI 标准扩展 25
2.6 编译器特性 39

第 3 章 操作指南

3.1 简介 41
3.2 安装和激活编译器 41
3.3 调用编译器 43
3.4 编写源代码 46
3.5 让应用程序执行所需的操作 56
3.6 了解编译过程 60
3.7 修复无法工作的代码 67

第 4 章 XC8 命令行驱动程序

4.1 简介 71
4.2 调用编译器 72
4.3 编译序列 75
4.4 运行时文件 81
4.5 编译器输出 84
4.6 编译器消息 86
4.7 XC8 驱动程序选项 91
4.8 选项说明 92
4.9 MPLAB IDE V8 Universal Toolsuite 等效选项 117
4.10 MPLAB X Universal Toolsuite 等效选项 124

第 5 章 C 语言特性

5.1 简介 131
5.2 ANSI C 标准问题 131
5.3 与器件相关的特性 133
5.4 支持的数据类型和变量 143
5.5 存储器分配和访问 165

5.6 操作符和语句 179
5.7 寄存器使用 181
5.8 函数 182
5.9 中断 189
5.10 main、运行时启动和复位 194
5.11 库程序 198
5.12 混合使用 C 代码和汇编代码 200
5.13 优化 208
5.14 预处理 210
5.15 链接程序 222

第 6 章 宏汇编器

6.1 简介 241
6.2 汇编器用法 241
6.3 选项 242
6.4 MPLAB XC8 汇编语言 246
6.5 汇编代码级优化 268
6.6 汇编列表文件 269

第 7 章 链接器

7.1 简介 277
7.2 操作 277
7.3 重定位和 psect 285
7.4 映射文件 286

第 8 章 实用程序

8.1 简介 291
8.2 库管理器 291
8.3 OBJTOHEX 295
8.4 CREF 297
8.5 CROMWELL 300
8.6 HEXMATE 303

附录 A 库函数

附录 B 错误和警告消息

附录 C 实现定义的行为

C.1 翻译 （G.3.1） 479
C.2 环境 （G.3.2） 479
C.3 标识符 （G.3.3） 480
C.4 字符 （G.3.4） 480
C.5 整型 （G.3.5） 481
C.6 浮点型 （G.3.6） 482
C.7 数组和指针 （G.3.7） 482
C.8 寄存器 （G.3.8） 482
C.9 结构、联合、枚举和位域 （G.3.9） 483
C.10 限定符 （G.3.10） 483
C.11 声明符 （G.3.11） 483

© 2013 Microchip Technology Inc.

C.12 语句 （G.3.12） 483
C.13 预处理伪指令 （G.3.13） 484
C.14 库函数 （G.3.14） 485
术语表 487
索引 507
全球销售及服务网点 518

注：

© 2013 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。请访问我们的网站（www.microchip.com）获取最新文档。

文档均标记有 “DS”编号。该编号出现在每页底部的页码之前。 DS 编号的命名约定为 “DSXXXXXA_CN”，其中 “XXXXX”为文档编号，“A”为文档版本。

欲了解开发工具的最新信息，请参考 MPLAB® IDE 在线帮助。从 Help （帮助）菜单选择 Topics （主题），打开现有在线帮助文件列表。

简介

本章包含使用 MPLAB® XC8 C 编译器前需要了解的一般信息。本章讨论的内容包括：

- 文档编排
- 本指南使用的约定
- 保修登记
- 推荐读物
- Microchip 网站
- 开发系统变更通知客户服务
- 客户支持
- 文档版本历史

文档编排

本文档介绍了如何使用 MPLAB XC8 C 编译器。本手册的内容编排如下：

- **第 1 章 编译器概述**
- **第 2 章 通用 C 接口**
- **第 3 章 操作指南**
- **第 4 章 XC8 命令行驱动程序**
- **第 5 章 C 语言特性**
- **第 6 章 宏汇编器**
- **第 7 章 链接器**
- **第 8 章 实用程序**
- **附录 A 库函数**
- **附录 B 错误和警告消息**
- **附录 C 实现定义的行为**
- **术语表**
- **索引**

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}`
省略号 ...	代替重复文字	`var_name [, var_name...]`
	表示由用户提供的代码	`void main (void) { ... }`

保修登记

请填写随附的保修登记卡 （Warranty Registration Card）并尽快寄出。寄出保修登记卡的用户将可收到新产品更新信息。可在 Microchip 网站上获得临时软件版本。

推荐读物

本用户指南介绍了如何使用 MPLAB XC8 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。

© 2013 Microchip Technology Inc.

MPLAB XC8 C 编译器的自述文件 （Readme）

有关使用 MPLAB XC8 C 编译器的最新信息，请阅读软件附带的相关 Readme 文件（HTML 文件）。自述文件包含了本用户指南中可能未提供的更新信息和已知问题。

自述文件

有关使用其他工具的最新信息，请阅读相应工具的自述文件，自述文件位于MPLAB IDE 安装目录的 Readmes 子目录下。自述文件包含了本用户指南中可能未提供的更新信息和已知问题。

MICROCHIP 网站

Microchip 网站 （www.microchip.com）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

开发系统变更通知客户服务

Microchip 的客户通知服务有助于客户了解 Microchip 产品的最新信息。注册客户可在他们感兴趣的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。

欲注册，请登录 Microchip 网站 www.microchip.com。在“支持”（Support）下，点击“变更通知客户”（Customer Change Notification）服务后按照注册说明完成注册。

开发系统产品的分类如下：

- **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™ 和 MPLAB ICE 2000 在线仿真器。
- **在线调试器**——Microchip 在线调试器的最新信息，包括 MPLAB ICD 3 在线调试器和 PICkit™ 3 debug express。
- **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 编辑器和 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 MPLAB REAL ICE 在线仿真器、MPLAB ICD 3 在线调试器和 MPLAB PM3 器件编程器等生产编程器，以及 PICSTART® Plus、 PICkit 2 和 PICkit 3 等开发编程器。

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。

也可通过 http://microchip.com/support 获得网上技术支持。

文档版本历史

版本 B （2012 年 7 月）

- 增加了 “操作指南”一章。
- 扩展了与 PIC18 勘误表相关的章节。
- 更新了与编译器优化设置相关的章节。
- 更新了 MPLAB v8 和 MPLAB X IDE 项目选项对话框。
- 增加了介绍 PIC18 far 限定符和 inline 函数限定符的章节。
- 扩展了介绍 main() 函数操作的章节。
- 扩展了关于低档器件的等效汇编符号的信息。
- 更新了预定义宏符号表。
- 增加了关于 `#pragma addrqual` 的章节。
- 增加了与内联函数有关的章节。
- 更新了与列表文件中的调用图关联的表和文本。
- 更新了库函数章节，使之与封装库一致。
- 增加了新的编译器警告和错误。
- 新增了介绍通用编译器接口 （Common Compiler Interface， CCI）标准的一章。

版本 A （2012 年 2 月）

本文档的初始版本。

© 2013 Microchip Technology Inc.

第 1 章 编译器概述

1.1 简介

本章对 MPLAB XC8 C 编译器进行概述，包含以下主题。

- 编译器说明和文档
- 器件说明

1.2 编译器说明和文档

MPLAB® XC8 C 编译器是一种独立式的优化 ANSI C 编译器。它支持所有 8 位 PIC® 单片机：PIC10、 PIC12、 PIC16 和 PIC18 系列器件，以及 PIC14000 器件。

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。

该编译器提供了 3 种工作模式：免费 （Free）、标准 （Standard）或专业 （PRO）。标准和专业工作模式是许可模式，需要使用一个序列号来启用它们。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。

1.2.1 约定

在本手册中，将使用 “编译器”这一术语。它可以指代包含 MPLAB XC8 C 编译器的应用程序集合的全集或子集。在指明操作由哪个应用程序执行无关紧要时，将归之于编译器。

类似地，“编译器”也常用于指代命令行驱动程序。虽然具体地说， MPLAB XC8 C 编译器软件包的驱动程序名为 xc8。**第 4.7 节 “XC8 驱动程序选项”**介绍了驱动程序及其选项。相应地，“编译器选项”通常与命令行驱动程序选项相关。

在类似方式下，“编译”指代将源代码生成为可执行二进制镜像中涉及的全部或部分步骤。

1.3 器件说明

该编译器支持采用低档、中档、增强型中档和 PIC18 内核的 8 位 Microchip PIC 器件。以下描述说明了这些器件内核的区别：

低档内核使用 12 位宽的指令集，并在 PIC10、 PIC12 和 PIC16 部件编号的器件上提供。

中档内核使用 14 位宽的指令集，并包含比低档内核更多的指令。它还具有更大的数据存储器存储区和程序存储器页。它在 PIC12、 PIC14 和 PIC16 部件编号的器件上提供。

增强型中档内核也使用 14 位宽的指令集，但包含了更多的指令和功能。PIC12 和 PIC16 部件编号中都有基于增强型中档内核的器件。

PIC18 内核指令集为 16 位宽，具有更多的指令和扩展的寄存器集。 PIC18 内核器件的部件编号以 PIC18 开头。

只要可能，编译器都会利用目标器件的指令集、寻址模式存储器和寄存器。

关于编译器支持的器件的完整列表的信息，请参见**第 4.8.21 节 “--CHIPINFO：显示支持器件的列表”**。

© 2013 Microchip Technology Inc.

第 2 章 通用 C 接口

2.1 简介

通用 C 接口 （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 XC8 C 编译器用户指南》的这一章讨论以下主题：

- 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

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，但如果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进行编译，但使用不同的编译器。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。

您可能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针对新器件和改进代码生成，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*、工具集和用于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距巨大的结果，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

这是标准未作任何要求的行为。

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 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. 典型的例子：中档 PIC® 单片机没有数据堆栈。由于面向该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。

© 2013 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上的实现定义的行为，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 是您选择追随并付诸实施的标准，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。当您使用非 CCI 特性并使能 CCI 时，编译器可能会发出警告或错误来进行提示。

© 2013 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。确认原先针对 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

在所有编译器的原先版本中，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 等效的命令行选项中。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 `< >` 中指定的任何头文件，搜索路径应为通过 `-I` 选项（或等效的 MPLAB IDE 选项）指定的路径，然后是标准的编译器包含目录。`-I` 选项将按指定它们的顺序进行搜索。

对于在引号字符 `" "` 中指定的任何文件，第一个搜索路径应为当前工作目录。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。如果它位于其他位置，则不符合 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所有其他选项，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。

© 2013 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 下会被视为惟一。

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

2.4.5.2 差别

原先的 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作出任何限制。

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

这与编译器原先实现的类型一致。

2.4.6.3 移植到 CCI

如果需要具有固定大小的 C 类型，无论目标器件如何，可以使用由 `<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示对象的符号。

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 XC16/32 上，可以使用 -funsigned-char 选项来更改普通 char 的类型，但由于 XC8 上不支持该选项，因此该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。

© 2013 Microchip Technology Inc.

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

以下代码显示了将发生截断的赋值操作。

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

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

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` 赋值之后的值将为 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按位与运算的示例。

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

在 CCI 下， `output` 赋值之后的值将为 -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进行访问时的成员转换

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` 的位域的类型将与使用 `unsigned int` 定义的相同。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;
};
```


© 2013 Microchip Technology Inc.

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。

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：

```
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

位域是否可以跨越存储单元边界在标准中属于实现定义的行为。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 XC8 时，存储分配单元为字节大小。位域 second 将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 的大小将为 2 个字节。

2.4.14.2 差别

这种分配方式与所有原先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，所定义的第一个位将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 将被分配到最低有效位。位域 mid 将被分配到接下来的低 6 位， hi 将被分配到同一存储单元字节的最高有效位。

2.4.15.2 差别

这与所有编译器的原先操作是相同的。

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 位编译器原先使用表达式 ((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，其大小将至少为 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。

© 2013 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 位编译器的原先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

将原先在 8 位编译器代码中使用的以下头文件：

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

或如以下示例所示的特定于器件系列的头文件：

```
#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() 构造限定符放置在绝对地址处。请注意，为进行绝对寻址， XC16/32 可能要求将变量或函数放置在某个特殊段中。基于堆栈的 （auto 和参数）变量无法使用 __at() 说明符。

2.5.2.1 示例

以下显示了设为绝对寻址的两个变量和一个函数。

```
int scanMode __at(0x200);
const char keys[] __at(123) = { 'r', 's', 'u', 'd'};

int modify(int x) __at(0x1000) {
   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，避免将对象和函数设为绝对寻址。

在 XC8 中，将如下所示的绝对对象定义：

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

更改为：

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

在 XC16/32 中，将如下所示的代码：

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

更改为：

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

2.5.2.4 注意事项

在使用 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可位于“far 存储器”中。究竟哪些存储器构成 far 存储器取决于目标器件，但它通常是需要较复杂代码来进行访问的存储器。如果表达式涉及的是使用 far 限定符的对象，表达式可能会生成较慢、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器，在这种情况下，使用该限定符将被忽略。基于堆栈的（`auto` 和参数）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。

© 2013 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，如以下示例：

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

更改为 __far，即 __far char template[20];

在 16 位和 32 位编译器中，更改每一处 far 属性，如以下示例：

```
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可能位于 “near 存储器”中。究竟哪些存储器构成 near 存储器取决于目标器件，但它通常是可以使用较简单代码来进行访问的存储器。如果表达式涉及的是使用 near 限定符的对象，表达式可能会生成较快、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器，在这种情况下，使用该限定符将被忽略。基于堆栈的（auto 和参数）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。

© 2013 Microchip Technology Inc.

2.5.4.3 移植到 CCI

对于 8 位编译器，更改每一处 near 限定符，如以下示例：

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

更改为 __near，即 __near char template[20];

在 16 位和 32 位编译器中，更改每一处 near 属性，如以下示例：

```
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 位编译器使用了 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了 persistent 属性，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，如以下示例：

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

更改为 __persistent，即 __persistent char template[20];

对于 16 位和 32 位编译器，更改每一处 persistent 属性，如以下示例：

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

更改为

```
int __persistent tblIdx;
```

2.5.5.4 注意事项

无。

© 2013 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可能位于特殊存储区。究竟哪些存储器构成 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器。此类存储器通常需要一些 DSP 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这种存储器；在这种情况下，使用这些限定符将被忽略。

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，从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，如以下示例：

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

更改为 __xdata 或 __ydata，即 __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可能位于某个特定的数据存储区。数字 num 代表存储区编号。究竟哪些存储器构成分区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，使用该限定符将被忽略。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 （尽管更有限）的存储器位置。

对于所有其他编译器，从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，如以下示例：

```
bank2 int logEntry;
```

更改为 `__bank()`，即 `__bank(2) int logEntry;`

2.5.7.4 注意事项

无。

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 位编译器上从未实现对齐功能。

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，如以下示例：

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

更改为 `__align`，即 `__align(4) char mode;`

2.5.8.4 注意事项

XC8 上尚未实现该功能。

© 2013 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将会生成警告。基于堆栈的（auto 和参数）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，如以下示例：

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

更改为 __eeprom，即 __eeprom char title[20];

对于 16 位编译器，更改每一处 eedata space 属性，如以下示例：

```
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，XC8 均未实现 __eeprom 限定符；对于其他 8 位器件，该限定符将按预期工作。

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 （XC8——仅限 PIC18）

high_priority

中断函数对应于高优先级中断源 （XC8）

save(*symbol-list*)

在进入时保存并在退出时恢复所列的符号 （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将会生成警告。如果 `__interrupt` 说明符的参数对于目标器件没有意义，编译器将发出警告或错误。

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;
   }
}
```

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，如以下示例：

```
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，如以下示例：

```
void __attribute__((interrupt,auto_psv,(irq(52)))) myIsr(void);
```


© 2013 Microchip Technology Inc.

更改为

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

对于 32 位编译器， __interrupt() 关键字可接受两个参数，即向量编号和 （可选） IPL 值。更改使用 interrupt 属性的代码，类似于以下示例：

```
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对齐空隙来填充结构，对此类器件使用该说明符将被忽略。

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

__pack 说明符是一个全新的 CCI 说明符，可用于 XC8。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 位编译器使用了 packed 属性来指示结构成员不与存储器空隙对齐。

2.5.11.3 移植到 CCI

对于 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，如以下示例：

```
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。

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 位编译器使用了 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，如以下示例：

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

更改为：

```
int __deprecate intMask;
```

2.5.12.4 注意事项

无。

© 2013 Microchip Technology Inc.

2.5.13 将对象分配到各段

__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 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功能解决的时候使用。

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 位编译器使用了 #pragma psect 伪指令来将对象重定向到一个新的段或 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用于分配该对象的 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作出任何假定。使用该 pragma 伪指令重定向到新 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() 分配不同 psect 的对象将通过运行时启动代码清零。

在当前的 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 位编译器使用了 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 XC8，更改每一处 #pragma psect 伪指令，例如

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

更改为 __section() 说明符，如下所示

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

对于 16 位和 32 位编译器，更改每一处 section 属性，如以下示例：

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

更改为：

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

2.5.13.4 注意事项

对于 XC8， __section() 说明符不能用于任何中断函数。

2.5.14 指定配置位

#pragma config 伪指令可用于编程器件的配置位。 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。

2.5.14.1 示例

以下显示了使用该 pragma 伪指令指定的配置位的示例。

```
#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 位编译器使用了 __CONFIG() 宏。

16 位编译器使用了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，例如

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

更改为 #pragma config 伪指令，如下所示

```
#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，如以下示例：

```
_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。这些宏可以用于基于编译器或目标器件而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__

© 2013 Microchip Technology Inc.

表 2-1：　　由 CCI 定义的列表宏

名称	定义时的含义	示例
__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__），其他一些与原先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 MOVLW 指令的示例。

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

2.5.16.2 差别

8 位编译器使用了 asm() 或 #asm ... #endasm 构造来插入行内汇编代码。

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，使该 #asm 块中的每条指令都放入它自己的 asm() 语句中，例如：

```
#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。

© 2013 Microchip Technology Inc.

2.6 编译器特性

以下条目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 Compiler （编译器）类别中的 *Use CCI Syntax*（使用 CCI 语法）。 MPLAB IDE v8 的 Compiler 选项卡下提供了具有相同名称的窗口小部件。

如果不使用该 IDE，则命令行选项为 `--CCI`（对于 XC8）或 `-mcci`（对于 XC16/32）。

2.6.1.1 差别

该选项原先从未实现过。

2.6.1.2 移植到 CCI

使能该选项。

2.6.1.3 注意事项

无。

注：

© 2013 Microchip Technology Inc.

第 3 章 操作指南

3.1 简介

本章包含关于在为 Microchip 8 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。一些主题在多个章节中具有索引。

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。www.microchip.com 上提供了指南 *Installing and Licensing MPLAB XC C Compilers*（DS52059）。它详细介绍了单用户和网络许可证，以及如何为评估目的而激活编译器。

3.2.2 如何确定编译器是否已成功激活？

如果您认为编译器可能未正确安装或无法正常工作，最好在 MPLAB IDE 之外验证其操作，以隔离可能出现的问题。请尝试从命令行上运行编译器，以检查操作是否正确。您不需要实际编译代码。

从终端或 DOS 提示符下，带选项 `--VER` 运行编译器驱动程序 xc8（见**第 4.2 节“调用编译器”**）。该选项指示编译器打印版本信息并退出。例如，在 Windows 中输入以下命令行，使用与安装相关的路径替换路径信息。

```
"C:\Program Files\Microchip\xc8\v1.00\bin\xc8" --ver
```

编译器应运行，打印信息栏并退出。该信息栏会指示其工作模式。确认工作模式是您请求的模式。请注意：如果未正确地激活，编译器将继续工作，但仅在免费（Free）模式下工作。如果显示一条错误，或编译器指示免费模式，则说明激活未成功。

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安装同一编译器的多个版本，并且可以通过在 MPLAB IDE 中更改选项在版本之间轻松地进行切换，请参见**第 3.3.4 节 “如何选择要用于编译项目的编译器？”**。

编译器应安装到名称与编译器版本相关的目录中。这将反映在安装程序指定的默认目录中。例如， 1.00 和 1.10 XC8 编译器通常放置在单独的目录中。

```
C:\Program Files\Microchip\xc8\v1.00\
C:\Program Files\Microchip\xc8\v1.10\
```


© 2013 Microchip Technology Inc.

3.3 调用编译器

本节讨论如何在命令行上和从 MPLAB IDE 中运行编译器。它从选项和编译过程本身的角度介绍如何让编译器执行所需的操作。

- *如何从 MPLAB X IDE 中进行编译？*
- *如何在命令行上进行编译？*
- *如何使用 make 实用程序进行编译？*
- *如何选择要用于编译项目的编译器？*
- *如何更改编译器的工作模式？*
- *在编译为使用调试器时需要做些什么？*
- *如何编译库？*
- *如何在项目中使用库文件？*
- *如何确定哪些编译器选项可用，以及它们的功能？*
- *如何确定 MPLAB IDE 中的编译选项的功能？*
- *MPLAB IDE 调试编译有何差别？*
- *如何阻止编译器使用某些特定的存储单元？*
- *编译器采用了哪些优化？*

3.3.1 如何从 MPLAB X IDE 中进行编译？

关于如何设置项目的信息，请参见 MPLAB X IDE 随附的文档。

如果已安装了一个或多个 XC8 编译器，请在 Project Properties （项目属性）对话框的 Configuration（配置）类别中选择要使用的编译器。然后，对应于该编译器的选项将显示在 XC8 Compiler （XC8 编译器）和 XC8 Linker （XC8 链接器）类别中。可以注意到其中的每个编译器类别都具有几个 Option （选项）类别。

3.3.2 如何在命令行上进行编译？

对于所有 8 位 PIC 器件，编译器驱动程序均名为 `xc8`；例如，在 Windows 中，它名为 `xc8.exe`。应调用该应用程序来完成编译的所有方面。它位于编译器分发版的 bin 目录中。请避免显式地运行各个编译器应用程序（如汇编器或链接器）。您可以通过一条命令来进行编译和链接，即使您的项目分散于多个源文件中。

第 4.2 节 “调用编译器”对该驱动程序进行了介绍。如果安装了多个驱动程序，请参见**第 3.3.4 节 “如何选择要用于编译项目的编译器？”**，确保可以运行正确的驱动程序。**第 4.7 节 “XC8 驱动程序选项”**详细介绍了驱动程序的命令行选项。**第 4.2.3 节 “输入文件类型”**列出并介绍了可以传递给驱动程序的文件。

3.3.3 如何使用 make 实用程序进行编译？

使用 make 实用程序 （如 `make`）进行编译时，编译通常作为一个两步过程执行：先生成中间文件，然后执行最终的编译和链接步骤来生成一个二进制输出。**第 4.3.3 节 “多步编译”**对此进行了介绍。

XC8 编译器采用了一种称为 OCG 的独特技术，该技术使用一种不同于传统编译器（包括 XC16 和 XC32）的中间文件格式。XC8 使用的中间文件格式是一个 p 代码文件（`.p1` 扩展名），而不是一个目标文件。生成目标文件作为中间文件来进行多步编译会使该技术的许多优点落空。

3.3.4 如何选择要用于编译项目的编译器？

编译和安装过程设计为允许您同时安装多个编译器。您可以在 MPLAB X IDE 中创建一个项目，然后只需通过在项目属性中更改设置即可使用不同编译器来编译项目。

要选择在 MPLAB X IDE 下编译项目时实际使用的编译器，请转至 Project Properties 对话框。在 Project Properties 对话框中选择 Configuration 类别（`Conf: [default]`）。最右侧的 Compiler Toolchain （编译器工具链）中将会显示 XC8 编译器的列表。选择您需要的 XC8 编译器。

选定之后，通过选择 XC8 全局选项、 XC8 Compiler 和 XC8 Linker 类别，将显示对应于该编译器的所有控件。它们会在右侧展现选项的窗格。请注意，每个类别都具有几个窗格，可以从靠近窗格顶部的下拉菜单中进行选择。

3.3.5 如何更改编译器的工作模式？

通过命令行编译项目时，可以将编译器的工作模式（免费、标准或专业，见**第 1.2 节 “编译器说明和文档”**）指定为一个命令行选项，请参见**第 4.8.37 节 “--MODE：选择编译器工作模式”**。如果在 MPLAB X IDE 下编译项目，在 XC8 编译器类别中的 Optimizations（优化）选项选择器下提供了Project Properties选择器，请参见**第4.10.2节“编译器类别”**。

您只能选择您的许可证允许您使用的模式。免费模式始终是可用的；如果您购买了标准或专业模式的许可证，则可以选择这些模式。

3.3.6 如何编译库？

请注意， XC8 采用了一种不同的代码生成框架 （OCG），除了传统编译器 （包括 XC16 和 XC32）使用的库文件之外，该框架还使用了一些额外的库文件。关于可用库类型和如何将它们纳入编译过程的一般信息，请参见**第 4.3.1 节 “编译器应用程序”**。

当您具有一些在应用程序之间通用的函数和数据时，您可以提供所有 C 源文件和头文件，使其他开发人员可以将它们复制到其项目中。或者，您也可以将这些源文件捆绑到某个库中，该库与随附的头文件可以链接到项目中。

库更便于使用，因为需要处理的文件较少。通过库来编译代码也会略快。但是，库需要进行维护。对于以 C 语言编写的库程序， XC8 必须使用 LPP 库；旧型的 LIB 库用于以汇编语言源代码编写的库程序。如果您的项目要迁移到新的编译器版本，则建议也重新编译这些库。

使用编译器驱动程序时，可以通过在命令行上列出要包含到库中的所有文件来编译库。所有这些文件都不应包含`main()`函数，也不应包含配置位的设置或任何其他此类数据。使用 `--OUTPUT=lpp` 选项（见**第 4.8.44 节 “--OUTPUT= type：指定输出文件类型”**）来指示需要的是一个库文件。例如：

```
XC8 --chip=16f877a --output=lpp lcd.c utils.c io.c
```

将创建一个名为 `lcd.lpp` 的库文件。您可以使用 `-O` 选项（见**第 4.8.10 节 “-O：指定输出文件”**）指定其他名称，也可以简单地重命名该文件。

© 2013 Microchip Technology Inc.

3.3.7 如何确定哪些编译器选项可用，以及它们的功能？

通过在命令行上使用 --HELP 选项，可以获取所有编译器选项的列表，请参见**第 4.8.33 节 “--HELP：显示帮助”**。如果向 --HELP 选项提供了一个参数来作为选项名称，则它将会给出关于该选项的具体信息。

另外，本用户指南的**第 4.8 节 “选项说明”**中列出了所有选项。如果在 MPLAB X IDE 中进行编译，则请参见**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**；或者如果在 MPLAB IDE V8 中进行编译，则请参见**第 4.9 节 “MPLAB IDE V8 Universal Toolsuite 等效选项”**。

3.3.8 如何确定 MPLAB IDE 中的编译选项的功能？

在大多数情况下，MPLAB IDE 编译选项中的窗口小部件和控件都直接映射到一个命令行驱动程序选项或子选项。在适当情况下，用户指南中列出所有命令行驱动程序选项的章节（**第 4.8 节 “选项说明”**）中提供了交叉引用，指向与从 IDE 中访问该选项相关的章节。对于 MPLAB X IDE（**第 4.10 节“MPLAB X Universal Toolsuite 等效选项”**）和 MPLAB IDE V8 （**第 4.9 节 “MPLAB IDE V8 Universal Toolsuite 等效选项”**）存在两个独立的章节。

3.3.9 MPLAB IDE 调试编译有何差别？

MPLAB IDE V8 工具条中的 Debug/Release（调试 / 发布）下拉窗口小部件指示编译是调试编译还是发布编译。在 MPLAB X 中，为编译项目和调试项目提供了独立的编译按钮和菜单项。

对于 IDE，两者有许多差别，但对于 XC8 编译器，两者之间的差别极小。主要差别是，当选择调试时，会将名为 __DEBUG 的预处理器宏设置为 1。如果不是调试编译，则不定义该宏。

您可以在源文件中使用 #ifdef 等伪指令（见**第 5.14.2 节 “预处理器伪指令”**），使编译器根据该宏来编译代码，从而可以在仍处于开发周期中时让程序具有不同的行为。在执行调试编译之后，可以更容易地跟踪一些编译器错误。

在 MPLAB X IDE 中，只有执行调试编译时，才会为调试器保留存储器 （如果选中）。在 MPLAB v8 中，如果在项目中选择了调试器硬件工具，则总是会保留存储器，请参见**第 3.5.3 节 “在编译为使用调试器时需要做些什么？”**。

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

本节讨论与 C 语言本身直接相关但人们经常会问到的源代码问题。

- *何时应对表达式进行强制类型转换？*
- *隐式类型转换是否会更改表达式的预期结果？*
- *如何在程序中输入非英语字符？*
- *如何使用在另一个源文件中定义的变量？*

3.4.1.1 何时应对表达式进行强制类型转换？

表达式可以作为使用强制类型转换操作符 （类型包含在圆括号中，如 `(int)`）的明确实例。在所有情况下，将一种类型转换为另一种类型时都必须谨慎，并且只有在绝对必要时才这么做。

假设具有以下示例：

```
unsigned long l;
unsigned int i;

i = l;
```

此处，一个 `long` 类型被赋值给一个 `int` 类型，并且该赋值将截断 `l` 中的值。编译器将自动执行类型转换，从表达式中赋值操作符右侧的类型 （`long`）转换为操作符左侧值的类型 （`int`）。这称为隐式类型转换。编译器通常会产生一条有关截断可能丢失数据的警告。

如果希望将 `long` 转换为 `int`，则在以上示例中并不需要强制转换为 `int` 类型，也不应使用强制类型转换。编译器知道两个操作数的类型，并将相应地执行转换。如果使用了强制类型转换，并且如果稍后更改了代码，则有可能会产生错误。例如，如果具有以下代码：

```
i = (int)l;
```

该代码以相同的方式工作；但如果在将来，举例来说，`i` 的类型更改为 `long`，则必须记住要调整强制类型转换或将其删除，否则 `l` 的内容将继续被赋值截断，这可能产生错误。最重要的是，如果替换为使用强制类型转换，将不会产生由编译器发出的警告。

© 2013 Microchip Technology Inc.

仅在编译器使用的类型不是所需类型的情况下使用强制类型转换。例如，假设将除法运算的结果赋值给浮点型变量：

```
int i, j;
float fl;

fl = i/j;
```

在此例中，先执行整型除法运算，然后舍入后的整型结果被转换为 float 格式。所以，如果 i 包含 7，j 包含 2，除法运算结果将为 3，它会被隐式转换为 float 类型（3.0），然后赋值给 fl。如果希望以 float 格式执行除法运算，则需要强制类型转换：

```
fl = (float)i/j;
```

（强制转换 i 或 j 将强制编译器将代码编码为浮点除法运算）。赋值给 fl 的结果现在为 3.5。

显式强制类型转换会禁止其他情况下可能产生的警告。这也是很多问题的根源。编译器产生的警告越多，在代码中发现潜在错误的几率就越高。

3.4.1.2 隐式类型转换是否会更改表达式的预期结果？

会！编译器将总是使用整型提升，没有任何办法可以禁止它，请参见**第 5.6.1 节 “整型提升”**。此外，二元操作符的操作数的类型通常会发生改变，从而使它们具有 C 标准指定的共同类型。更改操作数的类型会改变最终表达式的值，所以了解使用二元操作符时应用的类型 C 标准转换规则非常重要。您可以通过强制类型转换手动更改操作数的类型，请参见**第 3.4.1.1 节 “何时应对表达式进行强制类型转换？”**。

3.4.1.3 如何在程序中输入非英语字符？

ANSI 标准和 MPLAB XC8 不支持在源代码字符集的字符和字符串字面值中使用扩展字符集。关于如何使用转义序列输入这些字符的信息，请参见**第 5.4.6 节 “常量类型和格式”**。

3.4.1.4 如何使用在另一个源文件中定义的变量？

假定在另一个源文件中定义的变量不为 static（见**第 5.5.2.1.1 节 “静态变量”**）或 auto（见**第 5.5.2.2 节 “自动变量分配和访问”**），则在当前文件中添加该变量的声明将允许您对其进行访问。声明包含关键字 extern，以及在变量定义中指定的变量类型和名称，例如

```
extern int systemStatus;
```

这是 C 语言的一部分，您爱读的 C 语言教材可以为您提供更多信息。

声明在当前文件中的位置决定变量的作用域，即，如果将声明放在某个函数内，则会将该变量的作用域限制在该函数内；放置在函数之外将允许当前文件其余部分的所有函数访问该变量。

通常情况下，声明放在头文件中，然后通过 #include 将它们包含到 C 源代码中，请参见**第 5.14.2 节 “预处理器伪指令”**。

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 PIC 器件的功能需要编写的代码。

- *如何设置配置位？*
- *如何使用 PIC 单片机的 ID 存储单元？*
- *如何确定中档器件上的复位原因？*
- *如何访问 SFR？*
- *如何阻止编译器使用某些特定的存储单元？*
- *在编译为使用调试器时需要做些什么？*

3.4.2.1 如何设置配置位？

它们应在代码中使用宏或 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。关于它们的设置方式，请参见**第 5.3.5 节 “配置位访问”**。

3.4.2.2 如何使用 PIC 单片机的 ID 存储单元？

提供了一个用于对这些值进行编程的宏或 pragma 伪指令，请参见**第 5.3.7 节“ID 存储单元”**。

3.4.2.3 如何确定中档器件上的复位原因？

STATUS 寄存器中的 TO 和 PD 位可以用于确定复位的原因。但是，这些位会被在 `main` 执行之前执行的运行时启动代码快速覆盖，请参见**第 5.10.1 节 “运行时启动代码”**。您可以将 STATUS 寄存器内容保存到稍后可从 C 代码访问的某个存储单元中，从而让应用程序可以在再次运行时确定复位的原因。请参见**第 5.10.1.4 节 “STATUS 寄存器保存”**。

3.4.2.4 如何访问 SFR？

编译器附带了一些头文件 （见**第 5.3.3 节 “器件头文件”**），这些文件定义了一些映射到存储器映射 SFR 顶部的变量。由于它们是 C 变量，所以可以像任何其他 C 变量一样使用它们，访问这些寄存器无需任何新语法。

此外，还可以访问 SFR 内的位。头文件定义了独立的位宽度变量，它们映射到 SFR 中的位。此外，还可以通过一些结构来访问位域，这些结构作为一个整体映射到 SFR。您可以在代码中使用其中任意一种。请参见**第 5.3.6 节 “从 C 代码中使用 SFR”**。

为变量指定的名称通常与器件数据手册中指定的名称相同。如果无法识别这些名称，请参见**第 3.4.2.5 节 “如何确定用于表示 SFR 和位的名称？”**。

© 2013 Microchip Technology Inc.

3.4.2.5 如何确定用于表示 SFR 和位的名称？

特殊功能寄存器及其内的位均可通过映射到寄存器的特殊变量进行访问（**第 3.4.2.4 节“如何访问 SFR？”**）；但是，这些变量的名称有时会不同于您所用器件的数据手册中指示的名称。

您可以通过 <xc.h> 头文件确定特定于器件的头文件（该文件使您可以访问这些特殊变量），但更简单的方法是查看在先前编译之后留下的任意预处理文件。这些文件具有 .pre 扩展名，对于项目中的每个源文件，其中都会有一个文件的基本名称与之相同。在任何包含 <xc.h> 的源文件的预处理文件中查找，因为它将包含所有 SFR 变量及其内位的定义。

如果在MPLAB X IDE下进行编译，对于常规编译，预处理文件将留在项目 build/default/production 目录下；对于调试编译，将留在 build/default/debug 目录下。如果在命令行上进行编译，则通常留在源文件目录中。

3.4.3 存储器分配

以下是一些与源代码如何影响存储器分配有关的问题。

- *如何将变量定位到指定的地址处？*
- *如何将函数定位到指定的地址处？*
- *如何将变量放入程序存储器？*
- *如何阻止编译器使用某些特定的存储单元？*
- *为什么某些对象被定位到保留的存储器中？*

3.4.3.1 如何将变量定位到指定的地址处？

实现该目的最简单的方法是通过使用 @ *address* 构造将变量设为绝对变量，请参见**第 5.5.4 节“绝对变量”**。这意味着将使用您指定的地址，而不是变量在生成代码中的符号。由于可以指定地址，所以可以完全控制对象的存放位置，但还必须确保绝对变量不会发生重叠。已放入存储区中的变量可能会对其他变量的分配产生破坏，从而产生“无法找到空间”的错误，请参见**第 3.7.6 节“如何解决“无法找到空间 ...”错误？”**。关于如何移动自动变量的信息，另请参见**第 5.5.2.4 节“更改默认的自动变量分配”**；关于移动非自动变量的信息，另请参见**第 5.5.2.1.3 节“更改默认的非自动变量分配”**；关于移动程序空间变量的信息，另请参见**第 5.5.3.2 节“更改默认分配”**。

3.4.3.2 如何将函数定位到指定的地址处？

实现该目的最简单的方法是通过使用 @ *address* 构造将函数设为绝对函数，请参见**第 5.8.4 节“更改默认的函数分配”**。这意味着将使用您指定的地址，而不是函数在生成代码中的符号。由于可以指定地址，所以可以完全控制函数的存放位置，但还必须确保绝对函数不会发生重叠。已放入页中的函数可能会对其他函数的分配产生破坏，从而产生“无法找到空间”的错误，请参见**第 3.7.6 节“如何解决“无法找到空间 ...”错误？”**。

3.4.3.3 如何将变量放入程序存储器？

const 限定符意味着所限定的变量是只读的。因此，使用 const 限定的所有变量（自动变量或函数参数除外）都会被放入程序存储器，从而释放宝贵的数据 RAM，请参见**第 5.5.3 节“程序空间中的变量”**。使用 const 限定的变量也可以设为绝对变量，从而可以将它们定位到您指定的地址处，请参见**第 5.5.4.2 节“程序存储器中的绝对对象”**。

3.4.3.4 如何阻止编译器使用某些特定的存储单元？

在进行编译时，可以保留存储器。通过 --RAM 和 --ROM 选项，可以在编译时分别调整数据存储器和程序存储器的范围。请参见**第 4.8.48 节“--RAM：调整 RAM 范围”**和**第 4.8.49 节“--ROM：调整 ROM 范围”**。默认情况下，所有可用的片上存储器都可供使用，但这些选项允许您保留部分存储器。

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？*
- *如何访问变量的各个位？*
- *变量名和宏名可以多长？*
- *如何在中断与主干代码之间共享数据？*
- *如何将变量定位到指定的地址处？*
- *如何将变量放入程序存储器？*
- *如何将变量放入 PIC18 的外部程序存储器？*
- *如何才能循环移位一个变量？*
- *如何利用器件上的所有 RAM 存储区？*
- *如何利用增强型中档 PIC 器件上的线性存储器？*
- *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，确保在 MPLAB IDE 中查看浮点型变量时，它们的类型和大小与它们的定义匹配。对于 24 位浮点型变量（无论它们的类型是 float 还是 double），需要确保变量属性中的 Format（格式）设置为 IEEE float MPLAB IDE v8。在 MPLAB X IDE 中，将 Display Column Value As（列值显示方式）弹出式菜单设置为 IEEE float (24 bit)（IEEE 浮点型（24 位））。如果变量为 32 位浮点型对象，则在两种 IDE 中都将类型设置为 IEEE Float（IEEE 浮点型）。

对于 float 和 double 类型，浮点型的长度都可以调整，请参见**第 4.8.31 节“--FLOAT：选择浮点型的种类”**和**第 4.8.24 节 “--DOUBLE：选择双精度浮点型的种类”**。

因为浮点型变量仅使用有限的位数来表示它们被赋予的值，所以它们将包含其所赋予值的近似值，请参见**第 5.4.3 节 “浮点型数据类型”**。浮点型变量只能包含一个离散实数值集合中的一个值。如果尝试赋予不处于该集合中的值，它会被舍入为最接近的值。浮点型变量中尾数使用的位越多，集合中可以准确表示的值就越多，由于舍入产生的平均误差会减小。

每次执行浮点运算时，也会发生舍入。这也可能导致看起来不正确的结果。

© 2013 Microchip Technology Inc.

3.4.4.2 如何访问变量的各个位？

有几种方法可以实现该目的。最简单、可移植性最好的方法是定义一个整型变量，并使用一些宏通过掩码值和逻辑运算来读取、置 1 或清零变量中的位，如下所示。

```
#define  testbit(var, bit)   ((var) & (1 <<(bit)))
#define  setbit(var, bit)    ((var) |= (1 << (bit)))
#define  clrbit(var, bit)    ((var) &= ~(1 << (bit)))
```

以上代码分别用于：测试整型 var 中的位编号 bit 是否置 1；将 var 中相应的 bit 置 1；将 var 中相应的 bit 清零。或者，也可以定义一个由整型变量和带位域的结构组成的联合（见**第 5.4.4.2 节 “结构中的位域”**），例如

```
union both {
   unsigned char byte;
   struct {
      unsigned b0:1, b1:1, b2:1, b3:1, b4:1, b5:1, b6:1, b7:1;
   } bitv;
} var;
```

这使您可以将 byte 作为一个整体访问（使用 var.byte），也可以独立地访问该变量中的任意一位（使用 var.bitv.b0 至 var.bitv.b7）。

请注意，编译器支持位变量（见**第 5.4.2.1 节“位数据类型和变量”**）和结构中的位域。

3.4.4.3 变量名和宏名可以多长？

C 标准规定，只有标识符中一定数量的初始字符是有效的，但它并未实际说明这一数字，它会因编译器而异。对于 XC8，前 255 个字符是有效的，但可以使用 -N 选项减小它的长度，请参见**第 4.8.9 节 “-N：标识符长度”**。变量名中的字符数越少，代码的可移植性就越高。使用 -N 选项使编译器可以检查标识符是否符合特定长度。该选项会影响变量名和函数名，以及预处理器宏名。

如果两个标识符只有名称的非有效部分是不同的，则它们会被视为代表同一个对象，这几乎肯定会导致代码失败。

3.4.5 函数

本节讨论与函数相关的问题。

- *函数的最佳大小是多大?*
- *如何确定某个函数有多大?*
- *如何知道每个函数在使用哪些资源?*
- *如何确定变量和函数定位到何处?*
- *如何在 C 中使用中断?*
- *如何防止删除未用函数?*
- *如何将函数设为内联函数?*

3.4.5.1 函数的最佳大小是多大?

一般来说，函数的源代码应保持很小，这有助于可读性和调试。描述并调试执行少量任务的函数的操作会比较容易，并且它们可能成为编码错误来源的副作用通常较少。在嵌入式编程环境中，大量的小函数，以及执行它们所需的调用可能导致过多的存储器和堆栈使用，所以往往需要作出折衷。

PIC10/12/16 器件使用了程序存储器中的页，函数代码存储在页中并在其中执行。虽然编译器允许，并且也可以编码大小（生成的汇编代码的大小）超过一个程序存储器页的函数，但还是应避免这种大小的函数，并在可能情况下将它们拆分成更小的程序。汇编代码调用和跳转序列的目标单元处于其他页中时，代码长度会远高于目标处于同一页中时。如果某个函数大到需要跨越页边界，则需要在该函数内跳转的循环或其他代码构造可能会在每次迭代时使用更长的跳转形式，请参见**第 5.8.3 节 “可执行代码的分配”**。

内部存储器分页对 PIC18 器件的影响较小，其指令集允许调用和跳转到任意目标，无需额外开销，但仍应努力使函数保持尽可能小。

对于所有器件，函数越小，链接器就可以越容易地在不产生错误的情况下将其分配到存储器中。

3.4.5.2 如何防止删除未用函数?

如果在手写的汇编代码中引用了某个 C 函数的符号，则永远不会删除该函数，即使 C 代码并不调用它或永远不会使用其地址。

创建一个汇编语言源文件，并将该文件添加到项目中。您只需在该文件中引用该符号，所以该文件可以包含以下内容

```
GLOBAL _myFunc
```

其中， myFunc 是所涉及函数的 C 名称 （请注意汇编语言名称中的前导下划线，见**第 5.12.3.1 节 “等效汇编符号”**）。这就足以防止执行函数删除优化。

3.4.5.3 如何将函数设为内联函数?

您可以通过使用 inline 说明符要求编译器对某个函数进行内联。这只是提供给编译器的一个建议，可能并不总是会得到服从。不要将该说明符与 inline pragma 伪指令（**第 5.14.4.4 节 “#pragma Intrinsic 伪指令”**）相混淆，后者用于没有对应的源代码，并将在编译过程中由代码生成器特别扩展的函数。

© 2013 Microchip Technology Inc.

3.4.6 中断

本节讨论中断和中断服务程序问题。

- *如何在C 中使用中断？*
- *如何让中断程序更快？*
- *如何在中断与主干代码之间共享数据？*

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了何种中断硬件。低档 PIC 器件根本不实现中断；中档器件采用单个中断向量，PIC18 器件实现了两个独立的中断向量单元，并使用简单的优先级方案。

在C源代码中，可以通过使用 `interrupt` 限定符将某个函数编写为用作中断服务程序，请参见**第 5.9.1 节 “编写中断服务程序”**。这种函数会在执行函数主体代码之前 / 之后保存 / 恢复程序现场，并使用一条不同的返回指令，请参见**第 5.9.3 节 “现场切换”**。对于目标器件上实现的每个中断向量，对应的中断函数不得超过一个。

除了 `interrupt` 限定符之外，函数原型不能指定任何参数，并使用 `void` 返回类型。如果要在 PIC18 器件上实现低优先级中断函数，可以与 `interrupt` 限定符一起使用 `low_priority` 关键字。

中断函数内的代码可以执行您需要的任何操作，但关于增强实时性能的建议，请参见**第 3.6.6 节 “如何让中断程序更快？”**。

在发生任何中断之前，程序必须确保已正确配置了外设，并已允许中断，请参见**第 5.9.4 节 “允许中断”**。在 PIC18 器件上，必须通过写入相应的 SFR 来指定中断源的优先级。

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？*
- *编写汇编代码时，必须控制哪些方面？*

3.4.7.1 如何混合使用汇编代码和 C 代码？

理想情况下，所有手写汇编代码都应编写为可调用的独立程序。这可以提供一定程度的保护，防止编译器生成的汇编代码和手写汇编代码之间的相互作用。这种代码可以放入可添加到项目中的独立汇编模块中，请参见**第 5.12.1 节 “集成汇编语言模块”**。

如果需要，汇编代码可以使用两种方法之一嵌入到 C 代码中，请参见**第 5.12.2 节“#asm、#endasm 和 asm()”**。理想情况下，嵌入的代码仅限于诸如 `NOP`、`SLEEP` 或 `CLRWDT` 之类的指令。编译器已提供了嵌入所有这些指令的宏，请参见**附录 A “库函数”**。如果未采取预防措施，可更改寄存器内容和器件状态的更复杂行内汇编代码可能会导致代码失败，因此要谨慎使用。关于编译器使用的寄存器，请参见**第 5.7 节 “寄存器使用”**。

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。**第 6.4.9 节 “汇编器伪指令”**的子节中介绍了所有伪指令的操作。下面介绍了所需的常用伪指令。

所有汇编代码必须放在 psect 中，以便链接器可以将它作为一个整体处理和放入存储器。关于 psect 的一般信息，请参见**第 5.15.1 节“程序段”**；关于用于创建和指定 psect 的伪指令的信息，请参见**第 6.4.9.3 节 “PSECT”**。

另一条常用的伪指令是 GLOBAL （在**第 6.4.9.1 节 “GLOBAL”**中定义），它用于将符号设为可以从多个源文件进行访问。

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代码的等效符号，**第 5.12.3 节 “汇编代码和 C 代码之间的交互”**对它们进行了详细说明。

通过使用 GLOBAL 汇编器伪指令，向汇编器指示符号在其他位置定义，请参见**第 6.4.9.1 节 “GLOBAL”**。它是 C 声明的汇编等效形式，虽然它不提供任何类型信息。如果符号在与汇编代码相同的模块中定义，则不需要该伪指令，且不应使用。

从汇编代码中访问的所有 C 变量都被视为使用 `volatile` 限定，请参见**第 5.4.7.2 节 “volatile 类型限定符”**。在 C 代码中特别指定 `volatile` 限定符是首选的方式，因为它明确说明外部代码可能会访问该对象。

3.4.7.4 如何从汇编代码内访问 SFR？

在汇编代码中访问 SFR 的最安全方式是在汇编代码中定义等效于相应 SFR 地址的符号。编译器随附了一些头文件，这使您不需要自己定义这些符号，**第 5.12.3.2 节 “从汇编代码中访问寄存器”**对它们进行了详细介绍。

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。

3.4.7.5 编写汇编代码时，必须控制哪些方面？

如果手工编写汇编代码，则必须控制以下几个方面。

- 每次访问 RAM 变量时，必须确保先选择变量的存储区，然后再读取或写入存储单元。这通过一条或多条汇编指令来完成。确切的代码取决于您使用的器件和变量的位置。如果对象处于公共存储区 （它在 PIC18 器件上被称为快速操作存储区）中，或者如果您使用的指令采用完整地址（如 PIC18 器件上的 MOVFF 指令），则不需要进行存储区选择。请核对您所使用器件的数据手册，以查看您所用器件的存储器架构，以及控制存储区选择的指令和寄存器。未选择正确的存储区会导致代码失败。BANKSEL 伪指令可以用于简化该过程，请参见**第 6.4.1.2 节 “存储区和页选择”**。

© 2013 Microchip Technology Inc.

- 必须确保对要访问的 RAM 变量的地址进行掩码处理，从而仅使用存储区偏移量作为指令的文件寄存器操作数。如果使用的指令采用完整地址（如 PIC18 器件上的 MOVFF 指令），则不应这样做。请核对您所使用器件的数据手册，以确定所需的地址操作数指令。未对地址进行掩码处理可能会导致 fixup 错误（见**第 3.7.8 节 “如何解决 fixup 溢出错误？”**）或代码失败。
  BANKMASK 宏可以用于截断地址，请参见**第 5.12.3.2 节“从汇编代码中访问寄存器”**。
- 在调用或跳转到任何程序之前，必须确保使用相应的指令选择了该程序的程序存储器页。可以使用 PAGESEL 伪指令（见**第 6.4.1.2 节 “存储区和页选择”**），或者 FCALL 或 LJMP 伪指令（在 PIC18 器件上不需要，见**第 6.4.1.4 节 “长跳转和调用”**），它们将根据需要自动添加页选择指令。
- 必须确保为用于存储的所有 RAM 保留存储器。如果您仅访问在 C 代码中定义的变量，则保留已经由编译器执行。对于仅在汇编代码中使用的所有变量，必须使用相应的伪指令（如 DS 或 DABS）为它们保留存储器，请参见**第 6.4.9.10 节 “DS”**或**第 6.4.9.11 节 “DABS”**。通常，在 C 代码中定义对象会比在汇编代码中容易。
- 您必须将您编写的所有汇编代码放入 psect（关于执行该操作的伪指令，请参见**第 6.4.9.3 节 “PSECT”**；关于 psect 的一般信息，请参见**第 5.15.1 节 “程序段”**）。您定义的 psect 可能需要指定一些标志（选项）。请特别注意 delta、space 和 class 标志（见**第 6.4.9.3.4 节 “delta”**、**第 6.4.9.3.13 节 “space”**和**第 6.4.9.3.3 节 “class”**）。如果未正确设置它们，则几乎肯定会产生编译错误或代码失败。如果 psect 指定了某个类，并且您愿意将它放在该类定义的存储器范围中的任意位置（见**第 7.2.1 节 “-Aclass =low-high,...”**），则不需要链接任何附加的选项；否则，将需要使用链接器选项链接该 psect（关于链接 psect 的常用方式，请参见**第 7.2.19 节 “-Pspec”**；关于如何不直接运行链接器而指定该选项的信息，请参见**第 4.8.7 节 “-L-：直接调整链接器选项”**）。
  嵌入到 C 代码中的汇编代码将被放入与编译器生成的汇编代码相同的 psect，不应将它放入单独的 psect。
- 您必须确保编译器生成的代码不在使用您在汇编代码中写入的任何寄存器。如果在一个独立模块中编写汇编代码，则这不再是一个问题，因为编译器在默认情况下会假定所有寄存器都由这些程序使用（见**第 5.7 节 “寄存器使用”**中的寄存器）。行内汇编代码不存在任何假定（见**第 5.12.2 节 “#asm、#endasm 和 asm()”**），您必须小心地保存和恢复您使用（写入）的任何资源，以及周围的编译器生成代码已在使用的资源。

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。

- *什么原因会在输出端口上产生毛刺？*
- *如何链接自举程序和可下载应用程序？*
- *在编译为使用调试器时需要做些什么？*
- *如何在复位后直接执行代码？*
- *如何在中断与主干代码之间共享数据？*
- *如何防止代码被盗用？*
- *如何使用 printf 将文本发送到外设？*
- *如何校准器件上的振荡器？*
- *如何将变量放入 PIC18 的外部程序存储器？*
- *如何在代码中实现延时？*
- *如何才能循环移位一个变量？*

3.5.1 什么原因会在输出端口上产生毛刺？

在大多数情况下，这通常是由于使用普通变量来访问端口位或整个端口本身而导致的。这些变量应使用 `volatile` 进行限定。

对于映射到端口的变量，在其中存储的值（从而是写入到该端口的实际值）会被直接转换为电气信号。这些变量包含的值应仅在代码希望时才发生更改，并且它们从其当前状态更改为新值只经过单次变换，这两点至关重要。请参见**第 5.4.7.2 节 “volatile 类型限定符”**。编译器会尝试在一次操作中写入 volatile 变量。

3.5.2 如何链接自举程序和可下载应用程序？

如何实现这一点取决于您所用的器件和您的项目要求，但编译使用自举程序的应用程序的一般方法是为自举程序和应用程序分配分离的程序存储空间，从而使它们具有自己的专用存储器。通过这种方式，其中一个的操作不会影响另一个。这将要求自举程序或应用程序在存储器中进行偏移。即，复位和中断的位置相对于地址 0 进行偏移，所有程序代码则偏移相同的地址量。

在 PIC18 器件上，通常对应用程序代码进行偏移，自举程序在链接时不进行任何偏移，从而使它填充复位和中断代码的位置。自举程序的复位和中断代码仅仅包含将控制重定向到由应用程序定义的、已进行偏移的真正复位和中断代码。

在中档器件上，当使用中断时，通常无法执行这种处理。可以考虑这种处理方式：与复位关联的代码（它必须总是由自举程序定义）除外，对自举程序的所有其他代码进行偏移。应用程序代码可以定义在中断位置链接的代码。自举程序需要重映射任何试图覆盖自举程序所定义的复位代码的应用程序代码。

选项 `--CODEOFFSET`（见**第 4.8.22 节 “--CODEOFFSET：将程序代码偏移到某个地址”**）可以用于将程序代码（包括复位和向量）移动一个指定的量。该选项还可以对程序进行限制，防止程序使用从地址 0（复位向量）到偏移地址处的任何程序存储器。请总是检查映射文件（见**第 7.4.2 节 “内容”**），确保没有任何内容留在保留区域中。

© 2013 Microchip Technology Inc.

可以将自举程序的 HEX 文件的内容与应用程序的代码合并，其方法是在命令行上或在 MPLAB IDE 中将 HEX 文件添加为一个项目文件。这将产生一个在单个镜像中包含自举程序和应用程序代码的 HEX 文件。HEX 文件通过 HEXMATE 应用程序进行合并，请参见**第 8.6 节 “HEXMATE”**。请检查该应用程序有关重叠的警告，这些警告可能指示存储器同时由自举程序和可下载应用程序使用。

3.5.3 在编译为使用调试器时需要做些什么？

您可以使用调试器（如 ICD3 或 REAL ICE）来调试使用 XC8 编译器编译的代码。这些调试器会将器件的一些数据存储器和程序存储器留给自己使用，所以您的代码不能使用这些资源，这一点很重要。

有一个命令行选项（见**第 4.8.23 节 “--DEBUGGER：选择调试器类型”**）可以用于指示编译器要使用的调试器。然后，编译器可以保留由调试器使用的存储器，从而您的代码不会装入这些位置。

在 MPLAB X IDE 中，如果执行调试编译，则会指定相应的调试器选项。如果执行常规的 Build Project （编译项目）或 Clean and Build （清除并编译），则不会指定它。

在 MPLAB IDE v8 中，建议从 Build Options （编译选项）对话框的 Linker （链接器）选项卡的 Debugger（调试器）中选择 Auto（自动）。通过这种方式，向编译器指示的调试器将与为项目选择的相同。该选项总是会产生作用。对于发布编译，不要选择任何调试器。

由于调试器会占用一些器件存储器，所以可供程序使用的存储器会变少，在选择调试器时您的代码或数据可能无法再装入器件中。

请注意，调试器具体使用哪些存储单元属于 MPLAB IDE 而不是器件的特性。如果将一个项目迁移到新版本的 IDE，所需的资源可能会发生变化。因此，不应手动为调试器保留存储器，或在代码中对于使用哪些存储器作出任何假设。 MPLAB IDE 帮助文件中提供了调试器要求的汇总。

要验证编译器保留的资源是否与调试器所要求的匹配，请执行以下操作。在选择和不选择调试器的两种情况下编译您的代码，并保留为两种编译方式生成的映射文件的副本。比较两个映射文件中的链接器选项，并查看 `-A` 选项中的变化，请参见**第 7.2.1 节 “-Aclass =low-high,...”**。例如，不选择调试器时为 `CODE` 类定义的存储器可能使用以下选项指定：

```
-ACODE=00h-0FFh,0100h-07FFh,0800h-0FFFhx3
```

选择 ICD3 作为调试器时，它变为：

```
-ACODE=00h-0FFh,0100h-07FFh,0800h-0FFFhx2,01800h-01EFFh
```

可以看到，从 1F00 至 1FFF 的存储器范围已被编译器去除，无法供程序使用。另请参见**第 3.6.16 节 “为什么某些对象被定位到保留的存储器中？”**。

3.5.4 如何在复位后直接执行代码？

提供了一个特殊的 hook，让您可以简便地添加特殊的“上电”汇编代码，该代码将被链接到复位向量处，请参见**第 5.10.2 节 “上电程序”**。该代码将在运行时启动代码执行之前执行，因而它将在`main`函数之前执行，请参见**第5.10节“main、运行时启动和复位”**。

3.5.5 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（见**第 5.4.7.2 节 “volatile 类型限定符”**）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。

其他问题涉及到编译器 / 器件是否能够以原子方式访问数据。对于 8 位 PIC 器件，很少会出现这种情况。原子访问是指仅通过一条指令访问整个变量的访问。这种访问是不可中断的。您可以通过查看编译器在汇编列表文件中生成的汇编代码，确定是否是以原子方式访问某个变量，请参见**第 6.6 节 “汇编列表文件”**。如果是通过一条指令访问该变量，则它是原子的。由于访问变量的方式会因语句而不同，所以通常最好完全避免这些问题，即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，请参见**第5.9.4节“允许中断”**。

3.5.6 如何防止代码被盗用？

首先，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。要使它生效，需要正确地设置器件配置位，请参见**第 5.3.5 节 “配置位访问”**和您所使用器件的数据手册。

其次，您可以通过使用某个值填充程序存储器中的未用存储单元，而不是将它们保留为其默认的未编程状态，防止将第三方代码烧写到这些存储单元中。您可以选择一个对应于某条指令的填充值，或者将所有位都置 1，从而使得无法进一步修改这些值。（想象一下，如果程序不知何故达到并从这些填充值处开始执行，将会发生什么。将执行什么指令？）

编译器的 HEXMATE 实用程序（见**第 8.6 节 “HEXMATE”**）能够填充未用存储单元，该操作可以使用命令行驱动程序选项来请求，请参见**第 4.8.30 节 “--FILL：填充未用的程序存储器”**。由于 HEXMATE 仅处理 HEX 文件，所以该功能仅在生成 HEX/COF 文件输出（而不是二进制文件，举例来说）时可用，这是默认操作。

最后，如果希望将库文件或中间 p 代码文件提供给他人使用，但不希望他们可查看原始源代码，则可以使用 --SHROUD 选项混淆文件，请参见**第 4.8.54 节 “--SHROUD：混淆 p 代码文件”**。

3.5.7 如何使用 printf 将文本发送到外设？

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），请参见**附录 A “库函数”**。 printf 函数会执行所有格式化；然后它会调用一个名为 putch 的助手函数来发送格式化文本的每个字节。通过自定义 putch 函数，可以让 printf 将数据发送到任何外设或位置，请参见**第 5.11.1 节 “printf 程序”**。例如，您可以选择将 printf 输出送到 LCD、SPI 模块或 USART。

© 2013 Microchip Technology Inc.

编译器的 sources 目录中可以找到 putch 函数的桩 （stub）。将它复制到您的项目中，然后对它进行修改，使之将传递给它的单字节参数发送到所需的目标。在可以使用 printf 之前，需要以正常方式初始化您使用的外设。以下给出了一个在中档器件上用于 USART 的 putch 函数的示例。

```
void putch(char data) {
  while( !TXIF)  // check buffer
    continue;     // wait till ready
  TXREG = data;   // send data
}
```

您可以让 printf 将数据发送到多个目标之一，并使用一个全局变量来指示您的选择。让 putch 函数基于该变量的内容向多个目标之一发送字节。

3.5.8 如何校准器件上的振荡器？

一些器件允许对它们的内部振荡器进行校准，请参见您所使用器件的数据手册。由编译器生成的运行时启动代码（见**第 5.10.1 节 “运行时启动代码”**）在默认情况下将提供执行振荡器校准的代码。如果需要，可以使用选项禁止它，请参见**第 4.8.50 节 “--RUNTIME：指定运行时环境”**。

3.5.9 如何将变量放入 PIC18 的外部程序存储器？

如果所要做的只是将只读变量放入程序存储器，则可以使用 const 对它们进行限定，请参见**第 5.5.3 节 “程序空间中的变量”**。如果希望变量位于外部程序存储器中，则使用 far 限定符，并使用 --RAM 选项指定存储器，请参见**第 4.8.48 节 “--RAM：调整 RAM 范围”**。编译器允许对使用 far 限定的变量进行修改。请注意，访问这些变量需要的时间将多于内部数据存储器中的变量。外部存储器的访问模式可以通过选项指定，请参见**第 4.8.26 节 “--EMI：选择外部存储器接口工作模式”**。

3.5.10 如何在代码中实现延时？

如果需要精确的延时，或者如果在延时期间可以执行其他任务，则使用定时器产生中断是最好的办法。

如果这些在您的代码中都不是问题，则可以使用编译器内建的延时伪函数：_delay、__delay_ms 或 __delay_us，请参见**附录 A “库函数”**。它们都会扩展为行内汇编指令或（嵌套）指令循环，并将消耗指定数量的周期或时间。延时参数必须为常量，并对于 PIC18 器件约小于 179,200，对于其他器件约小于 50,659,000。

请注意，这些代码序列将仅使用NOP指令和/或构成循环的指令。这些伪函数仅用于PIC18的替代版本 （如 _delaywdt）还可能使用 CLRWDT 指令。另请参见**附录 A “库函数”**。

3.5.11 如何才能循环移位一个变量？

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。由于 PIC 器件具有一条循环移位指令，编译器将会查找实现循环移位的代码表达式（使用移位和逻辑或运算），并在可能时在生成的输出中使用循环移位指令，请参见**第 5.6.2 节 “循环移位”**。

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。

- *免费、标准和专业模式之间有何差别？*
- *如何让代码更小？*
- *如何减少 RAM 使用量？*
- *如何让代码更快？*
- *编译器如何在存储器中放置所有内容？*
- *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？*
- *如何利用器件上的所有 RAM 存储区？*
- *如何利用增强型中档 PIC 器件上的线性存储器？*
- *编译器支持哪些器件？*
- *如何知道编译器生成哪些代码？*
- *如何确定警告 / 错误消息的含义？*
- *如何确定某个函数有多大？*
- *如何知道每个函数在使用哪些资源？*
- *如何确定变量和函数定位到何处？*
- *为什么某些对象被定位到保留的存储器中？*
- *如何知道还有多少存储器可用？*
- *如何编译库？*
- *MPLAB IDE 调试编译有何差别？*
- *如何防止删除未用函数？*
- *如何在项目中使用库文件？*
- *编译器采用了哪些优化？*

3.6.1 免费、标准和专业模式之间有何差别？

这些模式（见**第 1.2 节“编译器说明和文档”**）主要差别在于编译时执行的优化。在免费（原先称为简化）和标准模式下工作的编译器可以针对专业模式支持的所有相同器件进行编译。在免费和标准模式下编译的代码可以使用所选器件的所有可用存储器。差别在于所生成的编译器输出代码的大小和速度。免费模式输出代码的效率将远低于标准模式生成的输出代码，而后者的效率又会低于专业模式下生成的输出代码。

所有这些模式均使用 OCG 编译器框架，所以整个 C 程序将在一步中完成编译，源代码不需要很多的非标准扩展。

有少量的命令行选项在免费模式下会被禁止，但这些选项与代码功能无关；它们仅仅涉及到编译器的执行方式。大多数客户永远不需要使用这些选项。这些选项包括 `--GETOPTION`（**第 4.8.32 节“--GETOPTION：获取命令行选项”**）和 `--SETOPTION`（**第 4.8.53 节 “--SETOPTION：为应用程序设置命令行选项”**）。

© 2013 Microchip Technology Inc.

3.6.2 如何让代码更小？

有许多方法可以实现这个目标，但结果会因项目而不同。使用汇编列表文件（见**第 6.6 节“汇编列表文件”**）来观察编译器生成的汇编代码，以验证以下技巧是否对于您的代码有用。

使用可能的最小数据类型，因为访问它们需要的代码较少（这还可以减少 RAM 使用量）。请注意，该编译器具有 `bit` 类型和非标准的 24 位整型（`short long`）。关于所有数据类型和大小，请参见**第 5.4 节 “支持的数据类型和变量”**。

此外，还存在两种大小的浮点型，它们将在同一节中讨论。如果可能，请尽量避免使用浮点型。可以考虑编写定点算术运算代码。

如果可能，请使用无符号类型，而不是有符号类型；特别是如果是在具有混合类型和大小的表达式中使用它们。在可能时，尽量避免对具有混合大小的操作数进行操作的操作符。

每次编写循环或条件代码时，请使用 “强”停止条件，即以下代码：

```
for(i=0; i!=10; i++)
```

优于：

```
for(i=0; i<10; i++)
```

相等性检查（`==` 或 `!=`）的实现效率通常高于较弱的 `<` 比较。

在某些情况下，使用递减至零的循环计数器的效率会高于从零开始递增相同迭代次数的计数器。如果循环索引是一个字节宽的类型，则更有可能出现这种情况。所以，可以将以上代码重写为：

```
for(i=10; i!=0; i--)
```

更改函数参数的顺序，使第一个参数为字节大小，可能会具有一点小的优势。如果第一个参数为字节大小，则会使用一个寄存器。例如，可以考虑使用：

```
char calc(char mode, int value);
```

它优于

```
char calc(int value, char mode);
```

确保已使能所有优化，请参见**第 4.8.42 节 “--OPT：调用编译器优化”**。了解编译器会执行哪些优化（见**第 5.13 节 “优化”**和**第 6.5 节 “汇编代码级优化”**），从而可以利用它们，而无需浪费时间在 C 代码中手动执行编译器已处理的优化，例如，不要将乘 4 运算转换为左移 2 位运算，因为已检测到这种优化。

3.6.3 如何减少 RAM 使用量？

使用可能的最小数据类型（这也可以减小代码长度，因为访问它们需要的代码会较少）。请注意，该编译器具有 bit 类型和非标准的 24 位整型 （short long）。关于所有数据类型和大小，请参见**第 5.4 节 “支持的数据类型和变量”**。此外，还存在两种大小的浮点型，它们将在同一节中讨论。

考虑使用 auto （自动）变量，而不是全局或 static （静态）变量，因为这些变量有可能可以共用分配给其他在同一时间处于不活动状态的 auto 变量的存储器。自动变量的存储区分配到已编译堆栈中，如**第 5.5.2.2 节 “自动变量分配和访问”**所述。

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。在传递较大结构时尤其如此，但即使在传递 long 变量时，也可以实现 RAM 节省。

在整个程序中不需要进行更改的对象可以通过使用 const 限定符放入程序存储器，请参见**第 5.4.7.1 节 “const 类型限定符”**和**第 5.5.3 节 “程序空间中的变量”**。这可以释放宝贵的 RAM，但会降低执行速度。

确保已使能所有优化，请参见**第 4.8.42 节 “--OPT：调用编译器优化”**。了解编译器会执行哪些优化 （见**第 5.13 节 “优化”**），从而可以利用它们，而无需浪费时间在 C 代码中手动执行编译器已处理的优化。

3.6.4 如何让代码更快？

在很大程度上，代码长度越短则代码速度越快，所以减小代码长度的工作通常可以减少执行时间，请参见**第 3.6.2 节 “如何让代码更小？”**。另请参见**第 3.6.6 节 “如何让中断程序更快？”**。但是，有一些序列可以通过一些方式在代码长度增大的代价下提高速度。

编译器优化设置中有一项用于优化速度（替代设置是优化空间），所以请确保选中该设置，请参见**第 4.8.42 节 “--OPT：调用编译器优化”**。这将使用在某些情况下速度更快但大小较大的替代输出。

一般情况下，可以从执行速度方面获得的最大好处来自于在项目中使用的算法。确定程序的哪些部分需要快速。查找可能为线性搜索数组的循环，然后选择一种替代搜索方法，例如哈希表和函数。在需要重新计算结果的情况下，考虑是否可以缓存它们。

3.6.5 编译器如何在存储器中放置所有内容？

在大多数情况下，与代码和数据关联的汇编指令和伪指令会被分组到一些称为 psect 的段中，然后它们会被定位到代表器件存储器的容器中。**第 5.15.1 节 “程序段”**给出了该过程的介绍性说明。例外情况是绝对变量（见**第 5.5.4 节 “绝对变量”**），它们会被放置在定义它们时的特定地址处，不放入 psect。

© 2013 Microchip Technology Inc.

3.6.6 如何让中断程序更快？

对于所有中断代码，考虑采用**第 3.6.2 节“如何让代码更小？”**中提出的建议（代码长度）。代码长度越短，代码速度往往就越快。

除了您在 ISR 中编写的代码之外，编译器还会生成用于现场切换的代码。它紧接在发生中断之后和中断返回之前执行，所以在处理中断所需的时间中必须包含它，请参见**第 5.9.3 节“现场切换”**。该代码是最优的，即该代码只会保存在 ISR 中使用的寄存器。因此，在 ISR 中使用的寄存器越少，即意味着要执行的现场切换代码可能越少。

中档器件只有几个由编译器使用的寄存器，所以只有很少的现场切换代码。针对增强型中档器件进行编译时，需要考虑保存的寄存器会更少。PIC18 器件可以从上面的建议中受益最大，因为它们在生成代码中使用更大的一组寄存器，请参见**第 5.7 节 “寄存器使用”**。

通常，较简单的代码需要的资源会少于较复杂的表达式。使用汇编列表文件查看编译器在中断代码中使用了哪些寄存器，请参见**第 6.6 节 “汇编列表文件”**。

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用的变量，总是使用 `volatile` 限定符（见**第 5.4.7.2 节“volatile 类型限定符”**），请参见**第 3.5.5 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 `const` 限定的对象将位于程序存储器中；其他对象将放在数据存储器中。**第 5.5.3.1 节“const 变量的长度限制”**讨论了程序存储器对象大小。数据存储器中的对象可大致分为自动和非自动两种，**第 5.5.2.3 节 “自动变量的长度限制”**和**第 5.5.2.1.2 节 “非自动变量长度限制”**分别讨论这些对象的大小限制。

3.6.8 代码将被应用哪些优化？

OCG 编译器中的优化可以大致上分为 C 代码级和汇编代码级的优化。**第 5.13 节 “优化”**对它们进行了介绍，并且可以通过**第 4.8.42 节“--OPT：调用编译器优化”**中详细说明的选项进行控制。

3.6.9 如何利用器件上的所有 RAM 存储区？

编译器会自动使用您为之编程的器件上的所有可用 RAM 存储区。只有在您希望更改默认存储器分配时，才需要执行操作。一些特殊的存储区限定符（见**“--RAM=default,+20000-2FFFF”**）和一个指示如何解释这些限定符的选项（见**第 4.8.16 节“--ADDRQUAL：设置编译器对于存储器限定符的响应”**）用于手动分配变量。

请注意，无法保证器件上的所有存储器都可以获得利用，因为数据和代码都以段或 psect 的形式进行打包。

3.6.10 如何利用增强型中档 PIC 器件上的线性存储器？

线性寻址模式是一种将分区的数据存储器作为一个连续线性块进行访问的方法，请参见**第 5.5.1 节 “地址空间”**。线性存储器的使用是完全自动的。大小大于数据存储区的对象可以使用一般方式定义，并使用线性寻址模式进行访问，请参见**第 5.5.2.3 节 “自动变量的长度限制”**和**第 5.5.2.1.2 节 “非自动变量长度限制”**。如果在存储器中的某个特定位置上定义了绝对对象，则可以使用线性地址或（如果您更喜欢）常规的分区地址，请参见**第 5.5.4.1 节 “数据存储器中的绝对变量”**。

3.6.11 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。要确定您的编译器是否支持某种器件，可以做几件事情，另请参见**第 5.3.1 节 “器件支持”**。

- 编译器的 docs 目录中提供了 HTML 列表。在您喜欢的网络浏览器中打开它们。它们名为 `pic_chipinfo.html` 和 `pic18_chipinfo.html`。
- 在命令行上使用 `-CHIPINFO` 选项运行编译器驱动程序（见**第 4.2 节 “调用编译器”**），请参见**第 4.8.21 节 “--CHIPINFO：显示支持器件的列表”**。屏幕上将打印所有器件的完整列表。

3.6.12 如何知道编译器生成哪些代码？

汇编列表文件（见**第 6.6 节 “汇编列表文件”**）会显示几乎整个程序的汇编输出，包括在程序中链接的库程序，以及大量的运行时启动代码，请参见**第 5.10.1 节 “运行时启动代码”**。如果使用的是 MPLAB IDE，则默认情况下会生成该列表文件。如果使用的是命令行，使用选项 `--ASMLIST` 将可以为您生成该文件，请参见**第 4.8.17 节 “--ASMLIST：生成汇编列表文件”**。汇编列表文件将具有 `.lst` 扩展名。

该列表文件会显示汇编指令、一些汇编伪指令和关于程序的信息，例如调用图（见**第 6.6.6 节 “调用图”**）、指针引用图（见**第 6.6.5 节 “指针引用图”**）和每个函数的信息。如果使能了汇编优化器，则该列表文件中将不会显示所有汇编伪指令（它们在中间汇编文件中生成）。如果希望查看由编译器生成的所有汇编伪指令，请暂时禁止汇编优化器（**第 4.8.42 节 “--OPT：调用编译器优化”**）。

3.6.13 如何确定某个函数有多大？

某个函数的大小（对于该函数生成的汇编代码量）可以通过汇编列表文件（见**第 6.6 节 “汇编列表文件”**）或由编译器生成的 “funclist” 文件确定。最新的编译器定义了一个其所赋值等于函数大小的符号。该符号具有 `__size_of_`*`func`* 的形式，其中的 *`func`* 是函数的名称。该符号的单位将与特定器件的程序存储器的寻址能力相同：对于 PIC10/12/16 为字，对于 PIC18 为字节。此外，您还可以搜索标记函数开头和结尾的标号。函数以标号 `_`*`func`*: 开始（其中，*`func`* 是函数的名称），并在标号 `__end_of_`*`func`* 之前结束。例如，函数 `main` 可能具有关联的符号 `__size_of_main`、`_main` 和 `__end_of_main`。它们可以在汇编列表文件末尾的符号表中找到。

© 2013 Microchip Technology Inc.

函数、存储器位置和大小的列表在名为 funclist 的文件中提供。每个函数都具有类似于以下内容的行。

```
_main: CODE, 2012 0 30
```

这表示，与函数 main 关联的生成汇编代码被放在 CODE 链接器类中（见**第 6.4.9.3.3 节“class”**），位于地址空间号 0 的地址 2012（十进制）处（见**第 6.4.9.3.13 节“space”**），长度为 30 （同样为十进制）字 / 字节。**第 5.15.1 节 “程序段”** 给出了 psect 的简介。

3.6.14 如何知道每个函数在使用哪些资源？

在汇编列表文件中，提供了对于每个 C 函数（包括库函数）打印的信息，请参见**第 6.6 节“汇编列表文件”**。该信息指示函数使用了哪些寄存器，它调用了哪些函数 （在调用图中也可以找到该信息，见**第 6.6.6 节“调用图”**），以及它需要多少字节的数据存储器。请注意，函数使用的自动变量、参数变量和临时变量可能会与其他函数重叠，因为它们都由编译器放在编译堆栈中，请参见**第 5.5.2.2.1 节 “编译堆栈操作”**。

3.6.15 如何确定变量和函数定位到何处？

您可以通过汇编列表文件 （见**第 6.6 节 “汇编列表文件”**）或映射文件 （见**第 7.4 节 “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；汇编列表文件中会列出所有符号 （包括局部符号），但仅对于该列表文件代表的代码。（每个汇编模块都具有自己的列表文件。）

C 标识符和汇编代码中使用的符号之间存在映射，这些符号就是在这两个文件中显示的符号，请参见**第 5.12.3.1 节 “等效汇编符号”**。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。

3.6.16 为什么某些对象被定位到保留的存储器中？

存储器保留选项 （见**第 3.4.3.4 节 “如何阻止编译器使用某些特定的存储单元？”**）会调整与链接器所使用的类关联的地址范围。大多数变量和函数都放到 psect 中 （见**第 5.15.1 节 “程序段”**），它们会被链接到这些类范围内的任意位置，所以会受这些保留选项影响。

一些 psect 会被显式地放置在某个地址处，而不是链接到某个地址范围内的任意位置，例如，包含在复位时执行的代码的 psect 将总是链接到地址 0，因为它是为 8 位器件定义的复位位置。这种 psect 不会受 --ROM 选项影响，即使您使用它来保留存储器地址 0。包含与复位和中断关联的代码的 psect 可以使用 --CODEOFFSET 选项进行移动，请参见**第 4.8.22 节 “--CODEOFFSET：将程序代码偏移到某个地址”**。

检查汇编列表文件 （见**第 6.6 节 “汇编列表文件”**），确定包含对象和代码的 psect 的名称。检查映射文件中的链接器选项 （见**第 7.4 节 “映射文件”**），确定是否已显式地链接 psect，或者是否在某个类中将它们链接到任意位置。另请参见链接器选项 -p （**第 7.2.19 节 “-Pspec”**）和 -A （**第 7.2.1 节 “-Aclass =low-high,...”**）。

3.6.17 如何知道还有多少存储器可用？

虽然由编译器在编译之后打印的存储器摘要（见**第 4.8.56 节 “--SUMMARY：选择存储器摘要输出类型”**中的选项）或 MPLAB IDE 中提供的存储器计量器均可显示已用存储器量和仍然可用的存储器量，但这两个功能都不会指示该存储器是一个连续块还是拆分成许多小块。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（见**第 3.7.6 节 “如何解决“无法找到空间 ...”错误？”**）。

映射文件中的“UNUSED ADDRESS RANGES”（未用地址范围）部分（见**第 7.4.2.5 节 “未用地址范围”**）会指示每个链接器类中具体有哪些存储器仍然可用。如果存在存储分区或分页，则还会指示该类中最大的连续块。

3.6.18 如何在项目中使用库文件？

关于如何编译自己的库文件的信息，请参见**第 3.3.6 节 “如何编译库？”**。在您进行编译时，编译器会自动在编译过程中包含任何适用的标准库，所以永远不需要您来控制这些文件。

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。可以在文件列表中相对于源文件的任何位置指定这些库文件，但如果存在多个库文件，则将按命令行中指定的顺序搜索它们。如果要编译为中间文件，即使用 `-PASS1` 选项（见**第 4.8.45 节 “--PASS1：编译为 p 代码”**），则不需要指定 LPP 库。例如：

```
xc8 --chip=16f1937 main.c int.c lcd.lpp
```

如果要使用MPLAB X IDE来编译项目，则将库文件添加到将会显示在项目中的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。

3.6.19 编译器采用了哪些优化？

优化在 C 代码级和汇编代码级编译中采用。**第 5.13 节 “优化”**和**第 6.5 节 “汇编代码级优化”**分别对它们进行了介绍。**第 4.8.42 节 “--OPT：调用编译器优化”**介绍了用于控制优化的选项。

3.6.20 在选择调试器时，为什么会出现存储器不足错误？

如果使用硬件调试器工具（如 REAL ICE 或 ICD3），它们都需要用于板上调试执行程序的存储器。当使用编译器的 `--DEBUGGER` 选项（见**第 4.8.23 节 “--DEBUGGER：选择调试器类型”**）或 IDE 等效形式选择某个调试器时，将从您的项目的可用存储器中去除调试所需的存储器。请参见**第 3.5.3 节 “在编译为使用调试器时需要做些什么？”**。

© 2013 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已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？*
- *如何阻止产生虚假警告？*
- *为什么甚至无法使 LED 闪烁？*
- *如何知道堆栈是否溢出？*
- *如何解决 “无法找到空间 ...” 错误？*
- *如何解决 “无法生成代码 ...” 错误？*
- *如何解决 fixup 溢出错误？*
- *调用编译器*
- *使用中断时，什么原因会导致变量损坏和代码失败？*
- *为什么某些对象被定位到保留的存储器中？*

3.7.1 如何确定警告 / 错误消息的含义？

每条警告或错误消息都具有说明，并且可能还会提供可能触发此类错误的示例代码，在消息章节中列出了这些消息，请参见**附录 B “错误和警告消息”**。编译器会打印每条消息，并在括号中包含一个惟一的 ID 号。使用该编号可以在手册中查找消息。该编号还使您可以通过使用选项和 pragma 伪指令来控制消息的行为，请参见**第 4.6.5 节 “改变消息的行为”**。

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，请参见**第 4.6 节 “编译器消息”**。如果在 MPLAB 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 语句横跨在源文件的多行上。编译器有可能无法确定错误所在行，直到它扫描后面的语句之后才能确定。如在以下代码中，

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

赋值语句缺少分号这一错误将会标记在包含 `if()` 语句的下一行上。

在其他情况下，错误可能来自汇编器，而不是代码生成器。如果汇编代码是从 C 源文件获得的，则编译器会尝试指示 C 源文件中对应于错误汇编代码的行。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。在这两种情况下，请记住错误中的信息涉及到的是汇编代码而不是 C 代码中的问题。

最后，还有一些根本不涉及到任何特定代码行的错误。编译器选项错误或链接器错误就是这些错误的示例。如果程序定义了太多变量，则不存在引发错误的特定代码行；程序作为一个整体使用了太多的数据。请注意，在某些情况下，可能会打印所处理的最后一个文件和源代码的名称和行号，即使该代码并不是错误的直接来源。

要确定产生错误或警告的应用程序，请记下消息中打印的惟一编号（见**第 4.6.1 节“消息概述”**）并查看手册的消息部分，请参见**附录 B“错误和警告消息”**。在每条消息说明的顶部（在最右侧的括号内）是产生该消息的应用程序的名称。知道产生错误的应用程序后，就可以更容易地追查问题。**第 4.3 节“编译序列”**中列出了编译器应用程序名称。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，请参见**第 6.6 节“汇编列表文件”**。关于链接器尝试将对象定位到何处的信息，请参见**第 7.4 节“映射文件”**中讨论的映射文件。

3.7.3 如何阻止产生虚假警告？

警告指示可能会导致代码失败的情况。在许多情况下，代码是有效的，警告是多余的。请总是检查您的代码，确认它不是错误的可能来源；在确实如此的情况下，有几种方法可以隐藏警告。

- 可以调整警告级别阈值，从而仅打印达到一定重要性的警告，请参见**第 4.6.5.1 节“禁止消息”**
- 可以禁止带有指定 ID 的所有警告
- 在某些情况下，可以对一些特定的源代码行使用 pragma 伪指令来禁止带有指定 ID 的警告，请参见**第 5.14.4.11 节“#pragma warning 伪指令”**

3.7.4 为什么甚至无法使 LED 闪烁？

即使您已经设置了 TRIS 寄存器并向端口写入某个值，还是有一些情况会使这种看似简单的程序无法工作。

- 请确保器件的配置寄存器已正确设置，请参见**第 5.3.5 节“配置位访问”**。请确保明确指定这些寄存器中的每一位，而不只是将它们保留为其默认状态。器件数据手册中介绍了所有配置功能。举例来说，如果指定振荡器源的配置位是错误的，器件时钟可能甚至无法运行。
- 如果使用的是内部振荡器，则除了配置位之外，可能还需要初始化一些 SFR 来设置振荡器频率和模式，请参见**第 5.3.6 节“从 C 代码中使用 SFR”**和您所使用器件的数据手册。
- 在配置位中关闭看门狗定时器，或在代码中清零看门狗定时器（见**附录 A“库函数”**），从而使器件不会发生复位。如果器件发生复位，则可能永远无法到达程序中用于使 LED 闪烁的代码行。关闭可能会导致器件复位的所有其他功能，直到测试程序正常工作为止。
- 端口位使用的器件引脚经常会与其他外设复用。例如，引脚可能连接到端口中的某个位，或者它可能是模拟输入，或者它可能是比较器的输出。如果连接到 LED 的引脚未在内部连接到您要使用的端口，则 LED 永远不会按预期的那样工作。器件数据手册中给出的端口功能表会显示每个引脚的其他用途，这可以帮助您确定引脚与哪个外设复用。

© 2013 Microchip Technology Inc.

- 请确保不存在“读 - 修改 - 写”问题。如果您使用的器件不具有单独的“锁存”寄存器（中档 PIC 器件就是这种情况），则可能会出现该问题，特别是在端口输出驱动大负载（如 LED）的情况下。您可能会看到，将某个位置 1 会将另一个位清零或者发生一些其他的异常事件。通过使用一个临时变量来创建自己的锁存器。对锁存器变量进行修改，而不是直接读取和写入端口。完成修改之后，将锁存器内容作为一个整体复制到端口中。这意味着您永远不会通过读取端口来对其进行修改。更多详细信息，请查看器件文献。

3.7.5 如何知道堆栈是否溢出？

8 位 PIC 器件具有一个容量有限的硬件堆栈，它仅用于函数（和中断函数）返回地址，请参见**第 5.3.4 节 “堆栈”**。如果函数调用和中断的嵌套太深，堆栈将发生溢出（发生折回并覆盖先前的条目）。然后，代码会在之后当它访问损坏的返回地址时发生失败（有时会是在调用序列的很久以后）。

编译器会尝试跟踪堆栈深度，并在需要时切换为不需要硬件堆栈的调用方法（仅限 PIC10/12/16 器件）。您可以在一定程度上控制在堆栈深度已明显溢出时发生的情况，请参见**第 4.8.50 节 “--RUNTIME：指定运行时环境”**和 `stackcall` 子选项。

调用图中会显示编译器已确定的调用层次结构和深度。该图显示在汇编列表文件中。要理解该图中的信息，请参见**第 6.6.6 节 “调用图”**。

由于编译器无法确定程序的运行时行为，它只能假设最坏的情况，并可能会在并未发生溢出的情况下报告可能发生溢出。但是，如果程序完全以 C 语言编写，则不会漏检任何溢出。编译器不会考虑使用堆栈的汇编代码，这一点必须考虑在内。

3.7.6 如何解决 “无法找到空间 ...”错误？

该消息具有许多不同变化形式，但本质上都在暗示一种类似的情况。它们都涉及到没有足够大的可用空间来放置数据块或指令块。由于存储器分页、分区或其他碎片，在看似有足够的存储器剩余时，也可能会发出该消息。关于特定错误编号的更多信息，请参见**附录 B “错误和警告消息”**。

3.7.7 如何解决 “无法生成代码 ...”错误？

这是一条捕获所有（catch-all）消息，如果编译器已用尽了所有可能方法来编译一条 C 语言表达式，则会产生该消息，请参见**附录 B “错误和警告消息”**。通常它并不表示代码中存在错误。无法编译该代码可能是由于编译器的缺陷，或者表达式需要的寄存器或资源多于代码中该点可用的寄存器或资源。这通常发生在低档器件上。在任何情况下，简化引起问题的表达式或将语句拆分成几条较小的语句，通常就可以继续编译。您可能需要使用另一个变量来保存复杂表达式的中间结果。

3.7.8 如何解决 fixup 溢出错误？

fixup 是将符号引用替换为实际地址的过程，如果分配给该符号的地址太大，导致无法装入汇编指令的地址字段中，则它会发生溢出。大多数 8 位 PIC 汇编指令会指定一个文件地址，它是一个用于进入当前选定存储区的偏移量。如果使用这些指令指定完整的无掩码地址，链接器会无法将大地址值编码到指令中，并产生该错误。例如，中档器件指令仅允许 0 至 0x7F 范围的文件地址。但是，如果这种器件具有 4 个 RAM 数据存储区，变量的地址范围可以为 0 至 0x1FF。举例来说，如果使用其中一条指令指定某个将定位到地址 1D0 处的变量的符号，当符号替换为其最终值时，该值无法装入指令的地址字段中。

在大多数情况下，这些错误都是由手写的汇编代码引起的。在编写汇编代码时，编程人员需要负责添加用于选择目标存储区的代码，然后对要在指令中使用的地址进行掩码处理，请参见**第 3.4.7.5 节 “编写汇编代码时，必须控制哪些方面？”**。这是一个涉及到汇编指令的问题，您需要找到出错的指令之后才能继续，记住这一点很重要。关于如何追踪引起问题的指令的具体详细信息，请参见**附录 B “错误和警告消息”**中的相关错误编号。

3.7.9 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。更多信息，请参见**第 3.5.5 节 “如何在中断与主干代码之间共享数据？”**。

© 2013 Microchip Technology Inc.

第 4 章 XC8 命令行驱动程序

4.1 简介

命令行驱动程序的名称为 xc8。可以调用 XC8 来执行编译的所有方面，包括 C 代码生成、汇编和链接步骤。即使使用 IDE 来协助进行编译， IDE 最终还是会调用 xc8。

虽然可以从命令行上显式地调用内部的编译器应用程序，但 xc8 驱动程序是使用编译器的建议方式，因为它可以隐藏所用的所有内部应用程序的复杂性，并为所有编译步骤提供一致的界面。

本章介绍驱动程序在编译过程中执行的步骤、驱动程序可以接受和生成的文件，以及控制编译器操作的命令行选项。此外，还介绍了这些命令行选项与 MPLAB IDE *Build Options*（编译选项）对话框中控件之间的关系。

《MPLAB XC8 C 编译器用户指南》的这一章讨论以下主题：

- 调用编译器
- 编译序列
- 运行时文件
- 编译器输出
- 编译器消息
- XC8 驱动程序选项
- MPLAB IDE V8 Universal Toolsuite 等效选项
- MPLAB X Universal Toolsuite 等效选项

4.2 调用编译器

本节说明如何在命令行上调用 xc8，以及它可以读取的文件。

4.2.1 驱动程序命令行格式

xc8 具有以下基本命令格式。

```
xc8 [options] files [libraries]
```

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，或者在执行应用程序时指定了完整路径。安装编译器时，可以通过在安装过程中的适当时候选择 *Add to environment*（添加到环境中）复选框，将编译器的位置添加到搜索路径中。

习惯上在文件名之前声明 *options*（通过一个前导短划线 “-”或双短划线 “–”标识）。但是，这不是强制性的。

第 4.7 节 “XC8 驱动程序选项”给出了选项的格式，以及选项的相应说明。

files 可以是各种各样的 C 和汇编源文件，以及预编译的中间文件，例如可重定位目标（.obj）文件或 p 代码（.p1）文件。虽然文件的列出顺序并不重要，但它可能影响代码或数据在存储器中出现的顺序，并且可能影响一些输出文件的名称。

除了标准 C 库之外，*libraries* 是用户定义的目标代码库文件或 p 代码库文件的列表，代码生成器（对于 p 代码库）或链接器（对于目标代码库）会在其中进行搜索。这些文件的顺序将决定对它们进行搜索的顺序。习惯上将 *libraries* 列表插入到源文件名列表之后。但是，这不是强制性的。

如果使用 make 系统编译代码，则建议熟悉**第 4.3.3 节 “多步编译”**介绍的独特的中间 p 代码文件格式。目标文件很少用于 MPLAB XC8 C 编译器，除非汇编源代码模块处于项目中。

4.2.1.1 长命令行

如果通过一个命令文件向 xc8 驱动程序传递选项，驱动程序将能够处理超出任何操作系统限制的命令行。命令文件通过 @ 符号指定，它后面应紧随（即，没有中间空格字符）包含要传递给驱动程序的命令行参数的文件的名称。

每个命令行参数必须通过一个或多个空格隔开，并且可以横跨多行，即通过使用空格和反斜杠字符来分隔每一行。文件可以包含空行，驱动程序将简单地跳过它们。

使用命令文件意味着编译器选项和源代码文件名可以永久存储，以备将来引用，而又没有创建 make 实用程序的复杂性。

在以下示例中，在文本编辑器中构造了命令文件 xyz.cmd，其中同时包含了编译项目所需的选项和文件名。

```
--chip=16F877A -m \
--opt=all -g \
main.c isr.c
```

保存它之后，就可以使用以下命令调用编译器：

```
xc8 @xyz.cmd
```


© 2013 Microchip Technology Inc.

4.2.2 环境变量

在 Windows 环境中，编译器将使用注册表来存储与编译器安装目录和激活细节，以及其他配置设置相关的信息。无论是在命令行上还是从 IDE 中运行编译器，都需要该信息。

在 Linux 和 Apple OS X 环境下，注册表将替换为存储相同信息的 XML 文件。

在非 Windows 主机上，编译器将按以下方式搜索 XML 文件：

1. 编译器会查找是否存在名为 XC_XML 的环境变量。如果存在，该变量应包含 XML 文件的完整路径 （包括文件名）。
2. 如果该变量未定义，则编译器会搜索名为 HOME 的环境变量。该变量通常包含用户主目录的路径。编译器会在 HOME 变量所指示的目录中查找名为 .xc.xml 的 XML。
3. 如果 HOME 环境变量未定义，编译器将尝试打开文件 /etc/xc.xml。
4. 如果这些方法均未找到 XML 文件，则会生成错误。

在命令行上运行编译器时，您可能希望设置 PATH 环境变量。这使您可以无需随驱动程序名称一起指定完整的编译器路径，即可运行编译器驱动程序。请注意，由 PATH 变量指定的目录仅用于查找编译器驱动程序。驱动程序运行之后，它将使用注册表或 XML 文件 （如上所述）来查找内部编译器应用程序，例如解析器、汇编器和链接器等。在 PATH 变量中指定的目录不会覆盖注册表或 XML 文件中包含的信息。MPLAB IDE 允许通过对话框选择编译器，并且编译器的执行并不依赖于 PATH 变量。

4.2.3 输入文件类型

xc8 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。表 4-1 列出了可识别的文件类型。从编译器的角度来看，扩展名的字母大小写形式并不重要，但大多数操作系统 shell 都区分大小写。

表 4-1： XC8 输入文件类型

文件类型	含义
.c	C 源文件
.p1	p 代码文件
.lpp	p 代码库文件
.as 或 .asm	汇编源文件
.obj	可重定位目标代码文件
.lib	可重定位目标库文件
.hex	Intel HEX 文件

举例来说，这意味着 C 源文件必须具有 .c 扩展名。汇编器文件可以使用 .as 或 .asm 扩展名。

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。这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。

源文件是包含程序的全部或部分程序的文件。它们可能包含 C 代码，以及预处理器伪指令和命令。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。所有的预处理器伪指令和命令（一些用于调试的命令除外）都已从这些文件中删除。然后，这些模块将传递给其余的编译器应用程序。因此，模块可能是几个源文件和头文件的合并结果。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。

© 2013 Microchip Technology Inc.

4.3 编译序列

在编译项目时，需要调用许多内部应用程序来完成这项工作。本节着眼于这些内部应用程序在何时执行，以及它如何与多个源文件的编译过程发生联系。如果您使用 make 系统来编译项目，应特别注意本节。

4.3.1 编译器应用程序

图 4-1 显示了主要的内部编译器应用程序和文件。

您可以将底层大框视为代表整个编译器，它通过命令行驱动程序 xc8 进行控制。只知道 C 源文件（显示在最左侧）被传递给编译器，并生成结果输出文件（此处显示为最右侧的 HEX 和 COFF 调试文件），您可能就会满意；但是，在内部会生成许多应用程序和临时文件。了解编译器的内部操作虽然并非必需，但有助于工具的使用。

为了简化编译器设计，一些内部应用程序提供了 PIC18 和 PIC10/12/16 两种形式。编译过程将基于目标器件执行相应的应用程序。事实上，xc8 驱动程序会将编译命令委托给两个命令行驱动程序之一：PICC 或 PICC18。该操作是透明的，可以将 xc8 视为完成所有工作的 “驱动程序”。

驱动程序将调用所需的编译器应用程序。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。

图 4-1： 编译器应用程序和文件

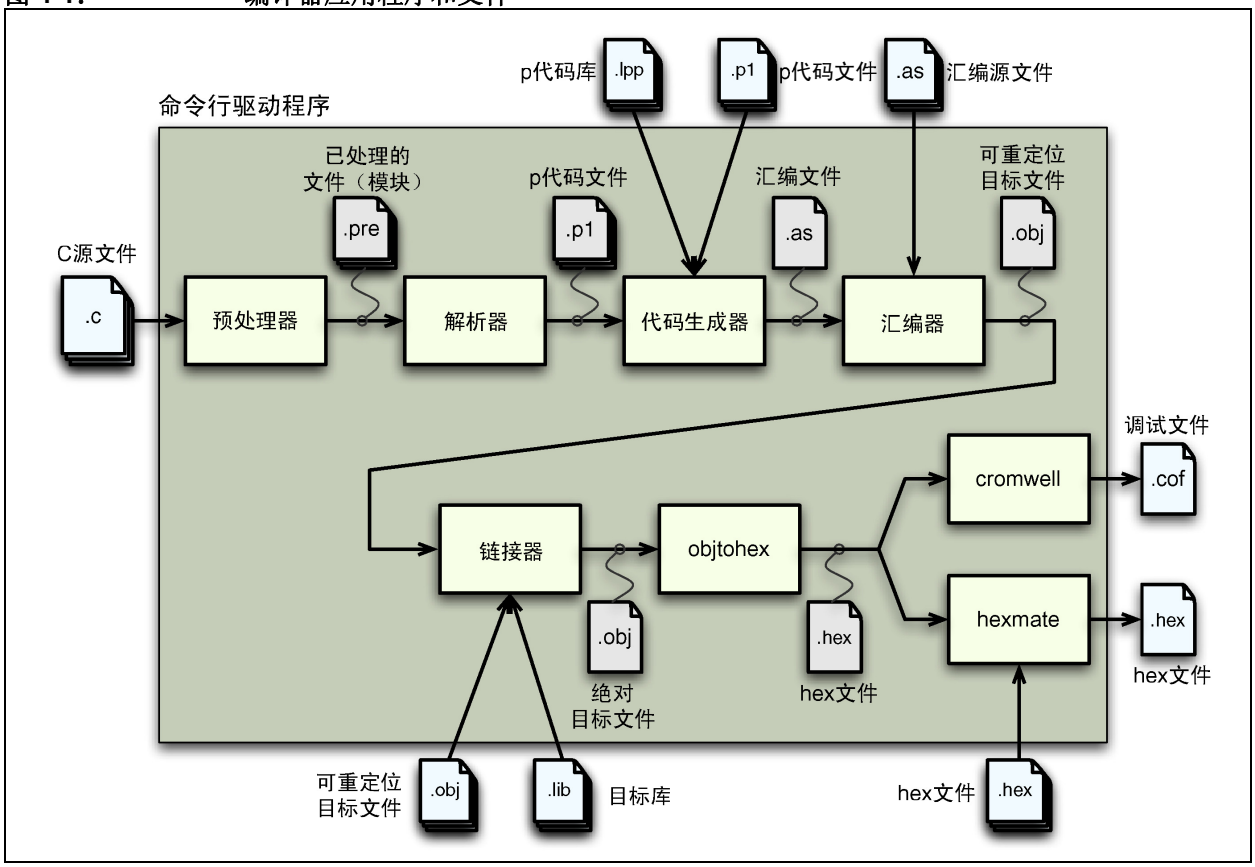


表 4-2 列出了编译器应用程序。所显示的名称是可执行文件的名称，可以在编译器的安装目录下的 bin 目录中找到它们。

表 4-2： 编译器应用程序名称

名称	说明
xc8（调用 PICC 或 PICC18）	命令行驱动程序；编译器的接口
CLIST	文本文件格式化程序
CPP	C 预处理器
P1	C 代码解析器
CGPIC 或 CGPIC18	代码生成器 （基于目标器件）
ASPIC 或 ASPIC18	汇编器 （基于目标器件）
HLINK	链接器
OBJTOHEX	用于创建 HEX 文件的转换实用程序
CROMWELL	调试文件转换器
HEXMATE	HEX 文件实用程序
LIBR	库管理器
DUMP	目标文件查看器
CREF	交叉引用实用程序

举例来说，C 源文件（.c 文件）先传递给 C 预处理程序 CPP。该应用程序的输出为 .pre 文件。然后，这些文件会传递给解析器应用程序 P1，它会生成扩展名为 .p1 的 p 代码文件输出。应用程序按照指定的顺序执行，并使用临时文件将一个应用程序的输出传递给下一个应用程序。

编译器不仅仅可以接受 C 源文件。表 4-1 列出了所有可能的输入文件类型；在上图的顶部和底部可以看到这些文件，它们被传递给不同的编译应用程序。这些应用程序对它们进行处理，然后应用程序输出加入到图中所示的正常流程中。

例如，汇编源文件被直接传递给汇编器应用程序[1]，根本不通过代码生成器进行处理。汇编器输出（扩展名为 .obj 的目标文件）以正常的方式传递给链接器。可以看到，在命令行上提供的所有 p 代码文件 （.p1 扩展名）或 p 代码库 （.lpp 扩展名）最初都传递给代码生成器。

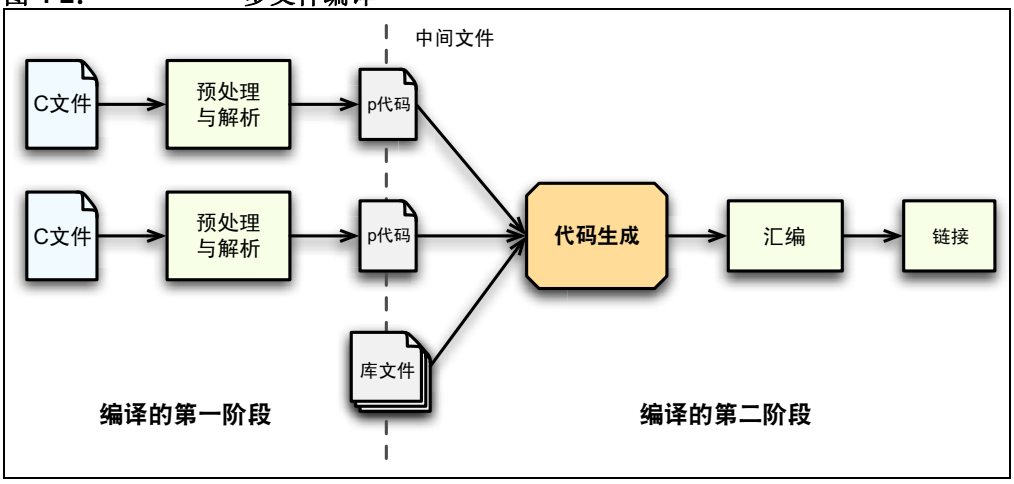
输入文件的其他示例包括目标文件 （.obj 扩展名）和目标库 （.lib 扩展名），这两种文件最初都传递给链接器；甚至是 HEX 文件 （.hex 扩展名），它传递给一个名为 HEXMATE 的实用程序，该实用程序在编译序列末尾运行。

该图中显示的一些临时文件实际上会被保留，可以在编译结束之后查看。此外，也有一些驱动程序选项可用于要求编译序列在特定应用程序之后停止，该应用程序的输出将成为最终的输出。

1. 如果选择了 -P 选项，汇编文件将先进行预处理，然后再传递给汇编器。

© 2013 Microchip Technology Inc.

图 4-2：　　多文件编译



4.3.2　　单步编译

图 4-1 显示了由每个应用程序生成的文件，以及这些应用程序的执行顺序。但是，其中并未显示在编译多个源文件时，如何执行这些应用程序。

假设有两个 C 源文件构成一个完整的项目，并且需要进行编译，如图 4-2 所示。如果这两个文件名为 `main.c` 和 `io.c`，可以使用单条命令对它们进行编译，例如：

```
xc8 --chip=16F877A main.c io.c
```

该命令将编译两个源文件，一直到产生最终输出，但在内部可以将该编译视为包含两个阶段。

第一阶段涉及到独立地处理每个源文件，并为每个源文件生成某种形式的中间文件。第二阶段涉及到组合所有这些中间文件，并通过进一步处理来形成最终的输出。中间文件是一个生成的特定临时文件，标志着编译的第一和第二阶段之间的中点。

`xc8` 使用的中间文件是由解析器生成的 p 代码（`.p1` 扩展名）文件输出，所以将对于每个 C 源文件生成一个 p 代码文件。如图中所示，需要先执行 `CPP`，然后再执行 `P1` 来形成该中间文件。（为清晰起见， `CPP` 和 `P1` 应用程序在图中使用同一个块表示。）

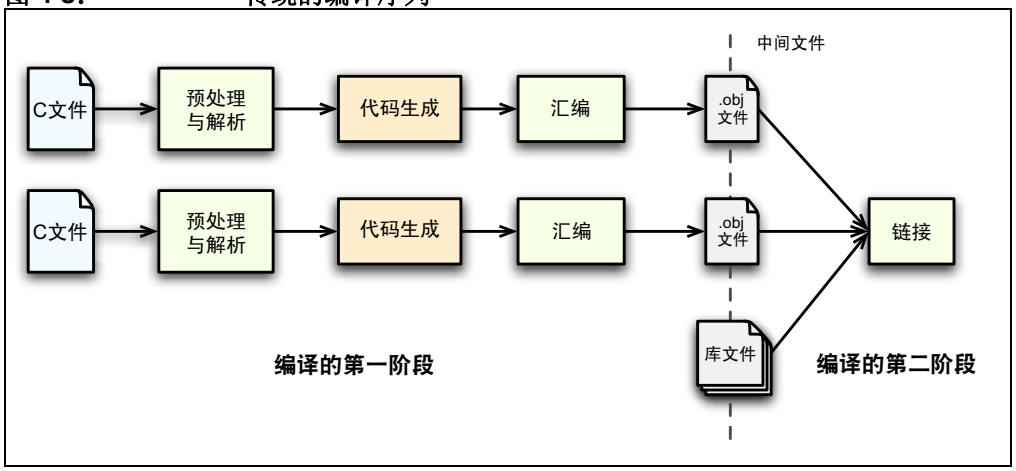
在第二阶段中，代码生成器会读取所有中间 p 代码文件，并生成单个汇编文件输出，然后将其传递给后续应用程序来生成最终的输出。

这种编译方法的理想特性就是代码生成器 （它是从 C 语言域变换到汇编代码域的主要应用程序）通过中间文件来查看整个项目的源代码。

传统编译器总是使用汇编器的目标文件输出作为中间文件。然后，由链接器对这些中间目标文件进行组合并进行进一步处理，以形成最终的输出。图 4-3 显示了这种编译方法，并且显示了代码生成器对于每个源文件执行一次。因此，代码生成器只能分析包含在当前正在编译的源文件中的那一部分项目代码。 MPLAB XC16 和 XC32 编译器以这种方式工作。

对MPLAB XC8 C编译器使用目标文件作为中间文件格式会使编译器用于优化代码的许多功能落空。如果使用 make 系统来编译项目，请总是使用 p 代码文件作为中间文件格式。

图 4-3：　　传统的编译序列



编译混合类型的文件时，仍然可以仅调用一次编译器驱动程序而实现编译。如**第 4.3 节“编译序列”**所述，驱动程序会将每个输入文件传递给相应的编译器应用程序。

例如，需要编译文件 `main.c`、`io.c`、`mdef.as` 和 `c_sb.lpp`。要在单个步骤中执行该操作，可以使用以下命令行。

```
xc8 --chip=16F877A  main.c  io.c  mdef.as  c_sb.lpp
```

如图 4-1 和图 4-2 所示，两个 C 文件（`main.c` 和 `io.c`）将被编译为中间 p 代码文件；这些文件与 p 代码库文件（`c_sb.lpp`）一起被传递给代码生成器。代码生成器的输出和汇编源文件 （`mdef.as`）将被传递给汇编器。

驱动程序将重新编译所有源文件，无论它们自上次编译以来是否发生了更改。要实现增量编译，必须采用 IDE （如 MPLAB® IDE）和 make 实用程序。另请参见**第 4.3.3 节“多步编译”**。

除非另外指定，否则将生成 HEX 文件和 Microchip COFF 文件作为最终的输出。所有中间文件在编译完成之后都会保留，但大多数其他临时文件会被删除，除非使用 `--NODEL` 选项 （见**第 4.8.40 节 “--NODEL：不删除临时文件”**），该选项会保留除运行时启动文件之外的所有生成文件。请注意，某些生成文件可能位于不同于项目源文件的其他目录中。请参见**第 4.8.43 节“--OUTDIR：指定输出文件的目录”**和**第 4.8.41 节“--OBJDIR：指定中间文件的目录”**，它们都可以控制一些输出文件的目标。

© 2013 Microchip Technology Inc.

4.3.3 多步编译

make 实用程序和 IDE（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。

MPLAB IDE 了解 xc8 采用的不同编译序列，可以为您处理该过程。从 MPLAB IDE 中，您可以选择增量编译（Build Project（编译项目）图标），或完全重新编译项目（Clean and Build Project （清除并编译项目）图标）。

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，您可能希望生成中间文件来构造您自己的库文件。但是，xc8 能够在单个步骤中构造库，因此这通常不需要。关于库创建的更多信息，请参见**第 4.8.44 节 “--OUTPUT=type：指定输出文件类型”**。

选项 --PASS1（**第 4.8.45 节 “--PASS1：编译为 p 代码”**）用于指示编译器，编译应在执行解析器之后停止。这将会在编译成功完成之后留下 p 代码中间文件。

例如，文件 main.c 和 io.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：

```
xc8 --chip=16F877A --pass1 main.c
xc8 --chip=16F877A --pass1 io.c
xc8 --chip=16F877A main.p1 io.p1
```

代码生成器需要在单个步骤中编译与项目关联的所有 p 代码文件或 p 代码库文件，注意到这一点非常重要。使用 --PASS1 选项时，代码生成器不会被调用，因此以上命令行不会违反这项要求。

对MPLAB XC8 C编译器使用目标文件作为中间文件格式会使编译器用于优化代码的许多功能落空。如果使用 make 系统来编译项目，请总是使用 p 代码文件作为中间文件格式。

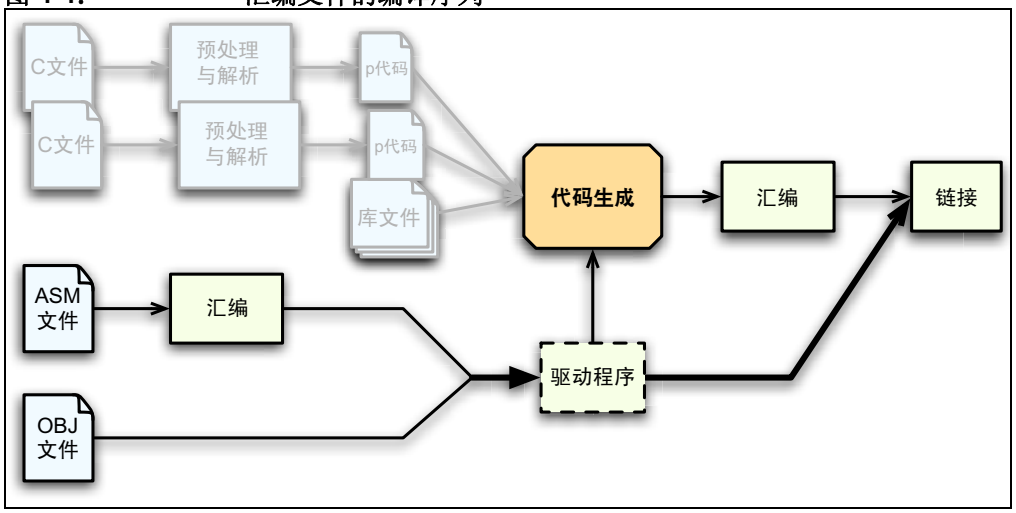
4.3.4 汇编源文件编译

由于代码生成器会执行传统上由链接器执行的许多任务，所以当项目中存在汇编源文件时，有可能发生问题。汇编文件传统上在 C 代码之后处理，但有必要先执行该操作，以便可以将汇编代码中包含的特定信息传送给代码生成器。

5.12.3 “汇编代码和 C 代码之间的交互”更详细地讨论了传递给代码生成器的特定信息。

当存在汇编源文件时，编译的顺序如图 4-4 所示。

图 4-4： 汇编文件的编译序列



先对所有汇编源文件进行汇编，以形成目标文件。然后，命令行驱动程序会扫描这些文件，以及属于项目一部分的所有其他目标文件，并在随后编译 C 文件时将该信息传递给代码生成器，如上文所述。

4.3.4.1 中间文件和汇编源文件

与汇编源文件相关联的中间文件格式与传统编译器中使用的相同；即目标文件（`.obj` 扩展名）。汇编文件永远不会传递给代码生成器，所以代码生成器技术不会改变编译这些文件的方式。

`-C` 选项（见**第 4.8.1 节 “-C：编译为目标文件”**）用于生成目标文件，并在汇编步骤之后停止编译。

4.3.5 printf 检查

在实际的代码生成阶段之前，会额外执行一次代码生成器。此次执行属于对 `printf` 库函数进行定制过程的一部分；更多详细信息，请参见**第 5.11.1 节 “printf 程序”**。

此次执行仅用于扫描 C 源代码中使用的 `printf` 占位符，如果在编译时选择了详细选项，就会看到代码生成器的此次执行，请参见**第 4.8.15 节 “-V：详细编译”**。

© 2013 Microchip Technology Inc.

4.4 运行时文件

除了在命令行上指定的 C 和汇编源文件之外，驱动程序还可能会将一些编译器生成的源文件和预编译的库文件编译到项目中。这些文件包含：

- C 标准库程序
- 隐式调用的算术运算程序
- 用户定义的库程序
- 运行时启动代码
- 上电程序
- printf 程序

严格地讲，上电程序既不是编译器生成的源文件，也不是库程序。它完全由用户定义，但是由于它与运行时启动模块的关系非常密切，所以在以下几节中将它与其他运行时文件一起进行讨论。

4.4.1 库文件

适用于所选目标器件的 C 标准库文件的名称和其他驱动程序选项由驱动程序确定，并传递给代码生成器和链接器。您不需要手动将库文件包含到您的项目中。p 代码库（.lpp 库）由代码生成器使用，目标代码库 （.lib 文件）由链接器使用。大多数库程序都源自 p 代码库。

默认情况下，xc8 会在编译器安装目录下的 lib 目录中搜索编译过程中所需的库文件。

4.4.1.1 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。**附录 A “库函数”**介绍了这些函数的范围。虽然 printf 函数被认为是一个库函数，但这些库文件中并不包含它的代码。该函数的 C 源代码将基于在分析用户 C 代码之后定制的一个特殊 C 模板文件而生成。关于使用 printf 库函数的更多信息，请参见 **“PRINTF 和 VPRINTF”**；关于在编译项目时如何定制 printf 函数的信息，请参见**第 5.11.1 节 “printf 程序”**。

这些库中还包含代码生成器的输出代码隐式调用的 C 程序。它们是执行诸如浮点运算、整型除法和类型转换之类任务的程序，可能并不直接对应于源代码中的 C 函数调用。

库名称格式为 *family-type-options*.lpp，适用以下规则。

- *family* 可以为 pic18（对于 PIC18 器件）或 pic（对于所有其他 8 位 PIC 器件）
- *type* 指示所提供的库功能的类型，可能为 stdlib（对于标准库函数）或 trace 等
- *options* 指示一些使用连字符分隔的名称，用以指示库的各种形式，以适应不同的编译器选项或模式；例如，htc 用于确保 HI-TECH C 兼容性，d32 用于表示 32 位双精度浮点型等

例如，用于使用24位double类型的低档和中档器件的标准库为pic-stdlib-d24.lpp。

所有库都位于编译器安装的 lib 目录中。搜索该目录可以获得所提供的所有库的完整列表。

4.4.1.2 用户定义的库

可以根据需要创建用户定义的库，并将其与程序进行链接。库文件更易于管理，可获得更快的编译时间，但必须与特定项目的目标器件和选项兼容。用户可能需要创建几个版本的库，以使它能够用于不同的项目。

库可以使用编译器和库管理器 LIBR 手动创建。关于库管理器和使用该应用程序创建库文件的更多信息，请参见**第 8.2 节“库管理器”**。或者，库文件也可以通过使用 --OUTPUT 选项指定库输出而直接从编译器中创建，请参见**第 4.8.44 节“--OUTPUT= type：指定输出文件类型”**。

在编译项目时应搜索的用户创建库可以与源文件一起在命令行上列出。

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。

对于未解析的符号，将会先扫描在命令行上指定的库文件，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。另请参见**第 5.15.4 节 “替换库模块”**。

4.4.2 启动和初始化

C 程序要求先初始化一些特定的对象，并且器件处于某种特定状态，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。**第 5.10.1 节“运行时启动代码”**特别详细介绍了该代码执行的操作，以及它如何与您编写的程序进行交互。

MPLAB XC8 C 编译器可以基于用户程序确定所需的运行时启动代码，然后在每次编译时生成该代码，而不是使用链接一个通用的预编译程序的传统方法。

生成运行时启动代码同时涉及到驱动程序和代码生成器。驱动程序会创建用于处理器件设置的代码，并将该代码放入一个独立的汇编启动模块。代码生成器会生成用于初始化 C 环境的代码，例如清零未初始化的 C 变量和复制已初始化的 C 变量。该代码与 C 程序的其余部分一起输出。

每次编译项目时，都会重新生成运行时启动代码。驱动程序创建的文件在编译之后可能会被删除，该操作可以通过 --RUNTIME 选项的 keep 子选项控制。驱动程序的默认操作是保留启动模块；但是，如果使用 MPLAB IDE 进行编译，文件将被删除，除非在 Project Properties（项目属性）对话框中另外指示。

如果保留启动模块，它将名为 startup.as，并位于当前工作目录中。如果使用 IDE 来执行编译，目标目录可由 IDE 自身决定。 MPLAB X IDE 会将该文件存储在项目目录的 dist/default/production 目录中。

© 2013 Microchip Technology Inc.

生成运行时启动代码是一个自动过程，不需要任何用户交互；但如果需要，可以使用 --RUNTIME 选项控制运行时代码的某些方面。**第 4.8.50 节 “--RUNTIME：指定运行时环境”**介绍了如何使用该选项。请参见**第 5.10.1 节“运行时启动代码”**，其中介绍了该模块中包含的代码的各个功能方面，以及它对程序操作的影响。

运行时启动代码在 main 函数之前执行，但如果需要在复位后立即执行任何特殊的初始化，则应使用后面的**第 5.10.2 节 “上电程序”**中介绍的上电功能。

4.5 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，一些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。

4.5.1 输出文件

许多输出文件的名称使用与它们的源文件相同的基本名称。例如，源文件 `input.c` 将创建一个名为 `input.p1` 的 p 代码文件。

一些输出文件包含项目范围的信息，并且不与任何一个特定输入文件（如映射文件）直接关联。如果未通过编译器选项指定这些输出文件的名称，其基本名称将基于命令行上列出的第一个 C 源文件产生。如果未指定该类型的文件，则名称基于命令行上的第一个输入文件 （无论其类型如何）。

如果使用 IDE （如 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范围输出文件的基本名称，除非用户另外指定。但是，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*这一术语指代在 IDE 中创建的项目的名称，或在命令行上指定的第一个 C 源文件的基本名称 （不带扩展名的文件名）。

编译器能够直接生成许多由 8 位 PIC 单片机开发工具使用的输出文件格式。

`xc8` 的默认行为是生成 Microchip 格式 COFF 和 Intel HEX 输出。除非通过驱动程序选项更改，否则这些文件的基本名称将为项目名称。默认输出文件类型可以通过编译器选项 （如 `--OUTPUT` 选项）控制。这些文件使用的扩展名是固定的，在**第 4.8.44 节 “--OUTPUT= type：指定输出文件类型”**中与该选项的说明一起列出了这些扩展名。

调试器使用 COFF 文件来获取关于项目的调试信息。

表 4-14 显示了使用 `--OUTPUT` 选项时，可供 `xc8` 使用的所有输出格式选项。“文件类型”列中列出了将用于输出文件的文件扩展名。

4.5.1.1 符号文件

`xc8` 会创建两个符号文件，它们用于生成调试输出文件，例如 COFF 和 ELF 文件。它们是由链接器生成的 SYM 文件 （`.sym` 扩展名）和由代码生成器生成的 SDB 文件（`.sdb` 扩展名）。

SDB 文件包含类型信息，SYM 文件包含地址信息。`CROMWELL` 应用程序（见**第 8.5 节 “CROMWELL”**）会将这两个文件与 HEX 文件进行组合来生成输出调试文件，例如 COFF 文件。

© 2013 Microchip Technology Inc.

4.5.2 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。

编译器选项 --ASMLIST（**第 4.8.16 节 “--ADDRQUAL：设置编译器对于存储器限定符的响应”**）用于生成列表文件，而 -M 选项（**第 4.8.8 节 “-M：生成映射文件”**）用于生成映射文件。

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，以及显示放置所有对象和代码的 psect 时，它是必需的。关于 psect 的介绍指南，请参见**第 5.15.1 节 “程序段”**。此外，关于该文件内容的更多信息，请参见**第 6.5 节 “汇编代码级优化”**。

对于整个 C 程序（包括 C 库文件）将会生成一个列表文件，并命名为项目名称加上扩展名 .lst。对于项目中编译的每个汇编源文件，会另外生成一个列表文件。

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。此外，它还会显示器件中的所有未用存储区和存储器碎片。关于该文件内容的完整信息，请参见**第 7.4 节 “映射文件”**。

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。该文件将被命名为项目名称加上 .map 扩展名的形式。

4.6 编译器消息

所有编译器应用程序（包括命令行驱动程序 xc8）在编译过程中都使用文本消息来报告反馈。编译过程使用集中式消息系统来生成消息，从而可以在编译过程的所有阶段保持一致性。本节将介绍消息系统，在**附录 B“错误和警告消息”**中可以找到所有警告和错误消息的完整列表。

4.6.1 消息概述

消息通过一个惟一编号进行引用，需要传达信息的编译器应用程序会将该编号传递给消息系统。对应于该编号的消息字符串从消息说明文件（Message Description File，MDF）获取，该文件存储在编译器安装目录的 dat 目录中。

当编译器应用程序请求某条消息时，将在对应于当前选定语言的 MDF 中查找其编号。如**第 4.6.2 节 “消息语言”**所述，用户可以改变消息的语言。

查找到之后，报警系统可以从 MDF 中读取消息类型和要显示的字符串。存在几种不同的消息类型，**第 4.6.3 节 “消息类型”**对它们进行了介绍；如同一节中所述，用户可以改写其类型。

用户还可以设置警告消息重要性阈值，从而仅显示用户认为重要的那些消息。此外，还可以禁止带有特定编号的消息。还可以使用 pragma 伪指令来禁止特定代码行内的特定消息编号。**第 4.6.5.1 节 “禁止消息”**对这些方法进行了说明。

假设已使能某条消息，并且它不是级别低于当前警告阈值的警告消息，则会显示消息字符串。

除了实际的消息字符串之外，还可能显示几个其他信息片段，例如消息编号、消息适用于的文件的名称、文件的行号和发出该消息的应用程序等。

如果消息属于错误，计数器将递增。达到一定错误数量之后，将停止当前模块的编译。可以通过使用 --ERRORS 选项，调整会导致这种终止的默认错误数量，请参见**第 4.8.29 节 “--ERRORS：最大错误数”**。该计数器会对于每个内部编译器应用程序发生复位，因此最大错误数量指定为 5 时，将允许解析器最多产生 5 个错误、代码生成器最多产生 5 个错误、链接器最多产生 5 个错误、驱动程序最多产生 5 个错误等。

虽然可以使用任何文本编辑器修改 MDF 中的信息，但不建议这样做。应仅使用以下各节中所述的选项和 pragma 伪指令来更改消息的行为。

© 2013 Microchip Technology Inc.

4.6.2 消息语言

xc8 对于显示消息支持多种语言。对于支持的每种语言，都存在一个 MDF。

在 Windows 下，可以在安装编译器时指定默认语言。

在命令行上，可以使用 --LANG 选项更改默认语言，请参见**第 4.8.35 节 “--LANG：指定消息的语言”**。或者，也可以通过一起使用 --LANG 选项与 --SETUP 选项永久性地更改它，这会将默认语言存储到注册表中（在 Windows 下）或在 XML 配置文件中（在其他系统上）。在以后的编译中，使用的默认语言将为指定的语言。

表 4-3 列出了适用于当前支持语言的 MDF。

表 4-3： 支持的语言

语言	MDF 名称
英语	en_msgs.txt
德语	de_msgs.txt
法语	fr_msgs.txt

如果选择了英语以外的其他语言，并且无法在相应的非英语 MDF 中找到该消息，报警系统会尝试在英语 MDF 中查找该消息。如果英语消息字符串不存在，则会显示类似以下内容的消息：

```
error/warning (*) generated, but no description available
```

其中，* 指示已生成并将打印的消息编号；否则，将显示所请求语言中的消息。

4.6.3 消息类型

有 4 种类型的消息。下面介绍了这些消息类型，以及编译器遇到每种消息类型时的行为。

建议性消息传达有关编译器遇到的情况或编译器即将执行某种操作的信息。显示该信息是为了引起“您的关注”，通常不需要执行任何操作。发出此类消息之后，编译将继续正常进行。

警告消息指示可以进行编译但异常的源代码或其他某种情况，可能会导致代码运行 时失败。用户应检查触发警告的代码或情况；但是，当前模块的编译将继续进行，所有剩余模块的编译也将继续进行。

错误消息指示源代码是非法的，或无法对该代码进行编译。编译过程会尝试对当前模块中的剩余源代码进行编译，但不会对更多的模块进行编译，编译过程之后将结束。

致命错误消息指示无法允许编译继续进行的情况，它要求立即停止编译过程。

4.6.4 消息格式

默认情况下，消息以人们可阅读的格式打印。该格式会因编译器应用程序而不同，因为每个应用程序报告的是关于不同文件格式的信息。

一些应用程序 （如解析器）通常能够查明需关注的区域，并具体到一个特定 C 源代码行的位置，而其他一些应用程序（如链接器）最多只能指示模块名称和记录编号，而不会与任何特定代码行直接关联。某些消息涉及到驱动程序选项的问题，没有任何办法可以将它们与任何源代码相关联。

有几种方法可以更改消息的显示格式，下面对它们进行了讨论。

驱动程序选项 -E（带或不带文件名）可以更改所有显示消息的格式。请参见**第 4.8.3 节“-E：将编译器错误重定向到一个文件”**。使用该选项可以生成更适合于机器解析但用户友好度较低的消息。通常每条消息都显示在单独一行上。使用 -E 选项时，所生成消息的一般格式为：

```
filename  line:  (message number)  message string  (type)
```

-E 选项还具有另一种作用。在使用它时，驱动程序会先检查是否已设置了特殊环境变量。如果已设置，则使用这些变量指定的格式作为所有编译器应用程序产生的所有消息的模板。表 4-4 列出了这些环境变量的名称。

表 4-4： 消息环境变量

变量	作用
HTC_MSG_FORMAT	所有建议性消息
HTC_WARN_FORMAT	所有警告消息
HTC_ERR_FORMAT	所有错误和致命错误消息

这些环境变量的值为用作消息格式模板的字符串。可以在字符串中放入 printf 样式的占位符，从而允许对消息格式进行定制。表 4-5 列出了占位符和它们代表的含义。

表 4-5： 消息占位符

占位符	替换
%a	应用程序名称
%c	列编号
%f	文件名
%l	行号
%n	消息编号
%s	消息字符串 （来自 MDF）

如果在 DOS 批处理文件中使用这些选项，则需要使用两个百分比字符来指定占位符，因为 DOS 会将单个百分比字符解释为参数，而不会将它传递给编译器。例如：

```
SET HTC_ERR_FORMAT="file %%f: line %%l"
```

接着，可以通过驱动程序选项 --MSGFORMAT、 --WARNFORMAT 和 --ERRFORMAT 来改写环境变量，请参见**第 4.8.28 节“--ERRFORMAT：定义编译器消息的格式”**。这些选项使用字符串作为其参数。选项字符串需要进行格式化，并且可以使用与对应变量相同的占位符。

© 2013 Microchip Technology Inc.

例如，对某个项目进行编译，但如下所示，解析器生成了一条警告，链接器生成了一条错误（编号分别为 362 和 492）。

```
main.c: main()
    17: ip = &b;
              ^ (362) redundant "&" applied to array (warning)
(492) attempt to position absolute psect "text" is illegal
```

可以注意到，解析器消息格式标识了引起问题的源代码的特定行和位置。

如果现在使用 -E 选项，并且编译器发出相同的消息，则编译器将输出：

```
main.c: 12: (362) redundant "&" applied to array (warning)
(492) attempt to position absolute psect "text" is illegal (error)
```

用户现在按以下方式使用 --WARNFORMAT：

```
--WARNFORMAT="%a %n %l %f %s"
```

重新编译时，将显示以下输出：

```
parser 362 12 main.c redundant "&" applied to array
(492) attempt to position absolute psect "text" is illegal (error)
```

可以注意到，警告的格式发生了改变，但错误消息的格式未改变。警告格式现在遵从环境变量的规定。应用程序名称（parser）替换了 %a 占位符，消息编号（362）替换了 %n 占位符等。

4.6.5 改变消息的行为

用户可以修改编译过程中各条消息的属性和消息系统的常规设置。驱动程序选项和C语言 pragma 伪指令都可以用来实现这一目的。

4.6.5.1 禁止消息

每条警告消息都具有一个默认数值，指示消息的重要性。该数值在 MDF 中指定，范围为 -9 至 9。数值越高，说明警告越重要。

可以通过使用 `--WARN` 驱动程序选项调整警告级别阈值来禁止警告消息，请参见**第 4.8.59 节 “--WARN：设置警告级别”**。级别低于当前阈值的所有警告都不会显示。

默认阈值为 0，这意味着在默认情况下将仅显示警告级别大于等于 0 的警告。该选项中的信息会传播到所有编译器应用程序，因此，在编译过程的所有阶段都可以观察到它的作用。

警告也可以通过使用 `--MSGDISABLE`选项禁止，请参见**第4.8.38节“--MSGDISABLE: 禁止警告消息”**。该选项可接受使用*逗号*分隔的消息编号的列表。列出的那些警告会被禁止，永远不会发出，无论当前警告级别阈值如何。

此外，还可以通过使用 `warning` pragma 伪指令禁止一些警告消息。该 pragma 伪指令只会影响由解析器或代码生成器产生的警告；即，直接与C代码关联的错误。关于该pragma 伪指令的更多信息，请参见**第 5.14.4.11 节 “#pragma warning 伪指令”**。

此外，也可以禁止错误消息；但是，需要通过更详细形式的命令来确认该操作。要在 `--MSGDISABLE` 命令中指定错误消息编号，编号后面必须跟随 `:off` 来确保禁止它。例如：`--MSGDISABLE=195:off` 将禁止错误编号 195。

> **注：** 禁止错误或警告消息并没有办法修复触发该消息的条件。使用这些选项时，总是需要格外小心。

4.6.5.2 更改消息类型

此外，还可以更改某些消息的类型。只能对由解析器或代码生成器生成的消息执行该操作。关于该 pragma 伪指令的更多信息，请参见**第 5.14.4.11 节 “#pragma warning 伪指令”**。

© 2013 Microchip Technology Inc.

4.7 XC8 驱动程序选项

本节介绍 xc8 命令行选项的一般形式，以及未为特定功能指定任何选项时编译器将执行的操作。

4.7.1 一般选项格式

所有单字母选项都通过前导*短划线*字符 - 标识，例如：-C。一些单字母选项可以指定额外的数据字段，数据字段紧跟在选项名称之后，没有任何*空格*，例如：-Ddebug。在本手册中，选项都使用大写形式，子选项都使用小写形式。

多字母或单词选项具有两个前导*短划线*字符，例如：--ASMLIST。（举例来说，由于具有双*短划线*，驱动程序可以确定选项 --DOUBLE 不是跟随着参数 OUBLE 的 -D 选项。）

其中一些单词选项可以使用子选项，这些子选项通常显示为后面跟随*等号*字符 = 的*逗号*分隔列表，例如：--OUTPUT=hex,cof。确切的选项格式会有所不同，以下几节中将详细介绍。

一些常用的子选项包括：default，它代表在该选项完全缺失时使用的默认规范；all，它表示应使能所有可用的子选项，就如同列出了其中每一个；以及 none，它表示应禁止所有子选项。例如：

```
--OPT=none
```

将关闭所有优化器。

一些子选项可以带有加号字符 + 作为前缀，表示它们是在所提供的其他子选项之外附加的；或带有减号字符 - 作为前缀，表示应将其排除。例如：

```
--OPT=default,-asm
```

表示使用默认优化，但应禁止汇编器优化器。如果等号之后的第一个字符是 + 或 -，则表示暗含 default 关键字。例如：

```
--OPT=-asm
```

与上一个示例作用相同。

关于选项和子选项的更多信息，请参见 --HELP 选项（**第 4.8.33 节 “--HELP：显示帮助”**）。

4.7.2 默认选项

如果从命令行上运行编译器驱动程序，并且未指定某个功能的选项，它将默认设为某种状态。此外，还可以将 default 子选项指定为双短划线选项，这也会调用默认的行为。您可以通过在命令行上使用 --HELP=*option* 来查看其默认行为，请参见**第 4.8.33 节 “--HELP：显示帮助”**。

如果从 MPLAB X IDE 中进行编译，则默认情况下，它会向编译器发出明确的选项（除非在 Project Properties 对话框中进行了更改），这些选项可能不同于命令行上的那些默认值。例如，除非在命令行上指定了 --ASMLIST 选项，否则编译器的默认操作是*不*生成汇编列表文件。但是，如果从 MPLAB X IDE 中进行编译，默认操作（事实上无法禁止它）是*总是*生成汇编列表文件。

如果从命令行上和 MPLAB X IDE 中编译相同的项目，请总是检查是否显式指定了所有选项。

4.8 选项说明

使用命令行驱动程序 xc8 可以对编译的大多数方面进行控制。驱动程序将配置并执行所有必需的应用程序，例如代码生成器、汇编器和链接器。

xc8 可以识别下表列出的编译器选项，下面几节中将对它们进行详细说明。选项的大小写形式并不重要，但是在涉及到文件名时，大多数操作系统中的命令 shell 都区分大小写。

表 4-6： 驱动程序选项

选项	含义
-C	编译为目标文件并停止
-D*macro*	定义预处理器宏符号
-E*filename*	重定向编译错误
-G[*filename*]	生成符号调试信息
-I*path*	指定包含路径
-Largument	设置链接器选项
-M[*filename*]	生成映射文件
-N*number*	指定标识符长度
-O*file*	指定输出文件名和类型
-P	预处理汇编源文件
-Q	安静模式
-S	编译为汇编文件并停止
-U*macro*	取消定义预处理器宏符号
-V	详细模式
--ADDRQUAL=*qualifier*	指定地址空间限定符处理
--ASMLIST	生成汇编列表文件
--CCI	实施并要求符合 CCI 规则
--CHAR=*type*	默认字符类型 （已取消）
--CHECKSUM=*specification*	计算校验和，并将结果存储在程序存储器中
--CHIP=*device*	选择目标器件
--CHIPINFO	打印器件信息
--CODEOFFSET=*value*	指定 ROM 偏移地址
--DEBUGGER=*type*	设置调试器环境
--DOUBLE=*size*	双精度浮点型的长度
--ECHO	回显命令行
--EMI=*mode*	选择外部存储器接口工作模式
--ERRATA=*type*	指定勘误规避措施
--ERRFORMAT=*format*	设置错误格式
--ERRORS=*number*	设置最大错误数
--FILL=*specification*	指定未用存储器的 ROM 填充值
--FLOAT=*size*	浮点型的长度
--GETOPTION=*argument*	获取高级选项
--HELP=*option*	帮助
--HTML=file	生成 HTML 调试文件
--LANG=*language*	指定语言
--MEMMAP=*mapfile*	显示存储器映射
--MODE=*mode*	选择工作模式

© 2013 Microchip Technology Inc.

表 4-6： 驱动程序选项 （续）

选项	含义
--MSGDISABLE=*list*	禁止警告消息
--MSGFORMAT=*specification*	设置建议性消息格式
--NODEL	不删除临时文件
--NOEXEC	不执行编译器应用程序
--OBJDIR=*path*	设置目标文件目录
--OPT=*optimizations*	控制优化
--OUTDIR=*path*	设置输出目录
--OUTPUT=*path*	设置输出格式
--PASS1	生成中间 p 代码文件并停止
--PRE	生成预处理源文件并停止
--PROTO	生成函数原型
--RAM=*ranges*	调整 RAM 范围
--ROM=*ranges*	调整 ROM 范围
--RUNTIME=*options*	指定运行时选项
--SCANDEP	扫描相依性关系
--SERIAL=*specification*	插入十六进制代码或序列号
--SETOPTION=*argument*	设置高级选项
--SETUP=*specification*	设置编译器
--SHROUD	掩饰 （混淆）生成的 p 代码文件
--STRICT	使用严格 ANSI 关键字
--SUMMARY=*type*	摘要选项
--TIME	报告编译时间
--VER	显示版本信息
--WARN=*number*	设置警告阈值级别
--WARNFORMAT=*specification*	设置警告格式

4.8.1 -C：编译为目标文件

-C 选项用于在执行汇编器之后停止编译，留下可重定位的目标文件作为输出。使用 make 实用程序编译汇编源文件时，经常会使用它。除非在命令行上提供了所有 C 源文件，否则不能使用它。使用 --PASS1 从 C 源文件生成中间文件，请参见**第 4.8.45 节 “--PASS1：编译为 p 代码”**。

关于生成和使用中间文件的更多信息，请参见**第 4.3.3 节 “多步编译”**。

4.8.2 -D：定义宏

-D 选项用于在命令行上定义预处理器宏，与在源代码中使用 #define 伪指令定义它的作用完全相同。该选项可采用两种形式之一：-Dmacro，它等效于将以下内容：

```
#define macro  1
```

放置在每个使用该选项编译的模块的顶部；或者 -Dmacro= *text*，它等效于：

```
#define macro  text
```

其中， *text* 是所需的替换文本。因此，以下命令：

```
xc8 --CHIP=16F877AA -Ddebug -Dbuffers=10 test.c
```

将使用所定义的宏编译 test.c，与在 C 源代码中包含以下伪指令的作用完全相同：

```
#define  debug    1
#define  buffers 10
```

将宏定义为 C 字符串字面值需要绕过正在使用的操作系统中的所有解释问题。要在 Windows 环境中传递 C 字符串 "hello world"（包括*引号*字符），请使用：
"-DMY_STRING=\\\"hello world\\\""（必须在整个选项两侧包含*引号*字符，因为宏定义中存在一个*空格*字符）。在 Linux 或 Mac OS X 下，请使用：
-DMY_STRING=\"hello\ world\"。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.3 -E：将编译器错误重定向到一个文件

该选项具有两个用途。第一个用途是更改显示消息的格式。第二个用途是可选地将消息定向到某个文件，因为在调用编译器时，一些编辑器不允许使用标准的命令行重定向工具。

使用 -E 选项时，所生成消息的一般格式为：

```
filename  line_number:  (message number)  message string  (type)
```

如果紧随 -E 之后指定一个文件名，它将被视为打印所有消息（错误和警告等）的文件的名称。例如，要编译 x.c 并将所有错误都重定向到 x.err，可以使用以下命令：

```
xc8 --CHIP=16F877AA -Ex.err x.c
```

-E选项还允许通过在错误文件名起始处指定一个加号字符+将错误追加到现有文件中，例如：

```
xc8 --CHIP=16F877AA -E+x.err y.c
```

如果希望编译几个文件，并将生成的所有错误合并到单个文本文件中，可以使用 -E 选项创建文件，然后在编译所有其他源文件时使用 -E+。例如，要编译一定数量的文件，并将所有错误合并到一个名为 project.err 的文件中，可以如下使用 -E 选项：

```
xc8 --CHIP=16F877AA -Eproject.err -O --PASS1 main.c
xc8 --CHIP=16F877AA -E+project.err -O --PASS1 part1.c
xc8 --CHIP=16F877AA -E+project.err -C asmcode.as
```

第 4.6 节“编译器消息”提供了关于该选项的更多信息，以及消息系统和其他相关驱动程序选项的概述。

© 2013 Microchip Technology Inc.

4.8.4 -G：生成源代码级符号文件

-G 选项允许指定用于*源代码级符号文件*的文件名（.sym 扩展名），该符号文件将用于支持的调试器和软件模拟器 （如 MPLAB IDE）。另请参见**第 4.5 节 “编译器输出”**。

如果未给出文件名，符号文件将使用项目名称 （见**第 4.2 节 “调用编译器”**）和 .sym 扩展名。例如，选项 -Gtest.sym 将生成一个名为 test.sym 的符号文件。使用 -G 选项生成的符号文件包含用于源代码级调试器的源代码级信息。

4.8.5 -I：包含搜索路径

使用 -I 可以指定额外的目录，用于搜索已通过 #include 伪指令包含的头文件。目录可以为绝对路径或相对路径。如果要搜索多个目录，可以多次使用 -I 选项。

即使未提供任何 -I 选项，也总是会搜索包含所有标准头文件的编译器 include 目录。如果使用*引号*字符而不是*尖括号*指定头文件名（如 #include "lcd.h"），则除了编译器的 include 目录之外，还会搜索当前工作目录。请注意，如果在 MPLAB IDE 中进行编译，搜索路径将相对于输出目录，而不是项目目录。

这些默认搜索路径将在搜索所有用户指定目录之后再进行搜索。例如：

```
xc8 --CHIP=16F877AA -C -Ic:\include -Id:\myapp\include test.c
```

将搜索目录 c:\include 和 d:\myapp\include 来查找包含到源代码中的所有头文件，然后再搜索默认的包含目录。

对于使用汇编 INCLUDE 伪指令包含到汇编源文件中的文件，该选项不起作用。请参见**第 6.4.10.4 节 “INCLUDE”**。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.6 -L：扫描库

-L 选项用于指定要由链接器扫描的额外库。使用 -L 选项指定的库将在标准 C 库之前扫描，从而允许获取其他版本的标准库函数。

-L 的参数是一个库关键字，在该关键字中将添加前缀 pic，代表器件范围、ROM 页数和 RAM 存储区数量的数字，以及后缀 .lib。

举例来说，针对 16F877A 进行编译时，选项 -L1 将扫描库 pic42c-l.lib，选项 -Lxx 将扫描名为 pic42c-xx.lib 的库。

所有库都必须位于编译器安装目录的 lib 目录中。

如上所述， -L 选项的参数*不是*一个完整的库文件名。如果希望链接器扫描名称不遵循上述命名约定或其位置不在 lib 子目录中的库，只需在命令行上与源文件一起包含库的名称，或将它们添加到项目中。

4.8.7 -L-：直接调整链接器选项

-L 驱动程序选项可用于指定将直接传递给链接器的选项。如果 -L 紧跟着以短划线字符 - 开头的文本，文本将被直接传递给链接器，不通过 xc8 进行解释。如果 -L 选项未紧跟一个短划线字符，则假定该选项是库扫描选项 （**第 4.8.6 节 “-L：扫描库”**）。

例如，如果指定了选项 -L-N，-N 选项会被传递给链接器，不通过驱动程序进行任何后续的解释。然后，如果调用链接器，链接器将在被调用时处理该选项，并执行相应的操作。

请小心使用命令行选项。链接器无法解释命令行驱动程序选项；类似地，驱动程序无法解释链接器选项。在大多数情况下，执行的总是命令行驱动程序 xc8。如果需要在 Project Properties 对话框的 Linker 类别中添加备用的链接器设置，则必须添加*驱动程序*选项（而不是链接器选项）。在链接过程中，驱动程序将使用这些驱动程序选项生成相应的链接器选项。 -L 选项是一种允许通过驱动程序选项指定链接器选项的方法。

-L 选项在链接包含非标准程序段（或 psect）的代码时特别有用；以下就会出现这种情况：程序包含手写的汇编代码，而汇编代码中包含用户定义的 psect （见**第 6.4.9.3 节 “PSECT”**和**第 5.15.1 节 “程序段”**），或者 C 代码使用了 #pragma psect 伪指令（见**第 5.14.4.8 节 “#pragma psect 伪指令”**）。如果没有该 -L 选项，就必须通过手动调用链接器来调整链接器选项。

该选项还可用于替换默认的链接器选项。如果从 -L 之后开始的第一个字符直到第一个等号字符 “=” 的字符串与默认选项中的某个 psect 或类名匹配，则 （默认选项中对该 psect 或类名的引用，以及该选项的剩余部分会被删除）默认链接器选项会被替换为由 -L 指定的选项。例如，如果默认链接器选项为：

```
-preset_vec=00h,intentry,init,end_init
```

驱动程序选项 -L-pinit=100h 将导致以下选项被传递给链接器：-pinit=100h -preset_vec=00h。请注意， end_init 链接器选项已经被完全删除。如果在 -L 选项中，第一个等号字符后面没有任何字符，则不会对将被删除的默认链接器选项进行任何替换。例如，驱动程序选项 -L-pinit= 将调整传递给链接器的默认选项 （如上所述），但 -pinit 链接器选项将被完全删除。

如果找不到这种默认链接器选项，不会生成任何警告。要删除或替换的默认选项必须包含等号字符。

4.8.8 -M：生成映射文件

-M 选项用于请求生成映射文件。映射文件由链接器生成，它包含关于对象在存储器中位置的详细信息。关于这些文件内容的信息，请参见**第 7.4 节 “映射文件”**。

如果选项未指定文件名，则映射文件的名称将使用项目名称（见**第 4.3 节“编译序列”**）和扩展名 .map。

在 MPLAB X IDE 中进行编译和使用 Universal Toolsuite （通用工具套件）时，在默认情况下将开启该选项。

© 2013 Microchip Technology Inc.

4.8.9 -N：标识符长度

该选项允许有效 C 标识符长度 （由函数和变量使用）相对于默认值 255 减小。该选项的有效长度为 32 至 255。该选项对于所有其他值不起作用。

该选项还可以控制由预处理器使用的标识符（如宏名）的有效长度。默认长度也是 255，最小可调节为 31。

如果使用了 `--STRICT` 选项，则默认的有效标识符长度将减至 31。使用较长标识符长度的代码，可移植性将较低。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.10 -O：指定输出文件

该选项用于指定输出文件的基本名称。如果未给出 `-O` 选项，则输出文件的基本名称将与项目名称相同，请参见**第 4.3 节 “编译序列”**。名称会受该选项影响的文件是那些不与任何特定源文件直接关联的文件，例如 HEX 文件、 MAP 文件和 SYM 文件。

通过在文件名之前包含所需的路径， `-O` 选项还可以更改输出文件所在的目录。然后，它也会指定由链接器或随后运行的应用程序生成的所有文件的输出目录。指定相对路径时，所有相对路径都是相对于当前工作目录。

例如，如果使用了选项 `-Oc:\project\output\first`，MAP 和 HEX 等文件将使用基本名称 `first`，并被放入目录 `c:\project\output`。

随文件名提供的任何扩展名都会被忽略。

指定 MAP 文件创建 （`-M`，见**第 4.8.8 节 “-M：生成映射文件”**）和 SYM 文件创建（`-G`，见**第 4.8.4 节 “-G：生成源代码级符号文件”**）的选项会覆盖通过 `-O` 提供的与 MAP 和 SYM 文件相关的任何名称或路径信息。

要更改所有输出和中间文件的写入目录，可以使用 `--OUTDIR` 选项，请参见**第 4.8.43 节 “--OUTDIR：指定输出文件的目录”**。请注意，如果 `-O` 指定的路径与 `--OUTDIR` 选项指定的路径不一致，将会导致错误。

4.8.11 -P：预处理汇编文件

`-P` 选项会使汇编器源文件在汇编之前进行预处理，从而允许对汇编器代码使用预处理器伪指令 （如 `#include`）和 C 语言样式的注释。

默认情况下，汇编器文件不进行预处理。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.12 -Q：安静模式

该选项将编译器置于安静模式，这会禁止显示 Microchip Technology Incorporated 版权声明。

4.8.13 -S：编译为汇编代码

-S 选项会在生成汇编输出文件之后停止编译。对于所有 C 源代码（包括 p 代码库代码）将生成一个汇编文件。

命令 xc8 --CHIP=16F877A -S test.c 会生成名为 test.as 的汇编文件，其中包含基于 test.c 生成的汇编代码。生成的文件是有效的汇编代码，它可以作为一个源文件传递给 xc8，但只有为了研究原因才应这样做。要在编译器中利用编译技术带来的好处，必须在单个步骤中编译并链接所有 C 源代码。如果要使用两步过程编译代码或使用中间文件，请参见 --PASS1 选项（**第 4.8.45 节 “--PASS1：编译为 p 代码”**）来生成中间文件。

该选项对于检查编译器的汇编代码输出非常有用。通过该选项生成的文件与通过 --ASMLIST 选项（见**第 4.8.16 节 “--ADDRQUAL：设置编译器对于存储器限定符的响应”**）生成的文件的不同之处在于它不包含操作码或地址，并且它可以用作后续编译的源文件。汇编列表文件更方便人们阅读，但它不是有效的汇编源文件。

4.8.14 -U：取消定义宏

-U 选项是 -D 选项的逆选项，用于取消定义已预定义的宏。该选项采用 -U*macro* 的形式，其中的 *macro* 是要取消定义的宏的名称。

例如，选项 -Udraft 等效于将以下内容：

```
#undef  draft
```

放在使用该选项编译的每个模块的顶部。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.15 -V：详细编译

-V 选项用于指定详细编译。使用该选项时，编译器将显示用于调用**第 4.3 节“编译序列”**中所述的每个编译器应用程序的命令行。

该选项会显示正在执行的编译器应用程序的名称，以及传递给该应用程序的所有命令行参数。该选项对于确认传递给编译器应用程序的选项和文件名非常有用。

如果使用该选项两次（-V -V），它将显示每个编译器应用程序的完整路径，以及完整的命令行参数。如果安装了多个编译器，这对于确保执行正确的编译器安装将非常有用。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

© 2013 Microchip Technology Inc.

4.8.16 --ADDRQUAL：设置编译器对于存储器限定符的响应

--ADDRQUAL 选项用于指示编译器对 C 源代码中的非标准存储器限定符的响应。

默认情况下，这些限定符会被忽略；即接受它们，不产生任何错误，但也没有任何作用。使用该选项允许编译器以不同的方式解释这些限定符。

near 限定符会受该选项影响。在PIC18器件上，该选项还会影响far限定符；对于其他 8 位器件， bank*x* 限定符 （bank0、 bank1 和 bank2 等）都会受影响。

表 4-7 详细介绍了子选项。

表 4-7： 编译器对于存储器限定符的响应

选择	响应
require	遵从限定符。如果无法满足它们，将产生一条错误。
request	如果可能，遵从限定符。如果无法遵从它们，将不生成任何错误。
ignore	限定符将被忽略，就如同未使用它们一样编译代码。
reject	如果遇到限定符，将立即生成一条错误。

例如，使用选项 --ADDRQUAL=request 时，编译器将尝试遵从所有非标准限定符，但如果无法满足它们，则会默默地忽略它们。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.17 --ASMLIST：生成汇编列表文件

--ASMLIST 选项指示 xc8 为正在编译的 C 和汇编源代码模块生成*汇编列表文件*。对于整个 C 程序 （包括来自 C 库函数的代码）生成一个汇编列表文件。

此外，还会为项目中的每个汇编源文件 （包括运行时启动代码）生成一个汇编列表文件。更多信息，请参见**第 4.4.2 节 “启动和初始化”**。

汇编列表文件使用 .lst 扩展名，由于还有其他信息放入这些文件，它们不能用作汇编源文件。

对于 C 源代码的列表，列表文件会同时显示原始 C 代码和编译器生成的相应汇编代码。关于这些文件内容的完整信息，请参见**第 6.5 节 “汇编代码级优化”**。

在汇编源代码的列表文件中会显示相同的信息。

在 MPLAB IDE 下进行编译时，默认情况下会开启该选项。

4.8.18 --CCI：实施并要求符合 CCI 标准

使能该选项将要求编译器检查所有源代码和编译器选项，确定它们是否符合通用编译器接口（CCI）标准。符合该标准的代码将可在所有 MPLAB XC 编译器之间移植。（描述 CCI 标准的文档在编写本用户指南时正在等待批准。）不符合 CCI 标准的代码或选项将使用编译器警告进行标记。

4.8.19 --CHECKSUM：计算校验和

该选项将对所指定的地址范围执行校验和计算，并将结果存储在指定的目标地址处。该选项的一般格式如下。

`-CHECKSUM=`*`start-end@destination`*`[,offset=][,width=`*`w`*`][,code=`*`c`*`][,algorithm=`*`a`*`]`

附加规范以*逗号*分隔列表的形式附加到该选项后面。此类规范包括：

width=*n* 选择校验和结果的宽度，以字节为单位。宽度为负值时，将以小尾数法字节顺序存储结果。允许 1 至 4 字节的结果宽度。

offset=*nnnn* 指定要与该校验和相加的初始值或偏移量。

algorithm=*n* 选择HEXMATE中实现的校验和算法之一。表8-9列出了可供选择的算法。

code=*nn* 是一个跟随在校验和结果中每个字节之后的十六进制代码。它使校验和结果的每个字节都可以嵌入一条指令中。

默认情况下，*`start`*、*`end`* 和 *`destination`* 属性是十六进制常量。如果未指定伴随的 `--FILL` 选项，则指定地址范围内的未用存储单元将使用 FFFh （对于低档器件） 、3FFFh （对于中档器件）或 FFFF （对于 PIC18 器件）填充。这是为了从公式中去除所有未知值，并确保校验和结果的准确性。

例如：

```
--checksum=800-fff@20,width=1,algorithm=8
```

将从地址 0x800 至 0xfff 计算 1 个字节的校验和，并将它存储在地址 0x20 处。将使用 Fletcher 的算法。请参见**第 8.6.1.5 节 “-CK”**。

校验和计算由 HEXMATE 应用程序执行。该驱动程序选项中的信息将在执行 HEXMATE 应用程序时传递给它。

4.8.20 --CHIP：定义器件

必须使用该选项来指定编译的目标器件或器件。它是编译代码时惟一的必需编译器选项。

要查看可以用于该选项的支持器件的列表，请使用**第 4.8.21 节 “--CHIPINFO：显示支持器件的列表”**中所述的 `--CHIPINFO` 选项。

4.8.21 --CHIPINFO：显示支持器件的列表

`--CHIPINFO` 选项将显示编译器支持的器件的列表。列出的名称是在 chipinfo 文件中定义并可用于 `--CHIP` 选项的那些芯片。

打印该列表之后，编译器执行将会终止。

© 2013 Microchip Technology Inc.

4.8.22 --CODEOFFSET：将程序代码偏移到某个地址

在某些情况下（如自举程序），需要将程序镜像移动到某个备用地址处。该选项用于指定程序代码镜像的基址，并保留从地址 0 到选项中指定地址的存储器。

使用该选项时，所有代码 psect（包括复位和中断向量以及常量数据）将会被调整到指定的地址处。该地址假定为一个十六进制常量。可以使用 0x 前缀或 h 后缀十六进制说明符，但不是必需的。

该选项与 --ROM 选项的区别在于它将移动与复位和中断向量关联的代码，这是使用 --ROM 选项无法完成的，请参见**第 4.8.49 节 “--ROM：调整 ROM 范围”**。

例如，如果指定了选项 --CODEOFFSET=600，复位向量将从地址 0 移到地址 0x600；中断向量将从地址 4 移到地址 0x604（对于中档 PIC 器件）或地址 0x608 和 0x618（对于 PIC18 器件）。在地址 0 和 0x600 之间将不放入任何代码。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.23 --DEBUGGER：选择调试器类型

该选项旨在用于确保与可用作调试器的开发工具的兼容性。 xc8 支持几种调试器，使用该选项会将编译器配置为符合选定调试器的要求。表 4-8 列出了该选项的可能选择。

表 4-8： 可选的调试器

子选项	选定的调试器
none	无调试器 （默认值）
icd2	MPLAB® ICD 2
icd3	MPLAB ICD 3
pickit2	PICkit™ 2
pickit3	PICkit 3
realice	MPLAB REAL ICE™ 在线仿真器

例如：

```
xc8 --CHIP=16F877AA --DEBUGGER=icd2 main.c
```

选择正确的调试器非常重要，因为它们会使用在省略该选项情况下可由项目使用的存储器资源。这种冲突会导致运行时失败。

如果不使用开发工具的调试功能 （例如，如果 MPLAB ICD 3 仅用作一个编程器），则调试器选项可以设置为 none，因为该工具不会使用存储器资源。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**。只有在 Project Preferences （项目首选项）中指示硬件工具之后， MPLAB X IDE 才会对调试编译自动应用该选项。

4.8.24 --DOUBLE：选择双精度浮点型的种类

该选项用于选择双精度浮点型的种类。默认情况下，编译器会为 double 类型选择缩短的 IEEE754 24 位实现。使用该选项，可以将它更改为完整的 32 位 IEEE754 实现。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.25 --ECHO：在处理之前回显命令行

使用该选项将导致驱动程序命令行在编译开始之前被回显到 stdout 流中。命令行中的每个标记将打印在单独一行中，并按它们在命令行中的放置顺序显示。

4.8.26 --EMI：选择外部存储器接口工作模式

一些 PIC18 器件上提供的外部存储器接口可以在几种模式下工作。该接口在 16 位模式下工作：字写和字节选择模式，也可以在 8 位模式下工作：字节写模式。这些模式通过表 4-9 中列出的模式表示。

表 4-9： 外部存储器接口模式

模式	工作模式
wordwrite	16 位字写模式 （默认值）
byteselect	16 位字节选择模式
bytewrite	8 位字节写模式

在写入外部数据接口时，选定的模式会影响生成的代码。在字写模式下，可能会添加假读和假写操作，以确保总是写入偶数个字节。在字节选择或字节写模式下，不会产生假读和假写操作，可以生成更高效的代码。

请注意，该选项不会预先配置器件，使之在选定模式下工作。关于用于配置器件外部接口模式的寄存器和设置，请参见器件数据手册。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

© 2013 Microchip Technology Inc.

4.8.27 --ERRATA：指定勘误规避措施

该选项用于指定一些软件规避措施来处理已记录在案的硅片勘误问题。有一组默认的勘误问题适用于每款器件，但这一组问题可以通过使用该选项和表 4-10 中列出的参数进行调整。

表 4-10： 勘误规避措施

符号	规避措施
4000	程序存储器访问 / 跳转跨过 4000h 地址边界
fastints	快速中断影子寄存器损坏
lfsr	LFSR 指令损坏
minus40	程序存储器在 -40 度下读取
reset	GOTO 指令不能位于复位向量处
bsr15	当 BSR 包含值 15 时的标志问题
daw	DAW 指令损坏
eedatard	在紧接 RD 置 1 之后的指令中读取 EEDAT
eeadr	不要在装入 EEADR 之后立即将 RD 位置 1
ee_lvd	LVD 必须在写入 EEPROM 之前稳定下来
fl_lvd	LVD 必须在写入闪存之前稳定下来
tblwtint	在 tblwt 指令之前清零中断寄存器
fw4000	闪存写入执行程序必须对 4000h 边界另一侧进行操作
resetram	异步时 RAM 内容会发生损坏。在写访问期间发生复位。
fetch	取指操作可能损坏。在所需存储单元应用 FFFFh （NOP）。

目前，规避措施仅对于 PIC18 器件采用。

要禁止所有软件规避措施，可以使用以下语句。

```
--ERRATA=none
```

例如，要应用默认的一组规避措施，但要特别禁止跨过 4000 跳转的勘误，可以使用以下语句。

```
--ERRATA=default,-4000
```

可通过将预处理器宏 ERRATA_TYPES （见**第 5.14.3 节 “预定义的宏”**）设置为某个值来指示要应用的勘误。表 4-10 列出的每个勘误代表该值中的一个位，表中最顶部的勘误为最低有效位。如果应用勘误，则相应位置 1。即，如果应用 `4000`、`reset` 和 `bsr15` 勘误，赋予 ERRATA_TYPES 的值为 0x31。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.28 --ERRFORMAT：定义编译器消息的格式

如果不使用 --ERRFORMAT 选项，编译器的默认行为是以 “人们可阅读”的形式显示所有错误。这种标准格式非常适合人们阅读错误输出，但是在支持编译器错误处理的环境中一般不使用。

该选项允许使用嵌入消息模板的特殊占位符来指定打印错误消息的确切格式。关于 xc8 采用的消息系统和可用于该选项的占位符的完整详细信息，请参见**第 4.6 节“编译器消息”**。

本节也适用于 --WARNFORMAT 和 --MSGFORMAT 选项，它们分别用于调整警告和建议性消息的格式。

如果使用 MPLAB IDE 进行编译，编译器消息的格式会自动配置为 IDE 期望的格式。如果使用该 IDE 进行编译，则建议不要调整消息格式。

4.8.29 --ERRORS：最大错误数

该选项用于设置每种编译器应用程序以及驱动程序在终止执行之前显示的最大错误数。默认情况下，每种应用程序最多显示 20 条错误消息。

关于 xc8 采用的消息系统的完整详细信息，请参见**第 4.6 节 “编译器消息”**。

4.8.30 --FILL：填充未用的程序存储器

该选项允许指定可用于填充所有未用程序存储单元的十六进制操作码。该选项利用了 HEXMATE 应用程序的功能，所以只有在生成 HEX 输出文件时（这是默认操作）它才可用。

该驱动程序功能使您可以在一个步骤中编译并填充未用存储单元。如果不希望使用该驱动程序选项，而是希望在编译之后填充未用存储单元，则需要显式地使用 HEXMATE 应用程序。请注意，不同于驱动程序 --FILL 选项， HEXMATE 中的相应选项为 -FILL（一个前导短划线）。另外请注意，HEXMATE 选项中不存在可以使用驱动程序选项指定的 unused 标记。

驱动程序选项的用法为：

```
--FILL=[const_width:]fill_expr[@address[:end_address]]
```

其中：

- *const_width* 的形式为 *wn*，并指示 *fill_expr* 中的每个常量的宽度（*n* 字节）。如果未指定 *const_width*，则默认值为架构的本机宽度。即，--FILL=w1:1 将使用值 0x01 填充每个字节。
- *fill_expr* 可以使用语法 （其中， *const* 和 *increment* 是 *n* 字节常量）：
  - *const* 会使用一个重复的常量填充存储器；即， --FILL=0xBEEF 会成为 0xBEEF, 0xBEEF, 0xBEEF, 0xBEEF
  - *const*+=*increment* 会使用一个递增常量填充存储器；即，--fill=0xBEEF+=1 会成为 0xBEEF, 0xBEF0, 0xBEF1, 0xBEF2
  - *const*-=*increment* 会使用一个递减常量填充存储器；即，--fill=0xBEEF-=0x10 会成为 0xBEEF, 0xBEDF, 0xBECF, 0xBEBF
  - *const*,*const*,...,*const* 会使用重复常量的列表填充存储器；即，--FILL=0xDEAD,0xBEEF 会成为 0xDEAD, 0xBEEF, 0xDEAD, 0xBEEF

© 2013 Microchip Technology Inc.

- 跟随在 fill_expr 之后的选项会导致以下行为：
  - @unused （或无任何内容）将使用 *fill_expr* 填充所有未用的存储器；即，--FILL=0xBEEF@unused 将使用 0xBEEF 填充所有未用的存储器。驱动程序会将它扩展到适当的范围，并传递给 HEXMATE。
  - @*address* 将使用 *fill_expr* 填充一个特定地址；即，--FILL=0xBEEF@0x1000 将在地址 1000h 处放入 0xBEEF
  - @*address*:*end_address* 将使用 *fill_expr* 填充某个存储器范围；即，--FILL=0xBEEF@0:0xFF 会在 0 和 255 之间的未用地址中放入 0xBEEF

所有常量都可以按照正常的 C 语法使用 （无符号）二进制、八进制、十进制或十六进制表示，例如， 1234 是十进制值， 0xFF00 是十六进制值， FF00 是非法的。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.31 --FLOAT：选择浮点型的种类

该选项允许选择 float 类型的长度。表 4-11 列出了可供选择的类型。

另请参见**第4.8.24节“--DOUBLE: 选择双精度浮点型的种类”**中的--DOUBLE选项。

表 4-11： 浮点型选择

子选项	作用
double	float 的长度与 double 类型的长度匹配
24	24 位浮点型 （默认值）
32	32 位浮点型 （IEEE754）

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.32 --GETOPTION：获取命令行选项

该选项用来检索用于指定编译器应用程序的命令行选项。然后，这些选项会被保存到给定文件中。该选项对于大多数项目都不需要，并且在编译器工作于免费模式 （见**第 4.8.37 节 “--MODE：选择编译器工作模式”**）时会被禁止。

该选项可接受应用程序名称和用于存储选项的文件名，例如：

```
--GETOPTION=hlink,options.txt
```

4.8.33 --HELP：显示帮助

该选项显示关于 xc8 编译器选项的信息。选项 --HELP 将显示所有可用选项。要了解关于某个特定选项的更多信息，请使用该选项的名称作为参数。例如：

```
xc8 --help=warn
```

将显示关于 --WARN 选项、可用的子选项，以及默认情况下使能哪些子选项的更多详细信息。

4.8.34 --HTML：生成 HTML 诊断文件

该选项将生成一系列 HTML 文件，这些文件可用于研究最新编译的编译结果。这些文件存储在输出目录中名为 `html` 的目录中。顶层文件（可以使用您喜欢的网络浏览器打开它）名为 `index.html`。

请在编译的所有阶段都使用该选项，以确保生成与所有编译阶段相关联的文件。

索引页是编译过程的图形表示。每个文件图标都可点击，并将打开该文件的内容。它可用于所有中间文件，甚至是以人们可阅读形式打开的二进制文件。此外，还可以单击每个应用程序图标，以显示包含关于该应用程序的选项和编译结果信息的页面。

所有预处理器宏 （预处理程序图标）和图形存储器使用映射 （链接器图标）的列表将提供其他情况下不容易获得的信息。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.35 --LANG：指定消息的语言

该选项允许对编译器进行配置，使之以非英语语言生成错误、警告和建议性消息。

英语是默认的语言，除非在安装时或通过使用 `--SETUP` 选项更改了该设置。一些消息总是仅以英语打印，无论使用该选项指定的语言如何。更多信息，请参见**第 4.6.2 节 “消息语言”**。

表 4-12 列出了当前支持的语言。

表 4-12： 支持的语言

子选项	语言
`en`, `english`	英语 （默认值）
`fr`, `french`, `francais`	法语
`de`, `german`, `deutsch`	德语

4.8.36 --MEMMAP：显示存储器映射

该选项将显示使用该选项指定的映射文件的存储器映射。打印的信息通过 `--SUMMARY` 选项进行控制 （见**第 4.8.56 节 “--SUMMARY：选择存储器摘要输出类型”**），例如：

```
xc8 --memmap=size.map --summary=psect,class,file
```

很少会需要使用该选项，但在显式地驱动链接器时 （即，不是通常由命令行驱动程序解析的方式），它会很有用。该命令将显示通常在编译结束时由驱动程序生成的存储器摘要。

4.8.37 --MODE：选择编译器工作模式

该选项用于选择编译器的基本工作模式。可用的类型包括 `pro`、`std` 和 `free`。对于旧项目，可以接受 `lite` 模式，表示免费工作模式。在专业模式下工作的编译器将使用完全优化，并生成长度最短的代码。标准模式使用有限的优化，免费模式只使用最小优化级别，并将生成相对较长的代码。

© 2013 Microchip Technology Inc.

只有编译器许可证状态允许的模式会被接受。例如，如果购买了标准编译器许可证，则编译器可以在标准或免费模式下运行，但无法在专业模式下运行。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.38 --MSGDISABLE：禁止警告消息

该选项允许在项目编译过程中禁止警告或建议性消息。`messagelist` 是要禁止的警告编号的*逗号*分隔列表。如果指定了某个错误的编号，该选项会忽略它。如果将消息列表指定为 `0`，则会禁止所有警告。关于禁止消息的其他方法，请参见**第 4.6.5 节 “改变消息的行为”**。

关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

4.8.39 --MSGFORMAT：设置建议性消息格式

该选项用于设置编译器产生的建议性消息的格式。警告和错误消息通过其他选项独立控制。关于更改这些消息的格式的信息，请参见**第 4.8.28 节 “--ERRFORMAT：定义编译器消息的格式”**和**第 4.8.60 节 “--WARNFORMAT：设置警告消息格式”**。

关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

如果使用 MPLAB IDE 进行编译，编译器消息的格式会自动配置为 IDE 期望的格式。如果使用该 IDE 进行编译，则建议不要调整消息格式。

4.8.40 --NODEL：不删除临时文件

在编译时指定 `--NODEL` 将指示 `xc8` 不要删除编译过程中创建的中间文件和临时文件。

4.8.41 --OBJDIR：指定中间文件的目录

该选项允许在 `xc8` 中指定用于放置其中间文件的目录。如果省略该选项，中间文件将创建在当前工作目录中。

该选项将不会设置输出文件的位置，需改为使用 `--OUTDIR`。更多信息，请参见**第 4.8.43 节 “--OUTDIR：指定输出文件的目录”**和**第 4.8.10 节 “-O：指定输出文件”**。

4.8.42 --OPT：调用编译器优化

--OPT选项用于控制所有编译器优化器。如果未指定该选项，或者它指定为--OPT=all，则将使能 space 和 asm 优化（见下文）。用户可以通过使用 --OPT=none 禁止优化，也可以控制各个优化器，例如：--OPT=asm 将仅使能一些汇编器优化。

表 4-13 列出了可用的优化类型。

表 4-13： 优化选项

选项名称	功能
asm	选择优化基于 C 源代码生成的汇编代码 （默认值）
asmfile	选择优化汇编源文件
debug	精确调试优先于优化
speed	优先考虑可产生更快代码的优化
space	优先考虑可产生更短代码的优化 （默认值）
all	使能所有编译器优化
none	不使用任何编译器优化

请注意，汇编源文件和基于 C 源代码生成的中间汇编文件的汇编器优化使用不同的子选项控制。

speed 和 space 子选项是对立的。空间优化是默认值。如果同时指定了 speed 和 space 子选项，则 speed 优化优先。如果请求 all 优化，则会使能 space 优化。这些优化会影响由汇编器执行的过程抽象，以及代码生成阶段的其他优化。

选择 asmfile 时将优化汇编源文件，否则编译器将不优化它。与之相对， asm 控制用于优化基于 C 代码生成的汇编代码，它是一项默认情况下使能的优化。

一些编译器优化可能会影响调试代码的能力。使能 debug 子选项可能会限制某些在其他情况下通常会执行并影响调试的优化。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.43 --OUTDIR：指定输出文件的目录

该选项允许为 xc8 指定用于放置其输出文件的目录。如果省略该选项，输出文件将创建在当前工作目录中。更多信息，另请参见**第 4.8.41 节 “--OBJDIR：指定中间文件的目录”**和**第 4.8.10 节 “-O：指定输出文件”**。

© 2013 Microchip Technology Inc.

4.8.44 --OUTPUT= type：指定输出文件类型

该选项用于指定输出文件的类型。如果未指定 --OUTPUT 选项，输出文件的名称将与项目名称相同（见**第 4.3 节 “编译序列”**）。

表 4-14 列出了可用的输出文件格式。通过提供一个*逗号*分隔的标记列表，可以指定多种输出格式。并不是所有格式都受 Microchip 开发工具支持。

指定库格式的那些输出文件类型会在执行编译最终阶段之前停止编译过程。因此，如果指定的输出文件格式列表包含诸如 lib 或 all 之类的格式，将不会生成其他格式。

表 4-14： 输出文件格式

类型标记	文件格式
lib	目标库文件
lpp	p 代码库文件
intel	Intel HEX （默认值）
tek	Tektronic
aahex	美国自动化符号 HEX 文件
mot	Motorola S19 HEX 文件
bin	二进制文件
mcof	Microchip COFF （默认值）

因此，例如：

```
xc8 --CHIP=16F877AA --OUTPUT=lpp lcd_init.c lcd_data.c lcd_msgs.c
```

会将三个指定文件编译为 LPP （p 代码）库。

4.8.45 --PASS1：编译为 p 代码

--PASS1 选项用于通过解析器生成 p 代码中间文件（.p1 文件），然后停止编译。如果要创建 p 代码库文件，则需要生成此类文件，但如果需要，编译器能够在一个步骤中生成库文件。关于指定库输出文件类型的信息，请参见**第 4.8.44 节 “--OUTPUT= type：指定输出文件类型”**。

4.8.46 --PRE：生成预处理的源代码

--PRE 选项用于生成扩展名为 .pre 的预处理 C 源文件（也称为模块或翻译单元）。对于确保预处理器宏扩展为您希望的内容，这可能很有用。使用该选项还可以创建不需要任何独立头文件的 C 源文件。如果将 .pre 文件重命名为 .c 文件，则可以将它们传递给编译器进行后续处理。需要将文件发送给一位同事，或者需要获取技术支持时，这非常有用，因为可以不必发送所有头文件（可能驻留在多个目录中）。

如果希望查看 printf() 系列函数的预处理源代码，*不要*使用该选项。该函数的源代码由编译器定制，但仅在代码生成器扫描完项目的 printf() 使用情况之后。因此，由于 --PRE 选项会在预处理器阶段之后停止编译，所以将不会执行代码生成器，并且不会处理任何 printf() 代码。如果省略该选项， printf() 的预处理源代码将自动保留在文件 doprnt.pre 中。

4.8.47 --PROTO：生成原型

--PROTO 选项用于生成 .pro 文件，其中将包含指定源文件中所有函数的 ANSI C 和 K&R 样式的函数声明。生成的每个 .pro 文件将使用与相应源文件相同的基本名称。原型文件将同时包含ANSI C样式的原型和旧式的C函数声明，它们将处于条件编译块内。

每个 .pro 文件的 extern 声明都应编辑到全局头文件中，然后可以将后者包含到项目的所有C源文件中。.pro文件可能还包含源文件局部函数的static声明。这些static声明应编辑到源文件的开头。

为了演示 --PROTO 选项的操作，输入以下源代码作为文件 test.c：

```
#include <stdio.h>
add(arg1, arg2)
int *   arg1;
int *   arg2;
{
        return *arg1 + *arg2;
}

void printlist(int * list, int count)
{
        while (count--)
                printf("d " *list++);
        putchar('\n');
}
```

如果使用以下命令进行编译：

```
xc8 --CHIP=16F877AA --PROTO test.c
```

xc8 将生成包含以下声明的 test.pro 文件，之后可以根据需要进行编辑：

```
/* Prototypes from test.c */
/* extern functions - include these in a header file */
#if     PROTOTYPES
extern int add(int *, int *);
extern void printlist(int *, int);
#else         /* PROTOTYPES */
extern int add();
extern void printlist();
#endif        /* PROTOTYPES */
```


© 2013 Microchip Technology Inc.

4.8.48 --RAM：调整 RAM 范围

该选项用于调整为目标器件指定的默认 RAM。默认存储器将包括为目标 PIC10/12/16 器件指定的所有片上 RAM，因此只有存在特殊的存储器要求时，才需要使用该选项。该选项通常用于保留存储器（减少可用的存储器量）。指定不处于目标器件中的额外存储器通常可以成功编译，但可能会在运行时导致代码失败。

每个目标器件的默认 RAM 存储器在 chipinfo 文件 `PICC18.INI`（适用于 PIC18 器件）或 `PICC.INI` （适用于所有其他 8 位器件）中指定。

严格地说，该选项指定可由可写 （基于 RAM）对象使用的存储区，它不一定是包含物理 RAM 的那些存储区。将放入该选项指定范围的输出通常是程序定义的变量。

例如，要在片上已存在的存储器之外指定额外的存储器范围，可以使用：

```
--RAM=default,+100-1ff
```

这会将 100h 至 1ffh 的范围添加到片上存储器中。要仅使用外部范围并忽略所有片上存储器，可以使用：

```
--RAM=0-ff
```

该选项还可用于保留已在 chipinfo 文件中定义为片上存储器的存储器范围。要实现这一点，需要提供使用减号字符 - 作为前缀的范围，例如：

```
--RAM=default,-100-103
```

将使用所有已定义的片上存储器，但不使用范围100h至103h中的地址来分配RAM对象。

该选项还可用于为 PIC18 器件上的 `far` 对象指定 RAM。这些对象存储在 PIC18 扩展存储器中。对于使用该选项指定的任何额外存储器，如果其地址高于片上程序存储器，则假定它是以 RAM 方式实现的扩展存储器。

例如，要指示已在扩展存储空间中的地址 0x20000 至 0x20fff 实现了该 RAM，可以使用以下选项。

```
--RAM=default,+20000-20fff
```

该选项将调整链接器类所使用的存储器范围 （见**第 7.2.1 节 “-Aclass =low-high,...”**），从而调节放在该类中的 psect 中的所有对象。psect 中包含的已通过链接器显式置于某个存储器地址，即不放入存储器类中的所有对象 （见**第 7.2.19 节 “-Pspec”**）都不会受该选项影响。关于 psect 的介绍指南，请参见**第 5.15.1 节 “程序段”**。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.49 --ROM：调整 ROM 范围

该选项用于更改为目标器件指定的默认 ROM。默认存储器将包括为目标 PIC10/12/16 器件指定的所有片上 ROM，因此只有存在特殊的存储器要求时，才需要使用该选项。该选项通常用于保留存储器（减少可用的存储器量）。指定不处于目标器件中的额外存储器通常可以成功编译，但可能会在运行时导致代码失败。

每个目标器件的默认 ROM 存储器在 chipinfo 文件 `PICC18.INI`（适用于 PIC18 器件）或 `PICC.INI`（适用于所有其他 8 位器件）中指定。

严格地说，该选项指定可由只读（基于 ROM）对象使用的存储区，它不一定是包含物理 ROM 的那些存储区。在生成可通过自举程序下载到系统中的代码时，目标存储器可能是某种形式的（易失性）RAM。将放入该选项所指定范围中的输出通常是可执行代码和使用 `const` 限定的所有数据变量。

例如，要在片上存储器之外指定额外的存储器范围，可以使用：

```
--ROM=default,+100-2ff
```

这会将 100h 至 2ffh 的范围添加到片上存储器中。要仅使用外部范围并忽略所有片上存储器，可以使用：

```
--ROM=100-2ff
```

该选项还可用于保留已在芯片配置文件中定义为片上存储器的存储器范围。要实现这一点，需要提供使用减号字符 - 作为前缀的范围，例如：

```
--ROM=default,-100-1ff
```

将使用所有已定义的片上存储器，但不使用范围 100h 至 1ffh 中的地址来分配 ROM 对象。

该选项将调整链接器类所使用的存储器范围（见**第 7.2.1 节 “-Aclass =low-high,...”**），从而调节放在该类中的 psect 中的所有对象。psect 中包含的已通过链接器显式置于某个存储器地址，即不放入存储器类中的所有对象（见**第 7.2.19 节 “-Pspec”**）都不会受该选项影响。关于 psect 的介绍指南，请参见**第 5.15.1 节 “程序段”**。

请注意，一些 psect 必须链接到某个阈值地址以上，最明显的就是一些保存常量数据的 psect。使用该选项去除高端存储器地址范围可能会使得无法放置这些 psect。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.50 --RUNTIME：指定运行时环境

`--RUNTIME` 选项用于控制作为运行时环境的一部分包含的内容。运行时环境封装在运行时提供、未由用户定义而是由编译器提供的所有代码，通常以库代码或编译器生成的源文件形式提供。

默认情况下会使能所有必需的运行时功能，该选项不是正常编译所必需的。

© 2013 Microchip Technology Inc.

请注意，清零或初始化变量的代码（默认情况下包含）会破坏 STATUS 寄存器的内容。对于中档和低档器件，如果需要使用该寄存器中的 TO 或 PD 位检查复位原因，则必须同时使能 resetbits 子选项。关于如何使用该功能，请参见**第 5.10.1.4 节“STATUS 寄存器保存”**。可用的子选项如表 4-15 所示。

表 4-15： 运行时环境子选项

子选项	控制	开启 （+）时的含义	默认状态
init	在主程序模块中提供用于将初始值的 ROM 镜像复制到 RAM 变量中的代码。	ROM 镜像被复制到 RAM 中，已初始化变量将在 main() 中包含它们的初始值。	开启
clib	通过链接器将库文件包含到输出代码中。	库文件被链接到输出中。	开启
clear	在主程序模块中提供用于清零未初始化变量的代码。	未初始化变量将被清零，并在 main() 中包含 0。	开启
config	使用默认配置字节对器件编程。	未指定的配置位将被赋予一个默认值。（仅限 PIC18）	关闭
download	调节将用于自举程序的 Intel HEX 文件。	Intel HEX 文件中的数据记录会被填充为 16 字节长度，并在 16 字节边界处对齐。启动代码将不采用某些寄存器中的复位值。	关闭
no_startup	启动模块是否与用户定义的代码链接。	将不链接启动模块。	关闭
osccal	使用振荡器常量初始化振荡器。	将校准振荡器。（仅限 PIC10/12/16）	开启
oscval: *value*	设置内部时钟振荡器校准值。	将使用所提供的 *value* 校准振荡器。（仅限 PIC10/12/16）	未指定
keep	编译后是否删除启动模块源文件 （startup.as）。	不删除启动模块。	开启
plib	是否链接外设库。	在编译中链接外设库。（仅限 PIC18）	开启
resetbits	在启动时保存掉电和超时 STATUS 位。	保存 STATUS 位。（仅限 PIC10/12/16）	关闭
stackcall	在硬件堆栈填满后，允许函数调用使用查表方法。	在堆栈未耗尽时通过 CALL 指令调用函数，之后通过查找表调用函数。（仅限 PIC10/12/16 器件）	关闭

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.51 --SCANDEP：扫描相依性关系

使用该选项时，将生成 .dep 和 .d 相依性关系文件。相依性关系文件会列出源文件所依赖的那些文件。当一个文件通过 #include 包含到另一个文件中时，就会产生相依性关系。 .d 文件格式由基于 GCC 的编译器使用，其中包含与 .dep 文件相同的信息。

如果使用了该选项，编译将在预处理阶段之后停止。

4.8.52 --SERIAL：在该程序存储器地址处存储一个值

该选项允许将一个十六进制代码存储在程序存储器中的特定地址处。该选项的典型任务可能将某个序列号定位到程序存储器中。

要存储的数据的字节宽度由选项中十六进制代码参数的字节宽度决定。例如，要在程序存储器地址 1000h 存储 1 字节值 0，可以使用 --SERIAL=00@1000。要将同一个值存储为一个 4 字节的量，可以使用 --SERIAL=00000000@1000。

该选项的功能与相应的 HEXMATE 选项完全相同。关于更多详细信息和可以用于该选项的高级控制，请参见**第 8.6.1.15 节 “-SERIAL”**。

驱动程序还会在存储该值的位置定义一个标号，可以从 C 代码中以 _serial0 形式引用它。要使能对该符号的访问，请记住要声明它，例如：

```
extern const int _serial0;
```

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.53 --SETOPTION：为应用程序设置命令行选项

该选项用于在编译时为指定的应用程序提供替代的命令行选项。该选项的一般格式如下：

--SETOPTION=*app*,*file*

其中，*app* 部分指定将接收新选项的应用程序，*file* 部分指定包含将传递给应用程序的额外选项的文件的名称。大多数项目都不需要该选项。

如果为某个部分指定了多个选项，则必须在选项文件新的一行中输入每个选项。该选项也可以用于从编译序列中去除某个应用程序。如果 *file* 参数指定为 *off*，则将跳过指定应用程序的执行。在大多数情况下，并不希望如此，因为几乎所有应用程序对于编译过程的成功都非常关键。禁止一个关键应用程序将导致灾难性失败。但是，如果最终结果并不依赖于某个非关键应用程序的功能，则允许跳过该非关键应用程序，例如 CLIST 或 HEXMATE。

4.8.54 --SHROUD：混淆 p 代码文件

在需要从机密源代码编译并分发 p 代码文件或 p 代码库时，应使用该选项。

C 注释（通常都包含到这些文件中）以及行号和变量名都会被删除或进行混淆处理，从而无法通过分发的文件重建原始源代码。

© 2013 Microchip Technology Inc.

4.8.55 --STRICT：严格符合 ANSI 标准

--STRICT 选项用于使所有特殊的非标准关键字严格符合 ANSI C 标准。

MPLAB XC8 C 编译器支持各种特殊关键字（如 persistent 类型限定符）。如果使用了 --STRICT 选项，这些关键字将更改为在关键字开头包含两个*下划线*字符（如 __persistent），从而严格符合 ANSI 标准。因此，如果使用了该选项，则需要在代码中使用限定符 __persistent，而不是 persistent。

请注意，使用该选项可能导致某些标准头文件（如 <xc.h>）出现问题，因为它们包含特殊的关键字。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.56 --SUMMARY：选择存储器摘要输出类型

使用该选项可以选择在编译之后显示的存储器摘要的类型。默认情况下，或如果选择了 mem 子选项，则会显示存储器摘要。它将显示所有存储空间的整体存储器使用情况。

通过使能 psect 子选项，可以显示 psect 摘要。它会显示各个 psect（由链接器对它们进行分组之后），以及它们涵盖的存储器范围。表 4-16 列出了可用的摘要类型。编译时打印的输出通常对应于存储器设置。

表 4-16： 存储器摘要子选项

子选项	控制
psect	将显示 psect 名称及其链接地址的摘要。
mem	将显示已用存储器的简明摘要。（默认值）
class	将显示每个存储空间中所有类的摘要。
hex	将显示组成最终输出文件的地址和 HEX 文件的摘要。
file	摘要信息将在屏幕上显示，并保存到文件中。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.57 --TIME：报告编译过程每个阶段所用的时间

编译时添加 --TIME 选项将生成显示完成编译过程每个阶段所用时间的摘要。

4.8.58 --VER：显示编译器的版本信息

--VER 选项将显示正在运行编译器的版本，然后退出编译器。

4.8.59 --WARN：设置警告级别

--WARN 选项用于设置编译器警告级别阈值。允许的警告级别范围为 -9 至 9。警告级别决定编译器对于可疑类型转换和构造的严格程度。每条编译器警告都具有指定的警告级别；警告级别越高，警告消息就越重要。如果警告消息的警告级别超出所设置的阈值，编译器将会打印警告。默认的警告级别阈值为 0，即允许所有正常的警告消息。

请小心使用该选项，因为一些警告消息指示代码很可能会在执行期间发生失败，或损害可移植性。

警告消息可以使用--MSGDISABLE选项单独禁止，请参见**第4.8.38节“--MSGDISABLE禁止警告消息”**。关于编译器消息系统的完整信息，另请参见**第 4.6 节“编译器消息”**。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**或**第 4.10 节 “MPLAB X Universal Toolsuite 等效选项”**。

4.8.60 --WARNFORMAT：设置警告消息格式

该选项用于设置编译器产生的警告消息的格式。关于该选项的更多信息，请参见**第 4.8.28 节“--ERRFORMAT：定义编译器消息的格式”**。关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

如果使用 MPLAB IDE 进行编译，编译器消息的格式会自动配置为 IDE 期望的格式。如果使用该 IDE 进行编译，则建议不要调整消息格式。

© 2013 Microchip Technology Inc.

4.9 MPLAB IDE V8 UNIVERSAL TOOLSUITE 等效选项

从 Microchip 的 MPLAB IDE 中进行编译时，执行的仍然是编译器的命令行驱动程序 xc8，通过它来编译程序。 Universal Toolsuite 插件可以管理用于访问编译器选项的 MPLAB IDE *Build Options*（编译选项）对话框，其中的大多数图形控件最终会调整驱动程序的命令行选项。在进行编译时，您可以在 MPLAB IDE 的 *Output*（输出）窗口中查看所使用的命令行选项。

以下对话框和描述说明了对话框控件和命令行选项之间的映射。由于该工具套件在所有 HI-TECH 品牌编译器之间通用，所以并不是所有选项都适用于 MPLAB XC8 C 编译器。

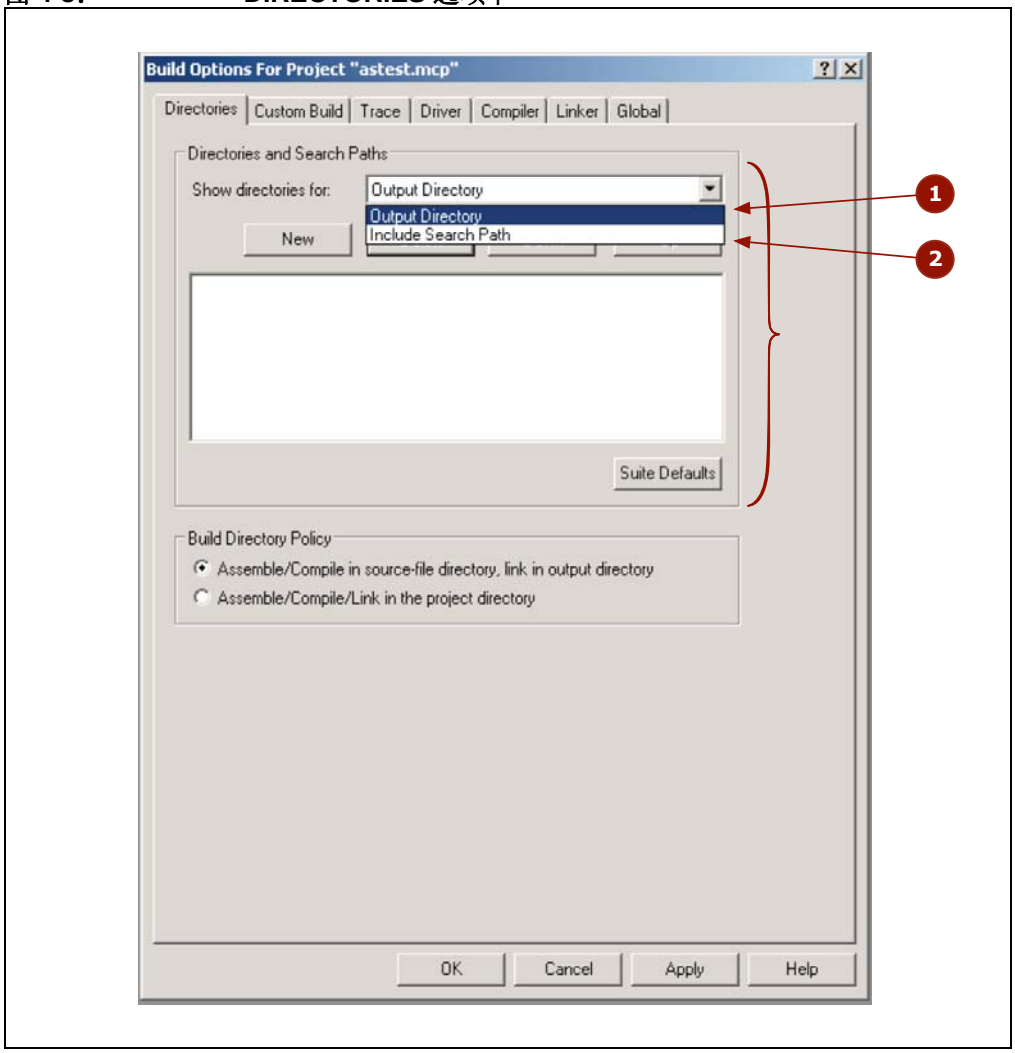
如果使用 MPLAB X IDE 进行编译，请参见**第 4.10 节“MPLAB X Universal Toolsuite 等效选项”**。

下面几页给出了 MPLAB IDE 图形用户界面的屏幕截图。此外还包含了选项卡、选项和选择的详细说明。

4.9.1 Directories （目录）选项卡

该对话框中的选项用于控制某些文件的输出和搜索目录。请参见图 4-5 和随后的等效命令行选项。

图 4-5： DIRECTORIES 选项卡



1. Output Directory （输出目录）：该选择项使用大括号内包含的按钮和域来指定编译器输出文件的输出目录。该选择项由 MPLAB IDE 在内部处理，不使用驱动程序选项；但是，它在功能上等效于 `--OUTDIR` 驱动程序选项 （见**第 4.8.43 节 “--OUTDIR：指定输出文件的目录”**）。
2. Include Search Path （包含搜索路径）：该选择项使用大括号内包含的按钮和域来指定包含文件（头文件）搜索目录。请参见**第 4.8.5 节 “-I 包含搜索路径”**。

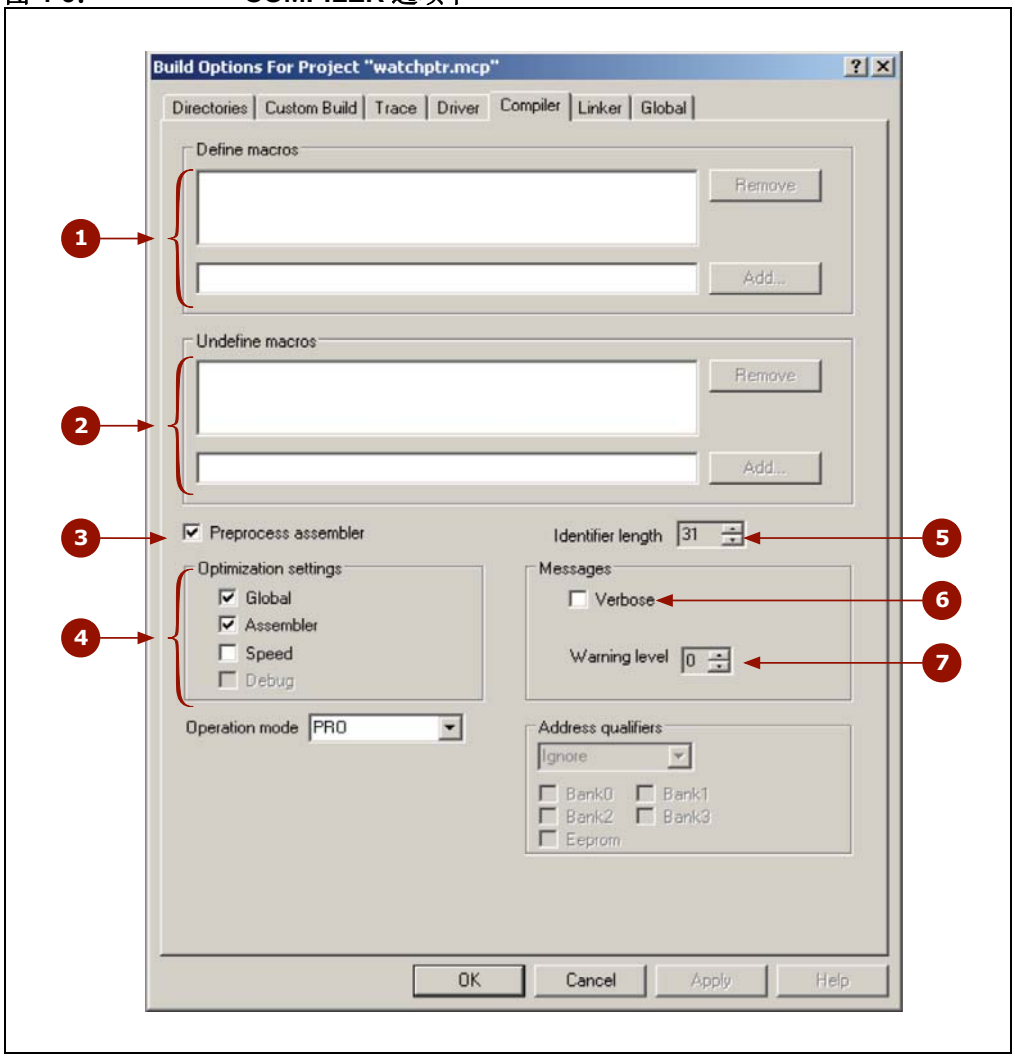
© 2013 Microchip Technology Inc.

4.9.2 Compiler （编译器）选项卡

该对话框中的选项用于控制一直到代码生成的各个编译方面。请参见图 4-6 和随后的等效命令行选项。

1. Define macros （定义宏）：大括号内包含的按钮和域可以用于定义预处理器宏。请参见**第 4.8.2 节 “-D：定义宏”**。
2. Undefine macros （取消定义宏）：大括号内包含的按钮和域可以用于取消定义预处理器宏。请参见**第 4.8.14 节 “-U：取消定义宏”**。
3. Preprocess assembler （预处理汇编器）：复选框控制预处理器是否扫描汇编源文件。请参见**第 4.8.11 节 “-P：预处理汇编文件”**。

图 4-6： COMPILER 选项卡



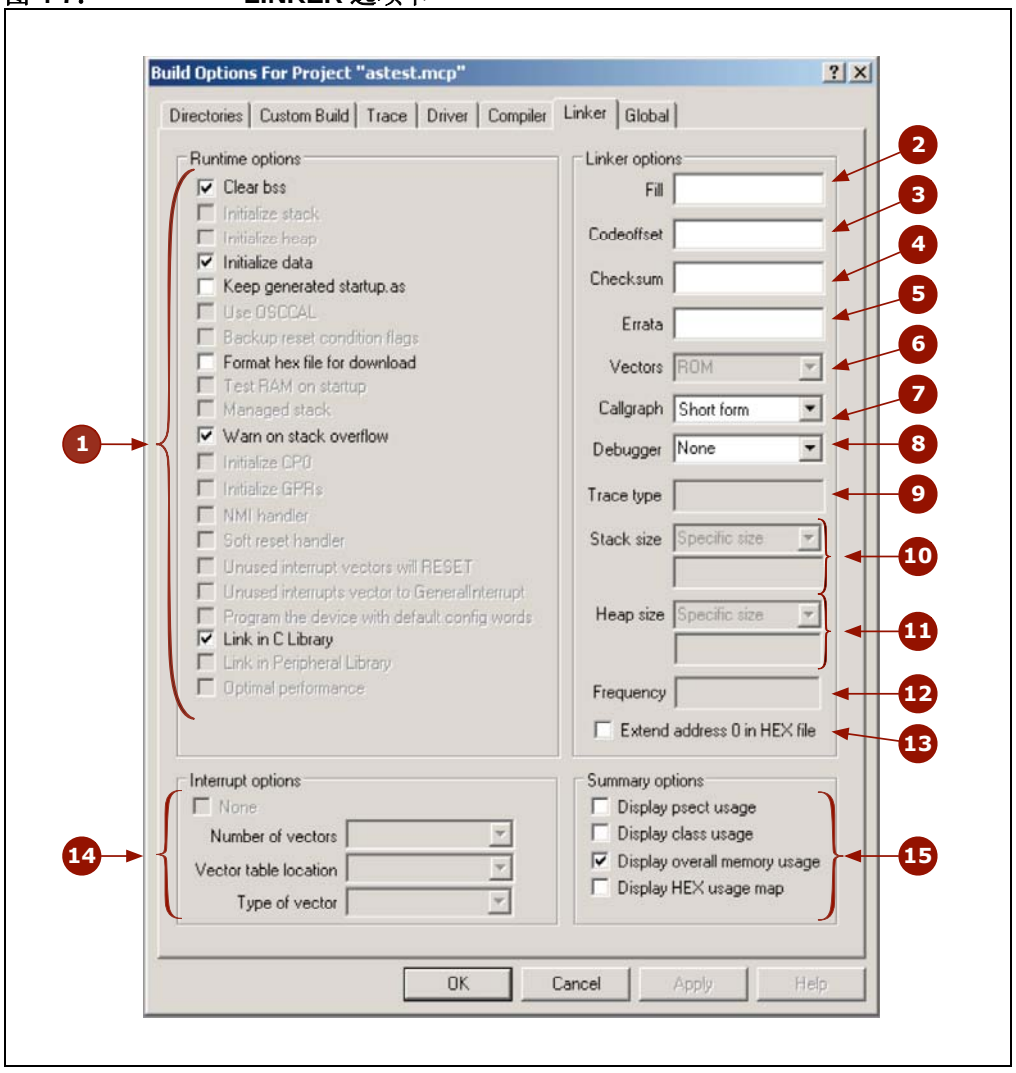
4. Optimization settings （优化设置）：这些控件用于调整编译器采用的不同优化。请参见**第 4.8.42 节 “--OPT：调用编译器优化”**。
5. Identifier length （标识符长度）：该选择项控制 C 源文件中的最大标识符长度。请参见**第 4.8.9 节 “-N：标识符长度”**。

6. Verbose （详细）：该复选框控制在编译时是否显示编译器应用程序的完整命令行。请参见**第 4.8.15 节 “-V：详细编译”**。
7. Warning level （警告级别）：该选择项用于设置警告级别打印阈值。请参见**第 4.8.59 节 “--WARN：设置警告级别”**。
8. Operation mode （工作模式）：该选择项使用户可以强制使用默认模式之外的其他可用工作模式 （例如，免费、标准或专业）。请参见**第 4.8.37 节 “--MODE：选择编译器工作模式”**。
9. Address qualifiers （地址限定符）：该选择项允许用户选择地址限定符的行为。请参见**第 4.8.16 节 “--ADDRQUAL：设置编译器对于存储器限定符的响应”**。

4.9.3 Linker （链接器）选项卡

该对话框中的选项用于控制编译的链接步骤。请参见图 4-7 和随后的等效命令行选项。

图 4-7： LINKER 选项卡



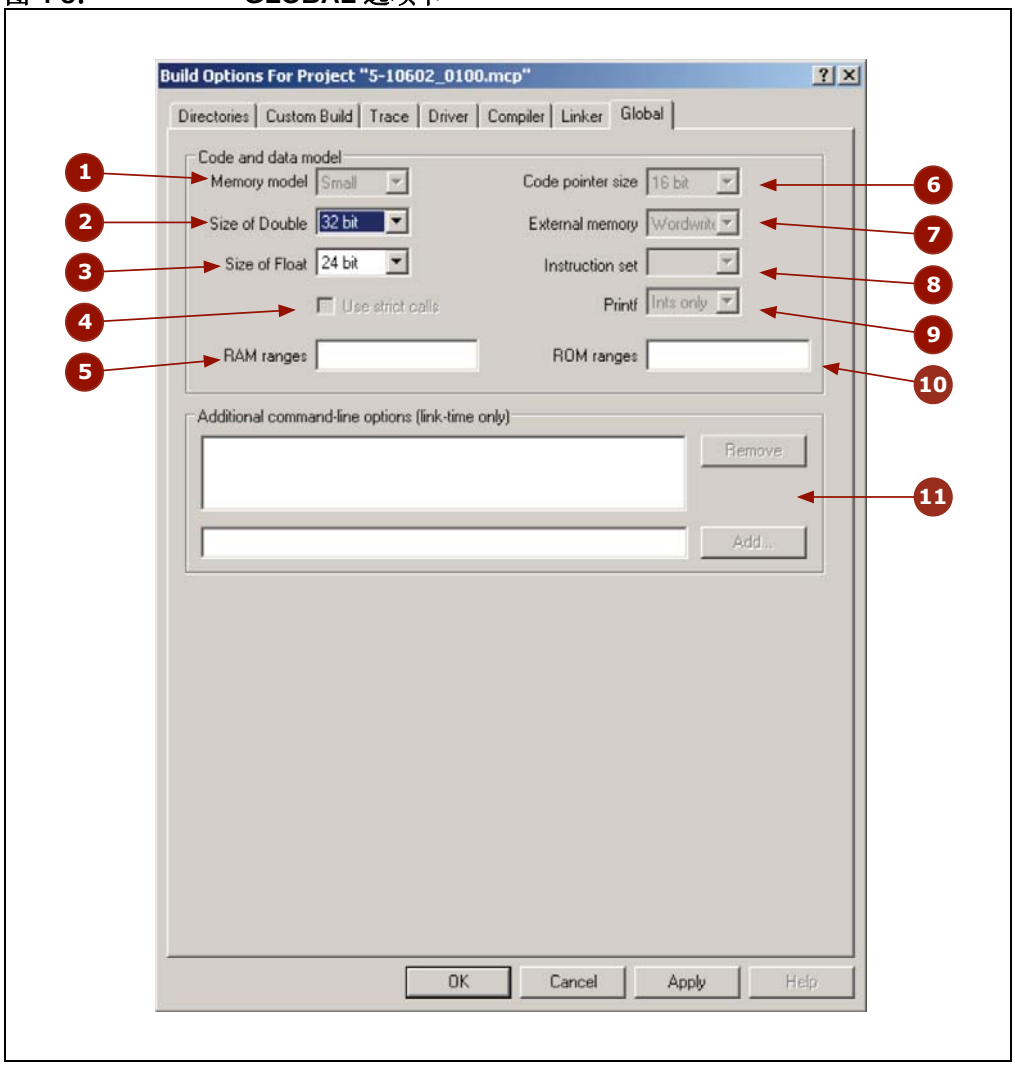
© 2013 Microchip Technology Inc.

1. Runtime options （运行时选项）：这些复选框用于控制编译器可以采用的许多运行时功能。请参见**第 4.8.50 节 “--RUNTIME：指定运行时环境”**。
2. Fill （填充）：该字段用于为未用存储单元指定一个填充值。请参见**第 4.8.30 节 “--FILL：填充未用的程序存储器”**。
3. Codeoffset （代码偏移量）：该字段用于指定程序的偏移量。请参见**第 4.8.22 节 “--CODEOFFSET：将程序代码偏移到某个地址”**。
4. Checksum （校验和）：该字段用于指定校验和规范。请参见**第 4.8.17 节 “--ASMLIST：生成汇编列表文件”**。
5. Errata （勘误）：该字段用于控制对勘误问题采用哪些软件规避措施。请参见**第 4.8.27 节 “--ERRATA：指定勘误规避措施”**。
6. Vectors （向量）：不适用。
7. Callgraph （调用图）：不适用。
8. Debugger （调试器）：该选择项用于选择硬件调试器的类型。请参见**第 4.8.23 节 “--DEBUGGER：选择调试器类型”**。
9. Trace type （跟踪类型）：尚未实现。仅实现了本机跟踪。
10. Stack size （堆栈大小）：不适用。
11. Heap size （堆大小）：不适用。
12. Frequency （频率）：不适用。
13. Extend address 0 in HEX file （在 HEX 文件中扩展地址 0）：该选项指定 Intel HEX 文件应将高端地址初始化为零。请参见**第 4.8.44 节 “--OUTPUT= type：指定输出文件类型”**。
14. Interrupt options （中断选项）：不适用。
15. Summary options （摘要选项）：这些复选框用于控制在编译之后打印哪些摘要。请参见**第 4.8.56 节 “--SUMMARY：选择存储器摘要输出类型”**。

4.9.4 Global （全局）选项卡

该对话框中的选项用于控制适用于代码生成和链接步骤（汇编的第二阶段）的各个编译方面。请参见图 4-8 和随后的等效命令行选项。

图 4-8： GLOBAL 选项卡



1. Memory model （存储器模型）：不适用。
2. Size of Double （双精度浮点型的长度）：该选择项用于选择 `double` 类型的长度。请参见**第 4.8.24 节 “--DOUBLE：选择双精度浮点型的种类”**。
3. Size of Float （浮点型的长度）：该选择项用于选择 `float` 类型的长度。请参见**第 4.8.31 节 “--FLOAT：选择浮点型的种类”**。
4. Use strict calls （使用严格调用）：不适用。
5. RAM ranges （RAM 范围）：该字段用于调整所使用的默认 RAM （数据空间）存储器。请参见**第 4.8.48 节 “--RAM：调整 RAM 范围”**。
6. Code pointer size （代码指针长度）：不适用。

© 2013 Microchip Technology Inc.

7. External memory （外部存储器）：该选项会影响在适用的 PIC18 器件上访问外部数据接口时生成的代码。请参见**第 4.8.26 节 “--EMI：选择外部存储器接口工作模式”**。
8. Instruction set （指令集）：不适用。
9. Printf：不适用。
10. ROM ranges （ROM 范围）：该字段用于调整所使用的默认 ROM （程序空间）存储器。请参见**第 4.8.49 节 “--ROM：调整 ROM 范围”**。
11. Additional command-line options （其他命令行选项）：使用该字段可以在 Build Options 窗口小部件指定的驱动程序选项之外添加其他驱动程序选项。这些选项只会在第二个编译器 （代码生成和链接）阶段传递给驱动程序，请参见**第 4.3.2 节 “单步编译”**。

4.10 MPLAB X UNIVERSAL TOOLSUITE 等效选项

在 MPLAB X IDE 下进行编译时，执行的仍然是编译器的命令行驱动程序，通过它来编译程序。MPLAB XC8 编译器插件控制用于访问编译器选项的 MPLAB X IDE Properties（属性）对话框，但这些图形控件最终可以调整在编译时传递给命令行驱动程序的命令行选项。在进行编译时，您可以在 MPLAB X IDE 的 Output 窗口中查看所使用的命令行选项。

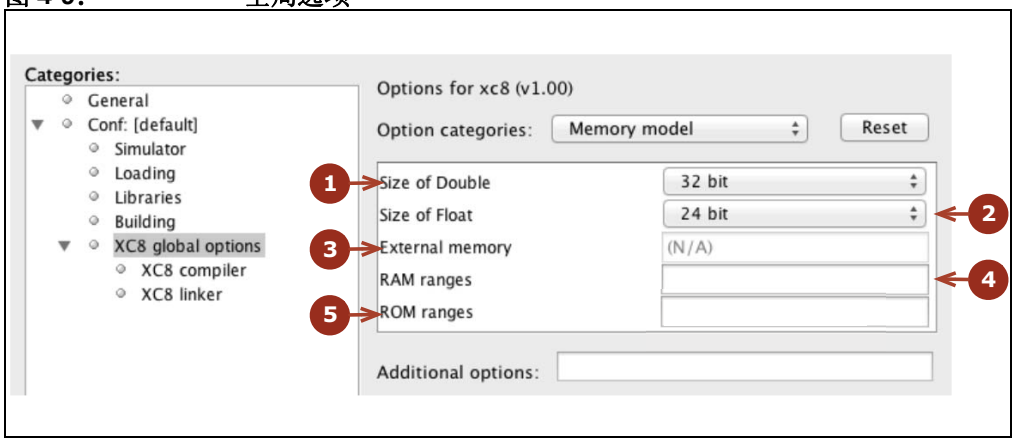
以下对话框和描述说明了 MPLAB X IDE 对话框控件和命令行选项之间的映射。

如果使用的是 MPLAB IDE v8，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**。

4.10.1 全局类别

该类别中的面板 （如图 4-9 所示）控制应用于整个项目的设置。

图 4-9： 全局选项



1. Size of Double：该选择项用于选择 `double` 类型的长度。请参见**第 4.8.24 节“--DOUBLE：选择双精度浮点型的种类”**。
2. Size of Float：该选择项用于选择 `float` 类型的长度。请参见**第 4.8.31 节“--FLOAT：选择浮点型的种类”**。
3. External memory：该选项用于指定如何执行外部存储器访问。它只会影响可以访问外部存储器的那些器件。请参见**第 4.8.26 节 “--EMI：选择外部存储器接口工作模式”**。
4. RAM ranges：该字段用于调整所使用的默认 RAM （数据空间）存储器。请参见**第 4.8.48 节 “--RAM：调整 RAM 范围”**。
5. ROM ranges：该字段用于调整所使用的默认 ROM （程序存储空间）存储器。请参见**第 4.8.49 节 “--ROM：调整 ROM 范围”**。

© 2013 Microchip Technology Inc.

4.10.2 编译器类别

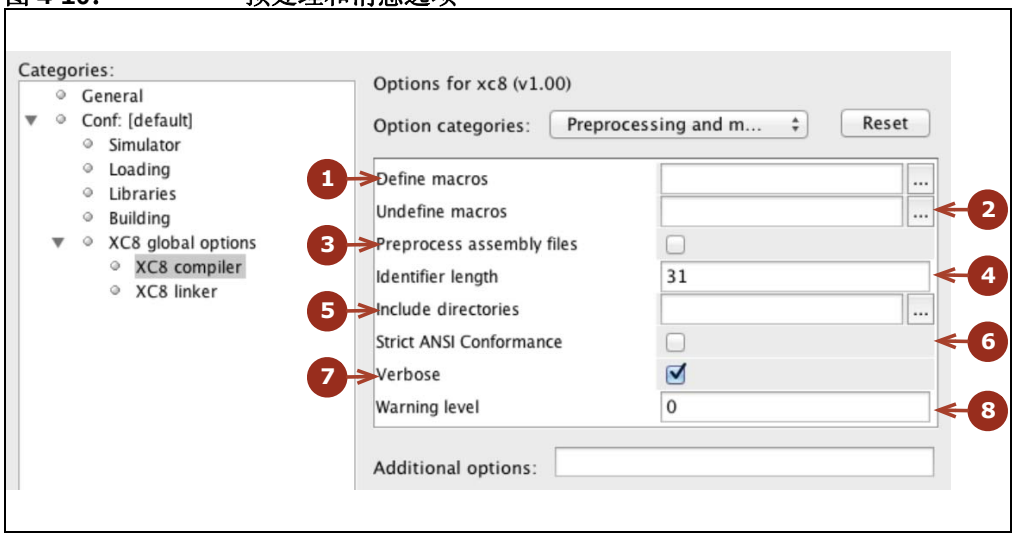
该类别中的面板控制 C 源文件编译的各个方面。

4.10.2.1 预处理和消息

这些选项涉及到 C 预处理程序和编译器生成的消息（更多信息，请参见**第 4.6 节 “编译器消息”**）。

请参见图 4-10 和随后的等效命令行选项。

图 4-10： 预处理和消息选项



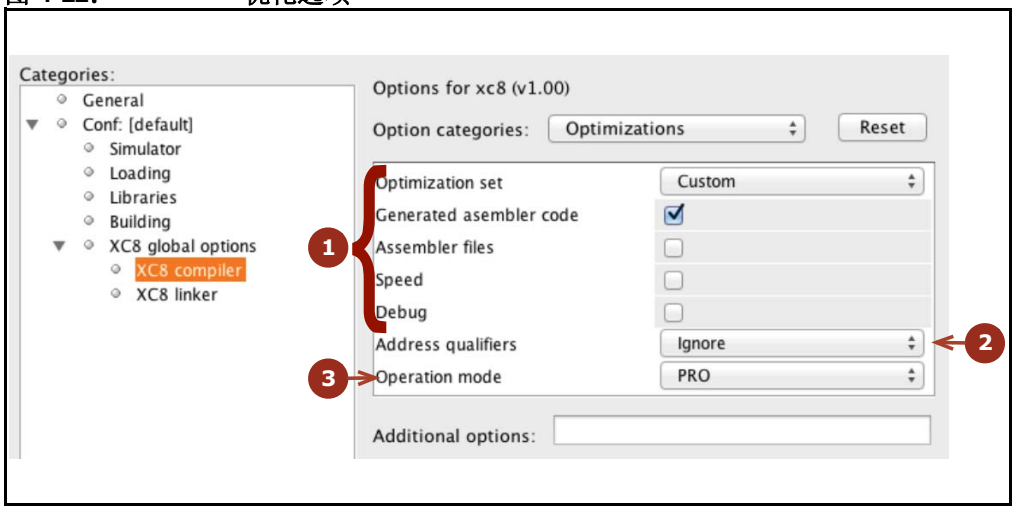
1. Define macros：按钮和域可以用于定义预处理器宏。请参见**第 4.8.2 节 “-D：定义宏”**。
2. Undefine macros：按钮和域可以用于取消定义预处理器宏。请参见**第 4.8.14 节 “-U：取消定义宏”**。
3. Preprocess assembley files （预处理汇编文件）：该复选框控制预处理器是否扫描汇编源文件。请参见**第 4.8.11 节 “-P：预处理汇编文件”**。
4. Identifier length：该选择项当前未实现。请参见**第 4.8.9 节 “-N：标识符长度”**。
5. Include directories （包含目录）：该选择项使用按钮和域来指定包含文件 （头文件）搜索目录。请参见**第 4.8.5 节 “-I：包含搜索路径”**。
6. Strict ANSI Conformance （严格符合 ANSI 标准）：它会强制编译器拒绝所有非标准关键字。请参见**第 4.8.55 节 “--STRICT：严格符合 ANSI 标准”**。
7. Verbose：该复选框控制在编译时是否显示编译器应用程序的完整命令行。请参见**第 4.8.15 节 “-V：详细编译”**。
8. Warning level：该选择项用于设置警告级别打印阈值。请参见**第 4.8.59 节 “--WARN：设置警告级别”**。

4.10.2.2 优化

这些选项（如图 4-11 所示）涉及到由编译器执行的优化。

1. Optimization controls（优化控件）：这些控件用于调整编译器采用的优化。请参见**第 4.8.42 节 “--OPT：调用编译器优化”**。选择优化集（`--OPT` 子选项：`all` 或 `none`），或下面的任意定制选项（`--OPT` 子选项：`asm`、`asmfile`、`speed/space` 和 `debug`）。
2. Address qualifiers：该选择项允许用户选择地址限定符的行为。请参见**第 4.8.16 节 “--ADDRQUAL：设置编译器对于存储器限定符的响应”**。
3. Operation mode：该选择项使用户可以强制使用默认模式之外的其他可用工作模式（例如，`free`、`std` 或 `pro`）。请参见**第 4.8.37 节 “--MODE：选择编译器工作模式”**。工作模式会影响 OCG 优化。

图 4-11： 优化选项



© 2013 Microchip Technology Inc.

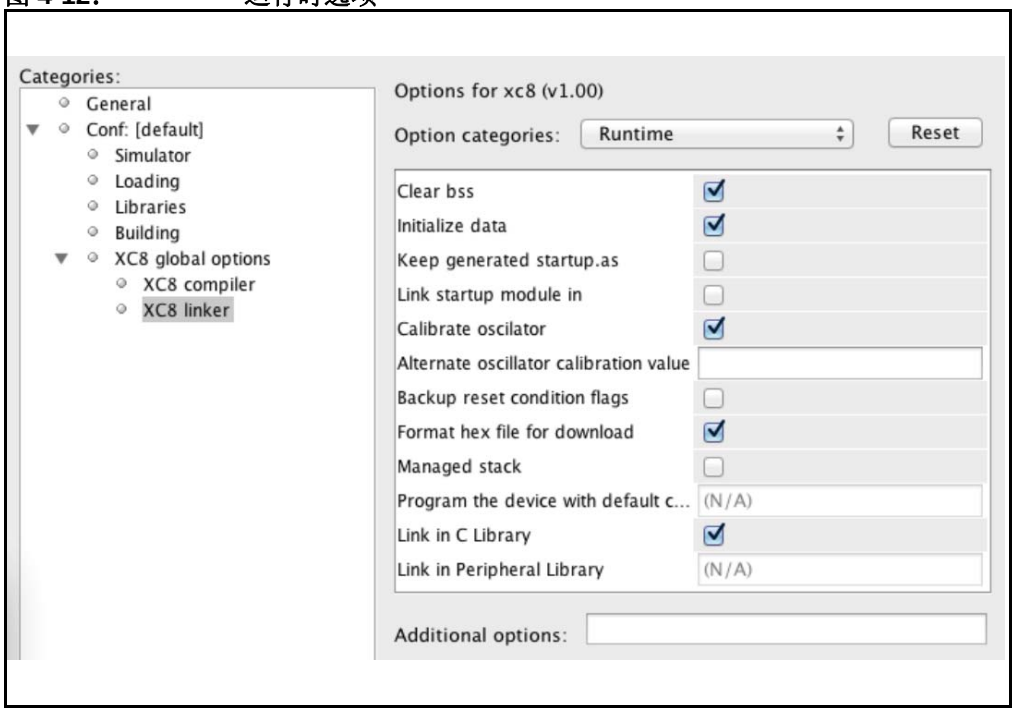
4.10.3 链接器类别

该对话框中的选项控制编译的第二阶段（包括代码生成和链接）的各个方面。

4.10.3.1 运行时

图 4-12 中的所有窗口小部件都对应于 `--RUNTIME` 选项的子选项，请参见**第 4.8.50 节“--RUNTIME：指定运行时环境”**。它们分别映射到 `--RUNTIME` 选项的 `clear`、`init`、`keep`、`no_startup`、`osccal`、`oscval`、`resetbits`、`download`、`stackcall`、`config`、`clib` 和 `plib` 子选项。

图 4-12： 运行时选项



4.10.3.2 报告

这些选项（如图 4-13 所示）涉及到编译期间和之后生成的信息。

1. Display memory usage after compilation（编译之后显示存储器使用情况）：这些复选框用于指定在编译之后显示哪些信息。它们对应于 `--SUMMARY` 选项的 `psect`、`class`、`mem` 和 `hex` 子选项，请参见**第 4.8.56 节 “--SUMMARY：选择存储器摘要输出类型”**。
2. Create summary file（创建摘要文件）：选择该复选框时会将您在上面选择的信息发送到文件以及标准输出。它对应于 `--SUMMARY` 选项的 `file` 子选项，请参见**第 4.8.56 节 “--SUMMARY：选择存储器摘要输出类型”**。
3. Create html files（创建 html 文件）：这将创建对先前编译进行总结的 HTML 文件，请参见**第 4.8.34 节 “--HTML：生成 HTML 诊断文件”**。

图 4-13： 报告选项

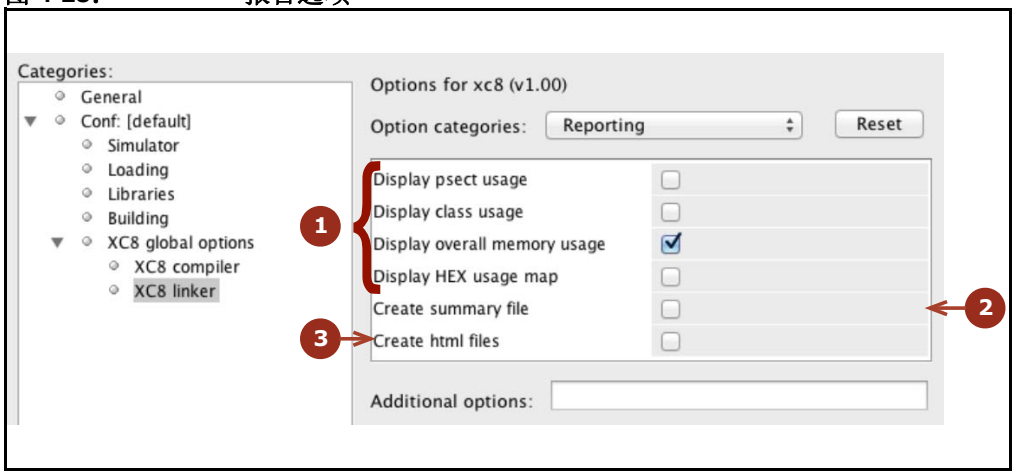
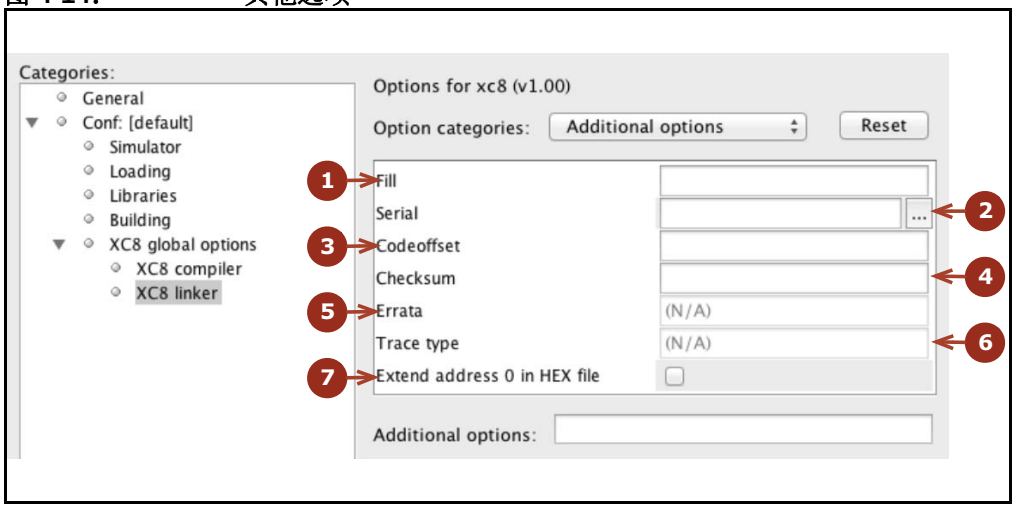


图 4-14： 其他选项



© 2013 Microchip Technology Inc.

4.10.3.3 其他

这些选项（如图 4-14 所示）涉及到一些其他选项。

1. Fill：该字段用于为未用存储单元指定一个填充值，请参见**第 4.8.30 节 “--FILL：填充未用的程序存储器”**。
2. Serial（序列）：该选项用于指定一个可以插入到输出 HEX 文件中的字符串。请参见**第 4.8.52 节 “--SERIAL：在该程序存储器地址处存储一个值”**。
3. Codeoffset：该字段用于指定程序的偏移量，请参见**第 4.8.22 节 “--CODEOFFSET：将程序代码偏移到某个地址”**。
4. Checksum：该字段用于指定校验和规范，请参见**第 4.8.19 节 “--CHECKSUM：计算校验和”**。
5. Errata：它用于定制由编译器应用的勘误规避措施，请参见**第 4.8.27 节 “--ERRATA：指定勘误规避措施”**。
6. Trace type：尚未实现。仅支持本机跟踪。
7. Extend address 0 in HEX file：该选项指定 Intel HEX 文件应将高端地址初始化为零，请参见**第 4.8.44 节 “--OUTPUT= type：指定输出文件类型”**。

注：

© 2013 Microchip Technology Inc.

第 5 章 C 语言特性

5.1 简介

MPLAB XC8 C 编译器支持许多特殊的 C 语言特性和扩展，这些特性和扩展旨在轻松执行针对 8 位 PIC 器件生成基于 ROM 的应用程序这一任务。本章将介绍特定于这些器件的特殊语言特性。

- ANSI C 标准问题
- 与器件相关的特性
- 支持的数据类型和变量
- 存储器分配和访问
- 操作符和语句
- 寄存器使用
- 函数
- 中断
- main、运行时启动和复位
- 库程序
- 混合使用 C 代码和汇编代码
- 优化
- 预处理
- 链接程序

5.2 ANSI C 标准问题

该编译器符合针对编程语言的 ISO/IEC 9899:1990 标准。它通常称为 C90 标准。在本手册中将它称为 ANSI C 标准。

下面的**第 5.2.1 节 “不符合 ANSI C 标准的方面”**中介绍了一些不符合 ANSI C 标准的方面。此外，还支持更新标准 C99 中的一些特性。

5.2.1 不符合 ANSI C 标准的方面

MPLAB XC8 C 编译器上实现的 C 语言在以下方面不符合 ANSI C 标准：函数递归。由于存储器有限和未实现硬件数据堆栈，所以不支持递归，并且函数不可重入。

编译器可以通过复制功能让从主干代码和中断代码中调用的函数看上去可重入。请参见**第 5.9.5 节 “函数复制”**。

5.2.2 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而不同。本手册将详细介绍编译器的确切行为，**附录 C “实现定义的行为”**也对其进行了全面总结。

5.2.3 通用编译器接口标准

该编译器符合 Microchip XC 编译器通用编译器接口标准（CCI）。它对 ANSI 标准进行了进一步细化，并尝试在整个 MPLAB XC 编译器系列之间标准化实现定义的行为和非标准扩展。

如果选择编写符合该标准的代码，则应使能一个编译器选项（见**第 4.8.18 节 “--CCI：实施并要求符合 CCI 标准”**）。这将指示编译器应强制要求符合标准。或者，您也可以继续使用编译器提供的非标准 ANSI 扩展来编写代码。

有一个独立文档（在编写本文档时正在等待批准）对该接口进行了介绍。

© 2013 Microchip Technology Inc.

5.3 与器件相关的特性

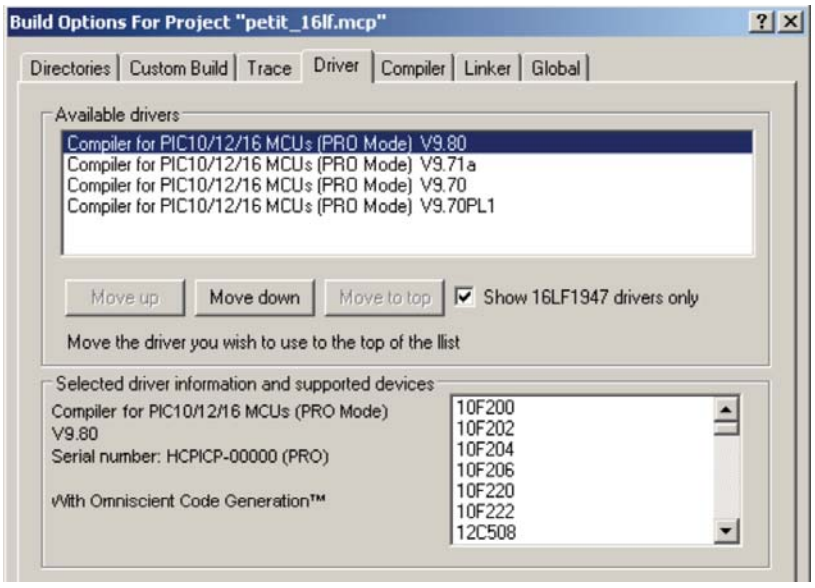
MPLAB XC8 具有一些与 8 位 PIC 架构和指令集直接相关的特性。以下几节对它们进行了详细介绍。

5.3.1 器件支持

MPLAB XC8 C 编译器旨在支持所有 8 位 PIC 器件。但是，这些系列中经常会发布新的器件。有几种方法可以检查您使用的编译器是否支持特定器件。

从 MPLAB IDE (v8) 中，打开 Build Options （编译选项）对话框。选择 Driver （驱动程序）选项卡。在 Available Drivers （可用驱动程序）字段中，选择要使用的编译器。对话框中央的 Selected Driver Information and Supported Device （选定驱动程序信息和支持的器件）区域将会显示该编译器支持的所有器件的列表。关于该对话框中的相关字段，请参见图 5-1。

图 5-1： 显示支持的器件



从命令行中，可以获得相同的信息。运行要使用的编译器，并向它传递选项 `--CHIPINFO`（见**第 4.8.21 节 “--CHIPINFO：显示支持器件的列表”**）。此时会打印所有器件的列表。

如果在 `--CHIPINFO` 选项之外还使用了 `-V` 选项，则可显示关于每款器件的更详细信息。

您也可以在您喜欢的网络浏览器中查看支持的器件。打开文件 `pic_chipinfo.html` 可查看所有支持的低档或中档器件的列表，打开文件 `pic18_chipinfo.html` 可查看所有 PIC18 器件的列表。这两个文件都位于编译器安装目录下的 DOCS 目录中。

5.3.2 指令集支持

编译器支持 PIC10/12/16 器件的所有指令集，以及标准 （传统） PIC18 指令集。目前不支持某些 PIC18 器件上提供的扩展指令模式。在适用的时候，需要确保将配置位设置为使用 PIC18 传统指令模式。

5.3.3 器件头文件

有一个头文件通常在编写每个 C 源文件时都需要包含。该文件为 <xc.h>，它是一个通用头文件，它会在您编译项目时包含其他特定于器件和特定于架构的头文件。

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 （如 CLRWDT）。

旧项目可继续使用 <htc.h> 头文件。

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。

编译器随附的头文件是特定于该编译器版本的头文件。未来的编译器版本可能随附经过修改的头文件。如果将编译器头文件复制到项目中，特别是如果修改了这些文件，请注意它们可能会与编译器的未来版本不兼容。

关于汇编包含文件（.inc）的信息，请参见**第 5.12.3.2 节“从汇编代码中访问寄存器”**。

5.3.4 堆栈

8 位 PIC 器件使用本用户指南中所称的硬件堆栈作为堆栈。该堆栈深度有限，并且不能直接操作。它仅用于存放函数返回地址，不能用于存放程序数据。[1] 编译器实现了用于存放自动变量的编译堆栈。关于它如何实现的信息，请参见**第 5.5.2.2.1 节 “编译堆栈操作”**。

您必须确保不超出最大硬件堆栈深度；否则，代码可能会发生失败。函数调用嵌套太深会使堆栈发生溢出。有一点很重要，就是需要考虑到隐式调用的库函数和中断，它们也会使用堆栈层级。通过对 --RUNTIME 编译器选项使用 stackcall 子选项，可以让编译器管理一些器件的堆栈使用，请参见**第 4.8.50 节“--RUNTIME：指定运行时环境”**。这可提供一种防止堆栈溢出的备用函数调用方法。

代码生成器会在汇编列表文件中提供一个调用图，请参见**第 6.6.6 节 “调用图”**。它将显示每个函数调用的堆栈层级，可以用作堆栈深度的指示。如果超出了最大堆栈深度，代码生成器还可能产生警告。

这两者都可用于*指示*堆栈使用情况。优化和使用中断会使程序使用的堆栈深度减小或增大，超出编译器决定的深度。

1. Microchip 16 位和 32 位器件允许使用其相应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，它是通过某种形式的入栈和出栈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。

© 2013 Microchip Technology Inc.

5.3.5 配置位访问

PIC 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。

对于 PIC18 器件，这些位可以使用配置 pragma 伪指令进行设置。（在未来版本中，也会对于其他 8 位器件引入 pragma 伪指令。）该 pragma 伪指令具有以下形式。

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

其中，*setting* 是配置设置描述符（如 WDT），*state* 是所需状态的文本描述（如 OFF）。*value* 字段是一个可以优先用于描述符的数值。

假设存在以下仅针对 PIC18 的示例。

```
#pragma config WDT = ON      // turn on watchdog timer
#pragma config WDT = 1       // an alternate form of the above
#pragma config WDTPS = 0x1A  // specify the timer postscale value
```

通过用逗号分隔每个设置 / 值对，可以使用一条 pragma 伪指令来编程多个设置。例如，以上示例可以使用一条 pragma 伪指令指定，如下所示。

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

可以使用一个数值来编程整个寄存器，而不需要单独指定各个设置，例如：

```
#pragma config CONFIG1L = 0x8F
```

每个寄存器的高半部分和低半部分必须单独编程。

与 PIC18 器件关联的设置和值可以通过一个 HTML 指南确定。打开文件 pic18_chipinfo.html，它位于编译器安装的 DOCS 目录中。单击您使用的目标器件，它将显示适用于该 pragma 伪指令的设置和值。更多信息，请查看您所使用器件的数据手册。

低档和中档器件的配置位可以使用 HI-TECH C® 中支持的 __CONFIG 宏设置，例如：

```
#include <xc.h>
__CONFIG(WDTDIS & HS & UNPROTECT);
```

要使用该宏，请确保在源文件中包含 <xc.h>。对于具有多个配置字的器件，随后每次调用 __CONFIG() 都将按顺序修改下一个配置字。通常它具有以下形式：

```
#include <xc.h>
__CONFIG(WDTDIS & XT & UNPROTECT);  // Program config. word 1
__CONFIG(FCMEN);
```

5.3.5.1 对于 PIC18 器件的遗留支持

您可以继续对旧项目使用 PIC18 配置宏。编译器支持 __CONFIG 和 __PROG_CONFIG 宏，它们分别用于指定配置位符号或配置字值。

用于 PIC18 器件的 __CONFIG 宏需要一个指定配置字单元编号的额外参数。例如：

```
__CONFIG(2, BW8 & PWRTDIS & WDTPS1 & WDTEN);  // specify symbols
```

或者，您也可以指定字面值 （如果偏好这种形式）：

```
__PROG_CONFIG(1, 0xFE57);    // specify a literal constant value
```

在 __PROG_CONFIG 宏中不能使用设置符号，在 __CONFIG 宏中不能使用字面值。

配置单元不需要按顺序进行编程。

对于新项目，请使用 pragma 伪指令而不是宏。要使用这些宏，请确保在源文件中包含 <xc.h>。这些宏的符号可以在 .cfgmap 文件中找到，这些文件包含在编译器安装的 dat/cfgmap 目录中。

5.3.5.2 配置注意事项

config pragma 伪指令和 __CONFIG 宏都不会生成可执行代码。它们都应放置在函数定义之外，以免影响函数代码的操作。

MPLAB IDE v8 具有一个对话框（*Config（配置）>Configuration bits...（配置位）*），它可用于在对器件进行编程时指定配置位。如果不选中该对话框中的 “Configuration Bits Set in Code”（在代码中设置配置位）复选框，则在代码中使用 __CONFIG 宏指定的所有配置位都会被忽略，并改为使用该对话框中的那些配置位。在处理 MPLAB IDE v8 项目时，请确保知道配置位设置的来源。

MPLAB X IDE 不允许对配置位进行调整。它们必须在源代码中使用 pragma 伪指令（或旧式的宏）指定。

配置字中的所有位都应进行编程，以防止意外的程序行为。不要将它们保留为默认 / 未编程状态。并不是所有配置位的默认状态都为逻辑高电平；一些配置位的默认状态为逻辑低电平。更多信息，请查看您所使用器件的数据手册。

© 2013 Microchip Technology Inc.

5.3.6 从 C 代码中使用 SFR

特殊功能寄存器 （Special Function Register， SFR）是一些控制 MCU 操作或器件上外设模块各个方面的寄存器。其中大多数寄存器都进行了存储器映射，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。（置于特定地址处的变量称为*绝对变量*，**第 5.5.4 节 “绝对变量”**对其进行了介绍）。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 变量在头文件中预定义，在源代码中包含 <xc.h> 头文件 （见**第 5.3.3 节 “器件头文件”**）之后即可访问它们。头文件中同时定义了位变量和带位域的结构，所以可以在源代码中使用其中任意一种形式来访问寄存器中的位。

对于映射到寄存器以及这些寄存器中的位的 C 变量，其名称将基于器件数据手册中指定的名称。但是，由于寄存器内的某些位名称可能会重复，所以命名可能会有差异。

存放位域的结构的名称通常为相应寄存器的名称后跟随 `bits`。例如，下面显示了一个代码示例，它包含了通用头文件、将 PORTA 作为一个整体清零、使用位变量将 PORTA 的 bit 0 置 1，以及使用结构 / 位域定义将 PORTA 的 bit 2 置 1。

```
#include <xc.h>
void main(void)
{
    PORTA = 0x00;
    RA0 = 1;
    PORTAbits.RA2 = 1;
}
```

要确认与所用器件有关的名称，请查看 <xc.h> 将包含的特定于器件的头文件，确定每个变量的定义。这些文件将位于编译器的 `include` 目录，并具有一个代表该器件的名称。器件和将通过 <xc.h> 包含的头文件的名称存在一对一关系，例如，针对 PIC16LF1826 器件进行编译时，<xc.h> 将包含头文件 <pic16lf1826.h>。请记住，不需要在源代码中包含这个特定于芯片的文件；它由 <xc.h> 自动包含。

从 C 代码或嵌入到 C 代码的汇编代码中访问一些 SFR 时需要小心。编译器使用一些寄存器来保存计算的中间值，直接写入这些寄存器可能导致代码失败。当 SFR 由于 C 或汇编代码直接写入它们而发生改变时，编译器并不会检测到发生了这种情况。**第 5.7 节 “寄存器使用”**列出了编译器使用的寄存器，并提供了进一步的信息。

编译器不会使用与外设关联的 SFR 来存放中间结果，您可以根据需要更改它们。请总是确保根据器件数据手册来确认外设模块的操作。

5.3.6.1 特殊的低档 / 中档寄存器问题

一些 SFR 不进行存储器映射，未在特定于器件的头文件中定义相应的变量，无法直接从 C 代码中访问。

例如，在低档器件上，W 寄存器不进行存储器映射。一些器件使用了 OPTION 和 TRIS 寄存器，这两个寄存器只能通过特殊指令进行访问，它们也不进行存储器映射。关于编译器如何访问这些寄存器的信息，请参见**第 5.3.9 节 “低档 PIC MCU 特殊指令”**。

5.3.6.2 特殊的 PIC18 寄存器问题

与 PIC18 关联的一些 SFR 可以通过组合来构成多字节值，例如，TMRxH 和 TMRxL 寄存器可以组合构成 16 位定时器计数值。根据器件和工作模式，可能会存在以某种特定方式读取这些寄存器的硬件要求，例如通常必须先读取 TMRxL 寄存器，然后再尝试读取 TMRxH 寄存器，才能获取有效的 16 位结果。

虽然可以定义一个字长的 C 变量来映射到此类寄存器 （即， `int` 变量 TMRx 映射到 TMRxL 和 TMRxH），编译器读取此类对象中字节的顺序会因表达式而不同。一些表达式要求先读取高字节 （MSB）；其他一些则先读取低字节 （LSB）。

建议使用芯片头文件中现有的 SFR 定义。SFR 的每个字节应按器件数据手册中要求的顺序直接访问。这可以产生更高的可移植性。

以下代码会将两个定时器寄存器复制到无符号的 C 变量 `count`，供以后使用。

```
count = TMR0L;
count += TMR0H << 8;
```

此外，还提供了一些宏来执行更常用定时器寄存器的读取和写入。请参见**附录 A “库函数”**中的宏 `READTIMER`*x* 和 `WRITETIMER`*x*。它们可确保使用正确的字节顺序。

© 2013 Microchip Technology Inc.

5.3.7 ID 存储单元

8 位 PIC 器件具有一些位于可寻址存储区之外的存储单元，可以用于存储程序信息，例如 ID 号。对于 PIC18 器件，config pragma 伪指令还可用于通过使用特殊寄存器名称将数据放入这些存储单元。该 pragma 伪指令的使用方式如下所示。

```
#pragma config IDLOCx = value
```

其中，*x* 是 ID 存储单元的编号（位置），*value* 是要放入该 ID 存储单元的半字节或字节。如果 *value* 大于目标器件上对于每个存储单元允许的最大值，值将被截断，并发出一条警告消息。每个 ID 存储单元的长度会因器件而不同。更多信息，请查看您所使用器件的数据手册。

例如：

```
#pragma config IDLOC0 = 1
#pragma config IDLOC1 = 4
```

将尝试使用 1 和 4 填充前两个 ID 存储单元。通过用逗号分隔每个寄存器 / 值对，可以使用一条pragma伪指令来编程多个存储单元。例如，以上示例可以使用如下形式指定。

```
#pragma config IDLOC0 = 1, IDLOC1 = 4
```

config pragma 伪指令不会生成可执行代码，所以理想情况下应放置在函数定义之外。

编译器还提供了对于 __IDLOC 宏的遗留支持。该宏必须用于低档和中档器件。该宏的使用形式如下：

```
#include <xc.h>
__IDLOC(x);
```

其中，*x* 是放入 ID 存储单元的半字节的列表。只会设定每个 ID 存储单元的低 4 位，所以以下代码：

```
__IDLOC(15F0);
```

将尝试使用以下值填充 ID 存储单元：1、 5、 F 和 0。

要使用该宏，请确保在源文件中包含 <xc.h>。

__IDLOC 宏不会生成可执行代码，所以理想情况下应放置在函数定义之外。

某些器件最高允许编程每个 ID 存储单元中的 7 个位。__IDLOC() 宏并不适用于此类器件，应改为使用 __IDLOC7(*a*,*b*,*c*,*d*) 宏。参数 *a* 至 *d* 为要编程的值。可以使用十进制或十六进制格式输入这些值，例如：

```
__IDLOC7(0x7f,1,70,0x5a);
```

在不允许对 ID 存储单元进行 7 位编程的器件上使用 __IDLOC7() 宏是不适合的。

5.3.8 位指令

只要有可能，MPLAB XC8 C 编译器都会尝试使用位指令，即使是对于非位整型值。例如，在使用按位操作符和掩码来更改整型内的某个位时，编译器将检查掩码值，以确定位指令是否可以实现相同的功能。

```
unsigned int foo;
foo |= 0x40;
```

将生成以下指令：

```
BSF _foo,6
```

要置 1 或清零整型内各个位，可以使用以下宏：

```
#define bitset(var, bitno)    ((var) |= 1UL << (bitno))
#define bitclr(var, bitno)    ((var) &= ~(1UL << (bitno)))
```

要对 foo 执行与以上相同的操作，可以如下使用 bitset 宏：

```
bitset(foo,6);
```

5.3.9 低档 PIC MCU 特殊指令

低档器件具有一些不处于正常 SFR 空间中的寄存器，这些寄存器无法使用普通的文件指令访问。它们是 OPTION 和 TRIS 寄存器。

这两个寄存器都是只写的，不能在读取其值的表达式中使用。它们只能使用特殊指令访问，编译器将自动使用这些指令。

映射到这些寄存器的变量的定义使用了 control 限定符。该限定符指示编译器，寄存器处于正常地址空间之外，需要一种不同的访问方式。对于所有其他寄存器，不应使用该限定符。

在写入这两个 SFR 变量中的任意一个时，编译器将使用相应的指令来装入值。例如，要向 TRIS 寄存器装入值，以下代码：

```
TRIS = 0xFF;
```

可能被编译器编码为：

```
MOVLW 0ffh
TRIS
```

对于具有多个输出端口的那些 PIC 器件，根据可用端口的确切数量，可能具有以下对象定义：TRISA、 TRISB 和 TRISC。这种对象的使用方式与以上所述的相同。

© 2013 Microchip Technology Inc.

5.3.9.1 振荡器校准常量

一些低档和中档器件随附了一个振荡器校准常量，它预先编程到器件的程序存储器中。可以从程序存储器中读取该常量，并写入 OSCCAL 寄存器来校准内部 RC 振荡器。

在一些低档 PIC 器件上，校准常量以 MOVLW 指令的形式存储在程序存储器顶部，例如 PIC10F509 器件。在复位时，会使程序计数器指向该指令，并在程序计数器回绕到 0x0000 （它是器件的实际复位向量）之前先执行它。默认的运行时启动程序 （见**第 5.10.1 节 “运行时启动代码”**）将自动包含在此类器件上发生复位之后将 WREG 中包含的值装入 OSCCAL 寄存器的代码。无需任何其他代码。

对于其他芯片 （如 PIC12F629 器件），振荡器常量也存储在程序存储器顶部，但以 RETLW 指令的形式。编译器的启动代码会自动生成代码来获取该值并执行配置。

在运行时，以 RETLW 指令形式存储的校准值可以使用 “函数” `__osccal_val()` 读取，因为有一个标号会被赋予 RETLW 指令地址。<xc.h> 提供了该函数的原型。例如：

```
calVal = __osccal_val();
```

通过对 `--RUNTIME` 选项 （见**第 4.8.50 节 “--RUNTIME：指定运行时环境”**）使用 `osccal` 子选项，可以关闭校准值的装入。

在运行时，该校准值可以使用宏 `_READ_OSCCAL_DATA()` 读取。为了能够使用该宏，请确保将 <xc.h> 包含到程序的相关模块中。该宏会返回校准常量，然后可以将它存储到 OSCCAL 寄存器中，如下所示：

```
OSCCAL = _READ_OSCCAL_DATA();
```

注： 存储校准常量的存储单元永远不会进行代码保护，如果对器件重新编程，它将会丢失。因此，如果使用窗口器件或闪存器件，则在擦除校准常量之前，必须先从先前的 ROM 单元中保存它。之后，必须与新的程序和数据一起，在同一存储单元重新写入该常量。
如果使用在线仿真器 （In-Circuit Emulator， ICE），则可能无法编程校准 RETLW 指令所用的存储单元。调用 `_READ_OSCCAL_DATA()` 宏将无法工作，几乎肯定不会正确返回。如果希望在 ICE 上测试包含该宏的代码，则必须在程序存储器中的相应存储单元编程一条 RETLW 指令。请记住在对实际器件进行编程时删除该指令，从而不会破坏校准值。

5.3.10 REAL ICE 支持

在使用 Microchip 的 REAL ICE 调试器时，编译器支持日志和跟踪功能 （仪器跟踪）。关于仪器跟踪功能的更多信息，请参见 REAL ICE 文档。

该编译器当前仅支持本机跟踪。并不是所有器件都支持插装跟踪，并且您使用的 IDE 同时也需要具有对于目标器件的插装跟踪支持。

日志和跟踪宏调用需要在MPLAB IDE中手动添加到您的源代码中。它们具有以下形式。

```
__TRACE(id);
__LOG(id, expression);
```

在进行编译时， MPLAB IDE v8 会自动将 `id` 替换为适当的值；但如果需要，您可以手动指定它们。跟踪 `id` 应为介于范围 0x40 至 0x7f 的常量，日志 `id` 为介于范围 0x0 至 0x7f 的常量。应为每个宏指定一个惟一编号，以便可以正确识别它。相同的有效编号可以同时用于跟踪和日志宏。

`expression` 可以为任意整型或 32 位浮点型表达式。通常情况下，该表达式只是某个变量的名称，因而将记录该变量的内容。

宏应放置在 C 源代码中的所需位置处。在执行它们时，它们将触发向调试器和 IDE 发送信息。添加跟踪和日志宏将增加代码的长度，因为它们会增加下载到器件的程序镜像中的内容。

以下给出了可添加的这些宏的示例。

```
inpStatus = readUser();
if(inpStatus == 0) {
    __TRACE(id);
    recovery();
}
__LOG(id, inpStatus);
```


© 2013 Microchip Technology Inc.

5.4 支持的数据类型和变量

5.4.1 标识符

C 变量标识符（以下对于函数标识符也是符合的）是一个由字母和数字组成的序列，其中的下划线字符 _ 算作一个字母。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。对于汇编域标识符则不是如此，它们通常以下划线开头，请参见**第 5.12.3.1 节 “等效汇编符号”**。

标识符区分大小写，所以 `main` 不同于 `Main`。

并不是标识符中的每个字符都是有效的。最大有效字符数可以使用一个选项设置，请参见**第 4.8.9 节 “-N：标识符长度”**。如果两个标识符只有最大有效字符数之后不同，则编译器会将它们视为相同的符号。

5.4.2 整型数据类型

MPLAB XC8 编译器支持 1、2、3 和 4 字节长的整型数据类型，以及单个位的类型。表 5-1 列出了数据类型及其相应的长度和算术类型。每个类型的默认类型均带有下划线。

表 5-1： 整型数据类型

类型	长度 （位）	算术类型
bit	1	无符号整型
signed char	8	有符号整型
unsigned char	8	无符号整型
signed short	16	有符号整型
unsigned short	16	无符号整型
signed int	16	有符号整型
unsigned int	16	无符号整型
signed short long	24	有符号整型
unsigned short long	24	无符号整型
signed long	32	有符号整型
unsigned long	32	无符号整型
signed long long	32	有符号整型
unsigned long long	32	无符号整型

`bit` 和 `short long` 类型是该实现中提供的非标准类型。`long long` 类型是 C99 标准类型。

所有整型值都使用最低位 （LSb）位于低地址的小尾数法格式表示。

如果类型中未指定符号，则除 `char` 类型 （总是为无符号）之外的类型均为有符号。`bit` 类型总是为无符号，符号位的概念毫无意义。

有符号值以二进制补码整型值的形式存储。

表 5-2 总结了这些类型能够存放的值的范围。该表中的符号是在源代码中包含 <limits.h> 之后可用的预处理器宏。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。

与 short long 类型关联的宏均为该实现中提供的非标准宏；与 long long 类型关联的那些宏均为 C99 标准定义的宏。

表 5-2： 整型值的范围

符号	含义	值
CHAR_BIT	char 的位数	8
CHAR_MAX	char 的最大值	127
CHAR_MIN	char 的最小值	-128
SCHAR_MAX	signed char 的最大值	127
SCHAR_MIN	signed char 的最小值	-128
UCHAR_MAX	unsigned char 的最大值	255
SHRT_MAX	short 的最大值	32767
SHRT_MIN	short 的最小值	-32768
USHRT_MAX	unsigned short 的最大值	65535
INT_MAX	int 的最大值	32767
INT_MIN	int 的最小值	-32768
UINT_MAX	unsigned int 的最大值	65535
SHRTLONG_MAX	short long 的最大值	8388607
SHRTLONG_MIN	short long 的最小值	-8388608
USHRTLONG_MAX	unsigned short long 的最大值	16777215
LONG_MAX	long 的最大值	2147483647
LONG_MIN	long 的最小值	-2147483648
ULONG_MAX	unsigned long 的最大值	4294967295
LLONG_MAX	long long 的最大值	2147483647
LLONG_MIN	long long 的最小值	-2147483648
ULLONG_MAX	unsigned long long 的最大值	4294967295

<stdint.h> 中也提供了一些宏，它们定义与固定宽度类型关联的值。

在指定有符号或无符号的 short int、short long int、long int 或 long long int 类型时，可以省略关键字 int。因此声明为 short 的变量将包含 signed short int，声明为 unsigned short 的变量将包含 unsigned short int。

一种常见的误解是，C char 类型纯粹用于 ASCII 字符操作。但是，C 语言甚至不保证默认字符表示形式为 ASCII。（该实现使用 ASCII 作为字符表示形式。）char 类型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 XC8 C 编译器，char 类型可用于多种用途——用作 8 位整型，用于存储 ASCII 字符，以及用于访问 I/O 单元。

© 2013 Microchip Technology Inc.

5.4.2.1 位数据类型和变量

MPLAB XC8 C 编译器支持 bit 整型，它可存放值 0 或 1。单个位变量可以使用关键字 bit 声明，例如：

```
bit init_flag;
```

这些变量不能为自动变量或函数参数，但可以使用 static 限定，从而可以在函数内局部定义它们。例如：

```
int func(void) {
    static bit flame_on;
    // ...
}
```

通过以通常方式在函数原型中使用 bit 关键字，函数可以返回一个位对象。值 1 或 0 将返回到在 STATUS 寄存器中的进位标志中。

位变量的行为在大多数方面类似于常规的 unsigned char 变量，但它们只能包含值 0 和 1，因此提供了一种便捷高效的方法来存储标志。 8 个位对象打包到存储器的每个字节中，所以它们不会消耗大量的内部 RAM。

只要可能，对位对象的操作都会使用单个位指令（bsf 和 bcf）执行，因而生成代码访问位对象的效率极高。

无法声明一个指向 bit 类型的指针或将位对象的地址赋予任何指针。也无法以静态方式初始化位变量，所以必须在代码自身中为它们赋予任意非零起始值（即，1）。除非使用 persistent 进行限定，否则在启动时，位对象会被清零。

将更长的整型赋予位变量时，将仅使用LSb。例如，如果位变量bitvar以如下形式赋值:

```
int data = 0x54;
bit bitvar;
bitvar = data;
```

赋值会将它清零，因为 data 的 LSb 为零。这使 bit 类型不同于 C99 标准 __Bool，后者为布尔型，而不是 1 位宽的整型。MPLAB XC8 编译器不支持 __Bool 类型。如果希望根据更长的整型是零（false）还是非零（true）而将某个位变量设置为 0 或 1，可以使用以下形式：

```
bitvar = (data != 0);
```

用于为位对象分配存储空间的 psect 使用 bit PSECT 伪指令标志声明，请参见**第 6.4.9.3 节 “PSECT”**。为位对象和 psect 指定的所有地址均为位地址。将这些地址与所有其他变量使用的字节地址进行比较时需要小心。

如果使用了 xc8 标志 --STRICT，则 bit 关键字将不可用。

5.4.3 浮点型数据类型

MPLAB XC8 编译器支持 24 位和 32 位浮点型。浮点型使用 IEEE 754 32 位格式或它经过改进（截断）的 24 位形式实现。表 5-3 列出了数据类型及其相应的长度和算术类型。

表 5-3： 浮点型数据类型

类型	长度 （位）	算术类型
`float`	24 或 32	实数
`double`	24 或 32	实数
`long double`	与 `double` 相同	实数

对于 `float` 和 `double` 值，24 位格式是默认格式。此外，还可以使用选项 `--FLOAT=24` 和 `--DOUBLE=24` 来显式指定这种格式。对于 `double` 值，如果使用了 `--DOUBLE=32` 选项，或对于 `float` 值，如果使用了 `--FLOAT=32`，则将对于它们使用 32 位格式。

变量可以分别使用 `float` 和 `double` 关键字进行声明，以存放这些类型的值。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。声明为 `long double` 的类型将使用与声明为 `double` 的类型相同的格式。所有浮点值都使用 LSb 位于低地址的小尾数法格式表示。

表 5-4 描述了这种格式，其中：

- “符号”为指示数字是正数还是负数的符号位。
- 指数为 8 位，它以加 127 的形式存储 （即，指数 0 存储为 127）。
- “尾数”为小数点右侧的尾数。在小数点左侧有一个隐含位，对于零值，隐含位为零，对于其他值，它总是为 1。零值通过零指数表示。

该数字的值为 $(-1)^{符号} \times 2^{(指数-127)} \times 1.$ 尾数。

表 5-4： 浮点型格式

格式	符号	偏移的指数	尾数
IEEE 754 32 位	x	xxxx xxxx	xxx xxxx xxxx xxxx xxxx xxxx
改进的 IEEE 754 24 位	x	xxxx xxxx	xxx xxxx xxxx xxxx

© 2013 Microchip Technology Inc.

下面的表 5-5 列出了 IEEE 754 32 位格式的一些示例。请注意，尾数列的最高位（MSb，即小数点左边的位）是隐含位，除非指数为零（这种情况下浮点值为零），否则假定它为 1。

表 5-5： 浮点型格式示例 IEEE 754

格式	数字	偏移的指数	1. 尾数	十进制
32 位	7DA6B69Bh	11111011b	1.01001101011011010011011b	2.77000e+37
		(251)	(1.302447676659)	—
24 位	42123Ah	10000100b	1.001001000111010b	36.557
		(132)	(1.142395019531)	—

使用以下过程来手动计算表 5-5 中的 32 位示例。

符号位为零；偏移的指数为 251，所以指数为 251-127=124。获取尾数中小数点右侧的二进制数。将它转换为十进制数，并除以 2^{23}（其中， 23 是尾数占用的位数），得到 0.302447676659。将该小数加上 1。然后，通过以下方式得到浮点数：

$-1^{0}\times2^{124}\times1.302447676659$

它将变为：

$1\times2.126764793256e+37\times1.302447676659$

它约等于：

2.77000*e*+37

二进制浮点值有时会引起误解。记住一点很重要，即并不是每个浮点值都可以通过一个有限长度的浮点数表示。数字中指数的长度将决定可以存放的数值范围，而尾数的长度则关系到可以精确表示的每个值之间的间距。因此，24 位格式可以表示与 32 位格式大致相同的数值范围，但这种格式可以表示的数值的间距较大。

例如，如果使用 24 位宽的浮点型，它可以精确存储值 95000.0。但是，它可以表示的下一个最高数字为 95002.0，这种类型无法表示这两个值之间的任何值，因为会发生舍入。这意味着进行浮点型比较的 C 代码可能无法产生预期的行为。例如：

```
volatile float myFloat;
myFloat = 95002.0;
if(myFloat == 95001.0)     // value will be rounded
   PORTA++;                // this line will be executed!
```

其中， `if()` 表达式的结果将为 true，虽然看起来所比较的两个值是不同的。

将它与具有更高精度的 32 位浮点型进行比较。它也可以将 95000.0 作为一个值精确地存储。可以表示的下一个最高值 （近似）为 95000.00781。

表 5-6 总结了浮点型格式的特征。该表中的符号是一些预处理器宏，它们在源代码中包含 `<float.h>` 之后可用。

有两组宏可分别用于 float 和 double 类型；其中， *XXX* 代表 FLT 和 DBL。例如，FLT_MAX 表示 float 类型的最大浮点值。根据 float 是 24 位还是 32 位宽格式，它会具有两个值。 DBL_MAX 代表 double 类型的相同值。

由于 ANSI 标准未详细规定浮点型数据类型的长度和格式，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。

表 5-6： 浮点型值的范围

符号	含义	24 位值	32 位值
*XXX*_RADIX	指数表示形式的基数	2	2
*XXX*_ROUNDS	加法的舍入模式	0	0
*XXX*_MIN_EXP	使 FLT_RADIX^{n-1} 为归一化浮点值的最小 *n*	-125	-125
*XXX*_MIN_10_EXP	使 10^n 为归一化浮点值的最小 *n*	-37	-37
*XXX*_MAX_EXP	使 FLT_RADIX^{n-1} 为归一化浮点值的最大 *n*	128	128
*XXX*_MAX_10_EXP	使 10^n 为归一化浮点值的最大 *n*	38	38
*XXX*_MANT_DIG	FLT_RADIX 尾数位数	16	24
*XXX*_EPSILON	与 1.0 相加不会得到 1.0 的最小数字	3.05176e-05	1.19209e-07

© 2013 Microchip Technology Inc.

5.4.4 结构和联合

MPLAB XC8 C 编译器支持 `struct` 和 `union` 类型。结构和联合的惟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。结构和联合的成员不能为 `bit` 类型的对象，但它们完全支持位域。

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。

5.4.4.1 结构和联合限定符

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

在此例中，整个结构都会被放入程序空间，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，但每个成员都是只读的。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 };
```

5.4.4.2 结构中的位域

MPLAB XC8 C 编译器完全支持结构中的位域。

位域总是分配到 8 位字中，虽然它在定义中通常使用 `unsigned int` 类型。

所定义的第一个位将为用于存储它的字的 LSb。声明位域时，如果当前的 8 位单元可容纳它，则会将它分配到其中；否则，会在结构中分配一个新的字节。位域永远不能跨越 8 位分配单元之间的边界。例如，以下声明：

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

将产生一个占用 1 个字节的结构。如果 `foo` 最终链接到地址 10H 处，位域 `lo` 将为地址 10H 的 bit 0； `hi` 将为地址 10H 的 bit 7。 `dummy` 的 LSb 将为地址 10H 的 bit 1，`dummy` 的 MSb 将为地址 10H 的 bit 6。

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，如果永远不会引用 `dummy`，以上结构可以声明为：

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

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

将正确初始化成员 `lo` 和 `hi`。

长度为 0 的位域是一种特殊情况。标准规定不能将任何其他位域填充前一个长度为 0 的位域（如果有）的分配单元。

© 2013 Microchip Technology Inc.

MPLAB XC8 编译器支持匿名联合。它们是不具有标识符的联合，无需引用外围的联合即可访问其成员。这些联合可以在放入结构中之后使用。例如：

```
struct {
       union {
       int x;
       double y;
   };
} aaa;

void main(void)
{
   aaa.x = 99;
   // ...}
```

此处，联合未命名，其成员可以作为结构的一部分进行访问。匿名联合不属于任何 C 标准的一部分，所以使用它们会限制任何代码的可移植性。

5.4.5 指针类型

MPLAB XC8 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。

为了降低存储器需求和减少执行时间，指针会被设为不同的长度和格式。MPLAB XC8 C 编译器使用先进的算法来跟踪所有指针的地址赋值，因而在定义指针变量时不需要非标准限定符。标准限定符 `const` 和 `volatile` 仍可使用，具有它们通常的含义。尽管如此，每个指针的长度将对于它在程序中的预期用途是最优的。

5.4.5.1 组合类型限定符和指针

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 * 操作符也用于指针解引用，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间距进行了强调：

```
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`（可变）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，某个函数使用指针作为参数，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。

© 2013 Microchip Technology Inc.

5.4.5.2 数据指针

MPLAB XC8 编译器会监视并记录对程序包含的每个数据指针的*所有*地址赋值。这包括将对象的地址赋予指针；将一个指针赋予另一个；定义指针时初始化它们；并考虑到指针为普通变量和函数参数，以及使用指针来访问基本对象、结构或数组的情况。

每个指针存放的地址的长度和格式都基于该信息。如果在代码中的不同位置将多个地址赋予某个指针，则会维护该指针可以寻址的所有可能目标的集合。该信息特定于程序中定义的每个指针，因此对于两个具有相同 C 类型的指针，由于指针在程序中的使用方式，它们可能存放具有不同长度和格式的地址。

编译器会跟踪所有目标的存储器位置，以及所有目标的长度，以确定指针的长度和作用域。目标的长度也很重要，特别是对于数组或结构。例如，它必须能够对指针进行递增，从而使它可以访问数组中的所有元素。

MPLAB XC8 C 编译器使用了几种指针分类，如以下所述。

对于低档和中档器件：

- 能够访问公共存储器和两个连续存储区（如存储区 0 和 1，或存储区 7 和 8 等）的 8 位指针
- 能够访问整个数据存储空间的 16 位指针
- 最多能够访问 256 字节程序空间数据的 8 位指针
- 最多能够访问 64 KB 程序空间数据的 16 位指针
- 能够访问整个数据存储空间和最多 64 KB 程序空间数据的 16 位混合目标空间指针

对于 PIC18 器件：

- 能够访问快速操作存储区的 8 位指针
- 能够访问整个数据存储空间的 16 位指针
- 最多能够访问 256 字节程序空间数据的 8 位指针
- 最多能够访问 64 KB 程序空间数据的 16 位指针
- 能够访问整个程序空间的 24 位指针
- 能够访问整个数据存储空间和最多 64 KB 程序空间数据的 16 位混合目标空间指针
- 能够访问整个数据存储空间和整个程序空间的 24 位混合目标空间指针

每个数据指针都将在初步扫描源代码之后分配一个可用的类别。编程人员无法通过任何机制指定所需的指针样式（除了通过为指针赋值）。C 代码必须向编译器传递所需的信息。

指针引用图会显示关于指针及其目标的信息，如**第 6.6.5 节 “指针引用图”**所述。该图打印在汇编列表文件中，它通过**第 4.8.17 节 “--ASMLIST：生成汇编列表文件”**所述的选项进行控制。

假设在开发早期阶段具有以下中档器件程序。它包含以下代码：

```
int i, j;

int getValue(const int * ip) {
    return *ip;
}

void main(void) {
    j = getValue(&i);
    // ... code that uses j
}
```

指针 ip 是函数 getValue() 的一个参数。指针目标类型使用了限定符 const，因为我们不希望指针被用于写入地址被传递给函数的任何对象。const限定没有任何其他用途，并不会改变指针变量的格式。

如果编译器将变量 i（在 main() 中定义）分配到数据存储器存储区 0，则还会注明指针 ip （getValue() 的参数）仅指向一个位于数据存储区 0 的对象。在这种情况下，指针 ip 会被设为一个 8 位宽的数据指针。生成 getValue() 中解引用 ip 的代码时，将假定地址只能为存储区 0 中的对象。

随着程序的开发，定义了另一个变量 x，并（对于编程人员是未知的）在数据存储区 2 中分配空间。 main() 函数现在具有如下形式：

```
int i, j;  // allocated to bank 0 in this example
int x;     // allocated to bank 2 in this example

int getValue(const int * ip) {
    return *ip;
}

void main(void) {
    j = getValue(&i);
    // ... code that uses j
    j = getValue(&x);
    // ... code that uses j
}
```

指针 ip 现在具有位于存储区 0 和存储区 2 中的目标。为了能够适应这种情况，指针会被设为 16 位宽，用于解引用指针的代码将相应地发生更改。这是在不对源代码进行任何修改的情况下发生的。

跟踪指针目标的一个积极方面是它较少地依赖于指针限定符。指针定义中仍然必须使用标准限定符 const 和 volatile，以分别指示只读或外部可修改的目标对象。但是，这严格符合 ANSI C 标准。不需要使用诸如 near 和 bank2 之类的非标准限定符来指示指针目标，它们没有任何作用，应避免使用它们。省略这些限定符可以获得可移植性和可读性更高的代码，并减少编译器发出无关警告的机会。

© 2013 Microchip Technology Inc.

5.4.5.2.1 指向两个存储空间的指针

如果向某个指针赋予在数据空间中分配存储器的一个或多个对象的地址，同时也赋予它一个或多个 const 对象的地址，该指针将归入**第 5.4.5.2 节 “数据指针”**列出的混合目标空间指针中的一种，并且地址将编码为可以在运行时确定目标存储空间的形式。这些指针类型的编码如下所述。

对于低档 / 中档 16 位混合目标空间指针，地址的 MSb （即， bit 15）指示地址引用的存储空间。如果该位置 1，则指示地址代表的是程序存储器中的对象；清零则指示数据存储器中的对象。该地址的剩余部分表示所指示存储空间中的完整地址。

对于 PIC18 16 位混合目标空间指针，地址高于最高数据空间地址时，表示它是一个处于程序存储空间中的对象；否则，该地址代表一个数据存储空间对象。

对于 PIC18 24 位混合目标空间指针， bit 21 指示该地址引用的存储空间。如果该位置 1，则指示该地址代表一个位于数据存储器中的对象；如果它清零，则指示程序存储器中的对象。该地址的剩余部分表示所指示存储空间中的完整地址。请注意出于效率方面的考虑，存储空间位的含义与低档和中档器件的是相反的。

为了扩展**第 5.4.5.2 节 “数据指针”**给出的中档器件示例，现在进一步开发该代码。现在使用某个位于程序存储器中的对象的地址来调用函数 getValue()，如下所示。

```
int i, j;  // allocated to bank 0 in this example
int x;     // allocated to bank 2 in this example
const int type = 0x3456;

int getValue(const int * ip) {
    return *ip;
}

void main(void) {
    j = getValue(&i);
    // ... code that uses j
    j = getValue(&x);
    // ... code that uses j
    j = getValue(&type);
    // ... code that uses j
}
```

同样，确定指针 ip 的目标，并且现在会将指针设为可同时访问数据存储器和程序存储器的类别。进行指针解引用的生成代码将可以根据地址确定所需的存储空间，并相应地访问其中任一空间。同样，这也是在不需要对指针定义进行任何更改的情况下发生的。

如果汇编代码引用了某个 C 指针，编译器会将该指针强制设为 16 位混合目标空间指针（对于低档或中档器件程序）或 24 位混合目标空间指针 （对于 PIC18 器件程序）。这些指针类型可以不受限制地访问所有存储区，并且可以正确工作，即使是在汇编代码中对指针进行赋值 （赋予格式正确的地址）。

5.4.5.3 函数指针

MPLAB XC8 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。它们通常用于调用存储在用户定义的 C 数组中的几个函数地址之一，该数组就如同一个查找表。

对于低档和中档器件，函数指针长度总是为 1 个字节，其中存放由编译器输出的跳转表中的偏移量。该跳转表包含指向目标函数的跳转。

对于 PIC18 器件，函数指针为 16 或 24 位宽。指针长度完全基于目标器件上可用的程序存储器量。

与数据指针一样，也会跟踪赋予函数指针的目标。与数据指针相关联的过程相比，这是一个较简单的过程，因为所有函数指令都必须位于程序存储器中。在指针引用图（如**第 6.6.5 节 “指针引用图”**所述）中，除了数据指针之外，还会显示函数指针，以及它们的所有目标。目标将为可通过指针调用的函数的名称。

低档和中档器件有一个重要的运行时特性，即如果某个函数指针包含 NULL （值 0），并使用它来间接调用函数，将会导致代码困在跳转到自身的循环中。这种无限循环可以用于检测这种错误情形。通常，通过 NULL 指针调用函数会导致代码崩溃或某种其他意外行为。无限循环跳转到的标号为 fpbase。

5.4.5.4 特殊指针目标

指针和整型是不可互换的。将整型常量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
```

整型常量 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。如果将整型地址赋予指针并进行解引用，则很有可能会发生代码失败，特别是对于具有多个存储空间的 PIC 器件。 0x123 是数据存储器还是程序存储器中的地址？在地址 0x123 找到的对象有多大？

将地址赋予一个指针时，请总是采用 C 对象的地址。如果在目标地址处未定义任何 C 对象，则在该地址定义或声明一个可用于该目的的对象。请确保该对象的长度与要通过指针访问的存储单元的范围匹配。

例如，要生成从程序存储器中地址 0x900 处开始的 1000 个存储单元的校验和。使用一个指针来读取该数据。可以尝试如下编写代码：

```
const char * cp;
cp = 0x900;  // what resides at 0x900???
```

并随数据递增指针。

但是，更好的解决方案是：

```
const char * cp;
const char inputData[1000] @ 0x900;
cp = &inputData;
// cp is incremented over inputData and used to read values there
```

在这种情况下，编译器可以确定目标的长度和存储空间。数组长度和类型指示指针目标的长度，对象 （不是指针）的 const 限定符指示目标位于程序存储空间。请注意，该实例中的 const 数组不需要指定初始值（见**第 5.4.7.1 节 “const 类型限定符”**）并可驻留在这些地址处的其他对象之上。

© 2013 Microchip Technology Inc.

如果指针需要访问数据存储器中的对象，则需要定义一个不同的对象来用作虚拟目标。例如，如果要计算从数据存储器中地址 0x90 处开始的 10 个字节的校验和，可以使用以下代码。

```
const char * cp;
char inputData[10] @ 0x90;
cp = &inputData;
// cp is incremented over inputData and used to read values there
```

用户定义的绝对对象不会被运行时启动代码清零，它们可以放置在其他绝对变量之上。

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。一个例外情况是地址可以延伸到超出数组末尾的一个元素。

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
```

永远不要将指针与整型常量进行比较。

NULL 指针是可以向指针赋予常量值的一个实例，它可以由编译器正确处理。 NULL 指针在数值上等于 0 （零），但这是 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。

如果 NULL 是惟一赋予指针的值，则会将指针设为尽可能小的长度。

5.4.6 常量类型和格式

常量用于在源代码中表示立即数，与可以存放相同值的变量形成对照。例如，123 是一个常量。

与所有值一样，常量必须具有 C 类型。除了常量的类型之外，还可以使用几种格式之一来指定实际值。

5.4.6.1 整型常量

整型常量的格式会指定其基数。 MPLAB XC8 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 5-7 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 5-7： 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常量的默认类型；例如 `23U`，其中的 `U` 为后缀。表 5-8 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常量，并且如果 `long int` 类型可以存放该常量，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 5-8： 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`

© 2013 Microchip Technology Inc.

以下给出了一个可能由于分配给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
}
```

常量 1 将被赋予 int 类型，因此移位操作的结果将为 int 类型。虽然将该结果赋予 long 变量 result，它的长度永远不会大于 int 的长度，无论对该常量进行了多少位的移位。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常量的类型，从而确保移位结果具有 unsigned long 类型。

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

5.4.6.2 浮点型常量

浮点型常量具有 double 类型，除非加上后缀 f 或 F，在这种情况下它是一个 float 常量。后缀 l 或 L 用于指定 long double 类型，MPLAB XC8 将其视为与 double 相同的类型。

5.4.6.3 字符和字符串常量

字符常量使用单引号字符 ' 包围，例如 'a'。字符常量具有 int 类型，虽然编译器之后可能会将其优化为 char 类型。

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\370rk";
printf("%s's Resum\351", name);   \\ prints "Bjørk's Resumé"
```

该实现不支持多字节字符常量。

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”。字符串常量的类型为 const char*，构成字符串的字符存储在程序存储器中，与使用 const 限定的所有对象一样。

将字符串字面值赋予一个未指定 const 目标的指针时，会产生一条常见的相关警告，例如：

```
char * cp = "hello world\n";
```

字符串字符不能进行修改，但这种类型的指针允许进行写入，因此会产生该警告。为了防止自己试图覆盖字符串，请如下使用限定符对指针目标进行限定。另请参见**第 5.4.5.1 节 “组合类型限定符和指针”**。

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

定义并使用字符串初始化数组 （即，不是指针）属于例外情况。例如：

```
char ca[]= "hello world\n";
```

实际上会将字符串字符复制到 RAM 数组中，而不是像前面的示例一样将字符的地址赋予指针。字符串字面值保持为只读，但该数组是可读写的。

对于具有相同字符序列的字符串， MPLAB XC8 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）。例如，在以下代码片段中

```
if(strncmp(scp, "hello", 6) == 0)
   fred = 0;
if(strcmp(scp, "world") == 0)
   fred--;
if(strcmp(scp, "hello world") == 0)
   fred++;
```

字符串 “world” 和字符串 “hello world” 中的最后 6 个字符 （最后一个字符为 nul 终结符）将由存储器中的相同字符表示。字符串 “hello” 不会与字符串 “hello world” 中的相同字符重叠，因为两者中的 nul 字符的位置不同。

编译器会连接两个相邻的字符串常量 （即，两个字符串 *仅*使用空格进行分隔）。因而：

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

将向指针赋予字符串 “hello world” 的地址。

© 2013 Microchip Technology Inc.

5.4.7 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 XC8 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 8 位 PIC MCU 架构。

5.4.7.1 CONST 类型限定符

MPLAB XC8 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。

如果用户定义的对象声明为 `const` 类型，则它们会被放入一个链接到程序空间的特殊 psect 中。限定为 `const` 类型的对象可以为绝对对象。`@ address` 构造用于将对象放置在程序存储器中的指定地址处，如以下示例所示，对象 `tableDef` 被放置在地址 0x100 处。

```
const int tableDef[] @ 0x100 = { 0, 1, 2, 3, 4};
```

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。但是，如果需要将某个对象放置在程序存储器中某个特定位置的其他对象之上，则可以定义未初始化的 `const` 对象，它会非常有用。通常，未初始化的 const 对象将被定义为绝对对象，如以下示例所示。

```
const char checksumRange[0x100] @ 0x800;
```

会将对象 `checksumRange` 定义为 0x100 字节的字符数组，位于程序存储器的地址 0x800 处。该定义不会在 HEX 文件中放入任何数据。

5.4.7.2 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 `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：

```
volatile static unsigned int  TACTL @ 0x160;
```

`volatile` 限定符并不保证所有访问都是原子操作，对于 8 位 PIC MCU 架构通常就不是。对于所有这些器件，每条指令最多都只能访问 1 个字节的数据。

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，因此仅在必需时才使用该限定符。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。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，这是非常有用的。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。如果项目中具有汇编代码，请参见**第 5.12.3.4 节 “未定义的符号”**。

5.4.8 特殊的类型限定符

MPLAB XC8 C编译器支持一些特殊的类型限定符，使用户可以控制将static和extern类变量放入特定的地址空间。

5.4.8.1 PERSISTENT 类型限定符

默认情况下，在启动时会清零所有未显式初始化的 C 变量。这与 C 语言的定义是一致的。但是，有些情况下希望在复位时保存一些数据。

persistent 类型限定符可用于限定运行时启动代码不应清零的变量。

此外，所有 persistent 变量都会被存储到与其他变量不同的存储区中。将使用不同的 psect 来存放这些对象。更多信息，请参见**第 5.15.2 节 “编译器生成的 psect”**。

该类型限定符不能用于自动类型的变量；但是，静态定义的局部变量可以使用 persistent 限定。例如，应编写：

```
void test(void)
{
    static persistent int intvar;  /* must be static */
    // ...
}
```

如果使用了 xc8 选项 --STRICT，该类型限定符将变为 __persistent。

5.4.8.2 NEAR 类型限定符

一些 8 位 PIC 架构实现了无论当前选定的存储区如何总是可以进行访问的数据存储器。这种*公共存储器*可用于减小代码长度和执行时间，因为在访问分区存储器中的数据时通常需要存储区选择指令，而在访问公共存储器时可以不需要它们。PIC18 器件将这种存储器称为快速操作存储区。中档和低档器件具有极少量的此类存储区 （如果确实存在）。 PIC18 器件具有较多的公共存储区，但其数量因器件而不同。更多信息，请查看您所使用器件的数据手册。

near 类型限定符可以用于将非自动变量放入公共存储区。

编译器会自动对经常访问的用户定义变量使用公共存储区，所以只有在对象需要进行特殊的存储器放置时才需要使用该限定符，例如，如果在假定 C 变量位于该存储区中的手写汇编代码中访问 C 变量。

© 2013 Microchip Technology Inc.

该限定符由编译器选项 --ADDRQUAL 控制，该选项决定它的作用，请参见**第 4.8.16 节“--ADDRQUAL：设置编译器对于存储器限定符的响应”**。基于该选项的设置，可能会绑定或忽略（这是默认操作）该限定符。被忽略的限定符不会产生错误或警告，但不会具有任何作用。

以下给出了使用 near 限定的 unsigned char 对象的示例：

```
near unsigned char fred;
```

使用 near 限定的对象不能为自动类型或函数参数，但可以使用 static 限定，从而可以在函数内局部定义它们，如下所示：

```
void myFunc(void) {
   static near unsigned char local_fred;
```

请注意，编译器可能会在公共存储区中存储一些临时对象，所以该空间可能不是全部都可供用户定义的变量使用。

如果使用了 xc8 选项 --STRICT，该类型限定符将变为 __near。

5.4.8.3 FAR 类型限定符

对于那些支持外部存储器的 PIC18 器件， far 类型限定符用于将非自动变量放入程序存储空间。针对 PIC10/12/16 目标器件进行编译时，它将被忽略。编译器假定变量将定位到在该存储空间中实现的 RAM 中。

far 变量的访问效率会低于内部变量，并会产生较长、较慢的代码。

该限定符由编译器选项 --ADDRQUAL 控制，该选项决定它在 PIC18 器件上的作用，请参见**第 4.8.16 节 “--ADDRQUAL：设置编译器对于存储器限定符的响应”**。基于该选项的设置，可能会绑定或忽略（这是默认操作）该限定符。被忽略的限定符不会产生错误或警告，但不会具有任何作用。

以下给出了一个将 unsigned int 对象放入器件外部程序存储空间的示例：

```
far unsigned int farvar;
```

使用 far 限定的对象不能为自动类型或函数参数，但可以使用 static 限定，从而可以在函数内局部定义它们，如下所示：

```
void myFunc(void) {
static far unsigned char local_far;
```

如果使用了 --STRICT，该类型限定符将变为 __far。

请注意，并不是所有 PIC18 器件都在其程序存储空间中支持外部存储器，因此，far 限定符并不适用于所有 PIC18 器件。在支持的器件上，必须先使用 --RAM 选项（**第 4.8.48 节 “--RAM：调整 RAM 范围”**）指定额外存储器需要映射到的地址范围。例如，要映射从 20000h 至 2FFFFh 的额外数据存储器，可以使用

```
--RAM=default,+20000-2FFFF
```

5.4.8.4 BANK0、 BANK1、 BANK2 和 BANK3 类型限定符

编译器可以识别 bank0、bank1、bank2 和 bank3 类型限定符，并允许一定程度地控制对象在器件数据存储区中的放置。针对 PIC18 目标器件进行编译时，接受这些限定符只是为了支持可移植性，它们对变量放置没有任何影响；在其他器件上，它们可用于定义由手写汇编代码假定位于特定存储区的 C 对象。编译器会自动将变量分配到所有数据存储区，因此通常不需要这些限定符。

虽然一些器件实现了超过 4 个数据 RAM 存储区，但当前未提供用于将变量放入这些高端存储区的存储区限定符。

这些限定符由编译器选项 --ADDRQUAL 控制，该选项决定它们的作用，请参见**第 4.8.16 节“--ADDRQUAL：设置编译器对于存储器限定符的响应”**。基于该选项的设置，可能会绑定或忽略（这是默认操作）这些限定符。被忽略的限定符不会产生错误或警告，但不会具有任何作用。

使用其中任意限定符限定的对象不能为自动类型或函数参数，但可以使用 static 限定，从而可以在函数内局部定义它们，如下所示：

```
void myFunc(void) {
   static bank1 unsigned char play_mode;
```

如果使用了 xc8 选项 --STRICT，这些限定符将变为 __bank0、__bank1、__bank2 和 __bank3。

5.4.8.5 EEPROM 类型限定符

只有对于低档和中档器件，编译器才会识别 eeprom 类型限定符，它指示对象应放置在 EEPROM 存储器中。并不是所有器件都实现了 EEPROM 存储器。更多信息，请查看您所使用器件的数据手册。

使用该限定符限定的对象不能为自动类型或函数参数，但可以使用 static 限定，从而可以在函数内局部定义它们，如下所示：

```
void myFunc(void) {
   static eeprom unsigned char inputData[3];
```

用于访问eeprom限定变量的生成代码会比用于访问基于RAM的变量的代码更长和更慢。

如果使用了 --STRICT 选项，该限定符将变为 __eeprom。

© 2013 Microchip Technology Inc.

5.5 存储器分配和访问

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的静态单元。以下几节分别介绍了这两组变量的存储器分配。

5.5.1 地址空间

所有 8 位 PIC 器件都采用哈佛架构，该架构具有独立的数据存储空间 （RAM）和程序存储空间 （通常为闪存）。此外，一些器件还实现了 EEPROM。

数据存储器使用分区来增加可用的存储器量（在数据手册中称为*通用寄存器文件*），而无需增加汇编指令宽度。存储区通过设置某个 SFR 中的一个或多个位来“选择”。（关于您所使用的器件的确切操作，请查看数据手册。）访问数据地址的大多数指令都仅使用当前选定存储区中的偏移量来访问数据。例外情况是 PIC18 指令 MOVFF，它需要完整的分区地址，并可独立于选定存储区工作。一些器件仅具有一个存储区，但许多器件具有多个。

通用 RAM 和 SFR 共用相同的数据空间，并可能会出现在所有可用存储区中。PIC18 器件将所有 SFR 都放置在一个数据存储区中，但中档和低档器件将 SFR 放置在每个存储区的低地址处。由于 SFR 在这些器件中的位置的原因，通用存储器会产生碎片，这会限制大多数 C 对象的长度。

增强型中档器件支持线性寻址模式，该模式允许将通用存储器作为一个连续块进行访问，从而克服碎片的问题。因此，针对这些器件进行编译时，允许的最大对象长度通常会增大。使用 PIC18 器件时，所定义的对象通常还可以使用整个数据存储器。请参见**第 5.5.2.3 节 “自动变量的长度限制”**和**第 5.5.2.1.2 节 “非自动变量长度限制”**。

许多器件都具有一些无论当前选择哪个存储区都可以进行访问的字节。这种存储器称为*公共存储器*。PIC18 数据手册将存储该存储器的存储区称为快速操作存储区，因此通常将它称为快速操作存储器。由于在访问这些存储单元之前无需用于选择存储区的代码，因而访问该存储器中对象的速度通常更快，并且生成的代码较短。如果可能，编译器将总是尝试使用该存储器。

程序存储空间主要用于可执行代码，但数据也可以位于其中。不同器件系列使用几种方法在该存储器中定位和读取数据，但定位到其中的所有对象都是只读的。

5.5.2 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` 限定的非自动变量（它们会被放入程序存储空间）或使用 `eeprom` 限定的变量。

由于自动变量和非自动变量的存储器分配方式根本上的不同，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。

> **注：** 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内部的变量。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），所以仅使用两个术语来描述它们可能会造成混淆。例如，在函数之外定义的静态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。

5.5.2.1 非自动变量分配

非自动变量（那些具有永久存储持续时间的变量）由编译器定位到任何可用的数据存储区。这是一个两阶段的过程：将每个变量放入某个适当的 psect，之后将该 psect 链接到预先确定的存储区。关于 psect 的介绍指南，请参见**第 5.15.1 节 “程序段”**。因此在编译期间，代码生成器可以确定每个变量将存放在哪一个存储区，并相应地对输出进行编码，但它不会知道该存储区内的确切位置。

编译器会尝试将所有变量定位到一个存储区中 （即，将所有变量都放入预定目标为该存储区的 psect 中），但如果它填满（即，如果编译器检测到 psect 大于存储区中的可用空间），变量将通过不同的 psect 定位到其他存储区。不需要使用限定符来将这些变量放入存储区 0 之外的存储区，但如果希望强制将变量放入某个特定存储区，则可以使用限定符。关于如何这样做的更多信息，请参见**“--RAM=default,+20000-2FFFF”**和**第 4.8.16 节 “--ADDRQUAL：设置编译器对于存储器限定符的响应”**。如果目标器件上提供了公共存储器，则也会考虑对于变量使用它。该存储器的长度可能有限，并可能保留用于特殊用途，因此只有一些变量可以分配到其中。

编译器会考虑 3 种类别的非自动变量，这 3 种类别都与程序开始时变量应包含的值有关。每个变量类别都具有相应的 psect，用于存放会为每个变量保留存储器的输出代码。下面列出了每个 psect 类别的基本名称。**第 5.15.2 节 “编译器生成的 psect”**给出了所有 psect 名称的完整列表。

nv 这些 psect 用于存储使用 `persistent` 限定的变量，运行时启动代码不应更改这些变量的值。它们在启动时不会被清零或以其他方式修改。

bss 这些 psect 包含所有未初始化变量（即定义时未赋值的变量）或应由运行时启动代码清零的变量。

data 这些 psect 包含所有已初始化变量 （定义时赋予一个非零初始值，且必须由运行时启动代码向其复制一个值的变量）的 RAM 镜像。

如**第 5.10 节 “main、运行时启动和复位”**所述，数据空间 psect 的基本名称总是与链接器类名联合使用，用以指示 psect 将定位到的 RAM 存储区。本节还列出了这些 psect 的其他变化形式，并说明了这些 psect 必须链接到何处。关于如何为变量赋予初始值的更多信息，请参见**第 5.15.2 节 “编译器生成的 psect”**。

© 2013 Microchip Technology Inc.

请注意，用于存放已初始化变量的 data psect 是存放 RAM 变量本身的 psect。存在一个与之对应的 psect （名为 idata），它会被放入程序存储器 （所以它是非 volatile 类型），用于存放由运行时启动代码复制到 RAM 变量的初始值。

所有非自动变量 （静态变量除外，它在**第 5.5.2.1.1 节 “静态变量”**中介绍）总是使用其词法名称加上前导 *下划线*字符作为用于该对象的汇编标识符。关于 C 域符号和汇编域符号之间映射的更多信息，请参见**第 5.12.3.1 节 “等效汇编符号”**。

5.5.2.1.1 静态变量

所有静态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。局部静态变量的作用域仅为定义它们的函数或块内部，但与自动变量不同，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。

程序将确保静态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。

属于静态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自动类型对象，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已初始化静态变量的方式与其他非自动已初始化对象相同，请参见**第 4.4.2 节 “启动和初始化”**。

第 5.12.3.1 节 “等效汇编符号”介绍了用于在汇编代码中访问静态对象的汇编符号。

5.5.2.1.2 非自动变量长度限制

编译器完全支持任何类型的数组 （包括聚合类型的数组） 。结构和联合聚合类型也是，请参见**第 5.4.4 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。

针对增强型中档 PIC 器件进行编译时，对象 （数组或聚合对象）的长度通常仅受可用数据存储器总量限制。如果单个对象无法装入任何可用的通用 RAM 范围，则会在几个 RAM 存储区中为它分配存储器，并使用器件的线性 GPR （通用 RAM）进行访问。

请注意，特殊功能寄存器 （位于数据存储空间）或通用 RAM 中的存储器预留可能会导致无法将对象分配到一个存储区的连续存储器中。在这种情况下，长度小于一个 RAM 存储区的对象也可能跨越多个存储区进行空间分配。用于访问多存储区对象的生成代码总是会较慢，关联的代码长度将大于完全包含在单个 RAM 存储区内的对象。

针对 PIC18 器件进行编译时，对象的长度通常也仅受可用的数据存储器限制。对象可以横跨多个数据存储区。

在低档和其他中档器件上，数组和结构会受每个 RAM 存储区中可用 GPR 存储器的最大长度限制，而不是剩余的存储器总量。如果定义的数组大于该长度，将会产生错误。

对于任何器件，在通用 RAM 中保留存储器（见**第 4.8.48 节 “--RAM：调整 RAM 范围”**），或在数据存储区中间定义绝对变量（见**第 5.5.4 节 “绝对变量”**）会进一步限制数据存储区中的连续存储器，可能会减小可以定义的对象的最大长度。

5.5.2.1.3 更改默认的非自动变量分配

有几种方法可以将非自动变量定位到那些由编译器所选位置之外的位置。

变量可以通过使用限定符放入其他存储空间。例如，如果希望将变量放入程序存储空间，则应使用 `const` 说明符（见**第 5.4.7.1 节 “const 类型限定符”**）。`eeprom` 限定符（见**第 5.4.8.5 节 “eeprom 类型限定符”**）可用于将变量分配到 EEPROM（如果目标器件上存在此类存储器）。

如果希望阻止变量使用一个或多个数据存储单元，从而可以将这些单元用于其他用途，最好使用存储器调整选项来保留存储器。关于如何这样做的信息，请参见**第 4.8.48 节 “--RAM：调整 RAM 范围”**。

如果只有几个非自动变量需要定位到数据存储空间中的特定地址处，则可以将变量设为绝对变量。这可以将各个变量显式地定位到存储器中的绝对地址处。**第 5.5.4 节 “绝对变量”**对绝对变量进行了介绍。将变量设为绝对变量之后，它们的地址将硬编码到生成的输出代码中，它们将不再放入 psect，不遵循正常的存储器分配过程。

通过更改默认的链接器选项，可以将不同类别的非自动变量所在的 psect（**第 5.5.2.1 节 “非自动变量分配”**介绍的 `nv`、`bss` 和 `data` psect）作为一个整体进行移动。例如，可以移动所有持久性变量。但是，通常这些 psect 只能在它们默认情况下分配的数据存储区内移动。关于更改 psect 的默认链接器选项的更多信息，请参见**第 5.10 节 “main、运行时启动和复位”**。代码生成器会对这些 psect 的位置作出一些假定，如果将它们移动到一个打破这些假定的位置，代码可能会发生失败。

非自动对象还可以通过使用 `psect` pragma 伪指令放置在特定位置处，请参见**第 5.14.4.8 节 “#pragma psect 伪指令”**。应基于位置要求来决定是通过这种方式还是使用绝对变量来定位变量。

© 2013 Microchip Technology Inc.

5.5.2.2 自动变量分配和访问

本节介绍自动变量（那些具有自动存储持续时间的变量）分配。此外还将介绍函数参数变量，它在存储持续时间和作用域方面的行为类似于自动变量。由编译器定义的临时变量也属于该类别。除了一点之外，它们与自动对象完全相同，即它们是由编译器而不是编程人员定义的，因此它们没有 C 名称。

`auto`（自动，*automatic* 的缩写）变量是局部变量的默认类型。除非显式地声明为静态类型，否则局部变量将设为自动类型。如果需要，可以使用 `auto` 关键字。

正如其名称所暗示的，自动变量会在函数执行时自动产生，然后在函数返回之后消失。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。编译器面向的所有器件都不具有可以按这种方式工作的数据堆栈。这些器件的硬件堆栈只能用于存放函数返回地址，也没有可以将数据放入该堆栈的指令。因此，编译器实现了一种替代的堆栈构造。所采用的堆栈机制称为*编译堆栈*，**第 5.5.2.2.1 节 “编译堆栈操作”**对其进行了详细介绍。

自动变量在编译堆栈中分配一个相对位置之后，将在数据空间中为堆栈本身分配存储器。其实现方式类似于为非自动变量分配存储器的方式：使用一个 psect 来存放堆栈，该 psect 将由链接器放入可用的数据存储器。用于存放编译堆栈的 psect 的基本名称为 `cstack`，并且类似于非自动变量 psect，基本名称总是与链接器类名联合使用，用以指示 psect 将定位到的 RAM 存储区。关于与该 psect 可以链接到的位置关联的限制，请参见**第 5.15.2 节 “编译器生成的 psect”**。

与参数不同，在函数中定义的自动变量不一定按声明的顺序分配存储器，而参数总是基于它们的词法顺序分配存储器。实际上，一个函数的自动变量可能会被分配到许多RAM存储区。

标准限定符 `const` 和 `volatile` 可同时用于自动变量，它们不会影响这些变量在存储器中的定位方式。这意味着，使用 `const` 限定的局部对象仍是自动对象，因此将在数据存储空间中的编译堆栈中分配存储器，而不是像非自动 `const` 对象一样在程序存储器中分配。

编译器会尝试将堆栈定位到一个数据存储区中，但如果它填满（即，如果编译器检测到堆栈 psect 太大），则会将堆栈构造为几个组成部分 （每个都具有自己的 psect）并将每个链接到不同的存储区。

在汇编代码中，每个自动对象都使用一个由代码生成器定义的特殊符号进行引用。如果访问在 C 源代码中定义的自动变量，则必须使用这些符号，**第 5.12.3 节 “汇编代码和 C 代码之间的交互”**对它们进行了介绍。

5.5.2.2.1 编译堆栈操作

编译堆栈由固定的存储区域组成，可供每个函数基于堆栈的变量使用。使用编译堆栈时，函数是不可重入的，因为每次调用函数时，每个函数中基于堆栈的变量将使用相同的固定存储区域。

调用图是生成编译堆栈的基础，它会定义函数调用的树状层次结构，即，它会显示每个函数可能调用哪些函数。

对于每个根函数，都会生成一个图。根函数通常不会被调用，但它通过其他方式执行，并包含一个程序入口点。函数 main() 是根函数的一个示例，每个项目中都会有该函数。发生硬件中断时执行的中断函数是另一个示例。

图 5-2： 调用图的构成

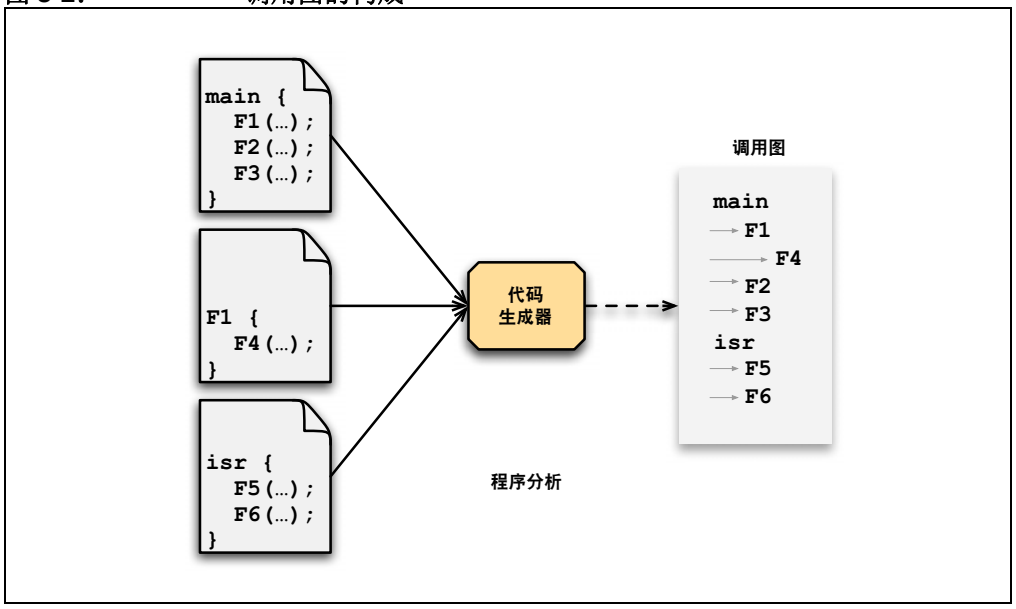


图 5-2 显示了代码生成器通过分析各个程序段来构成调用图。在原始的源代码中，函数 main() 调用了 F1()、F2() 和 F3()。F1() 调用了 F4()，但其他两个函数不进行任何调用。main() 的调用图显示了这些调用。符号 F1、F2 和 F3 全部在 main 下缩进一个级别。 F4 在 F1 下缩进一个级别。

这是静态调用图，显示了所有可能的调用。如果函数 F1() 的确切代码为以下形式：

```
int F1(void) {
  if(PORTA == 44)
    return F4();
  return 55;
}
```

函数 F4() 将总是显示在调用图中，虽然在实际源代码中是根据条件来执行它。因此，调用图将显示 *可能*会调用的所有函数。

在图中，还有一个中断函数 isr()，它也会生成独立的图。

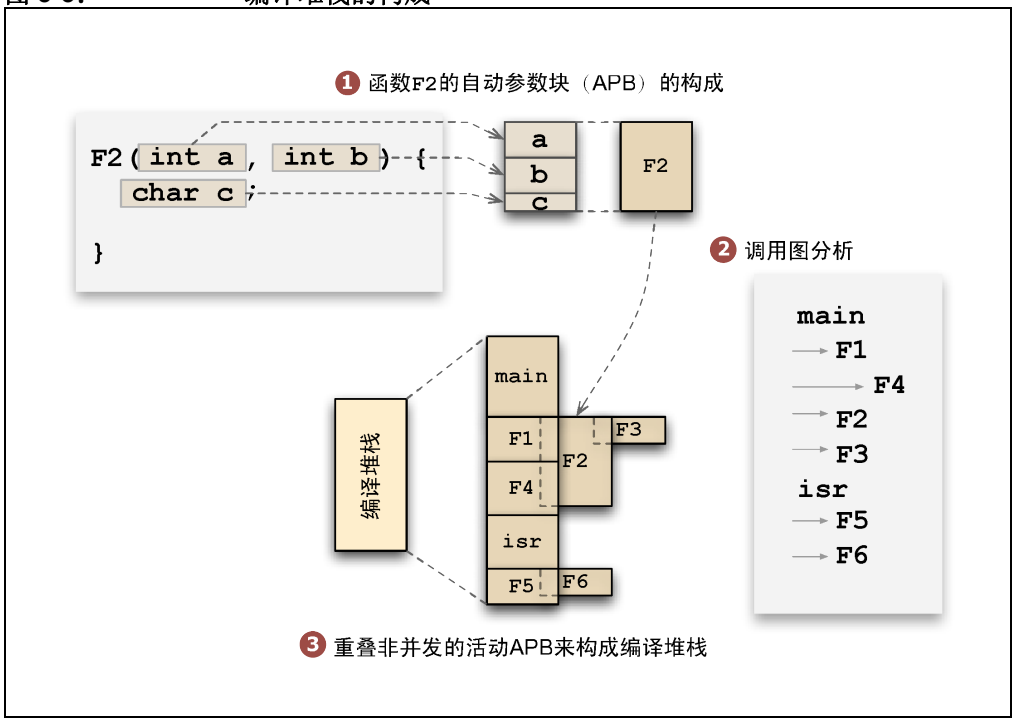
经常会使用主干代码一词，它指的是由于执行 main() 函数而执行的任何代码。在上图中，主干代码仅调用了 F1()、 F2()、 F3() 和 F4()。

中断代码一词指的是由于发生中断而执行的任何代码；在上图中，中断代码调用了 F5() 和 F6()。

© 2013 Microchip Technology Inc.

图 5-3 以图形方式显示了编译堆栈如何构成的一个示例。

图 5-3：　　编译堆栈的构成



对于程序中的每个函数，都会为其参数、自动和临时变量分配一个存储器块。每个块被称为自动参数块 （Auto-Parameter Block， APB）。图中显示了为函数 F2() 构成的 APB，该函数具有两个参数 a 和 b，以及一个自动变量 c。

先按函数参数在源代码中出现的词法顺序而严格确定的顺序对函数参数进行组合。在它们之后是所有自动对象；但是，自动对象可按任意顺序放置。所以可以看到，用于 a 的存储器之后是用于 b 的存储器，最后是用于 c 的存储器。

对这些变量组合之后，每个对象的确切位置在此时并不重要，可以通过一个块（该函数的 APB）表示该存储器。

对于程序中的所有函数，都会构造 APB。然后，通过分析调用图，在编译堆栈中为这些块分配位置或基值。

如果观察到以下情况，则可以节省存储器：如果两个函数永远不会在同一时间处于活动状态，则它们的 APB 可以进行重叠。

在图中所示的示例中， F4() 和 F1() 会在同一时间处于活动状态，实际上 F1() 调用了 F4()。但是， F2()、 F3() 和 F1() 永远不会在同一时间处于活动状态； F1() 必须在 main() 可以调用 F2() 或 F3() 之前返回。函数 main() 总是处于活动状态，所以其 APB 永远不能与其他函数重叠。

在编译堆栈中，可以看到为 main() 的 APB 分配了独立的存储器。 F1()、 F2() 和 F3() 的块在编译堆栈中全部彼此交叠，并具有相同的基值；但是，由 F1() 和 F4() 的 APB 所占用的存储器是独立的，不会重叠。

示例中还有一个中断函数 isr()，其调用图用于以相同的方式构造所有中断代码的 APB。由于 isr() 是一个图的根，所以将总是为它分配独立的存储器，并为中断函数的 APB 分配随后的存储器。

最终结果是一个构成编译堆栈的存储器块。然后，链接器可以将该块放入器件的存储器。

对于具有多个数据存储区的器件，编译堆栈可能会被构造为多个组成部分，每个部分放入不同的存储区中。编译器会尝试将编译堆栈分配到一个存储区中，但如果它填满，则会考虑其他存储区。构造堆栈的这些组成部分的过程是相同的，但每个函数可能会有多个APB，它们将基于堆栈组成部分的目标存储区中的剩余存储器而分配到其中一个堆栈组成部分中。

代码生成器会定义一些人们可阅读的符号，如果需要，可以从汇编代码中使用它们来访问编译堆栈中的自动和参数变量。关于 C 域和汇编域符号之间的完整信息，请参见**第 5.12.3 节 “汇编代码和 C 代码之间的交互”**。

5.5.2.3 自动变量的长度限制

编译堆栈会被构造为一个可放入可用数据存储区之一中的连续块。但是，如果堆栈对于该空间太大，则会将它构造为几个块，每个块定位到不同的存储区。因此，堆栈的总长度大体上仅受器件上的可用存储器限制。

不同于非自动变量，使用增强型中档器件的线性存储器访问编译堆栈中的自动变量的效率较低。因此，对于包括 PIC18 和增强型中档 PIC MCU 在内的所有器件，编译堆栈的每个组成部分都必须可完全装入目标器件上的一个数据存储区中（但是，可以具有多个组成部分，每个分配到不同的存储区）。这会限制堆栈中对象的长度，即长度不能超出它所分配到的存储区的最大可用空间。堆栈中的自动变量越多，对于大对象的空间限制就越多。回顾一下，中档器件上的 SFR 通常存在于每个数据存储区中，所以每个存储区中的最大可用 GPR 量通常小于这些器件的存储区长度。

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。这种变量仍然由函数局部使用，但不再是自动类型，并且它们的长度限制较小。它们将按**第 5.5.2.1 节 “非自动变量分配”**所述方式分配存储器。

5.5.2.4 更改默认的自动变量分配

由于自动变量放置在堆栈中，所以没有任何方法可以移动单个变量。它们无法设为绝对变量，也无法使用 `#pragma psect` 伪指令进行移动。

通过更改默认的链接器选项，可以将自动变量所在的 psect 作为一个整体进行移动。但是，这些 psect 只能在它们默认情况下分配的数据存储区内移动。关于更改 psect 的默认链接器选项的更多信息，请参见**第 5.10 节 “main、运行时启动和复位”**。代码生成器会对这些 psect 的位置作出一些假定，如果将它们移动到一个打破这些假定的位置，代码可能会发生失败。

© 2013 Microchip Technology Inc.

5.5.3 程序空间中的变量

只有不属于自动类型，并使用 const 限定的变量会被放入程序存储器中。使用 const 限定的所有自动变量都会与其他自动变量一起放入编译堆栈中，而编译堆栈的所有组成部分都只会被定位到数据存储空间中。

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。

在一些 8 位 PIC 器件上，器件无法直接读取程序空间。对于这些器件，编译器将通过 RETLW 指令（可以被调用，并会在 W 寄存器中返回一个数据字节）的方式在程序存储器中存储数据。编译器将生成所需的代码，使它看起来如同是直接读取程序存储器。

增强型中档 PIC 器件可以直接读取其程序存储器，虽然编译器通常仍然会以 RETLW 指令的形式存储数据。这种情况下，编译器可以生成通过调用这些指令来获取程序存储器数据的代码（与普通中档器件相同），也可以直接读取指令的操作数（RETLW 指令的 LSB）。在需要读取数据时，编译器可以选择效率最高的访问方法。

在 PIC18 器件上，数据可以按单个字节的形式存储在程序存储器中。它可以使用表读指令读取。

在所有器件上，访问位于程序存储器中的数据都会比访问数据存储器中的对象慢很多。与访问关联的代码也会较长。

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。可以通过定义未初始化的 const 对象来定义符号或标号，但它不会在输出文件中产生任何内容。未初始化的 const 对象通常设为绝对对象，请参见**第 5.5.4 节“绝对变量”**。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 just define a label
```

非自动 const 变量存放的数据将基于目标器件放入几个 psect 之一。关于与这些 psect 可以链接到的位置关联的限制，请参见**第 5.15.2 节 “编译器生成的 psect”**。

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 5.12.3 节 “汇编代码和 C 代码之间的交互”**。

5.5.3.1 CONST 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 5.4.4 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。

对于低档 PIC 器件，单个 const 对象的最大长度为 255 字节。但是，可以根据需要定义许多 const 对象，前提是其总长度不超出器件的可用程序存储器长度。请注意，与其他程序代码一样，还需要一些代码才能够访问程序存储空间中使用 const 限定的数据。因此，可能需要超出对象本身长度的额外程序存储空间。这些用于访问 const 数据的额外代码仅包含一次，无论 const 限定对象的数量如何。

对于所有其他 8 位器件， const 限定对象的最大长度仅受可用程序存储器限制。这些器件也需要使用额外代码来访问 const 数据。每次访问对象时，PIC18 器件都需要额外的代码，但它通常很短。中档器件需要包含一个较长的程序，但该代码也是仅包含一次，无论 const 限定对象的数量如何。

5.5.3.2 更改默认分配

如果只是希望阻止所有变量使用一个或多个程序存储单元，从而可以将这些存储单元用于其他用途，最好使用存储器调整选项来保留存储器。关于如何这样做的信息，请参见**第 4.8.49 节 “--ROM：调整 ROM 范围”**。

如果只有几个非自动 const 变量需要定位到程序存储空间中的特定地址处，则可以将变量设为绝对变量。这可以将各个变量显式地定位到存储器中的绝对地址处。**第 5.5.4 节 “绝对变量”**对绝对变量进行了介绍。将变量设为绝对变量之后，它们的地址将硬编码到生成的输出代码中，它们将不再放入 psect，不遵循正常的存储器分配过程。

通过更改默认的链接器选项，可以将不同类别的非自动 const 变量所在的 psect 作为一个整体进行移动。但是，对于这些 psect 可以移动到的位置存在一些限制。关于更改这些 psect 的默认链接器选项的更多信息，请参见**第 5.10 节“main、运行时启动和复位”**。

程序存储器中的变量还可以通过使用 psect pragma 伪指令放置在特定位置处，请参见**第 5.14.4.8 节 “#pragma psect 伪指令”**。应基于位置要求来决定是通过这种方式还是使用绝对变量来定位变量。

© 2013 Microchip Technology Inc.

5.5.4 绝对变量

大多数变量可以通过在其声明之后加上 `@ address` 构造而定位到绝对地址处；其中，*address* 是变量在存储器中定位到的位置。这种变量称为绝对变量。

5.5.4.1 数据存储器中的绝对变量

绝对变量主要用于使C标识符的地址等效于某个特殊功能寄存器，但也可以用于将普通变量放置在数据存储器中的绝对地址处。

例如：

```
volatile unsigned char Portvar @ 0x06;
```

将声明名为 `Portvar` 的变量，它会被定位到数据存储器中的 06h 处。编译器会为该对象保留存储空间 （如果地址处于通用 RAM 中），并使变量的标识符等效于该地址。

自动变量不能设为绝对变量，因为它们位于编译堆栈中。请参见**第 5.5.2.2.1 节 “编译堆栈操作”**。编译器不会检查绝对变量是否与其他绝对变量发生重叠，因此在选择变量位置时必须考虑这一点。定义多个绝对变量处于同一地址上并不会产生任何坏处，如果这正是您所需要的。编译器不会将普通变量定位到绝对变量之上，所以这些对象之间不会发生重叠。

> **注：** 定义绝对对象会产生存储器碎片，并可能使链接器无法定位其他对象。如果可能，请尽量避免使用绝对对象。如果必须定义绝对对象，请尝试将它们放置在存储区或页的两端，从而剩余的可用存储器不会碎片化为更小的块。

针对增强型中档 PIC 器件进行编译时，为某些对象分配的存储器可能横跨多个 RAM 存储区。此类对象在汇编代码中将仅进行间接访问，并将使用这些器件上实现的线性GPR存储器。编译器将在内部对于这些对象使用线性地址 （可以映射到普通的分区地址）。

在这些器件上，为绝对对象指定的地址可以为传统的分区存储器地址或线性地址。由于线性地址的起始地址高于最大分区地址，所以可以清楚地确定所期望的地址。在以下示例中：

```
int inputBuffer[100] @ 0x2000;
```

很显然， `inputBuffer` 应放置在线性地址空间的地址 0x2000 处，在传统分区地址空间中它是存储区 0 RAM 中的地址 0x20。关于您所选器件的确切详细信息，请参见器件数据手册。

RAM 中的绝对变量不能在定义它们时初始化。定义绝对变量，然后在主干代码中的适当点为它们赋值。

5.5.4.2 程序存储器中的绝对对象

使用 const 限定的非自动对象也可以通过相同方式设为绝对对象，但是，地址将指示程序存储器中的地址。例如：

```
const int settings[] @ 0x200 = { 1, 5, 10, 50, 100 };
```

会将数组 settings 放置在程序存储器中的地址 0x200 处。

已初始化和未初始化的 const 对象都可以设为绝对对象。只需在程序存储器中定义一个标号，而不在输出文件中产生内容时，后者会很有用。

变量还可以通过使用 psect pragma 伪指令放置在特定位置处，请参见**第 5.14.4.8 节“#pragma psect 伪指令”**。应基于位置要求来决定是通过这种方式还是使用绝对变量来定位变量。使用绝对变量是最简单的方法，但这种方法仅允许放置在必须在编译前已知的地址处。psect pragma 伪指令较复杂，但可以为链接器提供将新 psect 定位到存储器中的所有灵活性。例如，可以指定变量位于某个固定地址处，或将它们放置在其他 psect 之后，或将它们放置在编译器定义的或用户定义的地址范围中的任意位置。

5.5.5 EEPROM 中的变量

对于带有片上 EEPROM 的器件，编译器提供了几种方法来访问该存储器。您可以在该存储空间中定义命名变量，或使用块访问程序来读取或写入 EEPROM。以下几节介绍了 EEPROM 访问方法。

5.5.5.1 EEPROM 初始化

对于那些支持对其EEPROM数据区域进行外部编程的器件，可以使用__EEPROM_DATA( ) 宏将初始值放入可用于编程的 HEX 文件。该宏的使用方式如下所示。

```
#include <xc.h>
__EEPROM_DATA(0, 1, 2, 3, 4, 5, 6, 7);
```

该宏具有 8 个参数，代表 8 个数据值。每个值都应为一个字节长度。未使用的值应指定为零。

可以通过多次调用该宏来定义所需数量的 EEPROM 数据。建议将该宏放置在所有函数定义之外。

该宏不能用于在运行时写入 EEPROM 存储单元；它仅用于在编程时预先装入 EEPROM 内容。

为方便起见，使用宏 _EEPROMSIZE 来表示目标器件上可用的 EEPROM 字节数。

© 2013 Microchip Technology Inc.

5.5.5.2 EEPROM 访问函数

在程序执行期间，可以调用库函数 eeprom_read() 和 eeprom_write() 来读取和写入 EEPROM。在 PIC18 器件上，这些函数是基于外设库派生的。这些函数的原型如下。

```
#include <xc.h>
unsigned char eeprom_read(unsigned char address);
void eeprom_write(unsigned char address, unsigned char value);
```

这些函数会测试并等待对 EEPROM 的所有并发写操作完成，然后再执行所需的操作。eeprom_write() 函数将启动写入 EEPROM 的过程，该过程在 eeprom_write() 返回时并未完成。写入 EEPROM 的新数据将在稍后生效。关于您所用目标器件上 EEPROM 的确切信息，请参见器件数据手册。

此外，将预处理器符号 _EEPROMSIZE 与其中一些访问方法联合使用可能会方便一些。该符号定义选定芯片可用的 EEPROM 字节数。

5.5.5.3 EEPROM 访问宏

此外，还提供了 EEPROM 函数的宏版本。这些宏的 PIC18 版本纯粹只是调用等效的函数。其他 8 位 PIC 器件的宏版本执行与对应函数类似的操作，只是存在一些下面描述的时序问题。使用这些程序的宏形式可以更快地执行并可节省一个堆栈层级，但请注意重复使用它们会增加代码长度。

对于所有器件，这些宏的用法如下所示。

```
EEPROM_READ(address)
EEPROM_WRITE(address, value)
```

EEPROM_READ 宏会返回读取的字节。

对于低档和中档器件宏 EEPROM_READ()，需要注意它与函数版本存在另外一个非常重要的差异。不同于 eeprom_read()，该宏在选择和读取 EEPROM 之前不会等待所有并发 EEPROM 写操作完成。如果无法保证在调用 EEPROM_READ() 时所有 EEPROM 写操作都已完成，则应在执行 EEPROM_READ() 之前查询相应的标志。

例如：

```
xc.h    // wait for end-of-write before EEPROM_READ
    while(WR)
        continue;    // read from EEPROM at address
    value = EEPROM_READ(address);
```

5.5.6 寄存器中的变量

将变量分配到寄存器，而不是存储单元，可以使代码更有效率。对于 MPLAB XC8，无法直接控制变量在寄存器中的放置。`register` 关键字（它只能用于自动变量）将被默默地忽略，对变量的存储器分配没有任何影响。

在 PIC 低档和中档器件上存在极少量可用于缓存变量的寄存器，由于生成代码必须经常使用它们来实现其他用途，所以使用它们几乎没有任何优势。将变量装入寄存器所需的开销将会远远大于访问寄存器带来的任何优势。目前，针对 PIC18 器件编译的代码也不会利用以下所述寄存器之外的寄存器。

某些参数通过 W 寄存器而不是存储单元传递给函数；但是，这些值通常将通过函数内的代码回存到存储器中，从而使 W 寄存器可以用于与该函数关联的代码。关于哪些参数变量可使用寄存器的更多信息，请参见**第 5.8.5 节 “函数长度限制”**。

5.5.7 动态存储器分配

所有 8 位器件都不支持动态存储器分配（使用 `malloc` 等方式进行基于堆的分配）。这是因为数据存储器的数量有限，并且该存储器还进行了分区。由于动态存储器分配存在浪费资源的特点，所以不适合用于 8 位 PIC 器件架构。

5.5.8 存储器模型

MPLAB XC8 C 编译器不使用固定存储器模型来改变变量的存储器分配。存储器分配是完全自动的，不存在任何存储器模型控制。

© 2013 Microchip Technology Inc.

5.6 操作符和语句

MPLAB XC8 C 编译器支持所有 ANSI 操作符。其中一些操作符的确切结果由实现定义。**附录 C“实现定义的行为”**详细说明了实现定义的行为。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。

5.6.1 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。

如以上所述，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。

5.6.2 循环移位

C 语言未规定循环移位操作符；但是，它允许进行移位。编译器将会检测使用移位和逻辑操作符实现循环移位操作的表达式，并高效地编译它们。

对于以下代码：

```
c = (c << 1) | (c >> 7);
```

如果 c 为 unsigned 和非 volatile 类型，则编译器将会检测到所需的操作为循环左移 1 位，并使用 PIC MCU 循环移位指令对输出进行编码。循环左移 2 位可以使用类似以下代码实现：

```
c = (c << 2) | (c >> 6);
```

这种代码优化对于长度大于 char 的整型也有效。如果无法应用优化，或者该代码移植到另一个编译器，则会实现循环移位，但通常使用移位和按位或运算实现。

5.6.3 switch 语句

编译器可使用几种策略之一对 switch 语句进行编码。默认情况下，编译器基于 switch 语句内使用的 case 分支值来选择策略。每个 switch 语句将独立地分配策略。

可以通过使用 #pragma switch 伪指令来指示策略的类型。请参见**第 5.14.4.10 节 “#pragma switch 伪指令”**，其中还列出了可用的策略类型。每种类型可能会有多个与之相关联的策略。

汇编列表文件中会打印每个 switch 语句的信息，详细说明使用 switch 进行判定的值和所列出的 case 分支值。请参见**第 6.6.4 节 “switch 语句信息”**。

© 2013 Microchip Technology Inc.

5.7 寄存器使用

编译器基于 C 源代码生成的汇编代码将使用 PIC MCU 寄存器组中的一些特定寄存器。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，如果编译器生成的汇编代码向某个寄存器装入一个值，并且没有后续代码需要使用该寄存器，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。

如果中断代码使用了所列出的任何适用寄存器，则会在发生中断时通过硬件或软件保存和恢复它们。请参见**第 5.9.3 节 “现场切换”**。

表 5-9 列出了编译器使用的特殊寄存器。

表 5-9： 编译器使用的寄存器

适用器件	寄存器名称
所有 8 位器件	W
所有 8 位器件	STATUS
所有中档器件	PCLATH
所有 PIC18 器件	PCLATH, PCLATU
增强型中档和 PIC18 器件	BSR
非增强型中档器件	FSR
增强型中档和 PIC18 器件	FSR0L, FSR0H, FSR1L, FSR1H
所有 PIC18 器件	FSR2L, FSR2H
所有 PIC18 器件	TBLPTRL, TBLPTRH, TBLPTRU, TABLAT
所有 PIC18 器件	PRODL, PRODH

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 getInput(void)
{
    ZERO = 0x1;  // do not write using C code
    c = read();
#asm
#include <caspic.h>
    bcf ZERO_bit  ; do not write using inline assembly code
#endasm
    process(c);
}
```

MPLAB XC8 无法解释在 C 代码中遇到的行内汇编代码的寄存器使用。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。

5.8 函数

函数可以根据 C 语言按照通常的方式编写。以下几节介绍了与函数相关联的实现和特殊特性。

5.8.1 函数说明符

函数可以按照通常的方式使用标准说明符 `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。如果函数为 `static`，汇编代码中用于表示函数的等效符号可能会发生变化，请参见**第 5.12.3 节“汇编代码和 C 代码之间的交互”**。该说明符不会改变对函数进行编码的方式。

下面将介绍非标准限定符。

5.8.1.1 INTERRUPT 说明符

`interrupt` 说明符指示函数是一个中断服务程序，需要对它进行特殊编码，使之适应这项任务。**第 5.9.1 节 “编写中断服务程序”**详细介绍了中断函数。

5.8.1.2 INLINE 说明符

`inline` 函数说明符用于建议对指定函数的调用应尽可能快。如果满足某些条件，编译器可能能够对指定函数的主体进行内联。

以下给出了一个将函数设为内联函数的示例。

```
inline int combine(int x, int y) {
    return 2*x-y;
}
```

对于由编译器内联的所有函数，对它的所有调用都会被编码为如同调用被替换为所调用函数主体的形式。这将在汇编代码级执行；只有使能汇编优化器时，才会发生内联。即使对函数进行内联，编译器仍然可能对函数本身进行编码。

如果发生了内联，则程序的执行速度会提高，因为它可以消除与调用相关联的调用和返回序列。由于实际上不会执行任何调用指令，所以它也可以减少硬件堆栈的使用量。汇编列表文件中显示的调用图中不会反映任何堆栈使用量减少，因为该文件是在内联发生之前生成的。

如果发生了内联，并且与内联函数主体关联的汇编代码极短，而函数本身不会输出，则代码长度可能会减小。如果内联函数的主体大于它替换的调用 / 返回序列，并且不止一次调用函数，则代码的长度会增大。只有对于生成少量汇编代码的函数，才应考虑使用该说明符。请注意，函数主体中的 C 代码量并不是其生成汇编代码长度的良好指标（见**第 3.6.13 节 “如何确定某个函数有多大？”**）。

如果某个函数本身包含行内汇编代码，则不能对它进行内联。如果函数的汇编代码包含某些特定的汇编序列，则也可能无法对函数进行内联。如果函数引用了静态对象，为了符合 ANSI 标准，将会生成一条警告。如果内联未成功，也会发出警告。代码不应对内联是否成功，以及将执行与函数相关联的哪些汇编代码作出任何假定。

© 2013 Microchip Technology Inc.

该说明符执行的任务与#pragma inline伪指令相同，请参见**第5.14.4.4节“#pragma Intrinsic 伪指令”**。

5.8.2 外部函数

如果需要调用在程序 C 源代码之外定义的某个函数（它可能属于单独编译的代码的一部分，例如自举程序，或处于汇编代码中），则需要提供该函数的声明，使编译器知道如何对调用进行编码。

如果该函数需要参数或返回一个值，则编译器可能会使用一个符号来表示用于存储这些值的存储单元；要确定在这种参数传递中使用的是寄存器还是存储单元，请参见**第 5.8.6 节 “函数参数”**和**第 5.8.7 节 “函数返回值”**。通常情况下，编译器会在对 C 函数进行编码时定义该符号，但如果函数是外部函数，不由编译器进行编码，则必须手动定义符号值。如果使用了参数或返回值，并且它将存储在存储器中，则您的代码必须定义相应的符号并向它赋予相应存储单元的值。

可以通过外部编译（对函数进行编译）生成的映射文件或汇编代码确定其值。如果函数以 C 语言编写，则查找符号 ?_*funcName*；其中，*funcName* 是函数的名称。它可以在进行调用的程序中通过简单的 EQU 汇编器伪指令定义。例如，可以在 C 源代码中放置以下代码片断：

```
#asm
GLOBAL ?_extReadFn
?_extReadFn EQU 0x20
#endasm
```

或者，也可以直接将汇编代码包含在汇编模块中。这会将外部函数 extReadFn 的参数区域的基址定义为 0x20。

如果未定义该符号，编译器将发出未定义符号错误。如果需要，该错误可以用于验证由编译器用来对调用进行编码的名称。

建议不要通过将某个整型强制转换为函数指针来间接调用函数，但在这种情况下，编译器将使用符号名称中常量的值；例如，要调用某个位于地址 0x200 处的函数，将需要定义符号 ?0x200，并将它设为该函数的参数 / 返回值存储单元的位置。例如：

```
#asm
GLOBAL ?0x200
?0x200 EQU 0x55
#endasm
```

请注意，一个函数的返回值（如使用）将共用分配给该函数所有参数的相同存储单元，并且它们将使用相同的符号。

5.8.3 可执行代码的分配

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。

在低档和中档器件上，程序存储器会进行分页（比较：在数据存储空间中使用的分区）。该存储器仍然是顺序的（地址在跨越页边界时是连续的），但分页意味着从一个页中代码对另一个页中某个标号的任何调用或跳转都必须使用较长的指令序列来完成这一任务。关于程序存储器和指令集的更多信息，请参见器件数据手册。

PIC18 器件未实现任何程序存储器分页。 CALL 和 GOTO 指令是双字指令，它们的目标不受限制。相对转移指令的范围有限，但它不是基于任何分页边界。

与每个函数关联的生成代码最初由编译器放置在它自己的 psect 中，请参见**第 5.15.1 节 “程序段”**。这些psect具有诸如text*n*之类的名称；其中，n是一个数字，例如text98。但是，psect 之后可能会在编译过程中合并，从而在某个 psect 中可能会包含多个函数的生成代码。

当程序存储器进行分页时，同一 psect 内的函数可以使用较短形式的标号调用和跳转，因此最好将尽可能多函数的代码合并到同一 psect 中。这些 text psect 链接到程序存储器中的任意位置 （见**第 5.10 节 “main、运行时启动和复位”**）。

如果存放函数关联代码的 psect 的长度超出页长度，汇编器优化器会对它进行拆分。拆分的 psect 将具有 text*n*_split_*s* 形式的名称。例如，如果 text102 psect 超出页长度，它可能会拆分为 text102_split_1 和 text102_split_2 psect。该过程是完全自动的，但您需要知道，如果函数关联代码的长度大于页长度，代码的效率可能会稍降，因为需要使用较长的跳转和调用指令序列来将控制转移给其他页中的代码。

下面列出了每个 psect 类别的基本名称。**第 5.15.2.1 节 “程序空间 psect”** 列出了所有程序存储器 psect 名称的完整列表。

maintext 与特殊函数 main 关联的生成代码放置在该 psect 中。一些优化和特性将不会应用于该 psect。

text*n* 这些 psect（其中的 *n* 是十进制数字）包含不需要特殊链接位置的所有其他可执行代码。

5.8.4 更改默认的函数分配

如果只是希望阻止函数使用一个或多个程序存储单元，从而可以将这些存储单元用于其他用途，最好使用存储器调整选项来保留存储器。关于如何这样做的信息，请参见**第 4.8.49 节 “--ROM：调整 ROM 范围”**。

与 C 函数关联的汇编代码可以放置在一个绝对地址处。这可以通过使用 @ *address* 构造 （与用于绝对变量的方式类似）实现。此类函数称为*绝对函数*。

以下给出了将函数放置在地址 400h 处的绝对函数示例：

```
int mach_status(int mode) @ 0x400
{
/* function body */
}
```

如果检查汇编列表文件，可以看到与函数相关联的函数标号和第一条汇编指令被定位到 0x400 处。您可以使用汇编列表文件 （见**第 6.5 节 “汇编代码级优化”**）或映射文件（见**第 7.4 节 “映射文件”**）来确认函数是否按预期发生移动。

© 2013 Microchip Technology Inc.

如果对于中断函数使用该构造，它只会影响与中断函数主体关联的代码的位置。函数代码之前的中断现场切换代码将不会重定位，因为它必须链接到中断向量处。关于如何移动复位和中断向量位置的信息，另请参见**第 4.8.22 节 “--CODEOFFSET：将程序代码偏移到某个地址”**，它对于设计诸如自举程序之类的应用程序可能会很有用。

与绝对变量不同，与绝对函数关联的生成代码仍然放置在某个 psect 中，但 psect 仅供该函数专用。 psect 名称具有以下形式。**第 5.10 节 “main、运行时启动和复位”**列出了所有 psect 名称的完整列表。

***xxx*_text** 定义已设为绝对函数（即，放置在某个地址处）的函数的 psect。*xxx* 将为与该函数相关联的汇编符号。例如，如果函数 `rv()` 被设为绝对函数，与它关联的代码将出现在名为 `_rv_text` 的 psect 中。

函数还可以通过使用 `psect` pragma 伪指令放置在特定位置处，请参见**第 5.14.4.8 节 “#pragma psect 伪指令”**。应基于位置要求来决定是通过这种方式还是使用绝对函数来定位函数。

使用绝对函数是最简单的方法，但这种方法仅允许放置在必须在编译前已知的地址处。`psect` pragma 伪指令较复杂，但可以为链接器提供将新 psect 定位到存储器中的所有灵活性。例如，可以指定函数位于某个固定地址处，或将它们放置在其他 psect 之后，或将它们放置在编译器定义的或用户定义的地址范围中的任意位置。

5.8.5 函数长度限制

对于所有器件，函数的生成代码仅受可用程序存储器限制。在分页器件上，函数的长度可以大于一个页长度；但是，由于需要使用较长的调用序列来跳转到或调用其他页中的目标，这些函数的效率可能较低。更多详细信息，请参见**第 5.8.3 节 “可执行代码的分配”**。

5.8.6 函数参数

MPLAB XC8 使用固定约定来向函数传递参数。用于传递参数的方法取决于所涉及参数的长度和数量。

> **注：** 名称 “实参”（argument）和 “形参”（parameter）通常可以互换，但实参通常是指传递给函数的实际值，形参则是指由函数定义的用于存储实参的变量。

编译器将或者通过 W 寄存器传递参数，或者通过被调用函数的自动参数块 （APB）中的参数存储器传递参数。如果第一个参数的长度为一个字节，则它通过 W 寄存器传递。所有其他参数都通过 APB 传递。这适用于基本类型和聚合类型，例如结构。

参数将与函数的自动变量一起组合到 APB 中，并放入编译堆栈。关于编译堆栈的详细信息，请参见**第 5.5.2.2.1 节 “编译堆栈操作”**。参数变量将以相对于符号 `?_`*`function`* 的偏移量的形式进行引用；其中，*`function`* 是定义了该参数的函数的名称（即，被调用的函数）。

与自动变量不同，参数变量将严格按照它们在函数原型中的出现顺序分配存储器。这意味着参数将总是被放入同一存储区。函数的自动变量可以按任意顺序跨多个存储区分配。

对于具有可变参数列表（在原型中使用*省略号*定义，它们被称为非原型参数）的函数，可变参数将与命名参数一起放入参数存储器。

例如，假设存在以下 ANSI 样式的函数。

```
void test(char a, int b);
```

函数 `test()` 将在其函数自动参数块中接收参数 `b`，在 `W` 寄存器中接收 `a`。对该函数的调用：

```
test(xyz, 8);
```

将生成以下形式的代码：

```
MOVLW     08h       ; move literal 0x8 into...
MOVWF     ?_test    ; the auto-parameter memory
CLRF      ?_test+1  ; locations for the 16-bit parameter
MOVF      _xyz,w    ; move xyz into the W register
CALL      (_test)
```

在该示例中，参数 `b` 存放在存储单元 `?_test`（LSB）和 `?_test+1`（MSB）中。

总是可以在汇编列表文件中检查用于调用函数的确切代码或用于从函数中访问参数的代码。关于生成该文件的选项，请参见**第 4.8.17 节 “--ASMLIST：生成汇编列表文件”**。如果要编写必须使用参数调用函数的汇编程序，或在调用时接受参数的汇编程序，这会很有用。以上示例未考虑数据存储器分区或程序存储器分页的情况，这两种情况下可能需要额外的指令。

© 2013 Microchip Technology Inc.

5.8.7 函数返回值

函数返回值将使用 W 寄存器或函数自动参数块中的参数存储器传递给调用函数。对于返回其参数修改副本的函数，将返回值也定位到与参数所用存储器相同的存储器中可以减小代码长度。

8 位值将通过 W 寄存器从函数返回。返回值大于一个字节时，将通过函数的参数存储区返回，最低字 （lsw）位于最低的存储单元中。

例如，以下函数：

```
int return_16(void)
{
    return 0x1234;
}
```

将使用以下形式的代码退出：

```
MOVLW  34h
MOVWF  (?_return_16)
MOVLW  12h
MOVWF  (?_return_16)+1
RETURN
```

对于返回值长度大于 4 个字节的 PIC18 目标器件，参数区域的地址也会被放置在 FSR0 寄存器中。

5.8.8 调用函数

所有 8 位器件都使用硬件堆栈来存放函数返回地址。该堆栈的深度会因器件而不同。

通常情况下，在调用 C 函数时，使用 CALL 汇编指令来将控制转移到 C 函数。每个函数调用都会使用一个堆栈层级。该堆栈层级将在被调用程序执行 RETURN 指令之后释放。如果被调用函数在返回之前调用了另一个函数，则堆栈使用量会增长。如果硬件堆栈发生溢出，函数返回地址将被破坏，代码最终将发生失败。

--RUNTIME 选项的 stackcall 子选项控制在编译器检测到硬件堆栈由于嵌套调用过多而即将溢出时的编译器行为。关于该选项的详细信息，请参见**第 4.8.50 节 “--RUNTIME: 指定运行时环境”**。如果该子选项被禁止（默认状态），则在某个调用会导致超出堆栈深度时，编译器将发出警告，指示发生这种情况。对于 PIC18 器件，这是进行调用的惟一方法，但对于其他 8 位器件，编译器可以切换到另一种调用方法，详述如下。

如果使能了 stackcall 子选项，则编译器会自动切换为使用一种涉及到使用查表的方法（该方法不需要使用硬件堆栈），而不是发出一条警告。该特性对于 PIC18 器件不可用。

采用查表方法时，将通过直接跳转（不是调用）到函数地址来转移到该函数。在执行该操作之前，会在被调用函数中的一个临时位置存储一条特殊的 “返回”指令 （以跳转指令的形式实现）。该返回指令能够将控制返回给调用函数。

这种函数调用方式允许对函数进行深度嵌套，而不会使低档和中档器件上有限的可用堆栈发生溢出；但是，它需要牺牲一些存储器和程序速度。

5.8.8.1 函数内的存储区选择

函数可以也可能会在选择任意 RAM 存储区的情况下返回。关于 RAM 存储区的更多信息，请参见**第 5.5.1 节 “地址空间”**。

编译器会跟踪与每个函数关联的生成代码中所做的存储区选择，即使在有对其他函数的函数调用时。如果可以确定函数返回时所选的存储区，编译器将使用该信息来尝试删除多余的存储区选择指令，否则就可能会在生成代码中插入这些指令。

编译器无法跟踪以汇编代码编写的程序选择的存储区，即使是从 C 代码中调用它们。在此类程序返回时，编译器不会作出任何关于选定存储区的假定。

汇编列表文件中显示的与每个函数关联的 “Tracked objects”（被跟踪的对象）部分与这种存储区跟踪机制相关。关于这些文件内容的更多信息，请参见**第 6.5 节 “汇编代码级优化”**。

© 2013 Microchip Technology Inc.

5.9 中断

MPLAB XC8 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*中断服务程序*或 ISR。

> **注：** 低档器件不采用中断，因此以下几节仅适用于中档、增强型中档和 PIC18 器件。

在中档和增强型中档器件上只有一个中断向量。无论中断源如何，器件总是跳转到程序存储器中的一个特定位置，并从该地址处继续执行。该地址属于器件特性，无法更改。

每个中档器件中断源通常在 SFR 中具有一个可以禁止该中断源的控制标志。此外，还存在一个全局中断允许标志，它可以禁止所有中断源，并确保永远不会发生中断。中断源没有优先级。关于您所用器件如何处理中断的完整信息，请查看器件数据手册。

PIC18 器件具有两个独立的中断向量，以及决定执行哪个中断代码的优先级方案。两个中断会被指定为低优先级和高优先级。外设通过外设 SFR 中的设置与其中一种中断优先级 （向量）关联。

每个中断源可以通过 SFR 中与该中断源相关联的控制标志来禁止。除了全局中断允许标志之外，还具有可以禁止每个中断优先级的其他标志。

*中断代码*指的是由于发生中断而执行的任何代码，包括从 ISR 中调用的函数和库代码。中断代码在执行相应的中断返回指令时完成。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。

5.9.1 编写中断服务程序

函数限定符 `interrupt` 可以应用于 C 函数定义，从而使得在发生中断时执行它。编译器会以不同于任何其他函数的方式处理中断函数，即生成代码来保存和恢复所用的所有寄存器，并使用一条特殊指令返回。

如果使用了 xc8 选项 `--STRICT`，则 `interrupt` 关键字将变为 `__interrupt`。

中断函数必须声明为 `void interrupt` 类型，并且不能具有参数。这是对于中断函数有意义的惟一函数原型，因为永远不会在源代码中直接调用它们。

在 PIC18 器件上，中断函数默认设为高优先级。要创建一个低优先级的中断函数，除了在函数定义中使用 `interrupt` 之外，还需要使用限定符 `low_priority`。

中断函数不能直接从 C 代码中调用（因为它使用了不同的返回指令），但它们自己可以调用其他函数，包括用户定义的函数和库函数。

可能会有许多中断源共用同一中断向量，但只会有一个中断函数与每个中断向量相关联。然后，中断函数必须包含用于在继续操作之前确定中断源的代码。如果程序中的中断函数数量多于中断向量，将会产生错误。

以下给出了中断函数的示例。

```
int tick_count;

void interrupt tc_int(void)
{
    if (TMR0IE && TMR0IF) {
        TMR0IF=0;
        ++tick_count;
        return;
    }
    // process other interrupt sources here, if required
}
```

由编译器生成的代码将被放置在中断向量地址处，它将在任何必需的现场切换之后执行该函数。

请注意，中断函数中的代码会通过查看中断允许位（TMR0IE）和中断标志位（TMR0IF）来检查中断源；在此例中，中断源为一个定时器。中断允许标志检查是必需的，因为即使未将某个外设配置为允许产生中断，与该外设相关联的中断标志也会置为有效。

以下给出了一个可以针对 PIC18 器件编写的低优先级中断函数的示例。

```
void interrupt low_priority tc_clr(void) {
    if (TMR1IE && TMR1IF) {
        TMR1IF=0;
        tick_count = 0;
        return;
    }
    // process any other low priority sources here
}
```

5.9.2 指定中断向量

中断函数不能在运行时更改。即，不能具有备用中断函数，并在程序执行期间选择哪个函数将处于活动状态。如果程序中的中断函数数量多于中断向量，将会产生错误。

© 2013 Microchip Technology Inc.

5.9.3 现场切换

5.9.3.1 中断现场保护

发生中断时，硬件会自动保存一些寄存器。在由编译器生成的代码中，必须保存中断函数使用的所有寄存器或编译器临时对象，那些由硬件保存的除外。这即为*现场保护*或*现场切换*代码。

关于在发生中断时必须由硬件或软件保存和恢复的寄存器，请参见**第 5.7 节 “寄存器使用”**。

增强型中档 PIC 器件会在硬件中保存 W、STATUS、BSR 和 FSR*x* 寄存器（使用特殊的影子寄存器），因此不需要通过软件保存这些寄存器。惟一可能需要保存的寄存器是 BTEMP1[1]，它是一个行为类似于伪寄存器的编译器临时存储单元。这使增强型中档 PIC 器件上的中断函数可以非常快速、高效。

其他中档 PIC 处理器在发生中断时仅保存整个 PC（不包括 PCLATH 寄存器）。如果需要，编译器生成的代码必须保存 W、STATUS、FSR 和 PCLATH 寄存器，以及 BTEMP1 伪寄存器。

默认情况下， PIC18 高优先级中断函数将利用其内部影子寄存器来保存 W、 STATUS 和 BSR 寄存器。使用的所有其他寄存器通过软件保存。请注意，对于一些较老的器件，如果是针对 MPLAB ICD 调试器进行编译，编译器将不会使用影子寄存器，因为调试器本身会利用这些影子寄存器。一些勘误规避措施也会阻止使用影子寄存器，请参见**第 4.8.27 节 “--ERRATA：指定勘误规避措施”**。

对于低优先级 PIC18 中断，或无法使用影子寄存器时，将通过软件来保存中断代码使用的所有寄存器。

编译器可以准确地确定中断函数使用哪些寄存器和对象，或它调用的任何函数（基于由编译器生成的调用图），并相应地保存它们。

它*不会*扫描嵌入到中断函数中的汇编代码的寄存器使用情况。因此，如果在中断函数中包含了行内汇编代码，则可能需要添加额外的汇编代码来保存和恢复所使用的任何寄存器或存储单元。对于中断代码调用的任何汇编程序同样也是如此。

如果要由编译器保存 W 寄存器，它可能会被存储到公共 RAM 中保留的存储器中。如果所编写的代码针对的器件没有公共存储器，则会在所有RAM存储区中都保留一个字节，用作 W 寄存器的存储单元。

大多数需要保存的寄存器都在中断函数的 `auto` 区域中分配存储器。它们可以像任何其他自动变量一样对待，并使用相同的汇编符号。在中档器件上，W 寄存器存储在伪寄存器 BTEMP0 中。

5.9.3.2 现场恢复

在中断函数返回之前，软件会自动恢复由软件保存的所有对象。恢复的顺序与保存现场时所用的顺序相反。

1. BTEMP 寄存器是由编译器分配的存储单元，但它被视为一个寄存器，用于代码生成目的。并不是所有器件都使用它。

5.9.4 允许中断

包含 <xc.h> 之后，将有两个宏可供使用，它们控制所有可用中断的屏蔽。这两个宏为 ei() （用于允许或取消屏蔽所有中断）和 di() （用于禁止或屏蔽所有中断）。

在所有器件上，它们都会影响 INTCON 寄存器中的 GIE 位。在允许了程序中需要的中断对应的中断允许位之后，应使用这些宏。

例如：

```
ADIE = 1;  // A/D interrupts will be used
PEIE = 1;  // all peripheral interrupts are enabled
ei();      // enable all interrupts
// ...
di();      // disable all interrupts
```

注： 永远不要在中断函数自身内部重新允许中断。在执行 RETFIE 指令时，硬件会自动重新允许中断。在中断函数内重新允许中断可能导致代码失败。

5.9.5 函数复制

编译器假定随时都可能会发生中断。由于所有函数都不可重入（因为局部对象依赖于编译堆栈，见**第 5.5.2.2.1 节 “编译堆栈操作”**），如果某个函数既由中断函数调用，又由主干代码调用，则可能会导致代码失败。

MPLAB XC8 具有一项功能，即可以复制与从程序调用图中多个调用树中调用的所有函数关联的输出。调用图中将会具有一个与主干代码关联的调用树，以及一个用于中断函数的调用树 （如定义）。

主干代码将调用原始函数的输出，而中断将调用复制函数的输出。复制仅发生在被调用函数的输出中；不会复制 C 源代码本身。复制的代码和数据将使用不同的符号，并分配不同的存储器，所以是完全独立的。

这类似于编译器中未实现该功能的情况下您需要执行的过程：可以手动复制 C 函数，赋予不同的名称，一个从主干代码中调用；另一个从中断函数中调用。但是，您必须同时维护两个函数，并且如果代码移植到一个支持可重入的编译器上，将需要还原代码。

编译器生成的复制函数输出将对于它使用的汇编符号具有惟一标识符。标识符由原始输出中使用的相同名称加上前缀 i1 组成。在 PIC18 器件上，对于低优先级和高优先级中断，复制的 PIC18 函数将分别使用前缀 i1 和 i2。

从主干代码中调用的函数的输出将不使用任何前缀，汇编名称将为通常使用的那些名称。

为了进行说明，在一个程序中，函数 main 调用了名为 input 的函数。中断函数也会调用该函数。

© 2013 Microchip Technology Inc.

检查汇编列表文件将会看到原始函数和复制函数输出的汇编代码。对应于 C 函数 input()的输出将使用汇编标号_input。复制函数所使用的相应标号将为i1_input。如果原始函数引用了某个临时变量，生成的输出将使用符号 ??_input，而复制输出的符号则为 ??i1_input。即使函数输出内的局部标号也会以相同的方式进行复制。汇编列表文件中的调用图将显示对这两个函数的调用，就如同它们是独立编写的。在映射文件符号表中也可以看到这些符号。

该功能使编程人员可以对于采用可重入或不可重入模型的编译器使用相同的源代码。它并未处理递归调用函数的情形。

与库函数关联的代码使用相同的方式进行复制。这也适用于隐式调用的库程序，例如那些执行与 C 操作符相关联的除法或浮点运算的程序。

5.9.5.1 禁止复制

可以通过使用一条特殊 pragma 伪指令来禁止函数的自动复制。

只有源代码确保从任何主干代码中调用函数时不会发生中断，才应这样做。通常，这可以通过在调用函数之前禁止中断实现。在调用函数之后在函数内禁止中断并不足够；如果在执行函数时发生中断，代码可能会失败。关于如何禁止中断的更多信息，请参见**第 5.9.4 节 “允许中断”**。

该 pragma 伪指令为：

```
#pragma interrupt_level 1
```

该 pragma 伪指令应放置在不需要复制的函数的定义之前。该 pragma 伪指令只会影响其后定义的第一个函数。

例如，如果函数 read 仅在禁止中断时从主干代码中调用，则如果也会从中断函数中调用它，可以通过如下方式避免函数复制。

```
#pragma interrupt_level 1
int read(char device)
{
    // ...
}
```

在主干代码中，该函数通常以如下方式调用：

```
di();  // turn off interrupts
read(IN_CH1);
ei();  // re-enable interrupts
```

指定的级别值指示对于哪种中断不会复制函数。对于中档器件，级别应总是为 1；对于 PIC18 器件，它可以为 1 或 2，分别对应于低优先级或高优先级中断函数。要对于两种中断优先级禁止复制，可以使用 pragma 伪指令两次，分别指定级别 1 和 2。对于以下函数，如果也会从低优先级和高优先级中断函数中调用它，将不会复制它。

```
#pragma interrupt_level 1
#pragma interrupt_level 2
int timestwo(int a) {
   return a * 2;
}
```

5.10 MAIN、运行时启动和复位

标识符 main 是一个特殊的标识符。在程序中必须总是有且仅有一个名为 main() 的函数。这是在程序中执行的第一个函数。

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。以下几节介绍了该代码的操作和控制。

编译器会在main()的末尾插入特殊的代码，该代码在该函数结束时（即执行了main()内的 return 语句，或者代码执行到达 main() 的终止右大括号）执行。这段特殊代码会导致执行跳转到地址 0，即所有 8 位 PIC 器件的复位向量。这实质上是执行一种软件复位。请注意，软件复位后的寄存器状态可能不同于硬件复位后。

建议 main() 函数不要结束。在 main() 中的代码外围或代码末尾添加一个永远不会终止的循环 （如 while(1)），从而使该函数的执行永远不会终止。例如，

```
void main(void)
{
    // your code goes here
    // finished that, now just wait for interrupts
    while(1)
        continue;
}
```

5.10.1 运行时启动代码

C 程序要求先初始化一些特定的对象，并且器件处于某种特定状态，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
- 清零未初始化的全局变量
- 对寄存器或器件状态进行常规设置

与链接一个通用的预编译程序的传统方法不同，MPLAB XC8 使用了一种更高效的方法，即实际上根据用户的程序确定所需的运行时启动代码。**第 4.4.2 节 “启动和初始化”**介绍了所使用的文件和如何控制这一过程的详细信息。以下几节详细介绍了运行时启动代码实际上做些什么。

运行时启动代码在 main() 函数之前执行，但如果需要在复位后立即执行任何特殊的初始化，则应使用后面的**第 5.10.2 节 “上电程序”**中介绍的上电功能。

下表列出了启动和上电代码所使用的重要汇编标号。

表 5-10： 重要的汇编标号

标号	位置
reset_vec	复位向量单元处 （0x0）
powerup	上电程序开始处 （如使用）
start	startup.as 中的运行时启动代码的开始处
start_initialization	C 输出代码中的 C 初始化启动代码的开始处

© 2013 Microchip Technology Inc.

5.10.1.1 对象的初始化

运行时启动代码的一个任务就是确保所有已初始化变量在程序开始执行之前包含其初始值。已初始化变量是那些不属于自动对象并在其定义中赋予初始值的变量，例如以下示例中的 `input`。

```
int input = 88;
void main(void) { ...
```

此类已初始化对象具有两个组成部分：其初始值 （以上示例中的 0x0088），它存储在程序存储器中 （即，放置在 HEX 文件中）；以及变量在程序执行期间 （运行时）将驻留的并用于进行访问的 RAM 中保留的空间。

第 5.15.2 节 “编译器生成的 psect”介绍了用于存储这些组成部分的 psect。

运行时启动代码会将初始值的所有数据块从程序存储器复制到 RAM，从而使这些变量在执行 `main()` 之前包含正确的值。通过禁止 `--RUNTIME` 的 `init` 子选项可以忽略该操作。例如：

```
--RUNTIME=default,-init
```

缺少这部分运行时启动代码时，已初始化变量的内容在程序开始执行时将是无法预测的。依赖于变量包含其初始值的代码将发生失败。

由于自动对象是动态创建的，所以需要在定义这些对象的函数中放置用于执行对象初始化的代码。自动对象的初始值有可能会在函数的每个实例中发生更改，所以初始值不能在程序存储器中存储并复制。因此，运行时启动代码不会考虑已初始化的自动对象，而是由每个函数输出中的汇编代码进行初始化。

> **注：** 已初始化的自动变量可能会影响代码性能，特别是对象长度较大时。可以考虑改为使用全局或静态对象。

内容需要在复位时或甚至电源关闭时保存的变量应使用`persistent`限定符进行限定，请参见**第 5.4.8.1 节 “persistent 类型限定符”**。这种变量链接在存储器的不同区域，运行时启动代码不会以任何方式更改它。

如果对象进行了初始化，执行该操作的运行时启动代码会破坏 STATUS 寄存器的内容。对于某些器件，需要使用该寄存器中的 TO 和 PD 位来确定复位的原因。您可以选择获取该寄存器的副本，从而可以在稍后检查它。更多信息，请参见**第 5.10.1.4 节 “STATUS 寄存器保存”**。

5.10.1.2 清零对象

在程序开始执行之前，必须清零未初始化的非自动对象。该任务也是由运行时启动代码执行的。

未初始化变量是那些不属于自动对象并在其定义中未赋值的变量，例如以下示例中的 `output`。

```
int output;
void main(void) {...
```

此类未初始化对象只需对象在程序执行期间 （运行时）将驻留并用于进行访问的 RAM 中保留空间即可。

第 5.15.2 节 “编译器生成的 psect”介绍了用于存储这些组成部分的 psect，它们的名称通常基于首字母缩略词 “bss”（Block Started by Symbol，以符号开始的块）。

运行时启动代码会清零由未初始化变量占用的所有存储单元，因此它们在 `main()` 执行之前将包含零。

在复位时需要保存内容的变量应使用 `persistent` 进行限定。更多信息，请参见**第 5.4.8.1 节 “persistent 类型限定符”**。这种变量链接在存储器的不同区域，运行时启动代码不会以任何方式更改它。

如果对象进行了初始化，执行该操作的运行时启动代码会破坏 STATUS 寄存器的内容。对于某些器件，需要使用该寄存器中的 TO 和 PD 位来确定复位的原因。您可以选择获取该寄存器的副本，从而可以在稍后检查它。更多信息，请参见**第 5.10.1.4 节 “STATUS 寄存器保存”**。

5.10.1.3 振荡器校准

一些 PIC 器件随附了一个振荡器校准常量，它预先编程到器件的程序存储器中。如果需要，可以将该常量写入 OSCCAL 寄存器来校准内部 RC 振荡器。

代码会自动放置在运行时启动代码中，用以装入该校准值，请参见**第 5.3.9.1 节 “振荡器校准常量”**。

© 2013 Microchip Technology Inc.

5.10.1.4 STATUS 寄存器保存

--RUNTIME 选项的 resetbits 子选项（见**第 4.8.50 节 “--RUNTIME：指定运行时环境”**）可以在运行时启动代码的其余部分破坏 STATUS 寄存器之前保存 STATUS 寄存器内某些数据位。从复位条件中恢复之后，可以通过检查这些位的状态来确定复位的原因。

整个 STATUS 寄存器都会被保存到汇编变量 ___resetbits 中。可以使用以下声明从 C 代码中访问该变量：

```
extern unsigned char __resetbits;
```

编译器定义了汇编符号 ___powerdown 和 ___timeout 来表示 STATUS 寄存器内的掉电位和超时位的位地址，可以在需要时使用它们。可以使用以下声明从 C 代码中访问它们：

```
extern bit __powerdown;
extern bit __timeout;
```

在以上符号中，可以注意到 C 变量使用了两个前导下划线字符，而汇编等效符号使用了三个。关于映射的更多详细信息，请参见**第 5.12.3.1 节 “等效汇编符号”**。

关于在 MPLAB IDE 中使用该选项的信息，请参见**第 4.9 节“MPLAB IDE V8 Universal Toolsuite 等效选项”**。

5.10.2 上电程序

一些硬件配置需要进行特殊的初始化，通常是在复位之后的前几个指令周期内进行。为了实现该目的，通过*上电*程序提供了复位向量的 hook。

该程序可以在一个用户定义的汇编模块中提供，该模块将在复位后立即执行。powerup.as 文件中提供了一个模板上电程序，该文件位于编译器分发版的 sources 目录中。更多详细信息，请参见该文件中的注释。

应将该文件复制到您的工作目录，进行修改并作为一个源文件包含到项目中。不需要任何特殊的链接器选项或其他代码。编译器将检测您是否已经定义了上电程序，并将自动使用它，前提是该程序中的代码包含在名为 powerup 的 psect 中。

为确保正确工作（使用编译器生成的默认运行时启动代码时），代码必须使用 GOTO 指令结束，并且该指令指向名为 start 的标号。与所有用户定义的汇编代码一样，该文件中的所有代码也必须考虑到程序存储器分页和 / 或数据存储器分区，以及适用于您所用器件的任何勘误问题。程序的入口点已由运行时启动代码定义，因此不应在上电程序中使用 END 伪指令指定它 （如使用）。关于该汇编器伪指令的更多信息，请参见**第 6.4.9.2 节 “END”**。

5.11 库程序

5.11.0.1 使用库程序

在源代码中引用库函数之后，库函数（以及任何关联的变量）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。

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）。

在以下示例中，源代码中未包含 sqrt 的定义，因此编译器会通过搜索库来在其中查找定义。查找到之后，它会将 sqrt 的函数链接到程序中。

```
#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);
}
```

5.11.1 printf 程序

与 printf 函数关联的代码并未预编译成库文件。printf() 函数将基于在分析用户 C 代码之后定制的一个特殊 C 模板文件生成。关于使用 printf 库函数的更多信息，请参见 **“PRINTF 和 VPRINTF”**。

模板文件位于编译器分发版的 lib 目录中，名为 doprnt.c。它包含 printf() 函数最小程度的实现，但以条件代码的形式包含了更多高级功能，这些功能可以通过在对它（与您的代码一起）进行编译时定义的预处理器宏利用。

解析器和代码生成器会分析 C 源代码，搜索对 printf 函数的调用。对于所用调用，将会对 printf() 格式字符串中指定的占位符进行整理，以产生最终函数所需功能的列表。然后，使用初步分析指定的那些宏对 doprnt.c 文件进行预处理，从而创建一个针对所编译项目的定制 printf() 函数。进行解析之后，基于 doprnt.c 生成的 p 代码输出将在最终代码生成步骤中与 C 程序的其余部分进行组合。

例如，如果某个程序包含一个对 printf() 的调用，它具有以下形式：

```
printf(”input is: %d”);
```

编译器会注意到仅使用了 %d 占位符，链接到程序中的 doprnt.c 模块将仅包含处理十进制整型打印的代码。

假设现在对代码进行更改，并添加另一个对 printf() 的调用。新调用为以下形式：

```
printf(”output is %6d”);
```

现在，编译器将会检测到必须同时使能用于将十进制整型打印为特定宽度的额外代码。

© 2013 Microchip Technology Inc.

当检测到更多的 printf() 功能时， printf() 函数生成代码的长度将会增加。

如果printf()调用中的格式字符串不是以上形式的字符串字面值，而是一个指向字符串的指针，编译器将无法可靠地预测 printf() 用法，因而它会强制生成更完整的printf()版本。

但是，即使无法扫描 printf() 占位符，编译器仍然可以对于函数的用法作出某些假设。特别是，编译器会查看 printf() 的附加参数（跟在格式字符串表达式之后的参数）的数量和类型，以确定哪些占位符可能是有效的。这使生成的 printf() 程序即使在这种情况下也可以将长度和复杂性保持在最低程度。

例如，如果以如下形式调用了 printf()：

```
printf(myFormatString, 4, 6);
```

举例来说，编译器可以确定不需要任何浮点型占位符，并在 printf() 函数输出中忽略它们。由于格式字符串之后的参数是非原型参数，所以它们的类型必须与占位符匹配。

用户无法控制该操作的任何方面（除了通过调整对 printf() 的调用）；但是，可以观察程序所使用的实际 printf() 代码。如果编译某个使用 printf() 的程序，驱动程序会留下预处理的 doprnt.c 版本。该模块名为 doprnt.pre，处于工作目录中，它将显示实际包含到 printf 程序中的 C 代码。由于该代码已经过预处理，作为 C 预处理器执行的正常操作的一部分，缩进和注释都会被剔除。

5.12 混合使用 C 代码和汇编代码

通过使用两种不同的技术，可以将汇编语言代码与 C 代码混合使用：编写汇编代码，并将它放入一个独立的汇编模块，或将其作为行内汇编代码包含到 C 模块中。

> **注：** 项目中包含的汇编代码越多，维护它的难度和时间就越高。随着项目的开发，编译器可能会执行不同的优化，因为这些优化将基于整个程序。如果编译器进行了更新，由于经过更新的编译器的工作方式可能存在差异，汇编代码可能需要进行修改。这些因素不会影响以 C 编写的代码。
> 如果必须添加汇编代码，则最好将它编写为处于独立汇编模块中的自包含程序，而不是将它嵌入到 C 代码中。

5.12.1 集成汇编语言模块

整个函数可以用汇编语言进行编码，作为独立的 `.as` 或 `.asm` 源文件包含到项目中。它们将进行汇编并使用链接器组合到输出镜像中。

默认情况下，汇编器优化器不会对此类模块进行优化。可以通过使用 `--OPT` 选项使能优化，请参见**第 4.8.42 节 “--OPT：调用编译器优化”**。

以下是编写可通过 C 代码调用的汇编程序时必须遵循的准则。

- 为可执行汇编代码选择或定义适合的 psect （关于它们的介绍指南，请参见**第 5.15.1 节 “程序段”**。）
- 使用前导下划线字符选择程序的名称 （标号）
- 确保程序的标号可从其他模块进行全局访问
- 为程序选择相应的 C 等效原型，基于它来确定参数传递的模型
- 将参数和返回值限制为单字节长的对象（汇编程序不能定义驻留在编译堆栈中的变量。对于其他参数使用全局变量。）
- （可选）使用签名值来使能调用函数时的类型检查
- 使用存储区选择指令并对所有变量符号的地址进行掩码处理

以下示例将对于中档器件完成这些步骤。该过程对于其他器件是相同的。对于所有 C 函数和非静态全局变量的名称，将会执行映射。关于 C 标识符和汇编标识符之间映射的完整说明，请参见**第 5.12.3 节 “汇编代码和 C 代码之间的交互”**。

需要一个这样的汇编程序：可以将传递给程序的 8 位量与 PORTB 的内容相加，并将它作为 8 位量返回。

大多数编译器生成的可执行代码都放置在名为 `text`*n* 的 psect 中；其中， n 是一个数字。（见**第 5.15.2 节 “编译器生成的 psect”**。）我们将基于编译器使用的 psect 创建自己的 text psect。检查汇编列表文件，以查看基于 C 代码生成的汇编代码的 text psect 通常为何种形式。针对低档或中档器件进行编译时，可能会看到代码生成器生成如下所示的 psect。

```
PSECT text0,local,class=CODE,delta=2
```


© 2013 Microchip Technology Inc.

关于用于 PSECT 汇编器伪指令的标志的详细信息，请参见**第 6.4.9.3 节“PSECT”**。该 psect 名为 text0。它被标记为 local，这意味着它有别于其他具有相同名称的 psect。该标志在该示例中并不重要，如果需要，可以省略它。它位于 CODE 类中。该标志非常重要，因为意味着它将自动放置在为代码留出的存储区中。由于使用了该标志，所以不需要调整默认的链接器选项就可以将 psect 正确地放置在存储器中。最后一个选项 delta 值也是非常重要的，它指示将放置 psect 的存储空间是可字寻址的 （值为 2）。PIC10/12/16 程序存储空间是可字寻址的；数据空间是可字节寻址的。

对于 PIC18 器件，程序存储器是可字节寻址的，但指令必须按字对齐，所以会看到以下形式的代码。

```
PSECT text0,local,class=CODE,reloc=2
```

在此例中， delta 值为 1 （这是默认设置），但 reloc （对齐）标志设置为 2，以确保段从字对齐地址处开始。

我们只需选择一个不同的名称，所以可以选择名称 mytext 作为放置程序的 psect 的名称，从而对于中档示例得到：

```
PSECT mytext,local,class=CODE,delta=2
```

假定希望该程序在 C 代码域中名为 add。在汇编域中，必须选择名称 _add，因为它之后会映射为 C 标识符 add。如果选择 add 作为汇编程序，则永远无法从 C 代码中调用它。汇编程序的名称是将放置在汇编代码起始处的标号。我们使用的标号将具有以下形式。

```
_add:
```

我们需要能够从其他模块中调用它，因此通过使用 GLOBAL 汇编器伪指令 （**第 6.4.9.1 节“GLOBAL”**）将该标号设为可全局访问。

```
GLOBAL _add
```

通过编译一个与该汇编程序具有类似原型的虚拟 C 函数，我们可以确定签名值。该程序的 C 等效原型将具有以下形式：

```
unsigned char add(unsigned char);
```

检查汇编列表文件，查看此类函数的签名值。对于该步骤，将需要关闭汇编器优化器，因为优化器会从汇编列表文件中删除这些值。签名值并不是必需的，但它使链接器可以进行额外的类型检查。我们确定可以使用以下 SIGNAT 伪指令（**第 6.4.9.21 节“SIGNAT”**）。

```
SIGNAT _add,4217
```

W 寄存器将用于传入参数。关于用于传递参数的约定，请参见**第 5.8.6 节“函数参数”**。

以下给出了中档器件的一个完整程序的示例，它可以放置在汇编文件中并添加到项目中。GLOBAL 和 SIGNAT 伪指令不会生成代码，因而不需要处于 mytext psect 内，虽然如果愿意，您可以将它们放入其中。BANKSEL 伪指令和 BANKMASK 宏用于确保选择正确的存储区，并将所有地址掩码处理为适当的长度。

```
#include <xc.inc>

GLOBAL _add        ; make _add globally accessible
SIGNAT _add,4217   ; tell the linker how it should be called

; everything following will be placed into the mytext psect
PSECT mytext,local,class=CODE,delta=2
; our routine to add to ints and return the result
_add:
   ; W is loaded by the calling function;
   BANKSEL   (PORTB)             ; select the bank of this object
   ADDWF     BANKMASK(PORTB),w   ; add parameter to port
   ; the result is already in the required location (W)so we can
   ; just return immediately
   RETURN
```

要编译该代码，必须对汇编文件进行预处理，因为我们使用了 C 预处理器 #include 伪指令。请参见**第 4.8.11 节 “-P：预处理汇编文件”**。

要从 C 代码中调用某个汇编程序，必须提供该程序的声明。这可以确保编译器知道如何按参数和返回值对函数调用进行编码。

以下给出了先声明汇编程序操作，然后调用程序的 C 代码片断。

```
// declare the assembly routine so it can be correctly called
extern unsigned char add(unsigned char a);

void main(void) {
  volatile unsigned char result;

  result = add(5);  // call the assembly routine
}
```

5.12.2 #asm、#endasm 和 asm()

汇编指令也可以使用伪指令 #asm、#endasm 或语句 asm(); 直接嵌入到 C 代码。

#asm 和 #endasm 伪指令用于开始和结束要嵌入到代码生成器汇编输出中的汇编指令块。#asm 块在句法上不属于 C 程序的一部分，因此它不遵循正常的 C 控制流规则。这意味着不应在诸如 if()、while() 和 for() 语句之类的 C 构造内使用这种形式的行内汇编代码。但是，这是添加多条汇编指令的最简单方法。

asm(); 语句用于将汇编指令嵌入到 C 代码。这种形式的外观和行为都类似于 C 语句。指令放置在类似于函数调用括号的字符串内，虽然并不发生任何调用。通常在该字符串中放入一条指令，但可以通过用 \n 字符分隔指令来指定多条汇编指令（例如，asm(“MOVLW 55\nMOVWF _x”);）。如果在每个语句中放入一条指令，并使用多条语句，代码可读性将更高。

© 2013 Microchip Technology Inc.

您可以在 C 源代码中的任意位置使用 asm() 形式的行内汇编代码，因为它可以正确地与所有 C 控制流结构进行交互。

以下给出了同时使用了两种方法的示例：

```
unsigned int var;

void main(void)
{
    var = 1;
#asm          // like this...
    BCF 0,3
    BANKSEL(_var)
    RLF (_var)&07fh
    RLF (_var+1)&07fh
#endasm
             // do it again the other way...
    asm("BCF 0,3");
    asm("BANKSEL _var");
    asm("RLF (_var)&07fh");
    asm("RLF (_var+1)&07fh");
}
```

汇编器优化器永远不会优化行内汇编代码。

在使用行内汇编代码时，必须非常谨慎，以避免与编译器生成的代码发生交互。代码生成器无法扫描汇编代码的寄存器使用情况，因此如果汇编代码破坏或使用了寄存器，它将不会知道。

第 5.7 节“寄存器使用”说明了编译器使用的寄存器。如果对于周围代码中使用的寄存器存在疑问，可以使用 --ASMLIST 选项（见**第 4.8.16 节“--ADDRQUAL：设置编译器对于存储器限定符的响应”**）编译您的程序，并检查由编译器生成的汇编代码。请记住，当程序的其他部分发生更改时，编译器使用的寄存器和代码策略也会改变。

如果从主干代码和中断代码中调用某个 C 函数，它可能会被复制，请参见**第 5.9.5 节“函数复制”**。虽然使用一个特殊前缀来确保不复制由编译器生成的标号，但这不适用于 C 函数中的手写行内汇编代码中定义的标号。因此，如果包含行内汇编代码的函数可能会被复制，则不应在行内汇编代码中定义汇编标号。

5.12.3 汇编代码和 C 代码之间的交互

MPLAB XC8 C 编译器包含了几项功能，这些功能旨在使 C 代码可以符合用户定义的汇编代码的要求。此外也必须采取一些预防措施，以确保汇编代码不会干扰基于 C 代码生成的汇编代码。

命令行驱动程序可确保所有用户定义的汇编文件先进行处理，然后再开始编译 C 源文件。驱动程序能够读取并分析可重定位目标文件中的某些信息，并将该信息传递给代码生成器。该信息用于确保代码生成器考虑汇编代码的要求。关于编译序列的进一步信息，请参见**第 4.3.4 节 “汇编源文件编译”**。

5.12.3.1 等效汇编符号

大多数 C 符号都会映射到相应的等效汇编符号。

这种映射可以使在汇编域中定义的 “普通”符号无法干扰 C 域中的 “普通”符号。例如，如果在汇编域中定义了 main 符号，它与 C 代码中使用的 main 符号是截然不同的，它们将引用不同的位置。

C 函数的名称映射为汇编标号时，其名称将相同，但前面会带有 *下划线*。所以函数 main() 将定义汇编标号 _main。

低档 PIC 器件可能会对于函数使用备用汇编域符号。在这些器件上，调用指令的目标会被限制为程序存储器页的上半部分。因此，编译器会将函数编码分为两部分，如以下针对低档器件编译的 C 函数示例 add() 所示。

```
entry__add:
    LJMP    _add
```

标号 entry__add 是该函数的入口点，并总是位于链接到程序存储器页上半部分中的特殊 psect 中。与该标号关联的代码只是一个长跳转（见**第 6.4.1.4 节 “长跳转和调用”**），它跳转到位于其他位置并通过标号 _add 标识的实际函数主体。

如果要从汇编代码中调用程序，则必须知道器件中的这一限制，以及编译器对于 C 函数采用的规避措施。手写汇编代码应总是调用 entry__funcName 标号，而不是通常的汇编等效函数标号。

如果 C 函数使用 static 限定，并且程序中有多个使用相同名称的静态函数，第一个函数的名称将映射为正常的汇编符号，后续的函数则映射为以下形式的特殊符号：*fileName@functionName*；其中，*fileName* 是包含函数的文件的名称，*functionName* 是函数的名称。

例如，某个程序包含两个均名为 add 的静态函数的定义。一个位于文件 main.c 中，另一个位于 lcd.c 中。第一个函数将生成汇编标号 _add。第二个函数将生成标号 lcd@add。

非自动 C 变量的名称映射为汇编标号时，其名称将相同，但前面会带有下划线。所以，变量 result 将定义以下汇编标号：_result。

如果 C 变量使用 static 进行限定，则同样，程序中可能会具有多个使用完全相同 C 名称的变量。非局部静态变量适用与静态函数相同的规则。第一个变量的名称将映射为带有一个下划线前缀的符号；后续的符号将具有以下形式：*fileName@variableName*；其中，*fileName* 是包含变量的文件的名称，*variableName* 是变量的名称。

例如，某个程序包含两个均名为 result 的静态变量的定义。一个位于文件 main.c 中，另一个位于 lcd.c 中。第一个变量将生成汇编标号 _result。第二个变量将生成标号 lcd@result。

如果存在多个局部静态变量（即，它在函数定义中定义），则所有变量将具有以下形式的汇编名称：*functionName@variableName*。所以，如果函数 read 中有一个名为 output 的静态变量，函数 update 中定义了另一个同名的静态变量，则在汇编代码中，这些符号可以分别使用符号 read@output 和 update@output 访问。

© 2013 Microchip Technology Inc.

如果有多个静态函数具有相同的名称，并且它们包含同名静态变量的定义，则用于这些变量的汇编符号将具有以下形式：*fileName@functionName@variableName*。

两个静态变量或函数具有相同名称是合法的，但建议不要这么做，因为可能会访问错误的变量或调用错误的函数。

为了使参数和自动变量的访问更简便，将会定义一些特殊的等效形式，它们会将一个惟一符号映射到每个变量。符号具有以下形式：*functionName@variableName*。因此，如果函数 main 定义了名为 foobar 的自动变量，则在汇编代码中可以使用符号 main@foobar 来访问该 C 变量。

函数参数使用与自动变量相同的符号映射方式。如果名为 read 的函数具有名为 channel 的参数，则该参数的汇编符号为 read@channel。

函数返回值没有与它们相关联的 C 标识符。函数的返回值与函数的参数变量（如果它们存在）共用相同的存储器。用于返回值的汇编符号的形式为 ?_*funcName*；其中，*funcName* 是返回值的函数的名称。因此，如果函数 getPort 返回一个值，它将位于汇编符号 ?_getPort 包含的地址中。如果该返回值的长度为多个字节，则通过向该符号加上一个偏移量来访问每个字节，例如 ?_getPort+1。

如果编译器创建了临时变量来保存中间结果，这些变量的行为将类似于自动变量。由于没有相应的 C 变量，汇编符号将基于代表函数自动块的符号加上一个偏移量。该符号为 ??_*funcName*；其中，*funcName* 是使用了该符号的函数。例如，如果函数 main 使用了临时变量，它们可以通过相对于符号 ??_main 的偏移量的形式进行访问。

5.12.3.2 从汇编代码中访问寄存器

如果编写独立的汇编模块，SFR 定义将不会自动可访问。可以通过使用汇编头文件 <xc.inc> 来访问这些寄存器的定义。对于嵌入到 C 代码中的汇编代码，请不要使用该文件，因为它会与 <xc.h> 中的定义发生冲突。

使用汇编器的 INCLUDE 伪指令（见**第 6.4.10.4 节 “INCLUDE”**）或使用 C 预处理器的 #include 伪指令来包含该文件。如果使用后一种方法，请确保使用驱动程序选项 -P 进行编译，以便对汇编文件进行预处理，请参见**第 4.8.11 节 “-P: 预处理汇编文件”**。

该头文件中的符号类似于在包含 <xc.h> 时在 C 域中使用的标识符，例如 PORTA 和 EECON1 等。它们是不同域中的不同符号，但会映射到相同的存储单元。

寄存器内的位被定义为 *registerName,bitNumber*。例如，RA0 被定义为 PORTA,0。

以下给出了一个使用 SFR 的中档汇编模块的示例。

```
#include <xc.inc>
GLOBAL _setports

PSECT text,class=CODE,local,delta=2
_setports:
    MOVLW     0xAA
    BANKSEL   (PORTA)
    MOVWF     BANKMASK(PORTA)
    BANKSEL   (PORTB)
    BSF       RB1
```

如果要从嵌入到C代码的汇编代码中访问寄存器定义，可以使用另一个可用于该用途的头文件。将头文件 <caspic.h> 包含到汇编代码中。该文件只能用于低档和中档器件。PIC18 器件没有类似文件。

用于寄存器名称的符号将与由<xc.inc>定义的相同；但是，为寄存器内的位变量指定的名称将包含后缀 _bit。例如，前面给出的示例可以重写为如下形式的行内汇编代码。

```
#asm
    MOVLW     0xAA
    BANKSEL   (PORTA)
    MOVWF     BANKMASK(PORTA)
    BANKSEL   (PORTB)
    BSF       RB1_bit
#endasm
```

必须小心确保您不会破坏存放计算中间值的寄存器的内容。一些寄存器由编译器使用，直接写入这些寄存器可导致代码失败。当 SFR 由于汇编代码写入它们而发生改变时，代码生成器并不会检测到发生了这种情况。**第 5.7 节 “寄存器使用”**列出了编译器使用的寄存器，并提供了进一步的信息。

5.12.3.3 绝对 PSECT

一些从汇编代码初始编译中提取的信息（见**第 4.3.4 节 “汇编源文件编译”**）会涉及到绝对 psect，特别是使用 abs 和 ovrld PSECT 标志定义的 psect；关于该伪指令的信息，请参见**第 6.4.9.3 节 “PSECT”**。

MPLAB XC8 能够确定绝对 psect 的地址范围，并使用该信息来确保代码生成器基于 C 源代码生成的代码不使用汇编代码所需的存储器。在编译 C 源代码之前，代码生成器会保留汇编代码所使用的任何存储器。

以下给出了一个说明它如何工作的示例。汇编代码文件定义了一个必须定位到数据空间中地址 0x110 处的表。汇编文件包含：

```
PSECT lkuptbl,class=RAM,space=1,abs,ovlrd
ORG 110h
lookup:
    DS 20h
```

绝对 psect 总是从地址 0 开始。对于这种 psect，可以通过使用 ORG 伪指令来指定一个非零起始地址。关于该伪指令的重要信息，请参见**第 6.4.9.4 节 “ORG”**。

在编译项目时，将对该文件进行汇编，并在所产生的可重定位目标文件中扫描绝对 psect。由于该 psect 被标记为 abs 和 ovlrd 类型，所以会记录该 psect 的边界和空间——在此例中，存储空间 1 中从地址 0x110 至 0x12F 的存储器范围会被标注为已使用。该信息传递给代码生成器，以确保基于 C 代码生成的汇编代码不使用该地址范围。

链接器会处理程序存储器的所有分配，所以对于手写的汇编代码，只有位于数据存储器中的 psect 需要以这种方式定义。

© 2013 Microchip Technology Inc.

5.12.3.4 未定义的符号

如果需要同时可从汇编代码和 C 源代码中访问某个变量，在需要的情况下可以在汇编代码中定义它，但在 C 源代码中这样做会更容易。

如果在 C 源代码中定义了某个变量，但仅在汇编代码中引用它，可能会发生问题。在这种情况下，代码生成器会认为该变量未被使用并删除它。链接器将无法解析汇编代码引用的符号，会导致错误。

为了解决该问题，MPLAB XC8还会搜索汇编代码生成的目标文件，查找未定义的符号。请参见**第 4.3.4 节 “汇编源文件编译”**。这些将为汇编代码中已使用但未定义的符号。这些符号的信息将会传递给代码生成器，并且如果在 C 代码中遇到它们，变量会被自动标记为 volatile 类型。该操作与在源代码中使用 `volatile` 限定变量具有相同的效果，请参见**第 5.4.7.2 节 “volatile 类型限定符”**。

代码生成器永远不会删除限定为 `volatile` 类型的变量，即使它们在整个程序似乎未用。

例如，如果 C 程序以如下形式定义一个全局变量：

```
int input;
```

但只有汇编代码中使用了该变量。汇编模块可以使用 GLOBAL 汇编器伪指令简单地声明该符号，然后使用它。以下 PIC18 示例给出了访问该变量的汇编代码。

```
GLOBAL _input, _raster
PSECT text,local,class=CODE,reloc=2
_raster:
    MOVF   _input,w
```

编译器知道 C 符号 `input` 和相应汇编符号 `_input` 之间的映射 （见**第 5.12.3 节 “汇编代码和 C 代码之间的交互”**）。在该实例中，C 变量 `input` 不会被删除，将被视为如同使用 `volatile` 限定。

5.13 优化

MPLAB XC8 编译器中的优化可以大致分为 C 代码级优化 （在源代码转换为汇编代码之前对源代码执行）和汇编级优化 （对由编译器生成的汇编代码执行）。

C 代码级优化在代码生成阶段的早期执行，所以具有流程上的好处：执行一项优化可能意味着可以之后应用另一项。

由于这些优化是在生成调试信息之前应用的，所以对程序的源代码级调试通常几乎没有影响。

其中一些优化是代码生成过程不可或缺的组成部分，因此无法通过选项禁止。以下几节给出了关于如何废除一些具体优化的建议。

在标准模式下，特别是在免费模式下，其中一些优化会被禁止。即使使能了它们，也只有在满足非常特定的条件时才会应用它们。因此，您可能会发现一些代码行进行了优化，但有些则没有。

以下列出了可简化或更改 C 表达式的主要 C 代码级优化。

未用变量：程序中的未用变量会被删除。对于被删除的变量，将不会为它们保留存储器，它们不会出现在任何列表或映射文件中，也不会在调试信息中出现，因此无法在调试器中观察到。如果遇到未用变量，则会产生一条警告。
使用 `volatile` 限定的对象永远不会被删除，请参见**第 5.4.7.2 节“volatile 类型限定符”** 。获取变量的地址或在手写汇编代码中引用它的汇编域符号也构成对变量的使用。

冗余赋值：后续未被使用的冗余赋值会被删除，除非变量为 `volatile` 类型。赋值语句将被完全删除，就如同原始源代码中从未出现它。不会为它生成任何代码，您也无法在调试器中在该行上设置断点。

未用函数：程序中的未用函数会被删除。如果某个函数未被直接或间接地调用，也没有任何代码获取其地址，则认为它是一个未用函数。整个函数将被删除，就如同原始源代码中从未出现它。不会为它生成任何代码，您也无法在调试器中在该函数中的行上设置断点。
在独立的手写汇编模块中引用函数的汇编域符号可以防止它被删除。汇编代码只需在 `GLOBAL` 伪指令中使用该符号即可。

未用返回表达式：函数中的未用返回表达式会被删除。如果对某个函数的所有调用的结果都丢弃返回值，则返回值会被视为未用。与返回值计算关联的代码将被删除，函数将被编码为如同它的返回类型为 `void`。

常量传播：在可以确定变量的数值内容时，将执行常量传播。不属于 volatile 类型，可以准确确定其值的变量会被替换为数字值。对于未初始化的全局变量，将假定它们在进行任何赋值之前包含零。

© 2013 Microchip Technology Inc.

在读取之前赋值的变量：运行时启动代码不会清零或初始化在读取之前赋值的变量。该优化将仅考虑非自动变量，如果它们在其他代码可以读其值之前赋值，则它们会被视为 `persistent` 类型，请参见**第 5.4.8.1 节 “persistent 类型限定符”**。运行时启动代码不会清零所有 `persistent` 类型的对象，所以该优化可以加快程序启动时的执行速度。

仅带有目标的解引用指针可以被替换为直接访问目标对象。这适用于数据和函数指针。

由于 C 代码级优化在生成调试信息之前执行，所以它们对调试信息的影响通常较小。但是，如果某个变量被定位到寄存器中， IDE （如 MPLAB IDE）可能会在它们的 Watch（观察）视图中指示错误的值。这是由于用于向 IDE 传递调试信息的文件格式（当前为 COFF）存在限制而导致的。请检查汇编列表文件，查看所调试的程序中是否使用了寄存器。

第 6.5 节 “汇编代码级优化”介绍了汇编级优化。

5.14 预处理

在编译之前，需要对所有 C 源文件进行预处理。编译之后不会删除预处理的文件。它将具有 .pre 扩展名，并且与派生它的源文件具有相同的基本名称。

--PRE 选项可用于执行预处理，然后停止编译。请参见**第 4.8.46 节 “--PRE：生成预处理的源代码”**。

如果发出驱动程序选项 -P，则也可以对汇编器文件进行预处理。请参见**第 4.8.11 节 “-P：预处理汇编文件”**。

5.14.1 C 语言注释

MPLAB XC8 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

5.14.2 预处理器伪指令

除了标准伪指令之外，MPLAB XC8 还可接受几条专用的预处理器伪指令。下一页中的表 5-11 列出了所有这些伪指令。

带参数的宏扩展可以使用 # 字符将参数转换为一个字符串，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。另外请记住，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。

© 2013 Microchip Technology Inc.

表 5-11： 预处理器伪指令

伪指令	含义	示例
#	预处理器空伪指令，不执行任何操作	#
#assert	如果条件为 false，则产生错误	#assert SIZE > 10
#asm	指示行内汇编代码开始	#asm MOVLW FFh #endasm
#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
#endasm	终止行内汇编代码	请参见 #asm
#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
#*nn*	（其中， *nn* 是一个数字）#line *nn* 的缩写	#20
#pragma	特定于编译器的选项	请参见**第 5.14.4 节“pragma 伪指令”**
#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

预处理器域中的数值的类型和转换与 C 域中一样。预处理器值不具有类型，但会在预处理器对它们进行转换之后立刻获得类型。表达式可能会使其分配的类型发生溢出，与 C 表达式可能发生溢出的方式相同。

通过使用常量后缀，可以避免溢出。例如，在数字之后使用 L 表示在转换之后应将它解释为长整型。

例如：

```
#define MAX 1000*1000
```

和

```
#define MAX 1000*1000L
```

将分别定义值 0x4240 和 0xF4240。

5.14.3 预定义的宏

编译器驱动程序会为预处理器定义一些特定的符号，从而允许基于芯片类型等条件进行条件编译。表 5-12 列出了驱动程序定义的较常用符号。

表 5-12： 预定义的宏

符号	何时设置	用途
__*CHIPNAME*	选定芯片时	指示已选定的特定芯片类型，例如 __16F877。
__DATABANK	如果实现了 EEPROM 或闪存	标识 EEDATA/PMDATA 寄存器位于哪个存储区。
__DATE__	总是	指示当前日期，例如 May 21 2004。
__FILE__	总是	指示正在预处理的当前源文件。
__FLASHTYPE	如果实现了闪存	指示目标器件采用的闪存类型，请参见下面的 –PROGMEM。
__LINE__	总是	指示当前源代码行号。
__J_PART	如果为 PIC18 器件	指示器件为 “J” 系列器件。
__MPLAB_ICD*X*__	如果针对 MPLAB® ICD 或 MPLAB ICD 2/3 调试器进行编译	（其中， *X* 为空、 2 或 3。）赋值为 1，以指示代码是为用于 Microchip MPLAB ICD、 ICD 2 或 ICD 3 而生成的。
__MPLAB_PICKIT*X*__	如果针对 MPLAB® PICkit™ 2/3 进行编译	赋值为 1，以指示代码是为用于 Microchip MPLAB PICKIT 2 或 PICKIT 3 而生成的。
__MPLAB_REALICE__	如果针对 MPLAB® REAL ICE 进行编译	赋值为 1，以指示代码是为用于 Microchip MPLAB REAL ICE 而生成的。
__RESETBITS_ADDR	如果 -RUNTIME 选项要求保存 STATUS 寄存器	指示用于保存 STATUS 寄存器内容的地址。
__TIME__	总是	指示当前时间，例如 08:06:31。
__XC	总是	指示正在使用 Microchip 的 MPLAB XC 编译器。
__XC8	总是	指示正在使用用于 Microchip 8 位器件的 MPLAB XC 编译器。
__XC8_VERSION	总是	指示编译器的版本号乘以 1000，例如， v1.00 将表示为 1000。
_*CHIPNAME*	选定芯片时	指示已选定的特定芯片类型，例如 –16F877。
BANKBITS	总是	赋值为 0、 1 或 2，以指示 1、 2 或 4 个可用的存储区或 RAM。
BANKCOUNT	总是	指示实现的数据存储区的数量。
COMMON	如果存在公共 RAM	指示器件是否具有公共 RAM 区。

© 2013 Microchip Technology Inc.

表 5-12： 预定义的宏 （续）

符号	何时设置	用途
_EEPROMSIZE	总是	指示有多少个字节的 EEPROM 可用。
_ERRATA_TYPES	如果为 PIC18 器件	指示所应用的勘误规避措施，请参见 --ERRATA 选项章节 4.8.27。
_*FAMILY*_FAMILY_	如果为 PIC18 器件	指示 PIC18 系列。
_FLASH_ERASE_SIZE	总是	闪存擦除块的长度。
_FLASH_WRITE_SIZE	总是	闪存写入块的长度。
GPRBITS	总是	赋值为 0、 1 或 2，以指示 1、 2 或 4 个可用存储区或通用 RAM。
_HAS_OSCVAL_	如果目标器件具有振荡器校准寄存器	指示目标器件可能需要振荡器校准。
_HTC_EDITION_	总是	指示正在使用专业、标准还是免费编译器。将分别赋值为 2、1 或 0。
_HTC_VER_MAJOR_ _HTC_VER_MINOR_	总是	分别指示编译器版本号的整数和小数部分。
_HTC_VER_PATCH_ _HTC_VER_PLVL_	总是	指示编译器版本号的补丁级别。
MPC	总是	指示针对 Microchip PIC® MCU 系列进行编译。
_OMNI_CODE_	总是	指示正在使用 OCG 编译器进行编译。
_PIC12	如果为低档 （12 位指令）器件	指示选定器件是低档 PIC 器件。
_PIC12E	如果为增强型低档 （12 位指令）器件	指示选定器件是增强型低档 PIC 器件。
_PIC14	如果为中档 （14 位指令）器件	指示选定器件是中档 PIC 器件。
_PIC14E	如果为增强型中档 （14 位指令）器件	指示选定器件是增强型中档 PIC 器件。
_PIC18	如果为 PIC18 （16 位指令）器件	指示选定器件是 PIC18 器件。
PROGMEM	如果针对带闪存的中档器件进行编译	指示目标器件采用的闪存类型。值为 0xFF （未知）、 0xF0 （无）、0（只读）、1（带自动擦除的字写）、 2 （带自动擦除的块写）和 3 （带手动擦除的块写）。
_RAMSIZE	如果为 PIC18 器件	指示有多少个字节的数据存储器可用。
_ROMSIZE	总是	指示有多少个字的程序存储器可用。
EEPROMSIZE	总是	指示有多少个字节的 EEPROM 可用。
ERRATA_4000_BOUNDARY	如果应用了 ERRATA_4000	指示应用了 4000 字边界勘误。

表 5-12： 预定义的宏 （续）

符号	何时设置	用途
HI_TECH_C	总是	指示 C 语言种类与 HI-TECH C® 兼容。
MPLAB_ICD	如果针对 MPLAB® ICD 或 MPLAB ICD 2/3 调试器进行编译	赋值为 1 时，指示生成的代码用于 Microchip MPLAB ICD。赋值为 2 时，指示用于 MPLAB ICD 2； 3 则指示用于 MPLAB ICD 3。

除非另有说明，否则每个符号 （如定义）都等于 1。

5.14.4 pragma 伪指令

有一些特定的编译时伪指令可以用于修改编译器的行为。它们通过使用 ANSI 标准 #pragma 功能来实现。 pragma 伪指令的格式为：

```
#pragma keyword options
```

其中， *keyword* 是一组关键字之一，其中一些后面会跟随某些特定选项 *options*。表 5-13 列出了这些关键字。以下将详细介绍其他地方未介绍的那些关键字。

表 5-13： PRAGMA 伪指令

伪指令	含义	示例
addrqual	指定限定符的操作	#pragma addrqual=require
config	指定配置位	#pragma config WDT=ON
inline	内联函数 （如可能）	#pragma inline(getPort)
intrinsic	指定对函数进行内联	#pragma intrinsic(_delay)
interrupt_level	允许从中断和主干代码中进行调用	#pragma interrupt_level 1
pack	指定结构压缩	#pragma pack 1
printf_check	使能 printf 样式的格式字符串检查	#pragma printf_check(printf) const
psect	重命名编译器生成的 psect	#pragma psect nvBANK0=my_nvram
regsused	指定由函数使用的寄存器	#pragma regsused myFunc wreg,fsr
switch	指定 switch 语句的代码生成	#pragma switch direct
warning	控制消息参数	#pragma warning disable 299,407

© 2013 Microchip Technology Inc.

5.14.4.1 #PRAGMA ADDRQUAL 伪指令

该伪指令用于控制编译器对非标准存储器限定符的响应。该 pragma 伪指令是 --ADDRQUAL 选项在代码内的等效伪指令，它们使用相同的参数，请参见**第 4.8.16 节 “--ADDRQUAL：设置编译器对于存储器限定符的响应”**。

该pragma伪指令在整个C程序中有效，应发出一次（如需要）。如果多次发出该pragma伪指令，则最后一条 pragma 伪指令将决定编译器的响应。

例如：

```
#pragma addrqual=require
bank2 int foobar;
```

5.14.4.2 #PRAGMA CONFIG 伪指令

该伪指令用于指定 PIC18 目标器件的器件配置位。关于完整的详细信息，请参见**第 5.3.5 节 “配置位访问”**。

5.14.4.3 #PRAGMA INLINE 伪指令

`#pragma inline` 伪指令用于指示编译器，对指定函数的调用应尽可能快。该 pragma 伪指令与使用 `inline` 函数说明符具有相同的作用。

5.14.4.4 #PRAGMA INTRINSIC 伪指令

`#pragma intrinsic` 伪指令用于指示编译器，某个函数需要由编译器进行固有内联。该伪指令只能用于代码生成器可以在内部进行扩展的特殊函数，例如 `_delay` 函数。此类函数没有对应的源代码，由编译器进行特别处理。

> **注：** 对用户定义的函数使用该 pragma 伪指令并*不*意味着函数会被内联，并会产生错误。关于该操作，请参见**第 5.8.1.2 节 “inline 说明符”**中的 `inline` 函数说明符。

5.14.4.5 #PRAGMA INTERRUPT_LEVEL 伪指令

`#pragma interrupt_level` 伪指令可以用于阻止对从主干代码和中断代码中调用的函数进行函数复制。更多信息，请参见**第 5.9.5.1 节 “禁止复制”**。

5.14.4.6 #PRAGMA PACK 伪指令

所有 8 位 PIC 器件都只能对存储器执行字节访问，所以不需要对结构内的存储器对象进行任何对齐。该 pragma 伪指令在使用时不起任何作用。

5.14.4.7 #PRAGMA PRINTF_CHECK 伪指令

一些库函数可接受printf()形式的格式字符串跟随可变数量的参数。虽然格式字符串在运行时进行解释，但可以在编译时检查它与剩余的参数是否一致。

该伪指令用于对指定函数使能此项检查；例如，系统头文件 <stdio.h> 包含了以下伪指令：

```
#pragma printf_check(printf) const
```

以对于 printf() 使能此项检查。您也可以对可接受 printf 样式的格式字符串的任何用户定义的函数使用它。

函数名之后的限定符用于自动转换可变参数列表中的指针。以上示例会将 RAM 中所有指向字符串的指针强制转换为 （const char *）类型的指针。

请注意，只有警告级别设置为 -1 或更低时，该选项才会具有任何可见的效果。请参见**第 4.8.59 节 “--WARN：设置警告级别”**。

5.14.4.8 #PRAGMA PSECT 伪指令

通常，由编译器生成的目标代码会被拆分为多个标准 psect。**第 5.15.2 节“编译器生成的 psect”**对它进行了介绍。关于 psect 的介绍指南，请参见**第 5.15.1 节 “程序段”**。这对于大多数应用程序都是可以接受的，但在需要特殊的存储器配置时，有时会需要将变量或代码重定向到不同 psect。

包含代码和数据的一些编译器生成 psect 可以使用 #pragma psect 伪指令进行重定向。该 pragma 伪指令的一般格式如下：

```
#pragma psect standardPsect=newPsect
```

并指示代码生成器，正常情况下出现在标准 psect *standardPsect* 中的所有内容，现在将出现在名为 *newPsect* 的新 psect 中。该 psect 在标志和属性方面将会与 *standardPsect* 完全相同；但是，它将具有一个惟一名称。因此，您可以明确地定位该新 psect，而不会影响原始 psect 中所有内容的放置。

如果被重定向的标准 psect 的名称中包含一个计数器（例如，text0、text1 和 text2 等），则应在 psect 名称中的计数器位置使用占位符 %%u，例如 text%%u。它将匹配所有 psect，无论计数器值如何。例如，要重映射 C 函数，可以使用：

```
#pragma psect text%%u=lookupfunc
int lookup(char ind)
{
  ...
```

引用存储区编号的标准 psect 不使用计数器，不需要使用占位符来进行匹配。例如，要重定向来自存储区 1 中的未初始化变量，可以使用：

```
#pragma psect bssBANK1=sharedObj
int foobar;
```

该 pragma 伪指令不应用于存放已初始化变量的任何数据 psect（data 或 idata）。这些 psect 必须按特定顺序进行组装，而使用该 pragma 伪指令来重定向其内容会破坏这种顺序。对要链接到特定 RAM 存储区的基于 RAM 的 psect 使用该 pragma 伪指令是可以接受的，但新 psect 必须链接到同一个存储区。将新 psect 链接到一个不同存储区可能会导致代码失败。

© 2013 Microchip Technology Inc.

该 pragma 伪指令会影响它所在的整个模块，无论该 pragma 伪指令在文件中的位置如何。任何给定 psect 都应只在特定模块中重定向一次。即，不能对于模块的某些部分重定向标准 psect，然后切换回对于源代码的其余部分使用标准 psect。该 pragma 伪指令通常应放置在源文件的顶部。建议将要分离的代码或变量全部放置在它们自己的源文件中，以便很容易地区分它们。

要确定函数或对象通常定位到的 psect，可以在不使用该 pragma 伪指令的情况下以通常方式定义函数或对象。现在，检查汇编列表文件（见**第 6.5 节 “汇编代码级优化”**）来确定函数或对象通常定位到的 psect。请注意，对象和函数的位置可能会随所选的目标器件和编译器选项而不同。

检查在使用该 pragma 伪指令情况下的汇编列表文件或映射文件，以确保映射按预期的那样工作，并且函数或变量已被链接到指定地址处。

假设存在以下包含几个函数的原始程序。

```
#include <xc.h>

void init(void)
{
}

int getInput(void)
{
   return PORTA;
}

int masher(int val)
{
   return val;
}

void main(void)
{
   int input;

   init();
   while(1) {
      input = getInput();
      masher(input);
   }
}
```

编程人员确定需要将函数`masher()`链接到某个特定地址处，但他们不希望将该函数设为绝对函数。编程人员先编译他们的代码，然后通过检查汇编列表文件或映射文件来确定 masher 正常情况下所在的 psect。例如，映射文件会显示与该函数关联的符号，以及它所在的 psect。

```
                                    Symbol Table

_getInput             text13        07E5
_init                 text12        07E4
_main                 maintext      07ED
_masher               text14        07E8
...
```

所以它被分配到 text*n* psect。从该源文件中移除 masher() 函数的源代码，并放置在它自己的文件中。然后应用 psect pragma 伪指令，从而使这个新文件包含以下内容。

```
#pragma psect text%%u=myMasherPsect
int masher(int val)
{
   return val;
}
```

重新编译之后，该函数现在将定位到它自己的 psect 中。更新后的映射文件现在会显示以下内容。可以注意到，它的地址未发生改变。

```
                                    Symbol Table

_getInput             text13        07E5
_init                 text12        07E4
_main                 maintext      07ED
_masher               myMasherPsect 07E8
...
```

编程人员现在使用驱动程序选项 -L-pmyMasherPsect=1000h 将该 psect 定位到所需的位置。重新编译之后，映射文件现在会显示以下内容。可以注意到，其他函数的地址并未受该示例中的 pragma 伪指令影响。

```
                                    Symbol Table

_getInput             text13        07E5
_init                 text12        07E4
_main                 maintext      07ED
_masher               myMasherPsect 1000
...
```

变量还可以通过设为绝对变量放置在特定位置处，请参见**第 5.5.4 节 “绝对变量”**。对于函数也是如此。请参见**第 5.8.4 节 “更改默认的函数分配”**。应基于位置要求来决定是使用绝对对象还是通过 psect pragma 伪指令来定位函数或变量。

使用绝对函数和变量是最简单的方法，但这种方法仅允许放置在必须在编译前已知的地址处。psect pragma 伪指令较复杂，但可以为链接器提供将新 psect 定位到存储器中的所有灵活性。例如，可以指定函数或变量位于某个固定地址处，或将它们放置在其他 psect 之后，或将 psect 放置在编译器定义的或用户定义的地址范围中的任意位置。关于链接时可用的功能和选项，请参见**第 7 章“链接器”**。关于从驱动程序或 MPLAB IDE 中控制链接器的信息，另请参见**第 4.8.7 节 “-L-：直接调整链接器选项”**。

© 2013 Microchip Technology Inc.

5.14.4.9 #PRAGMA REGSUSED 伪指令

`#pragma regsused` 伪指令使编程人员可以向代码生成器指示它无法“看到”的函数的寄存器使用情况，举例来说，如果它们以汇编代码进行编写。对在 C 代码中定义的函数使用它时，它没有任何作用，但在这些情况下编译器可以准确地确定这些函数的寄存器使用情况，可以不需要该 pragma 伪指令。

编译器只会确定需要对于所定义的中断函数保存的那些寄存器和对象，使用该 pragma 伪指令使代码生成器还可以确定以汇编代码编写的程序的寄存器使用情况。

该 pragma 伪指令的一般格式为：

```
#pragma regsused routineName registerList
```

其中，*routineName*是定义了寄存器使用情况的函数或程序的C等效名称，*registerList* 是寄存器名称的空格分隔列表，如表 5-9 所示。

未列出的那些寄存器将被假定为未被函数或程序使用。代码生成器可能会在经历函数调用时使用其中任意寄存器来存放一些值。因此，如果程序实际上使用了这些寄存器，则可能产生不可靠的程序执行。

寄存器名称不区分大小写，如果无法识别寄存器名称，将产生一条警告。一个空列表表示指定的函数或程序不使用任何寄存器。如果不使用该 pragma 伪指令，编译器将假定外部函数使用所有寄存器。

例如，一个名为`_search`的程序是以PIC18汇编代码编写的。在C源代码中，可以编写：

```
extern void search(void);
#pragma regsused search wreg status fsr0
```

指示该程序使用了 W 寄存器、 STATUS 和 FSR0。此处， FSR0 将扩展为 FSR0L 和 FSR0H。如果需要，可以单独列出它们。

5.14.4.10 #PRAGMA SWITCH 伪指令

通常情况下，编译器会选择如何对 `switch` 语句进行编码，以产生可能的最小代码长度。 `#pragma switch` 伪指令可以用于强制编译器使用不同的编码策略。

switch pragma 伪指令的一般格式为：

```
#pragma switch switchType
```

其中，*switch_type* 是表 5-14 中列出的可用 switch 类型之一（对于 PIC18 器件，当前惟一实现的 switch 类型为 `space`）。

表 5-14： SWITCH 类型

switch 类型	说明
`speed`	使用最快的 switch 方法
`space`	使用代码长度最小的方法
`time`	使用固定延时的 switch 方法
`auto`	使用代码长度最小的方法 （默认）
`direct` （已弃用）	使用固定延时的 switch 方法
`simple` （已弃用）	顺序异或运算方法

对 #pragma switch 伪指令指定 time 选项会强制编译器生成查找表式的 switch 方法。执行每个 case 分支所需的时间是相同的，所以它在需要考虑时序的情况下会很有用，例如状态机。

该 pragma 伪指令会影响所有后续的代码。

auto 选项可用于恢复为默认的行为。

在汇编列表文件中会打印每个 switch 语句的信息，显示所选择的策略，请参见**第 6.6.4 节 “switch 语句信息”**。

5.14.4.11 #PRAGMA WARNING 伪指令

该 pragma 伪指令用于控制编译器的一些消息，例如警告和错误。关于编译器采用的消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

5.14.4.11.1 warning disable pragma 伪指令

通过使用 warning disable pragma 伪指令，可以禁止一些警告消息。

该 pragma 伪指令只会影响由解析器或代码生成器产生的警告；即，直接与 C 代码关联的错误。该 pragma 伪指令的位置仅对于解析器有意义；即，可以对于一个代码段禁止某个解析器警告编号，从而针对特定的警告实例。代码生成器生成的特定警告实例无法单独进行控制，该 pragma 伪指令将在整个模块的编译过程中保持有效。

已被禁止的那些警告的状态可以使用 warning push 和 warning pop pragma 伪指令进行保存和恢复。入栈和出栈可以进行嵌套，从而允许在很大程度上控制消息的行为。

以下示例说明了将某个 const 对象的地址赋予一个指向非 const 对象的指针时出现的警告。此类代码通常会产生警告编号 359。

```
int readp(int * ip) {
    return *ip;
}

const int i = 'd';

void main(void) {
    unsigned char c;
#pragma warning disable 359
    readp(&i);
#pragma warning enable 359
}
```

使用以下代码时，将观察到与它相同的作用。

```
#pragma warning push
#pragma warning disable 359
    readp(&i);
#pragma warning pop
```

此处，消息系统的状态通过 warning push pragma 伪指令进行保存。警告 359 被禁止，然后在触发该警告的源代码之后，通过使用 warning pop pragma 伪指令来恢复消息系统的状态。

© 2013 Microchip Technology Inc.

5.14.4.11.2 warning error/warning pragma 伪指令

此外，还可以更改某些消息的类型。

它只能通过使用 `warning pragma` 实现，并且它仅影响由解析器或代码生成器产生的消息。该 pragma 伪指令的位置仅对于解析器有意义；即，可以对于一个代码段更改某个解析器消息编号的类型，从而针对特定的消息实例。代码生成器生成的特定消息实例无法单独进行控制，该 pragma 伪指令将在整个模块的编译过程中保持有效。

以下示例显示了在前面示例中产生的警告对于函数 `main()` 中的实例被转换为错误。

```
void main(void) {
    unsigned char c;
#pragma warning error 359
    readp(&i);
}
```

编译该代码会产生错误，而不是通常情况下的警告。该错误将强制编译在当前模块结束之后停止，或如果已达到最大错误数则立即停止。

5.15 链接程序

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 psect 放置在存储器中。不使用任何链接描述文件。

用户可以调整传递给链接器的链接器选项，但只有在一些特殊情况下才需要这么做。更多信息，请参见**第 4.8.7 节 “-L-：直接调整链接器选项”**。

链接器会创建一个映射文件，其中将详细说明分配给各个 psect 和代码中一些对象的存储器。映射文件是查找存储器信息的最佳场所。关于该文件中详细信息的详细说明，请参见**第 7.4 节 “映射文件”**。

5.15.1 程序段

有些人对于 psect（程序段）究竟是什么感到非常困惑，而对于它们如何放置到存储器中则感到更加困惑。以下将通过类比和示例的形式来介绍编译器必须如何生成代码，以及如何将它分配到存储器中。理解它们对于汇编编程人员来说至关重要，并有助于理解由链接器发出的“无法找到空间”错误消息。与所有类比一样，它可能会产生误导，并且也只能到此为止，但它可以让代码生成、链接器和段的主要准则与人们普遍了解的知识产生关联。

阅读完本节之后，您将可以更好地理解：

- 为什么汇编代码要以段的形式进行填充和操作
- 为什么链接器将段填充到类中，而不是器件存储器中
- 为什么即使器件存储器还有足够的空间，还是会发出 “无法找到空间”错误消息

5.15.1.1 类比

我们的类比基于一家销售元器件的公司。全球各地的客户订购这些元器件。货物通过装运容器从中央仓库发送到一个区域办事处，然后再递送给客户。

在仓库中，通过机器人汇集每个订单中的元器件。机器人收集所需的元器件并放入一个大小足够的盒子中。盒子被贴上标签，然后放在传送带上。盒子上的标签指示目标城市和国家 / 地区，以及任何特殊的包装说明。

在传送带的另一端，包装机将读取盒子上的标签，并按目标城市对它们进行分类。因此，目标为同一个城市的所有盒子会被整理到一个储存仓中。

处理完一天的所有订单之后，每个储存仓中整理好的盒子先用塑料包装在一起。然后将这些包裹放入发往相应国家 / 地区的装运容器。由于同一个国家 / 地区可能会有多个目标城市，所以同一个装运容器中可能放入许多包裹。

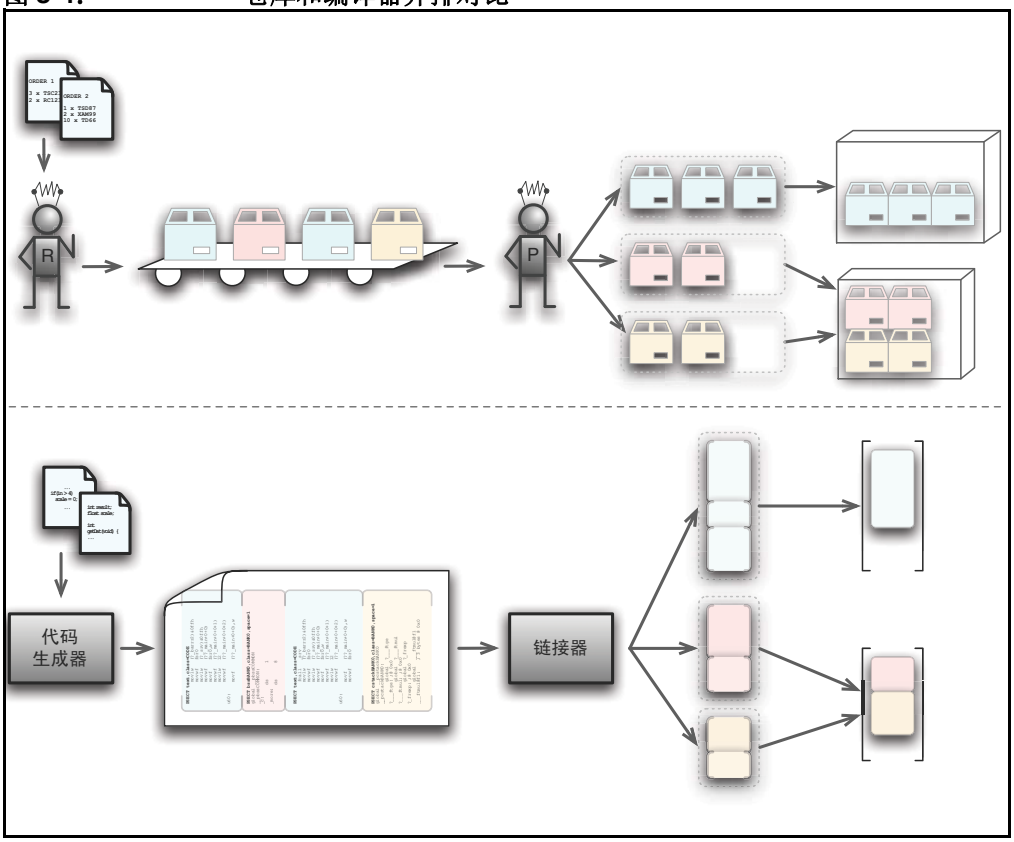
这就结束了仓库中富有收获的一天。

© 2013 Microchip Technology Inc.

5.15.1.2 编译器等效形式

现在，我们看一下这种情形与编译过程的异同。图 5-4 对这两个过程进行了对比。

图 5-4： 仓库和编译器并排对比



在这个类比中，将仓库比作编译器。机器人就像是编译器的代码生成器应用程序，后者将 C 代码转换为汇编代码；包装机就像是链接器，后者负责安排所有内容在存储器中的排列。

包装机实际上并不会递送由盒子组成的包裹；它仅仅是将它们放入仓库中的装运容器中。与之类似，链接器实际上并不会将段放入器件的存储器；它只是将它们安排在概念容器内。这个类比并不关心在仓库外会发生什么，也不关心编译过程之后的处理——那是另一个故事。

以下几节详细介绍了该过程不同阶段的要点。

5.15.1.2.1 订单和源代码

仓库和编译器都需要获取某种事务的说明，并使用该信息来生成最终的产品：仓库接收订单，并通过汇集元器件来履行订单；编译器读取 C 程序中的 （变量和函数）定义，并生成汇编代码来实现每个定义。

一个客户的订单可能对应任意数量的元器件。与之类似，每个 C 定义可能需要生成或多或少的汇编指令或伪指令。

订单随机到达仓库，但元器件总是运送到几个国家 / 地区之一。源代码 （变量和函数）也可以按任意顺序编写，但经过编译的代码总是分配到几个存储区之一。

5.15.1.2.2 盒子、标签 / 标号和段

在我们的类比中，每个订单的元器件都放入一个盒子中。与之类似，生成的汇编输出也会被放入称为程序段（或 psect， program section 的缩写）的盒子中。

将代码放置在段中有几个原因。

- 生成的汇编代码更易于管理。
- 段内的代码序列顺序将被保留。
- 具有相同名称的段的顺序将被保留。
- 代码可以简便地基于它需要驻留在存储器中的位置进行分类。
- 只需一条命令即可将整个段定位到存储器中。

必须保持连续的任何代码序列都放入一个段中。编译器的输出通常会出现在许多不同的段中，但具有相同名称的所有段都按生成它们的顺序进行整理。

编译器具有一个段名的列表，并基于它生成的代码来选择段，请参见**第 5.15.2 节“编译器生成的 psect”**。

一个段不是一个物理的盒子，而是一个用于在汇编代码中定义每个段起始位置的特殊伪指令。该伪指令指示前一个盒子结束，新的盒子开始。一个段可以存放任意数量的汇编代码，也可以为空。

仓库盒子和编译器段都会被加上标签 / 标号，在这两种情况下，标签 / 标号将指示目标，而不是内容。在图 5-4 中，使用颜色来标识盒子的目标城市，或段的名称。

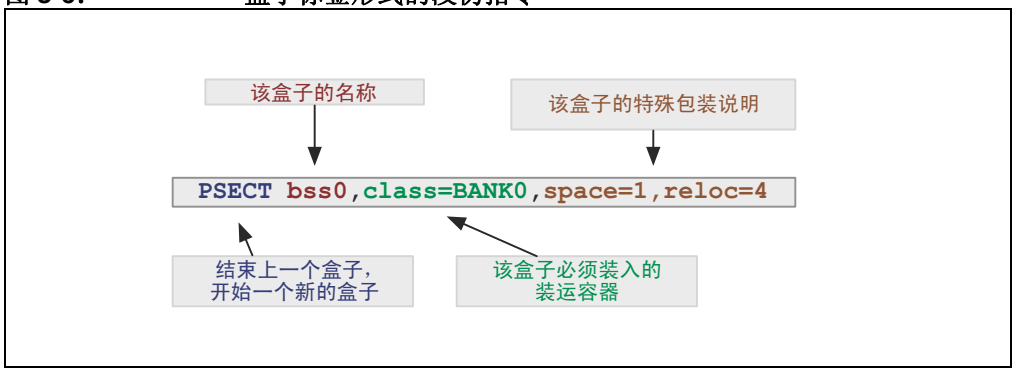
图 5-5 显示了典型盒子标签的形式。仓库中的包装机只关心容器中的打包盒子，所以除了城市和国家 / 地区之外，客户的实际地址在这里并不重要。

图 5-5： 典型盒子标签

```
BOX paris,country=FRANCE,fragile
```

图 5-6 给出了用于开始一个新段的汇编伪指令的示例。城市名称现在为段名，目标国家 / 地区则为链接器类。

图 5-6： 盒子标签形式的段伪指令



© 2013 Microchip Technology Inc.

5.15.1.2.3 转至传送带和目标文件

机器人汇集完订单并将元器件放入盒子之后，盒子中的内容就不再重要，仓库中剩下的活动将仅处理盒子打包。

编译器中存在类似的情形：汇编代码被放入段中之后，指令在链接过程中就不再重要。链接器仅处理各个段，并不在意每个段的内容。[1]

5.15.1.2.4 盒子和段的分类

在仓库中，所有盒子按目标城市进行分类。处理完一天的订单之后，所有这些类似的盒子将进行包装，从而可以方便地将它们聚拢在一起。

这一概念在编译器域中是相同的：段将基于段的名称进行分类，并由链接器合并到一个更大的段中。段的合并顺序将严格遵循它们的生成顺序。

5.15.1.2.5 装入容器和类

在仓库中，打包在一起的盒子会被装入装运容器。对于每个国家 / 地区，将提供一个或多个装运容器。如果同一国家 / 地区有多个城市的订单，则这些城市打包在一起的盒子可以共用相同的容器 （只要有空间）。

链接器将执行类似的操作：它会将经过整理的段安排到类中。通常情况下，每个类中会放入几个段。类代表器件存储器，但可能需要考虑寻址模式或目标器件中存在的分区 / 分页限制。类通常小于它们理论上定位到的存储空间。

想象一下一艘大型货船。虽然这艘船可能很长，但它可以容纳的最大物体将由它运输的装运容器的尺寸决定。船运公司可能会由于尺寸限制而无法运输某个物品，虽然该物品的尺寸只是船只的一小部分。

5.15.1.3 工作中的段

现在，已经对概念有了一个基本的了解，让我们通过一个简单示例来帮助了解它们在编译时的工作方式。

一名工程师编写了一个程序。程序到底做什么或者它如何编写并不重要。目标器件只有一个 RAM 存储区和一个闪存页。

在我们非常简化的示例中，编译器将从表 5-15 中列出的段中进行选择。该表还显示了与每个段关联的链接器类，以及类在器件中代表的存储器位置。

1. 在编译器内部，在代码生成器和链接器之间实际上存在几个步骤。汇编代码先传递给汇编器；但是，生成的目标文件 （传递给链接器）将保留所有段，在本练习中可以忽略它。
汇编器优化器会更改汇编指令，但仅在读取并解释代码之后进行——这是链接器无法完成的。修改后的代码在功能上与原始代码完全相同。其他优化会进一步合并或拆分各个段，但这对于概念基本理解不是很重要。

表 5-15： 简单示例的段名

段名	内容	链接器类	存储器位置
text	可执行代码	CODE	闪存
bss0	需要清零的变量	BANK0	RAM
data0	需要初始化的变量	BANK0	RAM
idata	已初始化变量的值	CODE	闪存

代码生成器开始读取 C 中的程序描述，并生成在目标器件上实现它需要的汇编代码。

代码生成器遇到的第一个情况就是无初始值的变量的 C 定义。像这样的所有变量需保存在一起，以便在启动时可以更简便、更高效地清零它们。代码生成器输出开始一个新段的伪指令。它的内部段名表指示 bss0 是适合的选择，并且该段放置在 BANK0 类中。该伪指令具有以下形式。

```
PSECT bss0,class=BANK0
```

该伪指令之后的代码定义变量：它仅包含一个标号和另一条用于保留所需存储器量的伪指令。它们具有以下形式：

```
PSECT bss0,class=BANK0
myVariable:
    DS 2        ; reserve 2 bytes for this object
```

代码生成器继续向前，并发现另一个未初始化变量。由于该变量也将使用相同的段，代码生成器可以继续添加到当前段，并立即输出另一个标号和伪指令来为第二个变量保留存储器。

现在，在 C 源代码中遇到一个函数定义。代码生成器会发现输出现在需要放置在 text 段中，并输出以下伪指令。

```
PSECT text,class=CODE
```

该伪指令将结束 bss0 段，并开始 text 段。随后的所有代码将包含在 text 段中。生成与该函数相关联的代码，并将其放入该段中。

继续向前，代码生成器遇到一个使用某个值进行初始化的变量。这些变量的初始值将存储在闪存中，并以块的方式复制到分配给它们的 RAM 空间中。通过对于 RAM 镜像使用一个段，对于初始值使用另一个段，编译器可以确保所有初始值将在链接之后对齐，因为各个段将按顺序进行整理。

编译器先输出对应于名为 data0 的段的伪指令。然后，它输出一个标号和伪指令来为变量在 RAM 中保留存储器。接着，它切换到 idata 段，该段将存放所有初始值。输出将初始值放置在该段中的代码。请注意，与已初始化变量关联的代码将输出到两个独立的段中。

```
PSECT data0,class=BANK0
_StartCount:
    DS 2       ; space in RAM for the variable
PSECT idata,class=CODE
    DB 012h    ;initial values places as bytes
    DB 034h    ; which will be copied to RAM later
```

代码生成器读取另一个已初始化变量。它选择 data0 段，输出标号和存储器保留伪指令，然后选择 idata 段并存储初始值。

© 2013 Microchip Technology Inc.

该过程一直继续，直到生成所有程序为止。输出将传递给链接器（通过汇编器），然后链接器会查找它所包含的段。

链接器会在遇到每个段时对其进行分类。整理所有 text 段，以及 bss0、data0 和 idata 段。对它们进行组合的顺序将与它们传递给链接器的顺序匹配，这在之后会作为代码生成器生成它们的顺序。现在，存在一个经过整理的 text、bss0、data0 和 idata 段。

这些段现在定位到存储器中。向链接器传递一些定义链接器类及其所代表的存储器范围的选项（来自编译器驱动程序），请参见**第 7.2.1 节 “-Aclass =low-high,...”**[1]。对于我们的器件，链接器选项具有以下形式。

```
-ACODE=0-7ffh
-ABANK0=20h-6fh
```

例如， BANK0 类涵盖从 20h 至 6fh 的存储器。您可以在映射文件中查看传递给链接器的所有选项，请参见**第 7.4 节 “映射文件”**。

链接器会将每个段“装入”与段所关联的类相关联的存储器。例如，它可能将 text 段放置在 CODE 类中的地址 0 处，然后紧跟着在地址 374h 处放置 idata 段。 bss0 和 data0 段将以类似方式放置在 BANK0 类中。

所有段现在都在类中分配了地址。现在可以确定符号的地址，并最终生成一个 hex 文件。编译器的工作就结束了。

5.15.1.4 更高级的主题

了解了基础知识之后，让我们了解一些关于链接的更复杂问题。

5.15.1.4.1 分配到特定位置

我们在前面的类比中了解到，在确定段需要放置在存储器中哪个位置的时候，链接器会将段放置在其相应的类中。大部分段都是这样进行链接的，但有一些例外情况。复位或中断时执行的代码就是这种例外情况的示例。不能仅仅将它们放置在类中的任意位置；它们必须定位到特定地址处。那么链接器如何定位它们呢？

要链接在某个特定地址处的所有代码都以通常的方式放入某个段中。这些段甚至会具有与它们关联的一个类，但在该类中任意位置的分配可以通过一个特殊的链接器选项进行改写，该选项会指示链接器将段放置在一个特定地址处。按照前面的类比，可以将特殊的链接器选项看做是向包装机提供明确的指示，要求在容器中的哪个位置放置盒子。在下一个示例中，我们将看到显式放置某个段的链接器选项。

```
-preset_vec=0h
-pint_text=04h
```

请注意，如果某个段以这种方式进行链接，链接器将严格遵循这些指示。如果要求将某个段放置在另一个段之上，它将发出警告；但由于并不存在容器（它实际上代表某个存储器范围），链接器将无法检查段是否延伸到超出器件的存储器限制。

5.15.1.4.2 类和容器的区别

容器和链接器类在一个重要方面存在区别：链接器类是概念性的，仅仅代表器件的存储器；它们不是物理存储装置。

1. 请记住这些是链接器选项，您不能直接将它们传递给编译器驱动程序。但是您可以将这些选项封装到驱动程序的 -L- 选项中，请参见**第 4.8.7 节 “-L-：直接调整链接器选项”**。

编译器可以，并且通常也会使用多个类来表示相同的存储器范围。**第 5.15.1.5 节“工作中的更高级的段”**对此进行了说明，其中的示例使用了 CODE 和 CONST 类来代表闪存。虽然类可能涵盖相同的存储范围，但通常容器的大小会有所不同。这使得可以容纳具有不同限制和要求的代码。

当类的存储器范围发生重叠时，将存储器分配给一个类时，在另一个类中它也会被标记为已用。实际上，通过链接器以任何方式分配任何存储器时，它会检查每个类，以确定它是否涵盖该存储器并将它标记为已用。这与物理容器的概念相差很大。

5.15.1.4.3 多仓容器

链接器类通常定义一个存储器范围，但有一些情况下类会定义多个存储器范围。可以将它看做是几个独立的容器，但具有相同的运送目的地。类中的存储器范围不需要是连续的。

编译器通常使用一个多范围的类来代表进行分页的程序存储器。存储器范围中的边界与页边界保持一致。这可以防止段跨页边界。

编译器可以对分区 RAM 使用类似的类，但如果每个变量的目标存储区对于代码生成器是已知的，则代码的长度将可以大大减小。通常，会看到为每个存储区定义了一个独立的类，以及与这些类相关联的专用段。代码生成器将为每个变量分配一个存储区，并选择这样一个段：该段的目标类将代表该存储区。

5.15.1.5 工作中的更高级的段

我们基于前面的示例继续编译。我们的目标器件现在具有两个 RAM 存储区和两个闪存页，表 5-16 显示了编译器现在使用的段的扩展列表。这些段引用新的类，表中也显示了这一点。

表 5-16： 扩展示例的段名

段名	内容	链接器类	存储器位置
text*n*	可执行代码	CODE	闪存
bss0	需要清零的变量	BANK0	RAM
bss1	需要清零的变量	BANK1	RAM
data0	需要初始化的变量	BANK0	RAM
idata	已初始化变量的值	CODE	闪存
reset_vec	与复位关联的代码	CODE	闪存
const	只读变量	CONST	闪存
init	运行时启动代码	CODE	闪存
int_text	中断函数代码	CODE	闪存

编译器的操作与前面的示例中相同，即在生成与当前编译的 C 源代码关联的汇编代码之前，先选择并生成段伪指令。

© 2013 Microchip Technology Inc.

与普通函数关联的汇编代码仍放置在 “text”段中，但由于现在有两个闪存页，所以必须确保这两个页均可使用。如果每个函数都被放置在同一个段中，链接器会对它们进行合并，该段将会不断增长，直到它太大而无法放入其中任意一页。为了确保所有“text”段不进行合并，将每个函数都放置在它自己惟一编号的段中：text0、text1 和 text2 等。由于这些段不具有相同的名称，链接器将不会合并它们。

定义 CODE 类的链接器选项现在具有以下形式：

```
-ACODE=0-7ffhx2
```

它指示链接器， CODE 代表两个存储器页：一个从 0 至 7ffh，另一个从 800h 至 fffh。

该定义指示在地址 800h 处存在某种形式的存储器边界 （器件内部存储器页），它与读为 -ACODE=0-fffh 的类定义具有很大的不同，后者涵盖相同的存储器范围，但它没有边界。链接器会尝试将每个 textx 段分配到一个页 （类存储器范围）中；如果它无法装入，则将尝试另一个。

如果遇到中断函数，则会为该代码选择 int_text 段。由于它独立于用于保存普通函数的段，所以可以将它显式地链接到中断向量位置。假定中断向量位于地址 4 处，定位该段的链接器选项将为以下形式，请参见**第 7.2.19 节 “-Pspec”**。

```
-pint_text=4h
```

为简单起见，在该示例中，已初始化变量的处理方式与前面示例中相同，虽然现在具有两个 RAM 存储区；即，它们总是分配在第一个 RAM 存储区中。

在前面的示例中，我们忽略了将初始值从闪存复制到 RAM 中的代码。该代码在复位之后，在函数 main 之前执行，并且构成运行时启动代码的一部分。它会从闪存中读取每个初始值，并将它写入 RAM 空间中的相应变量。如果变量在 RAM 中分配存储器的顺序与它们在闪存中分配初始值的顺序相匹配，则可以使用单个循环来执行复制。虽然变量在源代码中可能定义在很多地方，由于编译器使用段来存放与每个变量关联的代码，所以将会保留每个变量和值在存储器中的顺序。

该运行时启动代码将输出到名为 init 的段中。跳转到运行时启动代码的代码放置在 reset_vec 段，它将链接到复位位置。通过将这些段链接到同一个页中，从一个段跳转到另一个段的代码将会更短、更快。实现该目标的链接器选项具有以下形式：

```
-preset_vec=0
-pinit=int_text
```

它指示将复位向量代码链接到地址 0 处（这是复位向量位置），并且 init 段（其中包含运行时启动代码）应链接到紧随在 int_text 段中的中断代码之后的位置。如果 int_text 段为空；即程序中未定义中断代码，则 init 将放置在地址 4 处。

先前的所有未初始化变量都放置在 bss0 段中。现在，代码生成器会先检查存储区 0 的存储器是否实际具有足够的空间，然后再执行该操作。如果空间不足，则它会选择 bss1 段，该段最终会链接到存储区 1 的存储器中。代码生成器会跟踪分配到 RAM 的所有对象，所以它可以维护每个 RAM 存储区的可用存储器量。对于这些变量，链接器会将段分配到存储器中，但将由代码生成器决定每个变量使用哪个段。因此，两个应用程序都会在存储器分配过程中发挥作用。

在该示例中，还需要考虑存储在闪存而不是 RAM 中的 const 变量。每个数据字节都封装在一条返回字节到 W 寄存器中的 RETLW 指令中。需要通过代码来访问变量或数组的每个字节。实现该操作的一种方法是 “计算转移”，它涉及到向 W 寄存器装入数据块中的偏移量，并将该偏移量与 PC 相加。（Microchip 应用笔记 AN556 提供了关于如何对于几种器件实现该操作的示例。）计算转移要求目标地址（W 和 PC 相加的结果）不能跨越 256 字边界（即，地址 100h、 200h 和 300h 等）。可以使用一些段和一个类来满足该要求。

在该示例中创建一个名为 CONST 的新类，并如下进行定义

```
-ACONST=0-0ffhx16
```

它指示 CONST 是 100h 长的容器，但在存储器中有 16 个这样的容器，一个紧随另一个，所以 0-ffh 是一个容器，100-1ffh 是另一个容器，如此类推。我们让编译器将所有 RETLW 指令和计算转移代码都放入 const 段，它们将链接到该类中。段可以定位到 16 个容器的任意一个中，但必须完全装入一个容器中。

在该示例中，编译器仅允许一个 const 数据块。与 text 段一样，可以通过让每个 const 数据块处于一个惟一编号的段中来产生多个段 （例如， const1 和 const2 等）。因此，每个段可以保持独立，并分配到 CONST 类的任意存储器容器中。

5.15.1.6 解释常见的链接器错误和问题

我们也可以使用已有的知识来帮助解释一些常见的链接器问题和错误消息。

5.15.1.6.1 无法找到空间

一种常见的链接器错误是“无法在类 ‘XYZ’ 中找到可用于 psect ‘abc’ 的 x 个字”，我们现在可以将它看做是 “无法在容器 ‘FRANCE’ 中找到可用于 ‘paris’ 盒子的 3 立方英尺空间”。

该错误最明显的原因是容器经过稳定填充之后最终用尽了可用空间，即，需要存储的代码或数据总量超出了器件上的存储器量。

另一个原因是盒子大于它要放入的容器。如果是这种情况，段将永远无法装入，即使整个存储空间是空的。定义一个极大的 C 函数、RAM 或 const 数组，导致产生一个异常大的段时，就可能会出现这种情况。其他可能的原因包括很大的 switch 语句，或甚至是带有太多 case 标号的 switch。

© 2013 Microchip Technology Inc.

货船或货机上的货仓可能具有极大的空间，但货物总是打包到装运容器中，装运容器的尺寸将决定可以运输的物体的最大尺寸。与之类似，如果段必须先分配给一个类，则目标器件上的存储器总量是无关的。类看起来似乎有很多限制，但如果没有它们，代码的效率通常会降低，或可能无法配合工作。计算转移就是一个很好的示例。如果计算转移中的指令表跨越 100h 边界，则它将由于指令集的跳转范围有限而发生失败。

如果容器中剩有许多可用空间，但大小都不足以存放该段，则可能也会发生该空间错误。这可能会令人困惑，因为可用空间的总量可能大于要放入的段。类似于无法从包裹中移出一些盒子或进行拆包，如果某个段无法装入任何剩余空间中，而链接器无法对它进行拆分，将会产生错误。该错误消息会指示仍然可用的最大可用空间。

盒子上的标签可能指示特殊的包装说明，例如“易碎——置于容器顶部”或“此面向上”等；与之类似，段也可以指示特殊的存储器分配说明。最常见的要求之一是段必须从地址边界上开始，请参见**第 6.4.9.3.11 节 “reloc”**中的 `reloc` PSECT 标志。如果某个段必须放入某个类中，但也必须对齐到某个地址处，这会使它较难进行定位，并增大产生该错误的几率。如果其他段或对象已放置在容器中部（如**第 5.15.1.4.1 节 “分配到特定位置”**中所示），则也会出现这种情况。

5.15.1.6.2 未在正确的容器中运送

很显然，如果盒子放置在错误的容器中，将无法正确地递送它们。同样，如果将代码放入错误的类或地址中，代码可能也无法运行。编译器将总是确保正确地分配代码和数据，但您也可以手动更改链接器选项。

代码生成器假定各个段将定位到某些特定存储区中，而它生成的代码可能会依赖于这种假定成立。通常情况下，放置在 RAM 中的段必须保持处于它们原本预定的存储区中。包含可执行代码的段会较为宽容，但一些段会具有非常具体的要求。

请记住，链接器将允许您指示的任何分配方式；它无法通过任何方法检查您指示的信息是否正确。在改写链接器选项并重新定位某个段时，总是需要保持小心。**第 5.15.2 节 “编译器生成的 psect”**列出了编译器使用的常用段。它还说明了关于如何链接这些段的限制。

5.15.1.6.3 手动执行操作

目前为止，我们介绍了由代码生成器生成的汇编代码，以及它如何链接到存储器中；现在，我们考虑一下手写的汇编代码。

假设在仓库中某个订单不是由机器人进行处理，而是由一个工人处理。订购的元器件通过手工进行汇集，并通过手工包装到一个盒子中，然后与机器人包装的盒子一起放到传送带上。这非常类似于项目中的手写汇编代码：向链接器传递许多的段，一些由代码生成器创建，一些人工创建。

仓库工人或汇编编程人员可能会出错。在仓库中，工人可能不使用盒子并将元器件松散放在传送带上，或者可能使用了盒子但未贴上标签，或可能会贴上错误的标签。在编译器域中，编写的汇编代码可能未设置相应的段，或者它处于段中但具有错误的（或没有）类，或者段可能具有错误的分配说明。

如果汇编代码不处于某个段中，编译器实际上会将它放入一个默认段中。但由于不存在关于如何处理该段的具体说明，所以它可能被链接到任意位置。这些段就如同被贴上“运送到您所想要的任何地方”的标签。通常，需要将所有汇编代码放置在某个段中，但某些伪指令（如 GLOBAL 或 EQU）不会生成代码，可以放置在任意位置。

编写并定位手写汇编代码的最简单方法是将您创建的段与代表一个适合存储区的现有链接器类相关联。这意味着不需要更改默认的链接器选项。

关联使用 PSECT 伪指令的 class 标志来实现，请参见**第 6.4.9.3.3 节 “class”**。如果要将某个段放置在某个明确地址处，而不是将它放置在某个类中的任意位置，则仍应使用 class 标志。**第 5.15.3 节 “默认链接器类”**列出了链接器类及其所代表的存储器。

即使将代码放入某个段，并使用了相应的类，可能还需要使用其他标志来正确地链接代码。最重要的段标志包括 delta（见**第 6.4.9.3.4 节 “delta”**）、reloc（见**第 6.4.9.3.11 节 “reloc”**）和 space（见**第 6.4.9.3.13 节 “space”**）标志。如果未正确指定这些标志，未正确地定位代码，几乎可以肯定会发生失败。

5.15.2 编译器生成的 psect

代码生成器会将代码和数据放置到具有标准名称的 psect（程序段）中，它们随后通过默认的链接器选项进行定位。对于这些编译器生成的 psect，链接器对待它们的方式与您自己定义的 psect 没有任何不同。在汇编代码中，可以通过使用 PSECT 汇编器伪指令（见**第 6.4.9.3 节 “PSECT”**）来创建 psect。代码生成器使用该伪指令将它生成的汇编代码定向到适当的 psect。关于 psect 的介绍指南，请参见**第 5.15.1 节 “程序段”**。

一些 psect（特别是数据存储器 psect）使用特殊的命名约定。

例如， bss psect。使用名称 bss 是出于历史原因。它用于存放未初始化变量。但是，可能会有一些未初始化变量需要定位到存储区 0 的数据存储器中；其他一些可能需要定位到存储区 1 的存储器中。由于这两组变量需要放入不同的存储区，所以它们需要处于独立的 psect 中，从而使链接器可以独立地控制它们。此外，属于位变量的未初始化变量需要特别对待，所以它们需要自己的 psect。因此，会有许多不同 psect 使用相同基本名称，但通过加上前缀和后缀来使它们具有惟一性。

这些 psect 名称的一般格式为：

[bit]*psectBaseNameCLASS*[*div*]

其中，*psectBaseName* 是 psect 的基本名称，例如 bss。*CLASS* 是基于 psect 将链接到的链接器类（见**第 7.2.1 节 “-Aclass =low-high,...”**）派生的名称，例如 BANK0。如果 psect 包含位变量，则使用前缀 bit。所以可能会具有以下形式的 psect：bssBANK0、 bssBANK1 和 bitbssBANK0，由编译器定义用于存放未初始化变量。

例如，如果某个存储区中的某些存储单元是保留的，或已被绝对对象占用， psect 将无法放入完整的存储区。实际上，将使用一个独立的 psect 来表示已用存储器两侧的可用存储器。字母 l 和 h 用作 psect 名称中的 *div* 字段，以指示它是较低还是较高部分。因此，如果发生了拆分，可能会看到 bssBANK0l 和 bssBANK0h psect。

如果不确定哪个 psect 存放项目中的某个对象或代码，可以检查汇编列表文件（见**第 6.6.1 节 “一般格式”**）。

以下介绍了这些 psect 的内容，按 psect 的基本名称列出。

© 2013 Microchip Technology Inc.

5.15.2.1 程序空间 PSECT

checksum 该 psect 用于标记已使用 `--CHECKSUM` 选项请求的校验和位置，请参见**第 4.8.17 节 “--ASMLIST：生成汇编列表文件”**。校验和值将在链接器执行之后添加，所以不会在汇编列表文件中看到该 psect 的内容，也不会在映射文件中看到任何具体信息。
将该 psect 链接到非默认位置不会对校验和的存储位置产生任何影响，但映射文件会指示它被定位到新的地址。请不要更改与该 psect 相关的默认链接器选项。

cinit 由 C 初始化运行时启动代码使用。该 psect 中的代码是代码生成器随 C 程序的生成代码一起产生的输出，并且不会出现在运行时启动汇编模块中。
该 psect 可以链接在程序存储器页中的任意位置，前提是它不会干扰其他 psect 的要求。

config 用于存储配置字。
该 psect 必须存储在 HEX 文件中的特殊位置。请不要更改与该 psect 相关的默认链接器选项。

const 这些仅针对 PIC18 的 psect 存放使用 `const` 声明的对象和字符串字面值，它们是不可修改的。它在程序中的 `const` 数据总量超出 64k 时使用。
该 psect 可以链接到程序存储器页中的任意位置，前提是它不会干扰其他 psect 的要求。

eeprom（PIC18：`eeprom_data`）用于在 EEPROM 存储器中存储初始值。
请不要更改与该 psect 相关的默认链接器选项。

idata 这些 psect 包含所有已初始化变量的 ROM 镜像。这些 psect 将在启动时被复制到 data psect 中。在这种情况下，类名用于描述相应的基于 RAM 的数据 psect 的类。这些 psect 将存储在程序存储器中，而不是数据存储空间中。
这些 psect 隐式地链接到 CODE 链接器类中的某个任意位置。链接器选项可以进行更改，从而可以将该 psect 放置在程序存储器页内的任意地址处，前提是它不会干扰其他 psect 的要求。

idloc 用于存储 ID 单元字。
该 psect 必须存储在 HEX 文件中的特殊位置。请不要更改与该 psect 相关的默认链接器选项。

init 由运行时启动汇编模块中的汇编代码使用。该段和 `cinit` 段中的代码定义运行时启动代码。PIC18 器件还会使用 `end_init` psect，其中包含将控制转移到 main 函数的代码。
如果未定义中断代码，则复位向量处的代码会 “直落”到该 psect。建议不要更改与该 psect 相关的默认链接器选项，以防出现这种情况。

intcode 和 **intcodelo** 这两个 psect 分别包含高优先级（默认）和低优先级中断服务程序的可执行代码。这两个 psect 分别链接到地址 0x8 和 0x18 处的中断向量。
请不要更改与这两个 psect 相关的默认链接器选项。如果希望在使用自举程序时移动代码，请参见**第 4.8.22 节 “--CODEOFFSET：将程序代码偏移到某个地址”**。

intentry 包含链接到中断向量处的中断服务程序的入口代码。对于中档器件，该代码会保存必需的寄存器，并跳转到主中断代码；对于增强型中档器件，该 psect 将包含中断函数主体。（PIC18 器件使用 intcode psect。）
必须将该 psect 链接到中断向量处。请不要更改与该 psect 相关的默认链接器选项。如果希望在使用自举程序时移动代码，请参见 --CODEOFFSET 选项（**第 4.8.22 节 “--CODEOFFSET：将程序代码偏移到某个地址”**）。

jmp_tab 仅用于低档处理器，它是用于存储跳转地址和函数返回值的 psect。
请不要更改与该 psect 相关的默认链接器选项。

maintext 该 pect 将包含 main() 函数的汇编代码。main() 的代码会被单独放置，因为它包含程序入口点。
请不要更改与该 psect 相关的默认链接器选项，因为运行时启动代码可能会“直落”到该 psect，这将要求它链接到紧随在该代码之后的位置。

mediumconst 这些仅针对 PIC18 的 psect 存放使用 const 声明的对象和字符串字面值，它们是不可修改的。在程序中的 const 数据总量超出 255 字节，但不超出 64k 时使用。
该 psect 可以链接到程序存储器低 64k 中的任意位置，前提是它不会干扰其他 psect 的要求。对于 PIC18 器件，该 psect 的位置必须高于最高 RAM 地址。

powerup 包含用户提供的上电程序的可执行代码。
请不要更改与该 psect 相关的默认链接器选项。

reset_vec 该 psect 包含与复位向量关联的代码。
请不要更改与该 psect 相关的默认链接器选项，因为它必须链接到目标器件的复位向量位置。如果希望在使用自举程序时移动代码，请参见 --CODEOFFSET 选项（**第 4.8.22 节 “--CODEOFFSET：将程序代码偏移到某个地址”**）。

reset_wrap 对于低档 PIC 器件，该 psect 包含器件 PC 从位于程序存储器顶部的振荡器校准单元回绕到地址 0x0 之后执行的代码。
请不要更改与该 psect 相关的默认链接器选项，因为它必须链接到目标器件的复位向量位置。

smallconst 这些 psect 存放使用 const 声明的对象和字符串字面值，它们是不可修改的。在程序中的 const 数据总量不超出 255 字节时使用。
该 psect 可以链接到程序存储器中的任意位置，前提是它不会跨越 0x100 边界，并且它不会干扰其他 psect 的要求。对于 PIC18 器件，该 psect 的位置必须高于最高 RAM 地址。

strings strings psect 用于存放 const 对象。它还包含所有未命名的字符串字面值。该 psect 会被链接到 ROM 中，因为其内容不需要修改。
该 psect 可以链接到程序存储器中的任意位置，前提是它不会跨越 0x100 边界，或者它不会干扰其他 psect 的要求。

© 2013 Microchip Technology Inc.

stringtext 在针对低档器件进行编译时， stringtext psect 用于存放 const 对象。该 psect 会被链接到 ROM 中，因为其内容不需要修改。
该 psect 必须链接到每个程序存储器页的上半部分。

text*n* 这些 psect（其中的 *n* 是十进制数字）包含不需要特殊链接位置的所有其他可执行代码。
这些 psect 可以链接到程序存储器中的任意位置，前提是它们不会干扰其他 psect 的要求。

***xxx*_text** 定义已设为绝对函数（即，放置在某个地址处）的函数的 psect。*xxx* 将为与该函数相关联的汇编符号。例如，如果函数 rv() 被设为绝对函数，与它关联的代码将出现在名为 _rv_text 的 psect 中。
由于这些 psect 已放置在 C 源代码中指示的地址处，所以不应更改定位它们的链接器选项。

5.15.2.2 数据空间 PSECT

nv 这些 psect 用于存储使用 persistent 限定的变量。它们在启动时不会被清零或以其他方式修改。
这些 psect 可以链接到其目标存储区中的任意位置，并且不应与器件支持的任何公共（未分区的）存储器重叠（如果它是分区 psect）。

bss 这些 psect 包含所有未初始化变量。
这些 psect 可以链接到其目标存储区中的任意位置，并且不应与器件支持的任何公共（未分区的存储器）重叠（如果它是分区 psect）。

data 这些 psect 包含所有已初始化变量的 RAM 镜像。
这些 psect 可以链接到其目标存储区中的任意位置，并且不应与器件支持的任何公共（未分区的存储器）重叠（如果它是分区 psect）。

cstack 这些 psect 包含编译堆栈。堆栈中是整个程序的自动和参数变量。关于编译堆栈的信息，请参见**第 5.5.4 节 “绝对变量”**。
这些 psect 可以链接到其目标存储区中的任意位置，并且不应与器件支持的任何公共（未分区的）存储器重叠（如果它是分区 psect）。

5.15.3 默认链接器类

链接器使用类来表示存储器范围。关于 psect 和链接器类的介绍指南，请参见**第 5.15.1 节“程序段”**。

类通过由编译器驱动程序传递给链接器的链接器选项定义，请参见**第 7.2.1 节“-Aclass =low-high,...”**。 psect 通常在与它们关联的类中分配空间。关联使用 PSECT 伪指令的 `class` 标志来实现，请参见**第 6.4.9.3.3 节“class”**。或者，也可以使用链接器选项将 psect 明确地放置在某个类中，请参见**第 7.2.19 节 “-Pspec”**。

类可能代表单个存储器范围或多个范围。即使两个范围是连续的，一个范围结束、另一个范围开始的地址也会构成边界，放置在类中的 psect 永远不能跨越这种边界。您会看到一些类涵盖相同的地址，但被拆分为不同的范围，具有不同的边界。这是为了适应这样的 psect：其内容根据它们在存储器中的定位位置的假定进行编译。

从一个类中分配的存储器会被保留，防止指定了相同存储器的其他类使用。对于链接器，类名或它定义的存储器没有任何意义。

如果使用 `--ROM`或 `--RAM`选项来保留存储器，将会从这些类中减去存储器（见**第 4.8.49 节 “--ROM：调整 ROM 范围”**和**第 4.8.48 节 “--RAM：调整 RAM 范围”**）。当指定一个调试器 （如 ICD）时 （见**第 4.8.23 节 “--DEBUGGER：选择调试器类型”**），也可能会从与某些类关联的范围除去存储器，从而使您的程序不使用该存储器。

虽然可以手动调整与类关联的范围，但建议不要这样做。永远不要改变或删除类定义选项指定的地址边界。

以下是编译器可能定义的链接器类。对于每款器件，并不是所有类都可能出现。

5.15.3.1 程序存储器类

CODE 包含映射到目标器件上的页的范围。因此，它通常用于包含可执行代码的 psect。在低档器件上，它只能由通过跳转表访问的代码使用。

ENTRY 主要由低档器件使用，用于包含通过 CALL 指令 （调用只能针对页的上半部分）访问的可执行代码的 psect。该类以这种方式定义：它的长度为页，但它包含的 psect 将定位为使它从页的上半部分开始。
该类还用于中档器件中，并将包含许多范围，每个均为 0x100 字长，在 0x100 边界上对齐。因此，它对于内容不能跨越 0x100 字边界的 psect 很有用。

STRING 包含 0x100 字长并在 0x100 边界上对齐的范围。因此，它对于内容不能跨越 0x100 字边界的 psect 很有用。

STRCODE 定义涵盖整个程序存储器的单个存储器范围。它对于内容可以出现在任何页中并可以跨越页边界的 psect 很有用。

CONST 包含 0x100 字长并在 0x100 边界上对齐的范围。因此，它对于内容不能跨越 0x100 字边界的 psect 很有用。

© 2013 Microchip Technology Inc.

5.15.3.2 数据存储器类

RAM 包含涵盖目标器件所有通用 RAM 存储器、但不包括任何公共 （未分区）存储器的范围。因此，它对于必须放置在通用分区 RAM 中、但可以放置在任意存储区的 psect 很有用。

BIGRAM 包含单个存储器范围，该范围旨在涵盖增强型中档器件的线性数据存储器。它适用于内容使用线性寻址进行访问的任何 psect。

ABS1 包含涵盖目标器件所有通用 RAM 存储器，包括任何公共 （未分区）存储器在内的范围。因此，它对于必须放置在通用分区 RAM 中、但可以放置在任意存储区或公共存储器中的 psect 很有用。

BANK*x*（其中的 *x* 是存储区编号）每个均包含单个范围，该范围涵盖该存储区中的通用 RAM，但不包括任何公共 （未分区）存储器。

COMMON 包含单个存储器范围，该范围涵盖公共 （未分区） RAM （如存在）。

SFR*x*（其中的 *x* 是存储区编号）每个均包含单个范围，该范围涵盖该存储区中的 SFR 存储器。编程人员通常不使用这些类，因为它们不代表通用 RAM。

5.15.3.3 其他类

CONFIG 包含单个范围，该范围涵盖为 hex 文件中的配置位数据保留的存储器。编程人员通常不使用该类，因为它不代表通用 RAM。

IDLOC 包含单个范围，该范围涵盖为 hex 文件中的 ID 单元保留的存储器。编程人员通常不使用该类，因为它不代表通用 RAM。

EEDATA 包含单个范围，该范围涵盖目标器件的 EEPROM 存储器 （如存在）。该类通常用于将包含要编程到 EEPROM 中的数据的 psect。

5.15.4 替换库模块

MPLAB XC8 C 编译器附带了一个库管理器 LIBR，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**第 8.2 节 “库管理器”**。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。这是由于编译器扫描源文件和库文件的方式而导致的。

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。

如果某个源文件中定义了该符号，则编译器将永远不会实际搜索库来查找该符号。源代码和库文件中同时存在该符号，不会产生错误。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。

所有库源代码都是以 C 语言编写的，包含这些库程序的 p 代码库文件会被实际上传递给代码生成器，而不是链接器，但这两个应用程序都按上述的方式来解析库符号。

您不能使用上述方法将 C 库函数替换为以汇编代码编写的等效函数。如果需要这样，则需要使用库管理器编辑或创建一个新的库文件。

5.15.5 签名检查

编译器会自动生成所有函数的签名。签名是一个基于以下组合计算的 16 位值：函数的返回数据类型、其参数的数量，以及影响函数调用序列的其他信息。每当引用或定义一个函数时，都会生成该签名并放置目标代码。

在链接时，链接器将报告所有不匹配的签名，它表示函数定义之间存在差异。只有程序是预编译的目标文件或汇编程序时，MPLAB XC8 才会从链接器中发出不匹配错误。其他函数不匹配均由代码生成器报告。

有时需要编写通过使用 `extern` 声明从 C 中调用的汇编语言程序。此类汇编语言函数需要包含一个与用于调用它们的 C 原型兼容的签名。确定某个函数正确签名最简单的方法是编写一个具有相同原型的虚拟 C 函数，并使用 `--ASMLIST` 选项检查汇编列表文件（见**第 4.8.16 节 “--ADDRQUAL：设置编译器对于存储器限定符的响应”**）。

例如，假设具有名为 `_widget` 的汇编语言程序，它需要 `char` 参数，并返回 `char`。用于从 C 中调用该函数的原型为：

```
extern char widget(char);
```

在 C 代码中对 `_widget` 进行调用时，将会生成具有一个 `char` 参数和一个 `char` 返回值的函数的签名。为了匹配正确的签名，`widget` 的源代码需要包含定义相同签名值的汇编器 `SIGNAT` 伪指令。为了确定正确的值，需要在一个虚拟文件中编写以下代码：

```
char widget(char arg1)
{
}
```

在汇编列表文件中看到的所生成汇编代码将包含以下代码行：

```
SIGNAT  _widget,4217
```

`SIGNAT` 伪指令指示汇编器在 `.obj` 文件中包含一条记录，该记录将签名值 4217 与符号 `_widget` 相关联。值 4217 是具有一个 `char` 参数和一个 `char` 返回值的函数的正确签名。

如果将该伪指令复制到包含 `_widget` 代码的汇编源文件中，它会将正确签名与函数相关联，链接器将能够检查传递的参数是否正确。

如果 C 源文件包含以下声明：

```
extern char widget(long);
```

那么将生成一个不同的签名，链接器将报告签名不匹配，提醒您可能存在不兼容的调用约定。

© 2013 Microchip Technology Inc.

5.15.6 链接器定义的符号

链接器会定义一些特殊的符号，可以用于确定一些 psect 链接到存储器中的位置。如果需要，可以在代码中使用这些符号。

psect 的链接地址可以通过名称为 __L*name*（两个前导下划线）的全局符号获得，其中的 *name* 是 psect 的名称。例如， __LbssBANK0 是 bssBANK0 psect 的下边界。

psect 的最高地址（即链接地址加长度）使用符号 __H*name* 表示。

如果 psect 具有不同的装入和链接地址，则装入起始地址使用符号 __B*name* 表示。

并不是所有 psect 都会分配这些符号，特别是那些通过 -P 链接器选项不放置在存储器中的 psect。请参见**第 7.2.19 节 “-Pspec”**。 psect 名称可能会因器件而不同。

汇编代码可以通过全局声明这些符号来使用它们，例如：

```
GLOBAL __Lidata
```

而 C 代码可以通过声明如下形式的符号来使用它们。

```
extern char * _Lidata;
```

请注意，在 C 域中只有一个前导下划线，请参见**第 5.12.3.1 节 “等效汇编符号”**。由于该符号代表一个地址，所以典型的类型选择为一个指针。

注：

© 2013 Microchip Technology Inc.

第 6 章 宏汇编器

6.1 简介

MPLAB XC8 C 编译器随附了两个宏汇编器，用于汇编所有 8 位 PIC 器件的源文件。这两个汇编器的操作和汇编器伪指令几乎完全相同。使用编译器驱动程序编译项目时，它会调用相应的汇编器应用程序。

本章介绍汇编器的用法，以及汇编器可在源文件中接受的伪指令 （汇编器伪操作和控制）。

虽然“汇编器”一词几乎普遍用于描述将人们可阅读的助记符转换为机器码的工具，但“汇编程序”和“汇编语言”都用于描述这种工具读取的源代码。后者更为常用，本手册中使用它来描述语言。因而，您会看到*汇编语言*（或仅仅*汇编*）和*汇编列表*等词汇，而且看到*汇编器选项*、*汇编器伪指令*和*汇编器优化器*。

用户指南的这一章讨论以下主题：

- 汇编器用法
- 选项
- MPLAB XC8 汇编语言
- 汇编代码级优化
- 汇编列表文件

6.2 汇编器用法

对于 PIC18 器件，汇编器名为 ASPIC18；对于所有其他 8 位器件，名为 ASPIC。它可在 *Windows*、*Linux* 和 *Mac OS X* 系统上运行。请注意，除非存在错误或警告，否则汇编器不会产生任何消息——不存在“汇编完成”消息。

通常使用命令行驱动程序 `xc8` 来调用汇编器，因为可以向它传递汇编源文件作为输入；但是，本节列出了汇编器的选项，以备应对直接调用汇编器或使用命令行驱动程序选项 `--SETOPTION` 指定汇编器选项的情形，请参见**第 4.8.53 节 “--SETOPTION：为应用程序设置命令行选项”**。

汇编器的用法在所有可用操作系统下是类似的。所有命令行选项在大写或小写形式下都可识别。以下给出了基本命令格式：

```
ASPIC[18] [ options  ] files
```

`files` 是一个或多个汇编源文件的空格分隔列表。在指定多个源文件时，汇编器会将它们视为单个模块；即，对指定的所有源文件的组合执行单次汇编。这些文件必须以完整形式指定，不会假定任何默认扩展名或后缀。

`options` 是可选的汇编器选项的空格分隔列表，每个汇编器选项均带有单短划线 `-`（对于单字母选项，作为第一个字符）或双短划线 （对于多字母选项）。汇编器选项必须在命令行上在任何文件之前指定。

© 2013 Microchip Technology Inc.

6.3 选项

表 6-1 列出了 ASPIC[18] 可识别的所有命令行选项，随后提供了每个选项的完整说明。

表 6-1： ASPIC[18] 命令行选项

选项	含义	默认
-A	生成汇编程序输出	生成目标代码
-C*chipinfo*	定义 chipinfo 文件	dat\picc[18].ini
-E[*file* | *digit*]	设置错误目标 / 格式	
-F*length*	指定列表页面长度	66
-H	输出常量的 HEX 值	十进制值
-I	列出宏扩展	不列出宏
-L[*listfile*]	生成列表	无列表
-N	禁止合并优化	使能合并优化
-O	执行优化	无优化
-O*outfile*	指定目标名称	srcfile.obj
-R	指定非标准 ROM	
-T*width*	指定列表页面宽度	80
-V	生成行号信息	无行号
-VER=*version*	指定列表文件标题的完整版本信息	
-W*level*	设置警告级别阈值	0
-X	OBJ 文件中无局部符号	
--CHIP=*device*	指定器件名称	
--DISL=*list*	指定禁止的消息	不禁止任何消息
--EDF=*path*	指定消息文件位置	
--EMAX=*number*	指定最大错误数	10
--OPT=*optimization*	指定优化类型	
--VER	打印版本号并停止	

6.3.1 -A：生成汇编文件

如果使用了该选项，将生成一个汇编程序文件，而不是通常的目标文件格式。在使用 -O 优化选项检查生成的优化汇编程序时，这会非常有用。

默认情况下，输出文件的扩展名为 .opt，除非使用 -O*outfile* 输出选项来指定另一个名称。

6.3.2 -C：指定 chipinfo 文件

指定要使用的 chipinfo 文件。 chipinfo 文件名为 picc.ini，它位于编译器安装目录的 dat 目录中。该文件指定关于当前选定器件的信息。

© 2013 Microchip Technology Inc.

6.3.3 -E：指定错误格式 / 文件

错误消息的默认格式为以下形式：

filename: *line*: *message*

它表示，文件 *filename* 的第 *line* 行出现 *message* 类型的错误。

不带参数的 -E 选项将使汇编器对于错误和警告消息使用备用格式。

指定一个文件名作为参数会强制汇编器将错误和警告消息定向到具有指定名称的文件中。

6.3.4 -F：指定页面长度

默认情况下，汇编列表格式是不分页的；即，汇编列表输出是连续的。输出可以格式化为不同长度的页面。每个页面将以题头和标题开始 （如指定）。

-F选项用于指定页面长度。*length*值为零意味着输出不分页。长度以行数的形式指定。

6.3.5 -H：打印十六进制常量

该选项指定输出常量应以十六进制值而不是十进制值显示。请求使用该选项时，该选项会同时影响汇编列表文件和汇编输出。

6.3.6 -I：列出宏扩展

该选项会强制列出宏扩展和未汇编的条件语句，否则它们会被 NOLIST 汇编器控制禁止，请参见**第 6.4.10 节 “汇编器控制”**。要生成实际的列表输出，仍然需要 -L 选项。

6.3.7 -L：生成汇编列表

该选项用于请求生成汇编列表文件。如果指定了 *listfile*，则列表将写入该文件，否则它将被写入标准输出。

如果使用命令行驱动程序 xc8 和 --ASMLIST 驱动程序选项进行编译，则会应用该选项，请参见**第 4.8.16 节 “--ADDRQUAL：设置编译器对于存储器限定符的响应”**。

6.3.8 -O：优化汇编代码

它用于请求汇编器对汇编代码执行优化。请注意，使用该选项会使汇编过程变慢，因为汇编器必须对输入代码执行一遍额外的汇编。

使用该选项时，基于 C 源代码生成的汇编代码的调试信息可能会在调试器中变得不可靠。

如果使用命令行驱动程序 xc8 和 --OPT 驱动程序选项进行编译，则可以应用该选项，请参见**第 4.8.42 节 “--OPT：调用编译器优化”**。

6.3.9 -O：指定输出文件

默认情况下，汇编器通过从第一个源文件名中去除任何后缀或扩展名并追加 .obj 来确定要创建的目标文件的名称。 -O 选项使用户可以覆盖默认的文件名，并为目标文件指定新的名称。

6.3.10 -T：指定列表页面宽度

该选项用于指定汇编列表文件页面宽度（以字符为单位）。*width* 应为大于 41 的十进制数。默认宽度为 80 个字符。

6.3.11 -V：生成汇编调试信息

该选项将在汇编器生成的目标文件中包含行号和文件名信息。此类信息可供调试器使用。

请注意，行号将对应于汇编文件中的汇编代码行。在汇编由代码生成器生成的汇编文件时，不应使用该选项。在这种情况下，调试信息应与原始 C 源代码而不是中间汇编代码相关联。

6.3.12 -VER：指定版本信息

该选项用于将完整的版本信息（包括指示 Beta 测试版本或发行候选（Release Candidate）版本的可选文本）传递给汇编器。该信息仅在汇编列表文件的标题中使用，并不会反映在 --VER 选项的输出中。

6.3.13 -X：剔除局部符号

由汇编器创建的目标文件中包含符号信息，包括局部符号；即，既不属于公共符号也不属于外部符号的符号。-X 选项将阻止在目标文件中包含局部符号，从而减小文件大小。

6.3.14 --CHIP：指定器件名称

该选项定义正在使用的器件。器件类型还可以通过在汇编源文件中使用器件伪指令来指示，请参见**第 6.4.9.20 节 “PROCESSOR”**。此外，也可以通过编译器的 chipinfo 文件将您自己的处理器添加到编译器中。

如果使用命令行驱动程序 xc8 和 --CHIP 驱动程序选项进行编译，则会应用该选项，请参见**第 4.8.20 节 “--CHIP：定义器件”**。

6.3.15 --DISL：禁止消息

该选项主要由命令行驱动程序 xc8 用于禁止特定消息编号。它可接受要在编译过程中禁止的消息编号的*逗号*分隔列表。

如果使用命令行驱动程序 xc8 和 --MSGDISABLE 驱动程序选项进行编译，则会应用该选项，请参见**第 4.8.38 节 “--MSGDISABLE：禁止警告消息”**。

关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

© 2013 Microchip Technology Inc.

6.3.16 --EDF：设置消息文件路径

该选项主要由命令行驱动程序 xc8 用于指定消息描述文件的路径。默认文件位于编译器安装目录的 dat 目录中。

关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

6.3.17 --EMAX：指定最大错误数

该选项主要由命令行驱动程序 xc8 用于指定在汇编器终止之前可以遇到的最大错误数量。默认数量为 10 个错误。

如果使用命令行驱动程序 xc8 和 --ERRORS 驱动程序选项进行编译，则会应用该选项，请参见**第 4.8.29 节 “--ERRORS：最大错误数”**。

关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

6.3.18 --OPT：指定优化类型

该选项用于补充汇编器 -O 选项，它指示关于所需优化的特定信息。可以指定 speed、space 和 debug 子选项，以指示与过程抽象相关的首选项。

抽象将在设置 space 选项时使能；在设置 speed 时禁止。debug 子选项会限制一些优化的应用，否则这些优化可能会严重破坏由调试器使用的调试信息。

6.3.19 --VER：打印版本号

选择该选项会打印与汇编器应用程序的版本和内部版本有关的信息。即使命令行上存在其他选项和文件，汇编器也会在处理完该选项之后终止。

6.4 MPLAB XC8 汇编语言

以下介绍宏汇编器 ASPIC[18] 接受的源语言。所有操作码助记符和操作数语法都是特定于目标器件的。虽然在汇编器级别支持 PIC17 系列指令集，但代码生成器无法生成针对这些器件的代码，所以没有任何 C 项目可以面向这些器件。

本节将介绍附加的助记符和汇编器伪指令。

对于编译器生成的中间汇编代码和手写的汇编源代码，将使用相同的汇编器应用程序。

6.4.1 汇编指令差异

MPLAB XC8 汇编器使用相对于 Microchip 数据手册所规定的汇编语言稍作改造的汇编语言。以下将详细介绍对于指令格式的更改，以及除器件指令集外还可使用的伪指令。

这些存在差异的指令可以在嵌入到C代码中的行内汇编代码中或手写汇编模块中使用。

6.4.1.1 目标位置

某些 PIC 指令使用操作数 “,0” 或 “,1” 来指定操作结果的目标。 ASPIC[18] 使用可读性更高的操作数 “,w” 和 “,f” 来指定目标寄存器。

使用 “,w” 操作数时，将选择 W 寄存器作为目标；使用 “,f” 操作数或未指定目标操作数时，将选择文件寄存器。目标操作数中的字母大小写并不重要。例如：

```
MOVF   _foo,w
ADDWF  _foo,f        ;the ,f destination is optional
```

不能对 ASPIC[18] 使用数值目标操作数。

汇编器还使用目标选择操作数 “,b” 来指示在访问指定的文件寄存器地址时， PIC18 指令应使用存储区选择寄存器 （Bank Select Register， BSR）。“,c” 操作数指示地址处于公共存储器中，它在 PIC18 器件上被称为快速操作存储区。或者，也可以通过在操作数前面加上字符 “c:” 来指示该地址处于公共存储器中。这些操作数和前缀会影响指令中的 RAM 访问位。例如：

```
ADDWF  _foo,c
BTFSC  c:_foo,3
```

如果指令未指定目标选择操作数或公共存储器前缀，指令将使用分区访问。这些操作数和前缀不适用于 MOVFF 指令的操作数，该指令可接受两个未截断的地址，并总是独立于 BSR 工作。

如果指令地址是绝对地址，并且地址处于快速操作存储区中，则不需要公共存储器目标选择符 “,c” 或 “c:”。汇编器将根据地址确定所访问的是快速操作存储区。

© 2013 Microchip Technology Inc.

例如，以下指令显示了先将 W 寄存器内容传送到某个绝对存储单元，然后再传送到由某个标识符表示的地址。其中显示了这些指令的操作码（假定赋予 _foo 的地址为 0x516，赋予 _bar 的地址为 0x55）。

```
6EE5  MOVWF 0FE5h              ;write to access bank location 0xFE5
6E55  MOVWF _bar,c             ;write to access bank location 0x55
0105  BANKSEL(_foo)            ;set up BSR to access _foo
6F16  MOVWF BANKMASK(_foo),b   ;write to _foo (banked)
6F16  MOVWF BANKMASK(_foo)     ;defaults to banked access
```

请注意，前两条指令的 RAM 访问位（操作码的 bit 8）会清零，但在最后两条指令中它将会置 1。

6.4.1.2 存储区和页选择

BANKSEL 伪指令可以用于生成用以选择指定操作数的存储区的指令。操作数应为处于数据存储器中的某个对象的符号或地址。

根据目标器件，生成代码可能包含一条或多条将相应寄存器中的位置 1/ 清零的位指令，或使用 MOVLB 指令（对于增强型中档或 PIC18 器件）。由于该伪指令在中档或低档器件上会扩展为多条指令，所以在这些器件上，它不应紧随 BTFS*x* 指令。

例如：

```
MOVLW 20
BANKSEL(_foobar)    ;select bank for next file instruction
MOVWF _foobar&07fh  ;write data and mask address
```

与之类似， PAGESEL 伪指令可以用于生成用以选择地址操作数的页的代码。对于当前页，可以使用位置计数器 $ 作为操作数。

根据目标器件，生成代码可能包含一条或多条将相应寄存器中的位置 1/ 清零的指令，或使用 MOVLP 指令（对于增强型中档 PIC 器件）。由于该伪指令会扩展为多条指令，所以它不应紧随 BTFS*x* 指令。

例如：

```
FCALL _getInput
PAGESEL $        ;select this page
```

针对 PIC18 目标进行编译时，可以接受该伪指令，但它没有任何作用，不会生成任何代码。支持它纯粹是为了方便跨 8 位器件进行移植。

6.4.1.3 中断返回模式

RETFIE PIC18 指令可以跟随“f”（无逗号），以指示在执行时应获取影子寄存器内容并复制到其相应的寄存器。如果没有该修饰符，则不会从影子寄存器中更新寄存器。它代替器件数据手册中所述的“0”和“1”操作数。

以下同时给出两种形式的示例，以及它们生成的操作码。

```
0011  RETFIE f    ;shadow registers copied
0010  RETFIE      ;return without copy
```

低档和中档器件不允许这种语法。

6.4.1.4 长跳转和调用

汇编器可以识别几个会扩展为常规 PIC MCU 汇编指令的助记符。这些助记符为 FCALL 和 LJMP。在低档和中档器件上，这些指令分别扩展为常规的 CALL 和 GOTO 指令，并确保在目标地址处于程序存储器的另一个页中时生成必需的指令来设置 PCLATH（对于中档器件）或 STATUS（对于低档器件）中的位。是否生成页选择指令，以及具体将它们定位到何处，将取决于周围的源代码。页选择指令可能紧靠在调用或跳转之前，或作为先前 FCALL/LJMP 指令的一部分生成并紧随其后。

在 PIC18 器件上，提供这些助记符纯粹是为了与程序存储器较小的 8 位器件保持兼容，并且总是扩展为常规的 PIC18 CALL 和 GOTO 指令。

应尽可能使用这些附加的助记符，因为它们可以使汇编代码独立于要执行的程序的最终位置。如果确定调用或跳转处于当前页内，则会优化去除用以设置 PCLATH 位的额外代码。请注意，嵌入到 C 代码中的行内汇编代码从不优化，汇编模块则需要一个特定选项来使能优化，请参见**第 4.8.42 节 “--OPT：调用编译器优化”**。未经优化的 FCALL 和 LJMP 指令将总是生成页选择代码。

以下中档 PIC MCU 示例显示了汇编列表文件中的 FCALL 指令。可以看到 FCALL 指令已扩展为 5 条指令。在该示例中，存在两条置 1/ 清零 PCLATH 寄存器中位的位指令。在进行调用之后，为了重新选择在 FCALL 之前选择的页，也会置 1/ 清零该寄存器中的位。

```
13  0079  3021                    movlw   33
14  007A  120A  158A  2000       fcall   _phantom
          120A  118A
15  007F  3400                    retlw   0
```

由于 FCALL 和 LJMP 指令可能扩展为多条指令，所以在它们之前不应使用可以跳过的指令，例如 BTFSC 指令。

在 PIC18 器件上，常规 CALL 指令可以跟随“,f”，用以指示应将 W、STATUS 和 BSR 寄存器压入其相应的影子寄存器。它代替器件数据手册中所述的 “,1”语法。

6.4.1.5 相对转移

PIC18 器件实现了条件相对转移指令，例如 BZ 和 BNZ。与 GOTO 指令相比，这些指令的跳转范围相对有限。

请注意在某些情况下，汇编器可能会将相对转移指令更改为通过 GOTO 指令使用相反条件实现的相对转移。例如：

```
    BZ error
    ;next
```

可能变为：

```
    BNZ l18
    GOTO error
l18:
    ;next
```

它的功能是相同的，执行它是为了使条件转移可以使用与 GOTO 指令相同的目标地址范围。

© 2013 Microchip Technology Inc.

6.4.2 语句格式

表 6-2：“ASPIC[18] 语句格式”中列出了合法的语句格式。

label 字段是可选的，如果存在，则应包含一个标识符。标号可以自己单独一行出现，也可以在前面具有一个助记符，如第二种格式所示。

第三种格式仅对于某些汇编器伪指令是合法的，例如 MACRO、SET 和 EQU。*name* 字段是必需的，其中也应包含一个标识符。

如果汇编文件先由 C 预处理程序进行处理（见**第 4.8.11 节 “-P：预处理汇编文件”**），那么它可能还包含构成有效预处理器伪指令的行。关于这些伪指令的格式的更多信息，请参见**第 5.14.1 节 “C 语言注释”**。

对于语句的任何部分应出现在哪个列或行的哪个部分，不存在任何限制。

表 6-2： ASPIC[18] 语句格式

格式编号	字段 1	字段 2	字段 3	字段 4
格式 1	*label*:			
格式 2	*label*:	*mnemonic*	*operands*	*; comment*
格式 3	*name*	*pseudo-op*	*operands*	*; comment*
格式 4	*; comment only*			
格式 5	*empty line*			

6.4.3 字符

所使用的字符集为标准 7 位 ASCII。字母大小写对于标识符是有意义的，但对于助记符和保留字不是。制表符会被视为等同于空格。

6.4.3.1 分隔符

所有数字和标识符都必须使用空格、非字母数字字符或行结束符进行分隔。

6.4.3.2 特殊字符

有几个字符在某些上下文中是特殊的。在宏主体内，字符 & 用于标记连接。要在宏主体内使用按位操作符 &，需要改为通过使用 && 对它进行转义。在宏参数列表中，尖括号 < 和 > 用于括起宏参数。

6.4.4 注释

汇编注释使用不属于字符串或字符常量一部分的分号开头。

如果汇编文件先由 C 预处理程序进行处理（见**第 4.8.11 节 “-P：预处理汇编文件”**），则文件还可能包含使用标准 /* ... */ 和 // 语法的 C 或 C++ 样式的注释。

6.4.4.1 特殊注释字符串

代码生成器会在编译器生成的汇编指令之后追加几个注释字符串。这些注释通常由汇编器优化器使用。

注释字符串 ;volatile 用于指示在指令中访问的存储单元与 C 源代码中声明为 volatile 类型的变量关联。如果存在该字符串，则汇编器优化器不会删除对该存储单元看起来多余的访问。

该注释字符串也可以在手写汇编源代码中使用，以对于在汇编代码中定义和访问的存储单元实现相同的作用。

注释字符串 ;wreg free 放置在一些 CALL 指令上。该字符串指示未在 W 寄存器中装入函数参数；即，未在使用它。如果存在该字符串，则可能会对在函数调用之前向 W 寄存器装入内容的多余汇编指令进行优化。

6.4.5 常量

6.4.5.1 数字常量

汇编器会以有符号 32 位精度执行所有算术运算。

所有数字的默认基数为 10。可以通过尾随一个基数说明符 （如表 6-3 所列）来指定其他基数。

表 6-3： ASPIC[18] 数字和基数

基数	格式
二进制	数字 0 和 1 后面跟随 B
八进制	数字 0 至 7 后面跟随 O、 Q、 o 或 q
十进制	数字 0 至 9 后面跟随 D 或 d，或不跟随任何字符
十六进制	数字 0 至 9、 A 至 F，前面加上 0x 或后面跟随 H 或 h

十六进制数字必须具有一个前导数字 （如 0ffffh），以将它们与标识符区分开。大写或小写形式的十六进制数字均可接受。

请注意，二进制常量必须跟随大写 B，因为小写 b 用于临时 （数字）标号反向引用。

在表达式中，将以常规的格式接受实数，并将它解释为 IEEE 32 位格式。

6.4.5.2 字符常量和字符串

字符常量是包含在一对单引号 ' 中的单个字符。

多字符常量或字符串是包含在一对匹配引号内的字符 （不包括回车符或换行符）的序列。单引号 ' 或双引号 " 均可使用，但左引号和右引号必须相同。

© 2013 Microchip Technology Inc.

6.4.6 标识符

汇编标识符是用户定义的代表存储单元或数字的符号。符号可以包含任意数量的以下字符：字母、数字和特殊美元字符 $、问号 ? 和下划线 _。

标识符的第一个字符不能为数字。字母的大小写是有意义的，例如， Fred 与 fred 是不同的符号。以下给出了标识符的一些示例：

```
An_identifier
an_identifier
an_identifier1
$
?$_12345
```

标识符不能为汇编器伪指令、关键字或 psect 标志。

以至少一个下划线字符开头的标识符可以从 C 代码中访问。必须小心此类符号，以免它们与 C 代码标识符相互作用。不以下划线开头的标识符只能从汇编域访问。关于 C 域和汇编域之间的映射，请参见**第 5.12.3.1 节 “等效汇编符号”**。

6.4.6.1 标识符的意义

对于其他汇编器的用户，如果试图对标识符实现任何形式的数据类型定义，应注意该汇编器不对任何符号附加任何意义，也不对符号的用法作出任何限制或期望。

psect （程序段）的名称和普通符号占据独立的、重叠的名称空间，但除此之外，汇编器不关心符号是用于表示字节、字还是 “跑车”。不需要也未提供特殊的语法来定义位或任何其他数据类型的地址，如果在多个上下文中使用了某个符号，汇编器也不会发出任何警告。指令和寻址模式语法将提供汇编器生成正确代码所需的所有信息。

6.4.6.2 汇编器生成的标识符

如果在某个宏块中使用了 LOCAL 伪指令，则在该宏的每个扩展中，汇编器将生成一个惟一符号来替换每个指定的标识符。这些惟一符号将具有??*nnnn*的形式，其中的*nnnn*是一个 4 位数字。用户应避免定义具有相同形式的符号。

6.4.6.3 位置计数器

活动程序段内的当前位置可通过符号 $ 访问。该符号会扩展为当前正在执行的指令的地址 （这不同于执行该指令时程序计数器 （Program Counter， PC）寄存器所包含的地址）。因而：

```
GOTO $   ;endless loop
```

表示将跳转到自身并构成无限循环。通过使用该符号和偏移量，可以指定相对跳转目标。

与 $ 相加的所有地址偏移量都具有目标器件的本机寻址能力。所以对于低档和中档器件，偏移量是相对于当前位置的指令数量，因为这些器件具有可字寻址的程序存储器。对于 PIC18 指令（使用可字节寻址的程序存储器），该符号的偏移量代表相对于当前位置的字节数。由于 PIC18 指令必须按字对齐，相对于位置计数器的偏移量应为 2 的倍数。所有偏移量均向下舍入到最接近的 2 的倍数。例如：

```
GOTO   $+2   ;skip...
MOVLW  8     ;to here for PIC18 devices, or
MOVWF  _foo  ;to here for baseline and mid-range devices
```

在低档或中档器件上将跳过 MOVLW 指令。在 PIC18 器件上， GOTO $+2 将跳转到随后的指令处；即，行为类似于 NOP 指令。

6.4.6.4 寄存器符号

汇编模块中的代码可以通过包含预定义的汇编头文件来访问特殊功能寄存器。可以通过向汇编源文件添加以下行来包含相应的文件：

```
#include <xc.inc>
```

请注意，该文件必须使用 C 预处理器伪指令包含，因而在编译时必须使能预处理汇编文件的选项，请参见**第 4.8.11 节 “-P：预处理汇编文件”**。该头文件包含相应的命令，以确保在源文件中包含特定于目标器件的头文件。

这些头文件包含所有字节或多字节大小寄存器的 EQU 声明，以及字节寄存器内指定位的 #define 宏。

6.4.6.5 符号标号

标号是一个赋予值等于当前 psect 内当前地址的符号别名。标号直到链接时才会赋予一个值。

标号定义由任意有效汇编标识符后跟一个 *冒号* : 组成。定义可以自己单独一行出现，也可以位于语句之前。以下给出了两个穿插在汇编代码中的合法标号的示例。

```
frank:
    MOVLW   1
    GOTO    fin
simon44: CLRF _input
```

此处，标号 frank 最终将被赋予 MOVLW 指令的地址， simon44 将被赋予 CLRF 指令的地址。无论它们如何定义，汇编器生成的汇编列表文件将总是单独一行显示它们。

请注意，标号后的冒号对于 PIC18 汇编代码是必需的，但对于所有其他器件的汇编代码是建议使用的。无法解释为指令的符号都会被假定为标号。错误输入的汇编指令有时会被视为标号而不发出错误消息。因而，以下代码：

```
mistake:
    MOVLW 23h
    MOVWF 37h
    REUTRN     ; oops
```

会定义名为 REUTRN 的符号，而其本意是用作 RETURN 指令。PIC18 汇编代码不会发生这种情况，因为冒号是必需的；在到达包含 REUTRN 的行时，编译器将报告错误。

© 2013 Microchip Technology Inc.

在汇编代码中可以使用 （并最好使用）标号，而不是在其他指令中使用绝对地址。因此，它们可以用作跳转型指令的目标位置，或用于将某个地址装入寄存器。

类似于变量，标号也具有作用域。默认情况下，可以在定义它们的模块中的任意位置使用它们。它们可以由位于其定义之前的代码使用。要使标号可在其他模块中访问，请使用 GLOBAL 伪指令。更多信息，请参见**第 6.4.9.1 节 “GLOBAL”**。

6.4.7 表达式

指令和伪指令的操作数由表达式组成。表达式可以包含数字、标识符、字符串和操作符。

操作符可以为一元 （一个操作数，如 not）或二元 （两个操作数，如 +）。表 6-4 列出了表达式中允许的操作符。

表 6-4： ASPIC[18] 操作符

操作符	用途	示例
*	乘法	MOVLW 4*33,w
+	加法	BRA $+1
-	减法	DB 5-2
/	除法	MOVLW 100/4
= 或 eq	等于	IF inp eq 66
> 或 gt	有符号大于	IF inp > 40
>= 或 ge	有符号大于或等于	IF inp ge 66
< 或 lt	有符号小于	IF inp < 40
<= 或 le	有符号小于或等于	IF inp le 66
<> 或 ne	有符号不等于	IF inp <> 40
low	操作数的低字节	MOVLW low(inp)
high	操作数的高字节	MOVLW high(1008h)
highword	操作数的高 16 位	DW highword(inp)
mod	求模	MOVLW 77mod4
&	按位与	CLRF inp&0ffh
^	按位异或	MOVF inp^80,w
|	按位或	MOVF inp|1,w
not	按位取反	MOVLW not 055h,w
<< 或 shl	左移	DB inp>>8
>> 或 shr	右移	MOVLW inp shr 2,w
rol	循环左移	DB inp rol 1
ror	循环右移	DB inp ror 1
float24	24 位形式的实数操作数	DW float24(3.3)
nul	测试宏参数是否为空值	

控制表达式语法的常规规则适用。

所列出的操作符全都可以在常量和可重定位表达式中自由地组合使用。链接器允许对复杂表达式进行重定位，因而涉及到可重定位标识符的表达式的结果可能直到链接时才会求解得到。

6.4.8 程序段

程序段或 psect 只是一段代码或数据。它们是一种将程序的各个部分组合在一起的方法（通过 psect 的名称），即使源代码在源文件中可能在物理上并不相邻或甚至分布在几个模块中。关于 psect 的介绍指南，请参见**第 5.15.1 节 “程序段”**。

psect 通过名称标识，并具有多个属性。 PSECT 汇编器伪指令用于定义 psect。它可接受一个名称和一个（可选）标志的*逗号*分隔列表作为参数。关于代码生成器定义的所有 psect 的列表，请参见**第 5.15.2 节 “编译器生成的 psect”**。**第 7 章 “链接器”**提供了关于链接器操作和可用于控制在存储器中放置 psect 的选项的更多信息。

汇编器不会赋予 psect 名称任何意义，链接器也不会知道哪些 psect 是编译器生成的，哪些是用户定义的。除非定义为 abs （绝对）类型，否则 psect 将是可重定位的。

未显式放置到一个 psect 中的代码或数据将成为默认 （未命名） psect 的一部分。

6.4.9 汇编器伪指令

汇编器伪指令或伪操作的使用方式类似于指令助记符。 PAGESEL 和 BANKSEL 属于例外，这两条伪指令不会生成指令。DB、DW 和 DDW 伪指令会将数据字节放入当前 psect。表 6-5 列出了这些伪指令，后面几节对它们进行了详细介绍。

表 6-5： ASPIC[18] 汇编器伪指令

伪指令	用途
GLOBAL	使符号可供其他模块访问或允许引用其他模块的符号
END	结束汇编
PSECT	声明或恢复程序段
ORG	设置当前 psect 内的位置计数器
EQU	定义符号值
SET	定义或重新定义符号值
DB	定义常量字节
DW	定义常量字
DDW	定义双倍宽度的常量字 （仅限 PIC18 器件）
DS	保留存储
DABS	定义绝对存储
IF	条件汇编
ELSIF	备选条件汇编
ELSE	备选条件汇编
ENDIF	结束条件汇编
FNCALL	指示链接器一个函数调用另一个
FNROOT	指示链接器某个函数是调用图的 “根”
MACRO	宏定义
ENDM	结束宏定义
LOCAL	定义局部标签
ALIGN	使输出在指定边界处对齐
BANKSEL	生成用以选择操作数存储区的代码

© 2013 Microchip Technology Inc.

表 6-5： ASPIC[18] 汇编器伪指令 （续）

伪指令	用途
PAGESEL	生成用以设置该页 PCLATH 位的置 1/ 清零指令
PROCESSOR	定义该文件进行汇编所针对的特定芯片
REPT	重复代码块 *n* 次
IRP	使用列表重复代码块
IRPC	使用字符列表重复代码块
SIGNAT	定义函数签名

6.4.9.1 GLOBAL

GLOBAL 伪指令用于声明以逗号分隔的符号的列表。如果这些符号是在当前模块中定义的，则它们会被设为公共符号。如果这些符号不是在当前模块中定义的，则它们将可用于引用在外部模块中定义的公共符号。因而，要在两个模块中使用同一符号，必须至少使用GLOBAL伪指令两次：一次是在定义该符号的模块中，用以将该符号设为公共符号；另一次是在使用该符号的模块中，用以链接外部定义。

例如：

```
GLOBAL  lab1,lab2,lab3
```

6.4.9.2 END

END 伪指令是可选的，但如果存在，则应位于程序的最末尾。它将终止汇编，其后不能跟随任何内容，即使是空行。

如果将某个表达式作为参数提供，则将使用该表达式来定义程序的入口点。它存储在汇编器所生成的目标文件的起始记录中。它是否有任何用途将取决于链接器。

编译器定义的默认运行时启动代码将包含带有起始地址的END伪指令。由于每个项目只能指定一个起始地址，所以通常不需要定义该地址，但可以在任何文件中使用不带入口点的 END 伪指令。

例如：

```
END  start_label  ;defines the entry point
```

或

```
END               ;do not define entry point
```

6.4.9.3 PSECT

PSECT 伪指令用于声明或恢复某个程序段。关于 psect 的介绍指南，请参见**第 5.15.1 节 “程序段”**。

该伪指令可接受一个名称和一个（可选）标志的*逗号*分隔列表作为参数。表 6-6 列出了允许的标志，并指明了 psect 的属性。

psect 名称处于独立于普通汇编符号的名称空间中，所以 psect 可以使用与普通汇编标识符相同的标识符。但是， psect 名称不能为汇编器伪指令、关键字或 psect 标志。

声明某个 psect 之后，稍后可以使用另一条 PSECT 伪指令恢复它；但此时不需要重复指定标志，它们将从先前的声明中传递。如果遇到两条 PSECT 伪指令具有相互冲突的标志，则将产生错误。

表 6-6： PSECT 标志

标志	含义
abs	psect 是绝对的
bit	psect 存放位对象
class=*name*	指定 psect 的类名
delta=*size*	寻址单元的长度
global	psect 是全局的 （默认）
limit=*address*	psect 的高端地址限制
local	psect 不是全局的
merge=*allow*	允许或阻止合并该 psect
ovrld	psect 将与其他模块中的相同 psect 重叠
pure	psect 将是只读的
reloc=*boundary*	psect 在指定边界处开始
size=*max*	psect 的最大长度
space=*area*	表示 psect 将驻留的区域
split=*allow*	允许或阻止拆分该 psect
with=*psect*	将 psect 放入与指定 psect 相同的页

以下给出了一些使用 PSECT 伪指令的示例：

```
PSECT fred
PSECT bill,size=100h,global
PSECT joh,abs,ovrld,class=CODE,delta=2
```

6.4.9.3.1 abs

abs 标志用于将当前 psect 定义为绝对的；即，它从地址 0 开始。这并不意味着该模块在 psect 中产生的内容将从 0 开始，因为其他模块也可能在同一 psect 中产生内容。另请参见**第 6.4.9.3.9 节 “ovrld”**。

带有 abs 标志的 psect 是不可重定位的，如果发出尝试将这种 psect 放在任意位置的链接器选项，将产生错误。

6.4.9.3.2 bit

bit 标志指定 psect 存放 1 位长的对象。这种 psect 的 scale（比例）值将为 8，指示每个存储字节具有 8 个可寻址单元，与该 psect 关联的所有地址均为位地址，而不是字节地址。映射文件中会指示该比例值，请参见**第 7.4 节 “映射文件”**。

© 2013 Microchip Technology Inc.

6.4.9.3.3 class

`class` 标志指定该 psect 对应的链接器类名。类是一个可在其中放置 psect 的地址范围。

类名可用于在链接时定位局部 psect，因为它们并不总是可以通过 `-P` 链接器选项按它们自己的名称进行引用（如果具有多个同名的局部 psect，就会如此）。

需要仅将 psect 定位到某个地址范围中的任意位置，而不是某个特定地址时，类名也很有用。将某个类与某个已定义的 psect 关联通常意味着不需要提供定制的链接器选项即可将其放置在存储器中。

关于如何定义链接器类的信息，请参见**第 7.2.1 节 “-Aclass =low-high,...”**。

6.4.9.3.4 delta

`delta` 标志用于定义可寻址单元的长度。即，与每个地址相关联的数据字节数。

对于 PIC 中档和低档器件，程序存储空间是可字寻址的，因而该空间中的 psect 使用的 delta（增量）必须为 2。即，程序存储器中的每个地址在 HEX 文件中都需要 2 个数据字节来定义其内容。因此， HEX 文件中的地址将与程序存储器中的地址不匹配。

这些器件上的数据存储空间是可字节寻址的，因而该空间中的 psect 使用的 delta 必须为 1。这是默认的 delta 值。

PIC18 器件上的所有存储空间都是可字节寻址的，所以应对于这些器件上的所有 psect 使用 delta 值 1（默认值）。

使用相互冲突的 delta 值重新定义 psect 可能导致汇编器发出阶段错误。

6.4.9.3.5 global

定义为全局（`global`）类型的 psect 将在链接时与其他同名的全局 psect 进行合并。psect 将从进行链接的所有模块中进行组合。

除非使用了 `local` 标志，否则 psect 在默认情况下被视为全局类型。

6.4.9.3.6 limit

`limit` 标志用于指定 psect 可延伸到的最高地址的限制。如果将它定位到存储器中时超出该限制，将产生错误。

6.4.9.3.7 local

定义为局部（`local`）的 psect 在链接时不会与来自其他模块的其他局部 psect 进行合并，即使存在其他同名的局部 psect。如果在一个模块中存在两个局部 psect，它们将引用相同的 psect。局部 psect 不能与任何全局 psect 同名，即使它是处于另一个模块中。

6.4.9.3.8 merge

该标志可以赋值为 0 或 1，或不指定。当赋值为 0 时，汇编优化器在优化过程将永远不会合并 psect。如果赋值为 1，则 psect 可能被合并：如果其他 psect 属性允许该操作，并且优化器确定这样做存在优势。如果未指定该标志，则不会发生合并。

通常情况下，仅对基于代码的 psect（text psect）执行合并。

6.4.9.3.9 ovrld

定义为 ovrld 的 psect 会使每个模块生成的内容相互重叠，而不是在链接时进行连接。配合使用该标志与 abs 标志 （见**第 6.4.9.3.1 节 “abs”**）可定义真正的绝对 psect；即，在该 psect 中定义的所有符号都是绝对符号。

6.4.9.3.10 pure

pure 标志指示链接器，该 psect 不会在运行时修改，因此（举例来说）将放置在 ROM 中。该标志的用处有限，因为它依赖于实施它的链接器和目标系统。

6.4.9.3.11 reloc

reloc 标志用于指定将 psect 对齐到特定边界上。例如，标志 reloc=100h 将指定该 psect 的起始地址必须为 100h 的倍数。

PIC18 指令必须按字对齐，因而对于包含可执行代码的所有 psect，使用的 reloc 值必须为 2。所有其他段和所有其他器件的所有段可以使用默认 reloc 值 1。

6.4.9.3.12 size

size 标志用于为 psect 指定最大长度，例如 size=100h。从所有模块中组合 psect 之后，链接器将检查这一点。

6.4.9.3.13 space

space标志用于区分具有重叠地址但存在区别的存储区。定位到程序存储器和数据存储器的 psect 具有不同的 space 值；举例来说，它可以指示程序空间地址 0 是不同于数据存储器地址 0 的不同存储单元。

在所有 8 位 PIC 器件上，程序存储器使用的 space 值为 0，而数据存储空间使用的 space 值为 1。

具有分区数据空间的器件不使用不同的 space 值来标识每个存储区。对于该空间中的对象，将使用包含存储区编号的完整地址，因而可以惟一地标识每个存储单元。例如，存储区大小为 0x80 字节的器件将使用地址 0 至 0x7F 来表示存储区 0 中的对象，使用地址 0x80 至 0xFF 来表示存储区 1 中的对象，如此类推。

6.4.9.3.14 split

该标志可以赋值为 0 或 1，或不指定。当赋值为 0 时，汇编优化器在优化过程将永远不会拆分 psect。如果赋值为 1，则可能拆分 psect：如果其他 psect 属性允许进行拆分，并且 psect 太大，无法装入可用的存储器。如果未指定该标志，则将基于该 psect 是否可以合并来决定是否拆分该 psect，请参见**第 6.4.9.3.8 节 “merge”**。

6.4.9.3.15 with

with 标志用于将某个 psect 放置在与另一个 psect 相同的页中。例如，标志 with=text 将指定该 psect 应放置在与 text psect 相同的页中。

withtotal 一词指的是与其他 psect 放置一起的每个 psect 长度的总和。

© 2013 Microchip Technology Inc.

6.4.9.4 ORG

ORG 伪指令用于更改当前 psect 内的位置计数器的值。这意味着使用 ORG 设置的地址将相对于 psect 的基址，它直到链接时才会确定。

> **注：** 滥用 ORG 伪指令并*不会*将位置计数器移动到您指定的绝对地址处。只有放置该伪指令的 psect 是绝对且重叠时，才会将位置计数器移动到指定地址处。要将对象放置在特定地址处，可以将它们放置在它们自己的 psect 中，并使用链接器 -P 选项将该 psect 链接到所需的地址处，请参见**第 7.2.19 节 “-Pspec”**。在程序中通常不需要使用 ORG 伪指令。

ORG 的参数必须为绝对值或引用当前 psect 的值。在两种情况下，当前的位置计数器均设置为由参数确定的值。无法将位置计数器向后移动。例如：

```
ORG 100h
```

会将位置计数器移动到当前psect起始位置加100h处。实际位置只有到链接时才会知道。

要使用 ORG 伪指令将位置计数器设置为某个绝对值，必须从重叠的绝对 psect 内使用该伪指令。例如：

```
PSECT absdata,abs,ovrld
    ORG 50h
  ;this is guaranteed to reside at address 50h
```

6.4.9.5 EQU

该伪操作定义一个符号，并使其值等于一个表达式。例如

```
thomas EQU 123h
```

标识符 thomas 将赋值为 123h。只有符号先前未定义时， EQU 才是合法的。另请参见**第 6.4.9.6 节 “SET”**，它可用于重新定义值。

该伪指令执行与预处理器的 #define 伪指令类似的功能，请参见**第 5.14.1 节 “C 语言注释”**。

6.4.9.6 SET

该伪操作等效于 EQU （**第 6.4.9.5 节 “EQU”**），只是它允许重新定义符号，而不会产生错误。例如：

```
thomas SET 0h
```

该伪指令执行与预处理器的 #define 伪指令类似的功能，请参见**第 5.14.1 节 “C 语言注释”**。

6.4.9.7 DB

DB 伪指令用于以字节形式初始化存储单元。其参数是以*逗号*分隔的表达式列表，其中每个表达式将汇编为一个字节，并装入连续的存储单元中。

示例：

```
alabel: DB  'X',1,2,3,4,
```

如果程序存储器中地址单元的长度为 2 个字节 （对于低档和中档器件就是如此，请参见**第 6.4.9.3.4 节 “delta”**）， DB 伪操作将初始化一个字，其高字节设置为零。所以，以上示例将定义填充为以下字的字节。

```
0058 0001 0002 0003 0004
```

但在 PIC18 器件上 （PSECT 伪指令的 delta 标志应为 1），将不会发生填充，并且在 HEX 文件中将出现以下数据。

```
58 01 02 03 04
```

6.4.9.8 DW

DW 伪指令的工作方式类似于 DB，只是它会将表达式汇编为 16 位字。示例：

```
DW -1, 3664h, 'A'
```

6.4.9.9 DDW

DDW 伪指令的工作方式类似于 DW，只是它会将表达式汇编为双倍宽度 （32 位）的字。示例：

```
DDW 'd', 12345678h, 0
```

6.4.9.10 DS

该伪指令用于保留但不初始化存储单元。它的惟一参数是要保留的字节数。

该伪指令通常用于为基于 RAM 的对象在数据存储器中保留存储单元。如果在链接到程序存储器的 psect 中使用，它将会移动位置计数器，但不会在 HEX 文件输出中放入任何内容。请注意，由于程序存储器中地址单元的长度为 2 个字节 （见**第 6.4.9.3.4 节 “delta”**）， DS 伪操作实际上会保留整个字。

通常通过使用标号来定义变量，然后通过 DS 伪指令在标号位置处保留存储单元。

示例：

```
alabel: DS 23    ;Reserve 23 bytes of memory
xlabel: DS 2+3   ;Reserve 5 bytes of memory
```


© 2013 Microchip Technology Inc.

6.4.9.11 DABS

该伪指令用于在指定地址处保留一个或多个字节的存储器。该伪指令的一般格式为：

DABS *memorySpace*, *address*, *bytes*

其中，*memorySpace* 是一个编号，代表将保留的存储空间，*address* 是保留存储空间的地址， *bytes* 是要保留的字节数。

该伪指令与 DS 伪指令的不同之处在于它可以用于保留任意位置的存储器，而不仅仅是当前 psect 内的存储器。事实上，这些伪指令可以放置在汇编代码中的任意位置，而不会以任何方式在当前选定 psect 中产生内容。

存储空间编号与使用 space 标志选项为 psect 指定的编号相同 （见**第 6.4.9.3.13 节 “space”**）。

对于用户定义的每个绝对 C 变量，或对于代码生成器已分配地址的任何变量，代码生成器都会发出 DABS 伪指令。

链接器会从目标文件中读取此类与 DABS 相关的信息，并确保不将保留的地址用于其他存储器分配。

6.4.9.12 IF、 ELSIF、 ELSE 和 ENDIF

这些伪指令用于实现条件汇编。 IF 和 ELSIF 的参数应为绝对表达式。如果它不为零，则对从其后一直到下一个匹配 ELSE、 ELSIF 或 ENDIF 的代码进行汇编。如果表达式为零，则将跳过从其后到下一个匹配 ELSE 或 ENDIF 的代码。这些伪指令并未实现与 C 语句 if() 一样的运行时条件语句；它们在编译时进行求值。

在 ELSE 处，将反向解释条件编译的含义，而 ENDIF 将终止条件汇编块。

例如：

```
IF ABC
    GOTO aardvark
ELSIF DEF
    GOTO denver
ELSE
    GOTO grapes
ENDIF
```

在该示例中，如果 ABC 非零，则汇编第一条 GOTO 指令，但不汇编第二条或第三条。如果 ABC 为零， DEF 非零，则汇编第二条 GOTO 指令，但不汇编第一条和第三条。如果 ABC 和 DEF 均为零，则汇编第三条 GOTO 指令。请注意在以上示例中，只有一条 GOTO 指令会出现在输出中；具体哪一条将由赋予 ABC 和 DEF 的值决定。

条件汇编块可以进行嵌套。

6.4.9.13 MACRO 和 ENDM

这些伪指令用于定义汇编宏，并可以选择使用参数。关于值与标识符的简单关联，请参见**第 6.4.9.5 节 “EQU”**；关于预处理器的 #define 宏伪指令（它也可以使用参数），请参见**第 5.14.1 节 “C 语言注释”**。

MACRO 伪指令前面应为宏名，并 （可选）在后面跟随形参的*逗号*分隔列表。在使用宏时，应以与机器操作码相同的形式使用宏名，后面跟随用于替换形参的参数列表。

例如：

```
;macro: movlf
;args:  arg1 - the literal value to load
;       arg2 - the NAME of the source variable
;descr: Move a literal value into a nominated file register

movlf   MACRO   arg1,arg2
    MOVLW arg1
    MOVWF arg2 mod 080h
ENDM
```

在使用时，该宏将扩展为宏主体内的 2 条指令，并将形参替换为实参。因而：

```
    movlf 2,tempvar
```

将扩展为：

```
    MOVLW 2
    MOVWF tempvar mod 080h
```

& 字符可用于将宏参数与其他文本连接，但在实际扩展中会将它删除。例如：

```
loadPort MACRO port, value
    MOVLW value
    MOVWF PORT&port
ENDM
```

如果在调用时 port 为 A，则将装入 PORTA，如此类推。

通过使用两个分号 ;; 作为注释的开头，可以在宏扩展中禁止注释 （从而节省宏存储空间）。

在调用某个宏时，参数列表必须用*逗号*分隔。如果希望在参数中包含*逗号*（或其他分隔符，如空格），可以使用尖括号 < 和 > 将其括起。

如果参数前面具有一个百分号 %，则该参数将作为表达式进行求值，并以十进制数字而不是字符串的形式传递。如果宏主体内的参数求值会产生不同的结果，这将非常有用。

在宏中可以使用 nul 操作符来测试宏参数，例如：

```
IF nul     arg3  ; argument was not supplied.
    ...
ELSE             ; argument was supplied
    ...
ENDIF
```

关于在宏中使用惟一局部标号的信息，请参见**第 6.4.9.14 节 “LOCAL”**。

默认情况下，汇编列表文件将以未扩展格式显示宏；即，以调用宏的形式。通过使用 EXPAND 汇编器控制，可以在列表文件中显示宏的扩展形式，请参见**第 6.4.10.3 节 “EXPAND”**。

© 2013 Microchip Technology Inc.

6.4.9.14 LOCAL

LOCAL 伪指令用于为给定宏的每个扩展定义惟一标号。对于在 LOCAL 伪指令之后列出的所有符号，在对宏进行扩展时，都会被替换为汇编器生成的惟一符号。例如：

```
down MACRO count
    LOCAL more
    more: DECFSZ count
    GOTO more
ENDM
```

在进行扩展时，将包含汇编器生成的惟一标号来代替 more。例如：

```
down foobar
```

将扩展为：

```
??0001 DECFSZ foobar
    GOTO ??0001
```

如果第二次进行调用，标号 more 将扩展为 ??0002，并避免多次定义符号的错误。

6.4.9.15 ALIGN

ALIGN 伪指令会将其后的所有内容 （数据存储或代码等）对齐到当前 psect 内的指定偏移边界处。边界在伪指令之后以字节数的形式指定。

例如，要将输出对齐到 psect 内的 2 字节 （偶字节）地址处，可以使用以下代码。

```
ALIGN 2
```

请注意，如果 psect 以偶地址作为起始，之后的内容将仅以偶编号绝对地址作为起始；即，对齐是在当前 psect 内进行的。关于 psect 对齐，请参见**第 6.4.9.3.11 节 “reloc”**。

ALIGN 伪指令还可以用于确保 psect 的长度为某个特定数字的倍数。例如，如果将以上 ALIGN 伪指令放在某个 psect 的末尾，该 psect 的长度将总是为偶数个字节长。

6.4.9.16 REPT

REPT 伪指令用于临时定义一个未命名的宏，然后根据其参数对它进行多次扩展。

例如：

```
REPT 3
    ADDWF fred,w
ENDM
```

将扩展为：

```
    ADDWF fred,w
    ADDWF fred,w
    ADDWF fred,w
```

另请参见**第 6.4.9.17 节 “IRP 和 IRPC”**。

6.4.9.17 IRP 和 IRPC

IRP 和 IRPC 伪指令的工作方式类似于 REPT ；但是，它们不是将块重复固定的次数，而是对于参数列表的每个成员重复一次。

对于 IRP，其列表是一个常规的宏参数列表；对于 IRPC，它是一个参数中的每个字符。在每次重复时，都会将一个形参替换为实参。

例如：

```
IRP number,4865h,6C6Ch,6F00h
    DW number
ENDM
```

将扩展为：

```
    DW 4865h
    DW 6C6Ch
    DW 6F00h
```

请注意，您可以使用与常规宏相同的方式使用局部标号和尖括号。

IRPC 伪指令与之类似，只是它每次从一个非空格字符组成的字符串中替换一个字符。

例如：

```
IRPC char,ABC
    DB 'char'
ENDM
```

将扩展为：

```
    DB 'A'
    DB 'B'
    DB 'C'
```

6.4.9.18 BANKSEL

该伪指令可用于生成用以选择操作数存储区的代码。操作数应为处于数据存储器中的某个对象的符号或地址。请参见**第 6.4.1.2 节 “存储区和页选择”**。

6.4.9.19 PAGESEL

该伪指令可用于生成用以选择地址操作数页的代码。请参见**第 6.4.1.2 节 “存储区和页选择”**。

6.4.9.20 PROCESSOR

汇编器的输出可能因目标器件而不同。器件名称通常使用命令行驱动程序 xc8 的 --CHIP 选项设置（见**第 4.8.20 节 “--CHIP：定义器件”**），或使用汇编器的 --CHIP 选项设置（见**第 6.3.14 节 “--CHIP：指定器件名称”**），但也可以使用该伪指令设置，例如：

```
PROCESSOR 16F877
```

该伪指令将覆盖通过任何命令行选项选择的任何器件。

© 2013 Microchip Technology Inc.

6.4.9.21 SIGNAT

该伪指令用于将一个 16 位签名值与某个标号相关联。在链接时，链接器会检查为特定标号定义的所有签名是否相同，如果它们不同，将产生一个错误。 MPLAB XC8 使用 SIGNAT 伪指令来强制在链接时检查 C 函数原型和调用约定。

如果要编写从 C 中调用的汇编语言程序，请使用 SIGNAT 伪指令。例如：

```
SIGNAT _fred,8192
```

会将签名值 8192 与符号 _fred 关联。如果任何目标文件中存在 _fred 的不同签名值，链接器将报告错误。

确定程序的正确签名值的最简单方法是编写与汇编程序具有相同原型的 C 程序，并检查由代码生成器确定的签名值。这将显示在汇编列表文件中，请参见**第 4.8.16 节“--ADDRQUAL：设置编译器对于存储器限定符的响应”**和**第 6.5 节“汇编代码级优化”**。

6.4.10 汇编器控制

可以通过在汇编程序源代码中包含汇编器控制来控制汇编器操作。这些关键字在程序中的其他地方没有任何意义。控制通过在伪指令 OPT 后面跟随控制名称来进行调用。一些关键字会跟随一个或多个参数。例如：

```
OPT EXPAND
```

表 6-7 列出了这些关键字，并在后面进一步介绍了其中每个关键字。

表 6-7： ASPIC[18] 汇编器控制

控制	含义	格式
ASMOPT_ON, ASMOPT_OFF	启动和停止汇编优化	OPT ASMOPT_OFF ;protected code OPT ASMOPT_ON
COND*, NOCOND	在列表中包含 / 不包含条件代码	OPT COND
EXPAND, NOEXPAND	在列表输出中扩展 / 不扩展宏	OPT EXPAND
INCLUDE	以文本方式包含另一个源文件	OPT INCLUDE < *pathname* >
LIST*, NOLIST	定义列表输出的选项 / 禁止列表输出	OPT LIST [< *listopt* >, ..., < *listopt* >]
PAGE	在列表输出中开始新的一页	OPT PAGE
SPACE	向列表中添加空行	OPT SPACE 3
SUBTITLE	指定程序的子标题	OPT SUBTITLE “< *subtitle* >”
TITLE	指定程序的标题	OPT TITLE “< *title* >”

注 1： 列出的带有星号 （*）的选项为默认选项

6.4.10.1 ASMOPT_OFF 和 ASMOPT_ON

这些控制用于对于某些汇编代码段选择性地禁止汇编器优化器。在 ASMOPT_OFF 控制之后不会修改任何代码，直到遇到后续的 ASMOPT_ON 控制为止。

6.4.10.2 COND

列表输出中将包含所有条件代码。另请参见**第 6.4.10.6 节 “NOCOND”**中的 NOCOND 控制。

6.4.10.3 EXPAND

在 EXPAND 生效时，列表输出中将显示通过宏扩展生成的代码。另请参见**第 6.4.10.7 节 “NOEXPAND”**中的 NOEXPAND 控制。

6.4.10.4 INCLUDE

该控制将导致在汇编文件的此处以文本方式包含 *pathname* 指定的文件。INCLUDE 控制必须为行中的最后一个控制关键字，例如：

```
OPT INCLUDE "options.h"
```

驱动程序不会向汇编器传递任何搜索路径，所以如果包含文件不位于工作目录中，路径名必须指定确切的位置。

另请参见**第 4.8.11 节 “-P：预处理汇编文件”**中的驱动程序选项 -P，它会强制 C 预处理器对汇编文件进行预处理，从而允许使用预处理器伪指令，例如 #include（见**第 5.14.1 节 “C 语言注释”**）。

6.4.10.5 LIST

如果先前使用 NOLIST 控制关闭了列表， LIST 控制将独立地开启列表。

此外， LIST 控制还可以包含用于控制汇编和列表的选项。表 6-8 列出了这些选项。

表 6-8： LIST 控制选项

列表选项	默认	说明
c= *nnn*	80	设置页面（即列）的宽度。
n= *nnn*	59	设置页面长度。
t= *ON*|OFF	OFF	截断列表输出行。默认值将折行。
p=< *device* >	n/a	设置器件类型。
r=< *radix* >	HEX	将基数设置为 HEX、 dec 或 oct。
x= *ON*|OFF	OFF	开启或关闭宏扩展。

另请参见**第 6.4.10.8 节 “NOLIST”**中的 NOLIST 控制。

© 2013 Microchip Technology Inc.

6.4.10.6 NOCOND

使用该控制将阻止将条件代码包含在汇编列表文件输出中。另请参见**第 6.4.10.2 节 “COND”** 中的 COND 控制。

6.4.10.7 NOEXPAND

NOEXPAND 控制用于禁止汇编列表文件中的宏扩展。将改为列出宏调用。另请参见**第 6.4.10.3 节 “EXPAND”** 中的 EXPAND 控制。**第 6.4.9.13 节 “MACRO 和 ENDM”** 对汇编宏进行了介绍。

6.4.10.8 NOLIST

该控制用于从此处起关闭列表输出。另请参见**第 6.4.10.5 节 “LIST”** 中的 LIST 控制。

6.4.10.9 NOXREF

NOXREF 控制将禁止生成*原始*的交叉引用文件。另请参见**第 6.4.10.14 节 “XREF”** 中的 XREF 控制。

6.4.10.10 PAGE

PAGE 控制会导致在列表输出中开始新的页面。在源代码中遇到 Control-L（换页）字符时，也会导致新的页面。

6.4.10.11 SPACE

SPACE 控制将在列表输出中放入其参数所指定数量的空行。

6.4.10.12 SUBTITLE

SUBTITLE 控制用于定义一个出现在每个列表页面顶部，但处于标题下的子标题。应将该字符串括在单引号或双引号中。另请参见**第 6.4.10.13 节 “TITLE”** 中的 TITLE 控制。

6.4.10.13 TITLE

该控制关键字定义要出现在每个列表页面顶部的标题。应将该字符串括在单引号或双引号中。另请参见**第 6.4.10.12 节 “SUBTITLE”** 中的 SUBTITLE 控制。

6.4.10.14 XREF

XREF 控制等效于驱动程序命令行选项 --CR（见**第 4.8.23 节 “--DEBUGGER：选择调试器类型”**）。它会导致汇编器生成原始的交叉引用文件。实用程序 CREF 应用于实际生成格式化的交叉引用列表。

6.5 汇编代码级优化

除了由代码生成器直接对 C 代码执行的那些优化之外，汇编器也会对汇编代码执行优化，请参见**第 5.13 节 “优化”**。

默认情况下，xc8 驱动程序会指示汇编器对基于 C 代码生成的汇编代码进行优化，但不对手写汇编源代码模块执行优化。如果需要，也可以对后一种代码进行优化，请参见**第4.8.42 节 “--OPT：调用编译器优化”**。嵌入到 C 代码中的汇编代码（见**第 5.12.2 节 “#asm、#endasm 和 asm()”**）从不进行优化。

汇编器可执行的优化包括以下。

内联小程序，从而可以不需要调用该程序。该优化只会更改极小的程序（通常为几条指令），以便不影响代码长度。这可以加快代码执行速度，而不会显著增加代码长度。

过程抽象，对出现不止一次的汇编代码序列执行该优化。这实际上是一个反向内联过程。代码序列将被抽象到一个可调用程序中，并使用标号 PL*x*；其中，*x* 是一个编号；原始代码序列的每个实例都会被替换为对该程序的调用。这可以显著地减小代码长度，并会对代码速度产生微小的影响。

跳转至跳转类型的优化，主要用于整理与遵循某个通用模板且包含跳转到其他跳转指令的跳转指令的条件代码序列有关的输出。

无法到达的代码会被删除。代码可能会由于其他优化而变为孤立代码，无法在正常执行期间到达，例如返回指令之后的指令。所有标号都会被视为可能的入口点，标号之后的代码总是被视为可到达的。

窥孔优化，对每条指令执行。这些优化会考虑每条指令处的执行状态以及紧邻每条指令周围的执行状态（从而得此名称）。它们会在每个步骤更改或删除一条或多条指令。例如，如果已知 W 包含值 0，而某条指令将 W 传送到某个地址（MOVWF），这可能可以替换为 CLRF 指令。

psect 合并，可能可以通过执行该优化来实现其他优化。该优化将确保处于同一 psect 中的代码被定位到相同的程序存储器页，从而使 psect 内的调用和跳转不需要在执行之前设置页选择位。使用 LJMP 和 FCALL 指令的代码将可以从该优化中获益，请参见**第 6.4.1 节 “汇编指令差异”**。

汇编优化通常会影响工具（如 MPLAB IDE）中的调试，所以在调试代码时可能需要禁止它们（如可能），请参见**第 4.8.42 节 “--OPT：调用编译器优化”**。汇编器优化可以显著地减小代码长度，但通常对 RAM 使用影响甚微。

© 2013 Microchip Technology Inc.

6.6 汇编列表文件

汇编器会生成汇编列表文件（如果得到指示）。xc8 驱动程序选项 --ASMLIST 通常用于请求生成此类文件，请参见**第 4.8.17 节 “--ASMLIST：生成汇编列表文件”**。

汇编列表文件会显示编译器对于 C 和汇编源代码生成的汇编输出。如果使能了汇编器优化器，则汇编输出可能会与汇编源代码不同，因而对于汇编编程仍然是很有用的。

列表文件采用人们可阅读的格式，它无法进一步参与编译序列的任何部分。它与汇编输出文件的不同之处在于它包含地址和操作码数据。此外，汇编器优化器会简化一些表达式，并为清晰起见从列表文件中删除某些汇编器伪指令，虽然真正的汇编输出文件中会包含这些伪指令。如果使用汇编列表文件来查看由编译器生成的代码，则您可能希望关闭汇编器优化器，以便在该文件中显示编译器生成的所有伪指令。在继续开发时重新使能优化器。**第 4.8.42 节 “--OPT: 调用编译器优化”**提供了关于控制优化器的更多信息。

假设链接阶段已成功结束，链接器将会更新列表文件，使它包含绝对地址和符号值。因而，可以使用汇编列表文件来确定指令的位置和确切的操作码。

汇编器会对于传递给它的每个汇编文件生成一个汇编列表文件，因而将会对于项目中的所有 C 源代码 （包括基于 p 代码的库代码）生成一个文件。该文件还将包含一些 C 初始化代码，构成运行时启动代码的一部分。此外，将为每个汇编源文件生成一个文件。在每个项目中通常至少会有一个汇编文件，其中包含通常称为 startup.as 的运行时启动文件的部分内容。

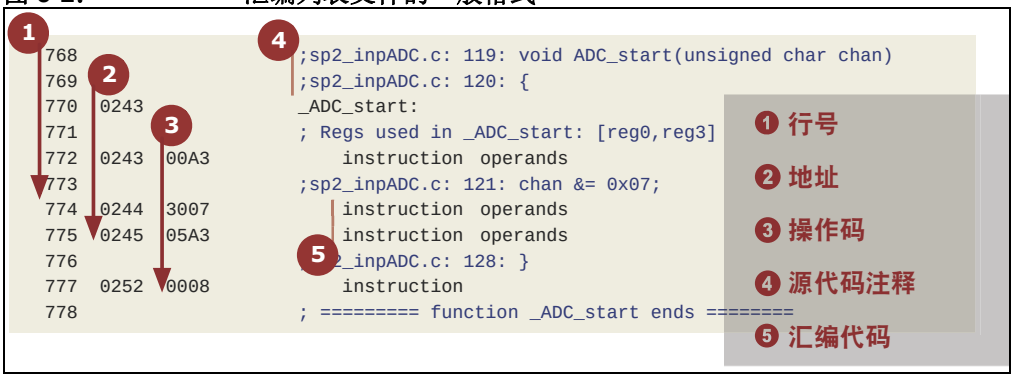
6.6.1 一般格式

图 6-1 显示了主列表所具有的格式。

行号仅仅与汇编列表文件有关，与 C 或汇编源文件中的行号无关。如果任何汇编代码以分号开头，则表示它是由代码生成器添加的注释。这种注释或者包含对应于所生成汇编代码的原始源代码，或者是由代码生成器插入的用于说明已执行的某个操作的注释。

在每个函数的输出之前，是由代码生成器总结的关于该函数的详细信息。该信息涉及到寄存器使用情况、局部变量信息、所调用的函数和调用它的函数。

图 6-1： 汇编列表文件的一般格式



6.6.2 psect 信息

汇编列表文件可以用于确定数据对象或代码段所放入的 psect 的名称（由编译器放入其中）。对于标号 （符号），请查看该文件末尾的符号表。它会指示它所在的 psect 的名称，以及与符号关联的地址。

对于其他代码，请在列表文件中查找代码。您通常可以搜索与代码相关联的 C 语句。查找该代码上面的第一条 PSECT 汇编器伪指令。与第一条伪指令关联的名称是放置代码的 psect 的名称，请参见**第 6.4.9.3 节 “PSECT”**。

6.6.3 函数信息

对于每个 C 函数，在函数的汇编标号 （搜索函数名紧跟一个冒号 :）之前打印的是有关函数所使用的资源的一般信息。**图 6-2：“函数信息”**显示了典型的打印内容。大部分信息都是不言自明的，但后面会跟随一些特别说明。

所示的存储单元使用格式 *offset*[*space*]。例如， 42[BANK0] 存储单元表示变量位于存储区 0 存储空间中，并且它出现在该空间中的编译堆栈部分的 42 字节偏移位置处，请参见**第 5.5.2.2.1 节 “编译堆栈操作”**。

每当显示指针变量时，它们往往伴随着指针可以引用的目标，位于箭头 -> 之后，请参见**第 6.6.5 节 “指针引用图”**。该信息的自动变量和参数部分特别有用，因为指针的长度是动态的，请参见**第 5.4.5 节 “指针类型”**。该信息会显示分配给每个指针变量的实际字节数。

被跟踪的对象通常不使用。它指示当前选定的 RAM 存储区在进入函数和在其退出点时的已知状态。它还会指示在函数中更改或未更改的存储区选择位。

硬件堆栈信息会显示该函数单独占用了多少个堆栈层级，以及该函数和它调用的任何函数使用的总层级数。

使用不可重入模型的函数是那些将自动和参数变量分配到编译堆栈中的函数，因此是不可重入的。

© 2013 Microchip Technology Inc.

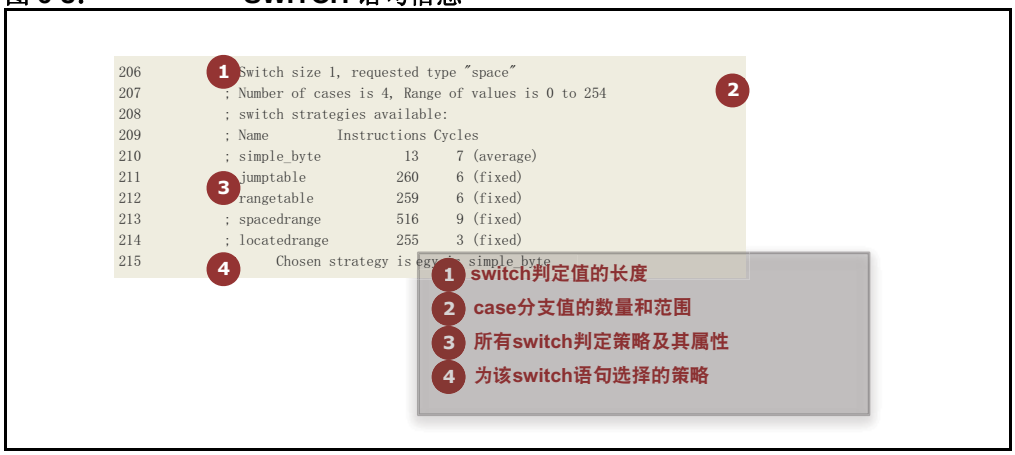
图 6-2：　　函数信息


4064 ********** function _render **********
4065 ;; Defined at:
4066 ;; line 29 in file "draw.c"
4067 ;; Parameters: Size Location Type
4068 ;; None
4069 ;; Auto vars: Size Location Type
4070 ;; lll 4 42[BANK0] long
4071 ;; x 2 46[BANK0] volatile int
4072 ;; cp 1 41[BANK0] PTR unsigned char
4073 ;; -> inputData(2),
4074 ;; Return value: Size Location Type
4075 ;; None void
4076 ;; Registers used:
4077 ;; wreg, fsr0l, fsr0h, status,2, status,0, pclath, cstack
4078 ;; Tracked objects:
4079 ;; On entry : 17F/0
4080 ;; On exit : 0/0
4081 ;; Unchanged: FFE00/0
4082 ;; Data sizes: COMMON BANK0 BANK1 BANK2
4083 ;; Params: 0 0 0 0
4084 ;; Locals: 0 7 0 0
4085 ;; Temps: 0 5 0 0
4086 ;; Totals: 0 12 0 0
4087 ;;Total ram usage: 12 bytes
4088 ;; Hardware stack levels used: 1
4089 ;; Hardware stack levels required when called: 4
4090 ;; This function calls:
4091 ;; _lrv
4092 ;; ___altofl
4093 ;; ___awdiv
4094 ;; ___awmod
4095 ;; This function is called by:
4096 ;; _main
4097 ;; This function uses a non-reentrant model
1 函数的名称
2 定义的文件名和行号
3 参数的长度、位置和类型
4 自动变量的长度、位置和类型
5 返回值的长度、位置和类型
6 函数代码使用的寄存器
7 进入和退出时选择的GPR存储区
8 整个函数的RAM存储器摘要
9 硬件堆栈要求
10 该函数调用的函数
11 哪些函数调用了该函数
12 该函数的编码方式

6.6.4 switch 语句信息

与每条 switch 语句的生成代码一起显示的是关于该语句如何进行编码的信息。编译器可以对于 switch 语句使用几种策略。编译器将确定适当的策略（见**第 5.6.3 节“switch 语句”**），或者您也可以使用 pragma 伪指令（见**第 5.14.4.10 节“#pragma switch 伪指令”**）来指示对于特定类型策略的偏好。打印的信息将类似图 6-3 所示的形式。

图 6-3： SWITCH 语句信息



6.6.5 指针引用图

汇编列表文件中包含的另一项重要信息是指针引用图（在列表文件中查找 *pointer list with targets:*（带目标的指针列表））。它会列出程序中包含的每个指针，以及指针可以在整个程序中引用的每个目标。除了指针变量本身的长度和类型之外，还会指示每个目标的长度和类型。

例如，以下显示了 C 代码中名为 task_tmr 的指针，它是函数 timer_intr() 的局部指针。它是一个指向 unsigned int 类型的指针，它为一个字节宽。该指针只有一个目标，它是名为 task 的结构中的成员 timer_count。该目标变量驻留在 BANK0 类中，它为两个字节宽。

```
timer_intr@task_tmr   PTR unsigned int  size(1); Largest target is 2
                          -> task.timer_count(BANK0[2]),
```

指针引用图会同时显示指向数据对象的指针和指向函数的指针。

© 2013 Microchip Technology Inc.

6.6.6 调用图

汇编列表文件中的另一个重要信息块是调用图（在列表文件中查找 *Call Graph Tables:*（调用图表））。它是针对使用编译堆栈的目标器件生成的，以方便局部变量（如函数参数和自动变量）的使用。关于编译堆栈操作的更多详细信息，请参见**第 5.5.2.2.1 节“编译堆栈操作”**。

列表文件中的调用图会显示由代码生成器整理及分析的信息，它们主要用于支持对函数的自动参数块（APB）进行重叠。通过研究调用图，可以获得以下信息。

- 程序中标记调用树顶部的“根”节点并自发调用的函数
- 链接器认为在程序执行过程中被调用或可能已被调用的函数
- 程序的函数调用的层次结构
- 每个函数的 APB 中的自动变量和参数区域的长度
- 每个函数的 APB 在编译堆栈中的偏移量
- 估算的调用树深度

6.6.6.1 以下将对这些特性进行讨论。

典型调用图的形式如图 6-4 所示。

图 6-4： 调用图格式

```
Call Graph Tables:

--------------------------------------------------------------------------------
(Depth) Function                Calls       Base Space   Used Autos Params    Refs
--------------------------------------------------------------------------------
(0) _main                                                  12    12      0   34134
                                           43 BANK0        5     5      0
                                            0 BANK1        7     7      0
                              _aOut
                            _initSPI
--------------------------------------------------------------------------------
(1) _aOut                                                   2     0      2      68
                                            2 BANK0        2     0      2
                                _SPI
                  _GetDACValue (ARG)
--------------------------------------------------------------------------------
(1) _initSPI                                                0     0      0       0
--------------------------------------------------------------------------------
(2) _SPI                                                    2     2      0      23
                                            0 BANK0        2     2      0
...

Estimated maximum stack depth 6
--------------------------------------------------------------------------------
```

该图以函数 `main()` 开始。请注意，函数名将总是以汇编形式显示，因此函数 `main()` 将显示为符号 `_main`。`main()` 总是为调用树的根。中断函数将构成独立的树。

其下将显示 main() 调用或可能调用的所有函数。它们在图中被组合到橙色框中。函数被包含到调用图中并不意味着会实际调用该函数，而是表示有可能会调用该函数。例如，如下代码：

```
int test(int a) {
  if(a)
    foo();
  else
    bar();
}
```

会将 foo() 和 bar() 列在 test() 下，因为两者都可能被调用。如果 a 总是为 true，那么将永远不会调用函数 bar()，即使它出现在调用图中。

除了这些函数之外，还包含与在 main() 的编译堆栈中分配的存储器有关的信息。该存储器将用于在 main() 中定义自动、临时和参数变量。自动和临时变量之间的惟一区别是自动变量由编程人员定义，临时变量由编译器定义，但两者的行为方式相同。

在 main() 的橙色框中，可以看到它定义了 10 个自动和临时变量。它未定义任何参数（main() 从不会有参数）。对于 main() 中的局部对象，在汇编代码中共进行了 24 次引用。

编译堆栈可以分解为多个组成部分，放置在多个存储空间中，以利用目标器件的所有可用存储器，而不是仅在一个存储空间中进行存储器分配。每个函数的存储器摘要行下会显示这种分解。在该示例中，显示了 main() 的一些局部对象放置在公共存储器中，但其他一些放置在存储区 0 数据 RAM 中。

Used（已用）列指示编译堆栈的每个部分使用了多少字节的存储器，*Space*（空间）列指示所放入的空间。*Base*（基）值指示该块在编译堆栈的相应部分中的偏移量。例如，该图指示 main() 在位于公共存储器中的编译堆栈部分中，在偏移量 4 处分配了 6 个字节的存储器。此外，它还在位于存储区 0 的编译堆栈组成部分中，在存储区 0 存储器的偏移量 16 处分配了 4 个字节的存储器。

在 main() 的信息下面（橙色框之外），可以看到对于 main() 调用的函数（即 rv()、rvx() 和 rvy()）重复了相同的信息。缩进用于指示函数在调用图中达到的最大深度。图中的箭头突出显示这种缩进。

在调用图中每个树之后，会指示该树可能实现的最大调用（堆栈）深度。它可以用作程序堆栈使用量的指示。由于几个原因，无法给出程序堆栈使用总量的明确值：

- 调用树的某些部分可能永远无法到达，这会减少树的堆栈使用量。
- 无法确定中断（或其他）树在 main() 树中产生的内容，因为无法预知中断（或其他函数调用）在 main 调用树中的发生点。
- 汇编器优化器可能会将函数调用替换为至函数的跳转，这会减少树的堆栈使用量。
- 汇编器的过程抽象优化可能会增加对所抽象程序的调用。（请进行检查，以确保它不会超出最大堆栈深度。）

© 2013 Microchip Technology Inc.

如果似乎已经超出最大堆栈深度，代码生成器也会产生一条警告。出于上述原因，该警告也仅用于指示堆栈的潜在问题。

6.6.6.2 调用图关键路径

在列表文件中，紧靠调用图表之前的是在调用图中确定的存储器使用情况的关键路径。对于每个存储空间和每个调用图，都会打印一条关键路径。查找类似 *Critical Paths under _main in BANK0* 形式的行，对于该示例，它表示 `main` 函数（一个调用图的根）在存储区 0 存储器中的关键路径。函数 `main` 存在一个调用图，每个中断函数存在另一个调用图，而对于器件定义的每个存储空间都会显示每个调用图的关键路径。

此处的关键路径代表以一个连续块的形式堆叠在一起的 APB 的最大范围。基本上，它会确定这样的函数：对于程序在该特定存储空间中使用的存储器，函数的 APB 使用了其中的存储器。如果可以减少这些函数在相应存储空间中的存储器使用量，则可以影响程序在该存储空间中的存储器使用总量。

该信息可能以如下形式显示。

```
3793 ;; Critical Paths under _main in BANK0
3794 ;;
3795 ;;   _main->_foobar
3796 ;;   _foobar->___flsub
3797 ;;   ___flsub->___fladd
```

在该示例中，显示了从函数 `main` 开始的所有调用图路径，其中 `main` 调用 `foobar`、`foobar` 调用 `flsub`、 `flsub` 再调用 `fladd` 的路径使用了存储区 0 RAM 中的最大存储器块。调用图表中会显示每个函数的确切存储器使用量。

函数使用的、但不处于关键路径中的存储器将与关键路径中的存储器完全重叠。降低它们的存储器使用量不会影响整个程序的存储器使用量。

注：

© 2013 Microchip Technology Inc.

第 7 章 链接器

7.1 简介

本章介绍链接器背后的理论以及链接器的用法。

链接器的应用程序名称为 HLINK。大多数情况下都不需要直接调用链接器，因为编译器驱动程序 xc8 会自动使用所有必需的参数执行链接器。直接使用链接器并不简单，通常只有那些具有扎实的编译器和链接知识的用户才应尝试这么做。编译器通常会对程序的链接方式作出一些假定。如果 psect 未正确链接，可能会导致代码失败。

如果直接使用链接器是绝对必要的，最佳的方法是复制由编译器驱动程序构造的链接器参数作为起点，然后适当地对其进行修改。这可以确保提供必需的启动模块和参数。

《MPLAB XC8 C 编译器用户指南》的这一章讨论了以下主题：

- 操作
- 重定位和 psect
- 映射文件

7.2 操作

链接器命令采用以下形式：

```
hlink [options] files
```

options 是零个或多个链接器选项，其中每个选项都会以某种方式修改链接器的行为。*files* 是一个或多个目标文件，以及零个或多个目标代码库名称（.lib 扩展名）。p 代码库 （.lpp 扩展名）总是传递给代码生成器应用程序，不能传递给链接器。

表 7-1 列出了链接器可识别的选项，随后的各个段落对它们进行了介绍。

表 7-1： 链接器命令行选项

选项	作用
-8	使用 8086 样式的段 : 偏移地址形式
-A*class=low-high* , . . .	指定类的地址范围
-C*x*	调用图选项
-C*psect=class*	指定全局 psect 的类名
-C*baseaddr*	生成以 *baseaddr* 为基址的二进制输出文件
-D*class=delta*	指定类的 delta （增量）值
-D*symfile*	生成旧式的符号文件
-E*errfile*	将错误消息写入 *errfile*
-F	生成仅具有符号记录的 .obj 文件
-G *spec*	指定段选择器的计算
-H *symfile*	生成符号文件
-H+ *symfile*	生成增强型符号文件

表 7-1： 链接器命令行选项 （续）

选项	作用
-I	忽略未定义的符号
-J *num*	设置在中止之前的最大错误数
-K	阻止重叠函数参数和自动变量区域
-L	保留 .obj 文件中的重定位项
-LM	保留 .obj 文件中的段重定位项
-N	按地址顺序对映射文件中的符号表排序
-Nc	按类地址顺序对映射文件中的符号表排序
-Ns	按空间地址顺序对映射文件中的符号表排序
-M*mapfile*	在指定文件中生成链接映射
-O*outfile*	指定输出文件的名称
-P*spec*	指定 psect 地址和顺序
-Q*processor*	指定器件类型 （仅出于美化原因）
-S	禁止符号文件中的符号列表
-S*class=limit[,bound]*	指定属于某个类的 psect 的地址限制和起始边界
-U*symbol*	以未定义形式在表中预先输入符号
-V*avmap*	使用文件 *avmap* 来生成 *Avocet* 格式的符号文件
-W*warnlev*	设置警告级别 （-9 至 9）
-W*width*	设置映射文件宽度 （>=10）
-X	从符号文件中删除所有局部符号
-Z	从符号文件中删除不重要的局部符号
--DISL=*list*	指定禁止的消息
--EDF=*path*	指定消息文件位置
--EMAX=*number*	指定最大错误数
--NORLF	不重定位列表文件
--VER	打印版本号并停止

如果标准输入是一个文件，则假定该文件包含命令行参数。可以通过在前一行的末尾加上*反斜杠* \ 来进行分行。通过这种方式，可以发出几乎无限长的 HLINK 命令。例如，名为 x.lnk 并包含以下文本的链接命令文件：

```
-Z -OX.OBJ -MX.MAP \
-Ptext=0,data=0/,bss,nvram=bss/. \
X.OBJ Y.OBJ Z.OBJ
```

可以通过以下方法之一传递给链接器：

```
hlink @x.lnk
hlink < x.lnk
```

有几个链接器选项要求指定存储器地址或长度。它们的语法全都是类似的。默认情况下，数字将被解释为一个十进制值。要强制将它解释为十六进制数字，应添加一个 H 或 h 后缀；例如， 765FH 将被视为十六进制数字。

© 2013 Microchip Technology Inc.

7.2.1 -A*class =low-high*,...

psect 通常根据传递给 -P 选项（见**第 7.2.19 节 “-Pspec”**）的信息进行链接，但有时需要将属于某个类的 psect 链接到多个非连续的地址范围。该选项用于将多个地址范围指定为属于某个类。例如：

```
-ACODE=1020h-7FFEh,8000h-BFFEh
```

指定将CODE类中的psect链接到给定的地址范围（除非它们以其他方式进行特别链接）。

如果存在一些相同的连续地址范围，则可以使用 x 字符后接重复计数的方式来指定它们。例如：

```
-ACODE=0-0FFFFhx16
```

指定存在 16 个连续区域，每个均从地址 0 开始，长度为 64 KB。虽然这些范围是连续的，但没有任何 psect 会跨越 64k 边界，因此该选项产生的 psect 放置方式将与指定以下选项时不同

```
-ACODE=0-0FFFFFh
```

该选项不包含 64k 倍数边界。

-A 选项不会指定与地址关联的存储空间。将某个 psect 分配给某个类之后，该 psect 的空间值会被分配给该类，请参见**第 6.4.9.3.13 节 “space”**。

7.2.2 -C*x*

该选项现已废弃。

7.2.3 -C*psect=class*

该选项用于将某个 psect 与某个特定类相关联。通常，在命令行上不需要该选项，因为 psect 类在目标文件中指定。请参见**第 6.4.9.3.3 节 “class”**。

7.2.4 -D*class=delta*

该选项用于定义属于指定类成员的 psect 的 delta 值。delta 值应为一个数字，表示 psect 内每个可寻址对象单元的字节数。大多数 psect 都不需要该选项，因为它们已定义了 delta 值。请参见**第 6.4.9.3.4 节 “delta”**。

7.2.5 -D*symfile*

使用该选项可生成旧式的符号文件。旧式的符号文件是一个 ASCII 文件，其中每行都具有符号的链接地址，之后跟随符号名称。

7.2.6 -E*errfile*

来自链接器的错误消息会被写入标准错误流。在 DOS 下，没有方便的方法可以将它重定向到一个文件（如果标准输出已被重定向，编译器驱动程序会重定向标准错误）。该选项会使链接器将所有错误消息都写入指定的文件，而不是屏幕，后者是默认的标准错误目标。

7.2.7 -F

通常，链接器会生成同时包含程序代码和数据字节，以及符号信息的目标文件。有时需要生成仅包含符号的目标文件，用于在再次运行链接器时提供符号值。 -F 选项将禁止数据和代码字节进入输出文件，仅留下符号记录。

在需要将一个项目（即，独立编译）的一部分与其他项目共用时，可以使用该选项，自举程序和应用程序都可能会出现这种情况。一个项目的文件使用该链接器选项进行编译来生成仅包含符号的目标文件；然后，将该目标文件与其他项目的文件进行链接。

7.2.8 -G*spec*

使用分段或需进行存储区切换的 psect 链接程序时，链接器可以通过两种方式为每个段分配段地址或选择器。段定义为一组连续的 psect，序列中的每个 psect 的链接和装入地址都与组中的上一个 psect 相连。段的段地址或选择器是在链接器处理段类型重定位时获得的值。

默认情况下，段选择器通过将段的基本装入地址除以段的重定位份额而得到；重定位份额基于在汇编器级别为 psect 指定的 reloc= 标志值，请参见**第 6.4.9.3.11 节“reloc”**。 -G 选项可用作一种计算段选择器的替代方法。 -G 的参数为以下形式的字符串：

```
A /10h-4h
```

其中，*A* 表示段的装入地址，/ 表示除法运算。它表示“获取 psect 的装入地址，除以十六进制数字 10，然后减去 4”。这种形式可以修改为以下形式：以 *N* 代替 *A*，以 * 代替 /（表示乘法运算），并加上而不是减去一个常量。标记 *N* 会被替换为由链接器分配的段编号。例如：

```
N*8+4
```

表示“获取段编号，乘以 8 并加上 4”。其结果为段选择器。该特定示例将按序列 4, 12, 20... 为已定义的段数量分配段选择器。

映射文件中会显示每个psect的选择器。请参见**第7.4.2.2节“按模块列出的psect信息”**。

7.2.9 -H*symfile*

该选项将指示链接器生成符号文件。可选参数 *symfile* 指定用于接收数据的文件的名称。默认文件名为 l.sym。

7.2.10 -H+*symfile*

该选项将指示链接器生成增强型符号文件，该文件会在标准符号文件的内容之外，提供与每个符号关联的类名，以及一个列出每个类名及其所占用存储器范围的段区域。如果代码要使用调试器运行，则建议使用该格式。可选参数 *symfile* 指定用于接收符号文件的文件。默认文件名为 l.sym。

© 2013 Microchip Technology Inc.

7.2.11 -I

通常，无法解析对未定义符号的引用属于致命错误。使用该选项会导致将未定义符号改为视为警告。

7.2.12 -J*errcount*

链接器会在错误达到一定数量之后停止处理目标文件（而不是产生警告）。默认数量为 10，但是 -J 选项可以用于更改它。

7.2.13 -K

对于使用编译堆栈的较早编译器，链接器会尝试重叠函数自动变量和参数块，以减少所需的 RAM 总量。对于调试目的，可以使用该选项禁止该功能；但是，这样做会增加数据存储器要求。

在编译堆栈由代码生成器执行分配时，该选项不起作用。 OCG （专业 / 标准 / 免费模式）编译器就是如此，不应使用该选项。

7.2.14 -L

在链接器生成输出文件时，它通常不会保存任何重定位信息，因为文件现在是绝对文件。在某些情况下，会在装入时对程序进行进一步的“重定位”。 -L 选项将对于输入中的每个重定位记录在输出文件中生成一个空的重定位记录。

7.2.15 -LM

类似于上面的选项，该选项会在输出文件中保存重定位记录，但仅保存段重定位记录。

7.2.16 -M*mapfile*

该选项会导致链接器在指定文件或标准输出（如果省略了文件名）中生成链接映射。**第 7.4 节 “映射文件”**说明了映射文件的格式。

7.2.17 -N、 -Ns 和 -Nc

默认情况下，映射文件中的符号表将按名称排序。 -N 选项将使它基于符号的值以数字顺序排序。 -Ns 和 -Nc 选项的作用类似，但符号将按其空间值或类进行组合。

7.2.18 -O*outfile*

该选项用于指定链接器的输出文件名。默认的输出文件名为 l.obj。使用该选项将覆盖默认值。

7.2.19 -P*spec*

psect 基于通过 -P 选项传递给链接器的信息链接在一起，并分配地址。 -P 选项的参数基本上由以下形式的*逗号*分隔序列组成：

```
-Ppsect =lnkaddr+min/ldaddr+min,psect=lnkaddr/ldaddr,...
```

它具有几种变化形式，但基本上都会列出每个 psect 及其所需的链接和装入地址，以及最小值。所有值都可以省略，这种情况下将根据前面的值应用默认值。

如果存在最小值*min*，它的前面需要带有 + 符号。它用于设置链接或装入地址的最小值。地址将按下面介绍的方式计算，但如果它小于最小值，则会将它设置为等于最小值。

链接和装入地址将为数字或者其他 psect、类或特殊标记的名称。

如果链接地址为负数，则 psect 将以相反的顺序进行链接，即 psect 的顶端出现在指定地址减 1 处。跟随在负地址后的 psect 将放置在存储器中第一个 psect 之前。

如果省略了 psect 的链接地址，则将根据上一个 psect 的顶端来获取它。例如，在以下示例中：

```
-Ptext=100h,data,bss
```

text psect 会被链接在 100h 处（其装入地址默认设为相同地址）。data psect 将被链接（并装入）到等于十六进制数字 100 加上 text psect 长度的地址处，如果 data psect 具有与之关联的 reloc 值，则还会根据需要对该地址进行舍入（见**第 6.4.9.3.11 节 “reloc”**）。类似地， bss psect 将与 data psect 相连。同样：

```
-Ptext=-100h,data,bss
```

将按升序链接 bss、 data 和 text， text psect 的顶端出现在地址 0ffh 处。

如果完全省略装入地址，则它默认设为与链接地址相同。如果提供了*斜杠* / 字符，但未在其后提供地址，装入地址将与上一个 psect 相连。例如：

```
-Ptext=0,data=0/,bss
```

将导致 text 和 data 的链接地址为零； text 的装入地址将为零， data 的装入地址从 text 末尾开始。 bss psect 的链接和装入地址都与 data 相连。

装入地址可以替换为*点号*字符 .。它指示链接器将该 psect 的装入地址设置为与其链接地址相同。链接或装入地址也可以为另一个（先前链接的） psect 的名称。这会显式地将当前 psect 与先前指定的 psect 相连，例如：

```
-Ptext=0,data=8000h/,bss/. -Pnvram=bss,heap
```

该示例显示了 text 位于地址零处，data 链接到 8000h 处但装入到 text 之后； bss 链接并装入到 8000h 加上 data 长度的地址处，nvram 和 heap 与 bss 相连。请注意，此处使用了两个 -P 选项。多个 -P 选项将按顺序进行处理。

如果使用了 -A 选项 （见**第 7.2.1 节 “-Aclass =low-high,...”**）来指定某个类的地址范围，则该类名可以用于代替链接或装入地址，并将在该类对应的地址范围之一中查找空间。例如：

```
-ACODE=8000h-BFFEh,E000h-FFFEh
-Pdata=C000h/CODE
```

这会将 data 链接到 C000h 处，但在与 CODE 类关联的地址范围中查找用于装入它的空间。如果该类中没有足够大的可用空间，将产生错误。请注意，在此例中， data psect 仍将组装到一个连续的块中，而类 CODE 中的其他 psects 将分布到适合它们的地址范围中。这意味着，如果类 CODE 中具有两个或多个 psect，它们可能会混杂在一些地址范围中。

对于通过 -P 选项分配的所有 psect，都会从相同存储空间中与类关联的地址范围中减去它们的装入地址范围。举例来说，这使得可以使用 -A 选项指定一个范围，而无需提前知道其他 psect 将需要该范围低端部分的多少空间。

© 2013 Microchip Technology Inc.

映射文件中将显示 psect 的最终链接和装入地址。请参见**第 7.4.2.2 节 “按模块列出的 psect 信息”**。

7.2.20 -Q*processor*

该选项用于指定器件类型。它纯粹用于放入映射文件中的信息。该选项的参数是一个描述器件的字符串。没有任何行为会因器件类型而发生改变。

7.2.21 -S

该选项可阻止将相关符号信息包含到链接器生成的符号文件中。但仍然会包含段信息。

7.2.22 -S*class =limit[,bound]*

属于某个类的 psect 可能具有与之关联的高端地址限制。以下示例将属于 CODE 类的 psect 的最高地址限制为小于 400h。

```
-SCODE=400h
```

请注意，要设置某个 psect 的上限，必须在汇编代码中使用 psect `limit` 标志设置，请参见**第 6.4.9.3.6 节 “limit”**。

如果使用了边界参数，则属于该类的 psect 的地址将从边界地址的倍数处开始。以下示例将属于 FARCODE 类的 psect 放置在 1000h 倍数地址处，但最高地址限制为 6000h。

```
-SFARCODE=6000h,1000h
```

7.2.23 -U*symbol*

该选项用于将指定符号作为未定义符号输入到链接器的符号表中。这对于以下情形很有用：完全从库中进行链接，或从库中链接某个模块，并且顺序安排为在默认情况下链接后一个模块。

7.2.24 -V*avmap*

要生成 Avocet 格式的符号文件，需要向链接器提供一个映射文件，以使它能够将 psect 名称映射为 Avocet 存储器标识符。avmap 文件通常通过编译器提供，或由编译器驱动程序根据需要自动创建。

7.2.25 -W*num*

`-W` 选项可以用于设置警告级别（范围为 -9 至 9）或映射文件的宽度（对于 *num* >= 10 时的值）。

`-W9` 将禁止所有警告消息。`-W0` 是默认值。将警告级别设置为 -9（`-W-9`）将提供最全面的警告消息。

7.2.26 -X

使用该选项可以从符号文件中禁止局部符号。全局符号将总是出现在符号文件中。

7.2.27 -Z

一些局部符号是编译器生成的，调试过程并不关心它们。该选项将从符号文件中禁止具有单个字母字符后跟一个数字字符串形式的所有局部符号。可以用于开始一个不重要符号的字母集合当前为 `klfLSu`。`-Z` 选项将剔除以其中字母之一开始并后跟数字字符串的所有局部符号。

© 2013 Microchip Technology Inc.

7.2.28 --DISL=*message numbers*：禁止消息

该选项主要由命令行驱动程序 xc8 用于禁止特定消息编号。它可接受要在编译过程中禁止的消息编号的*逗号*分隔列表。

如果使用命令行驱动程序 xc8 和 --MSGDISABLE 驱动程序选项进行编译，则会应用该选项，请参见**第 4.8.38 节 “--MSGDISABLE：禁止警告消息”**。

关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

7.2.29 --EDF=*message file*：设置消息文件路径

该选项主要由命令行驱动程序 xc8 用于指定消息描述文件的路径。默认文件位于编译器安装目录的 dat 目录中。

关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

7.2.30 --EMAX=*number*：指定最大错误数

该选项主要由命令行驱动程序 xc8 用于指定在汇编器终止之前可以遇到的最大错误数量。默认数量为 10 个错误。

如果使用命令行驱动程序 xc8 和 --ERRORS 驱动程序选项进行编译，则会应用该选项，请参见**第 4.8.29 节 “--ERRORS：最大错误数”**。

关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

7.2.31 --NORLF：不重定位列表文件

使用该选项可以阻止链接器对汇编器所生成的汇编列表文件应用 fixup。该选项通常由命令行驱动程序 xc8 在执行预链接阶段时使用，但在执行最终的链接步骤时会忽略它，以使列表文件显示最终的绝对地址。

如果要尝试解决 fixup 错误，则应禁止该选项，从而 fixup 汇编列表文件，并可以为该文件计算绝对地址。如果编译器驱动程序检测到存在等于 1 的预处理器宏 __DEBUG，则在编译时将禁止该选项。在 MPLAB IDE 中选择 *Debug*（调试）编译时，该宏会置 1，所以如果遇到此类错误，请总是选择它。

7.2.32 --VER：打印版本号

选择该选项会打印与链接器的版本和内部版本有关的信息。即使命令行上存在其他选项和文件，链接器也会在处理完该选项之后终止。

© 2013 Microchip Technology Inc.

7.3 重定位和 PSECT

本节介绍链接器需要处理的输入文件。

链接器可以读取可重定位目标文件和目标文件库 （.lib 扩展名）。库文件是打包到单个单元内的一组目标文件，所以基本上只需考虑目标文件的格式。

每个目标文件都包含一定数量的记录。每个记录都具有一个类型，指示它存放何种信息。一些记录类型存放关于目标器件及其配置的一般信息，其他记录类型可能存放数据和其他信息 （如程序调试信息）。

目标文件中的许多信息都与 psect 有关。 psect 是一种汇编域构造，实质上是某种内容（指令或数据）组成的块。所有会在程序中产生内容的对象都位于 psect 中。关于 psect 的介绍指南，请参见**第 6.4.8 节 “程序段”**。有一种特定的记录类型用于存放 psect 中的数据。每个目标文件的大部分内容都由包含可执行代码和变量等信息的 psect 记录组成。

现在，了解一下链接器执行的基本任务，它们包括：

- 将所有可重定位目标文件合并为一个
- 将目标文件中包含的 psect 重定位到存储器中
- fixup psect 中的符号引用

通常至少会向链接器传递两个目标文件。其中一个由项目中的所有 C 代码 （包括 C 库代码）生成。由于代码生成器会编译并组合程序的所有 C 代码，并仅生成一个汇编输出，所以只会有一个这种文件。传递给链接器的另一个文件为从运行时启动代码生成的目标代码，请参见**第 4.4.2 节 “启动和初始化”**。

如果项目中存在汇编源文件，则还会对于每个源文件生成一个目标文件，并将它们传递给链接器。在项目中还可以指定现有的目标文件或目标文件库，如果它们存在，也会将它们传递给链接器。

链接器的输出也是一个目标文件，但永远都只会生成一个文件。该文件是一个绝对文件，因为链接器已经执行了重定位。输出文件将包含来自所有输入目标文件合并在一起的信息。

重定位需要将 psect 数据放入目标器件的存储器。

目标器件的存储器规范通过链接器选项的方式传递给链接器。这些选项由命令行驱动程序 xc8 生成。不存在任何链接描述文件，也无法在任何源文件中指定这些选项。默认链接器选项很少需要调整，但如果需要，可以小心地使用驱动程序选项 -L- 进行更改，请参见**第 4.8.7 节 “-L-：直接调整链接器选项”**。

将 psect 放置在实际存储器之后，就可以将 psect 数据中的符号引用替换为绝对值。该过程称为 fixup。

对于目标文件中的每个 psect 记录，都具有一个相应的重定位记录，指示 psect 记录中的哪些字节（或位）需要在完成重定位之后进行调整。重定位记录还会指定如何确定这些值。如果由链接器确定的值太大，无法放入在 psect 中为值保留的“孔洞”中，则会出现链接器 fixup 溢出错误。关于确定这些错误原因的信息，请参见 **“(477) fixup overflow in expression (location 0x* (0x*+*), size *, value 0x*) （链接器）”**。

7.4 映射文件

映射文件包含有关重定位 psect 和赋予这些 psect 内符号的地址的信息。

7.4.1 生成

如果通过 IDE （如 HI-TIDE™ 或 MPLAB IDE）执行编译，则默认情况下会生成一个映射文件，无需调整编译器选项。如果从命令行上使用驱动程序，则需要使用 `-M` 选项来请求生成映射文件，请参见**第 7.2.16 节 “-Mmapfile”**。映射文件使用扩展名 `.map`。

映射文件由链接器生成。如果编译过程在执行链接器之前停止，则不会生成任何映射文件。即使遇到错误，链接器也仍然会生成映射文件，这使您可以使用该文件来追踪错误的原因。但是，如果链接器最终报告错误太多，从而未完成运行，则映射文件将不创建或不完整。可以在命令行上使用 `--ERR0RS` 选项 （见**第 4.8.29 节 “--ERRORS：最大错误数”**）来增加链接器退出之前的错误数。

7.4.2 内容

映射文件中的各个部分如下所示 （按出现顺序）。

- 编译器名称和版本号
- 用于调用链接器的命令行的副本
- 链接的第一个文件中的目标代码的版本号
- 机器类型
- 按 psect 的父目标文件排序的 psect 摘要
- 按 psect 的类排序的 psect 摘要
- 段摘要
- 未用地址范围摘要
- 符号表

以下几节显示了示例映射文件的某些部分，以及说明性文本。

7.4.2.1 一般信息

映射文件顶部是与链接器执行有关的一般信息。

如果安装了多个编译器版本，则在分析程序时，请总是确认映射文件中显示的编译器版本号，以确保执行所需的编译器。

使用 `--CHIP` 选项 （**第 4.8.20 节 “--CHIP：定义器件”**）或在 IDE 中选择的器件应出现在 Machine type （机器类型）条目之后。

object code version （目标代码版本）与汇编器生成的可重定位目标文件所用的文件格式有关。除非独立地更新了汇编器或链接器，否则不需要关注它。

© 2013 Microchip Technology Inc.

一个典型映射文件的开头可能为以下形式。为了清晰起见，该示例进行了缩减。

```
--edf=/home/jeff/Microchip/XC8/1.00/dat/en_msgs.txt -cs -h+main.sym -z \
  -Q16F946 -ol.obj -Mmain.map -ver=XC8 -ACONST=00h-0FFhx32 \
  -ACODE=00h-07FFhx4 -ASTRCODE=00h-01FFFh -AENTRY=00h-0FFhx32 \
  -ASTRING=00h-0FFhx32 -ACOMMON=070h-07Fh -ABANK0=020h-06Fh \
  -ABANK1=0A0h-0EFh -ABANK2=0120h-016Fh -ABANK3=01A0h-01EFh \
  -ARAM=020h-06Fh,0A0h-0EFh,0120h-016Fh,01A0h-01EFh \
  -AABS1=020h-07Fh,0A0h-0EFh,0120h-016Fh,01A0h-01EFh -ASFR0=00h-01Fh \
  -ASFR1=080h-09Fh -ASFR2=0100h-011Fh -ASFR3=0180h-019Fh \
  -preset_vec=00h,intentry,init,end_init -ppowerup=CODE -pfunctab=CODE \
  -ACONFIG=02007h-02007h -pconfig=CONFIG -DCONFIG=2 -AIDLOC=02000h-02003h \
  -pidloc=IDLOC -DIDLOC=2 -AEEDATA=00h-0FFh/02100h -peeprom_data=EEDATA \
  -DEEDATA=2 -DCODE=2 -DSTRCODE=2 -DSTRING=2 -DCONST=2 -DENTRY=2 -k \
  startup.obj main.obj

Object code version is 3.10

Machine type is 16F946
```

Linker command line（链接器命令行）会显示对于最后一次编译传递给链接器的所有命令行选项和文件。请记住，这些是链接器选项，不是命令行驱动程序选项。

链接器选项一定是很复杂的。幸运的是，很少需要从它们的默认设置调整它们。它们由命令行驱动程序 xc8 基于所选的目标器件和指定的驱动程序选项构造。通常，可以通过查看映射文件中的链接器选项来确认驱动程序选项是否有效。例如，如果指示驱动程序保留某个存储区，将会看到所使用的链接器选项中发生变化。

如果必须更改默认的链接器选项，可以使用驱动程序 -L- 选项间接通过驱动程序来实现，请参见**第 4.8.7 节 “-L-：直接调整链接器选项”**。如果使用该选项，请总是确认映射文件中出现正确的变化。

7.4.2.2 按模块列出的 PSECT 信息

映射文件中的下一个部分会列出在输出中产生内容的那些模块，以及关于这些模块定义的 psect 的信息。关于 psect 的介绍说明，请参见**第 5.15.1 节 “程序段”**。

该部分以包含以下题头的行开始：

```
Name   Link  Load  Length  Selector  Space  Scale
```

该行下最左侧是目标文件的列表。这些目标文件包括从源代码模块生成的文件和那些从目标库文件（.lib 扩展名）中提取的文件。对于后一种情况，库文件的名称会打印在目标文件列表之前。请注意，由于代码生成器会组合所有 C 源文件（和 p 代码库），所以只有一个代表整个程序 C 代码部分的目标文件。对应于运行时启动代码的目标文件通常会出现在该列表中。

映射文件该部分中的信息可以用于确认某个模块是否在输出文件中产生内容，以及确定每个模块定义的确切 psect。

其中会显示从每个目标文件中链接到程序中的所有 psect（在 *Name*（名称）列下），以及关于该 psect 的信息。

链接器可以处理两种地址：链接和装入。一般来说， psect 的链接地址是用于在运行时访问它的地址。

装入地址 （通常与链接地址相同）是 psect 在输出文件 （HEX 或二进制文件等）内的起始地址。如果某个 psect 用来存放一些位，则装入地址是无关紧要的，并改为用于存放已转换为字节地址的链接地址 （以位为单位）。

psect 的 *Length* （长度）将以 psect 所使用的单位显示。

Selector （选择器）较不常用，针对 PIC 器件进行编译时不用关心它。

Space （空间）字段非常重要，因为它指示 psect 所放置的存储空间。对于具有独立存储空间的哈佛架构机器 （如 PIC10/12/16 器件），该字段必须与地址一起使用，用以指定确切的存储位置。空间 0 指示程序存储器，空间 1 指示数据存储器。请参见**第 6.4.9.3.13 节 “space”**。

psect 的 *Scale*（比例）指示每个字节的地址单位数量。如果 scale 为 1，它将保留为空；对于存放位对象的 psect，它将显示为 8。存放位对象的 psect 的装入地址将用于显示已转换为字节单位的链接地址，而不是装入地址。请参见**第 6.4.9.3.2 节 “bit”**。

例如，映射文件中显示了以下内容。

```
         Name  Link  Load  Length Selector  Space  Scale
ext.obj  text    3A    3A      22       30      0
         bss     4B    4B      10       4B      1
         rbit    50     A       2        0      1      8
```

这表明链接器处理了一个名为 `ext.obj` 的文件。（它可能是从 C 代码或名为 `ext.as` 的源文件生成的。）

该目标文件包含 `text` psect，以及名为 `bss` 和 `rbit` 的 psect。

psect `text` 链接在地址 3A 处，`bss` 链接在地址 4B 处。乍看之下，考虑到 text 为 22 个字长，这似乎会发生问题；但是，由于 space 标志指示它们处于不同的存储区 （`text` 的 space 标志为 0， `bss` 的 space 标志为 1），所以不占用相同的存储器。

psect `rbit` 包含位对象，这可以通过查看它的 scale 值（值为 8）确认。同样，乍看之下，似乎可能发生 `rbit` 被链接到 `bss` 之上的问题。它们的 space 标志是相同的，但由于 `rbit` 包含位对象，其链接地址将以位为单位。`rbit` psect 的装入地址字段显示的是已转换为字节单位的链接地址，即 50h/8 => Ah。

目标文件列表下可能会存在标号 *COMMON*。它会显示从整个程序范围的 psect （特别是由编译堆栈使用的 psect）在程序中产生的内容。

7.4.2.3 按类列出的 PSECT 信息

映射文件中的下一个部分会显示相同的 psect 信息，但将按 psect 类进行分组。

该部分以包含以下题头的行开始：

```
TOTAL   Name  Link  Load  Length
```

在这下面是类名，后面跟随属于该类的 psect，请参见**第 6.4.9.3.3 节 “class”**。这些 psect 与上一个部分中按模块列出的那些 psect 是相同的；该部分中不包含任何新信息，只是不同的呈现方式。

© 2013 Microchip Technology Inc.

7.4.2.4 段列表

在映射文件中，类列表后面是段列表。段是同一存储空间中的连续psect的概念性分组，链接器使用它们来协助进行 psect 放置。不存在段汇编器伪指令，也无法以任何方式来控制段。

该部分以包含以下题头的行开始：

```
SEGMENTS   Name   Load   Length   Top   Selector   Space   Class
```

段的名称将来自连续分组中链接地址最低的 psect。这可能会导致与同名 psect 产生混淆。不要阅读映射文件这一部分中的 psect 信息。

用户通常可以忽略映射文件的这一部分。

7.4.2.5 未用地址范围

存储器摘要的最后部分将显示尚未分配、因而未使用的存储器。链接器可以知道代码生成器分配的所有存储器 （对于绝对变量），所以该可用空间是准确的。

该部分跟随以下题头：

```
UNUSED ADDRESS RANGES
```

其后为类的列表，以及每个类中仍然可用的存储器。如果某个类中具有多个存储器范围，则每个范围将单独打印一行。其中不会显示类中的任何分页边界，但 *Largest block*（最大的块）列将显示在考虑到存储器范围中的所有分页之后的最大连续可用空间。如果要确定为什么 psect 无法放入存储器的原因 （例如，无法找到空间类型的错误），则需要研究一下这个重要的信息。

请注意，与类关联的存储器可以与其他类中的存储器重叠，因而可用空间总量并不只是将所有未用范围相加。

7.4.2.6 符号表

映射文件中的最后一个部分会列出程序定义的全局符号。该部分具有以下题头：

```
Symbol Table
```

之后是两个列，其中将按字母顺序列出符号。对于链接器，所有从 C 代码生成的符号将总是与其汇编器等效符号名称一起显示。请参见**第 5.12.3 节 “汇编代码和 C 代码之间的交互”**。

该表中列出的符号包括：

- 全局汇编标号
- 全局 EQU/SET 汇编器伪指令标号
- 链接器定义的符号

汇编符号通过 GLOBAL 汇编器伪指令来设为全局符号，请参见**第 6.4.9.1 节“GLOBAL”**了解更多信息。

链接器定义的符号的行为类似于 EQU 伪指令；但是，它们是由链接器在链接过程中定义的，它们的所有定义都不会出现在任何源文件或中间文件中。请参见**第 5.15.6 节 “链接器定义的符号”**。

每个符号都会显示它们被放入的 psect，以及已为符号分配的值 （通常为一个地址）。符号中不会包含指示它是代表代码还是数据，或者指示它所驻留的存储空间的信息。

如果某个符号的 psect 显示为 (abs)，这意味着该符号不直接与 psect 相关联。绝对 C 变量或在汇编代码中使用 EQU 伪指令定义的所有符号就是如此。

请注意，在每个汇编列表文件中也会显示一个符号表。（关于生成这些文件的信息，请参见**第 4.8.16 节 “--ADDRQUAL：设置编译器对于存储器限定符的响应”**。）这些符号表与映射文件中显示的符号表的区别在于，它们还会列出局部符号，并仅显示在相应模块中定义的符号。

© 2013 Microchip Technology Inc.

第 8 章 实用程序

8.1 简介

本章介绍一些与编译器一起捆绑的实用应用程序。

其中一些应用程序可能无法在编译时正常调用，但可以手动执行，以执行特定的任务。

《MPLAB XC8 C 编译器用户指南》的这一章介绍以下应用程序：

- 库管理器
- OBJTOHEX
- CREF
- CROMWELL
- HEXMATE

8.2 库管理器

库管理器程序 LIBR 具有将几个文件组合为单个文件 （称为库）的功能。在一个项目中，可能希望使用库的原因包括：

- 要链接的文件会减少
- 可以更快地访问文件内容
- 库使用的磁盘空间较少

库管理器可以从 p 代码文件 （.p1 扩展名）编译 p 代码库 （.lpp 扩展名），或从目标文件 （.obj 扩展名）编译目标代码库 （.lib 扩展名）。只有在所有库源代码都以 C 编写时，才应创建 p 代码库。对于要编译到库中的汇编代码，应使用目标代码库。

对于两种库类型，将仅提取程序所需的那些模块，并将其包含在程序输出中。

8.2.1 库格式

库中的模块只是进行简单连接，但在库文件开头处会维护库中的模块和符号的目录。由于该目录的大小小于模块之和，所以在第一个阶段中链接器可以通过仅读取目录而不是所有模块来执行更快的搜索。在第二个阶段中，它只需读取所需的那些模块，忽略其他模块。这全都可以尽可能减少链接时的磁盘 I/O。

应当注意，库格式并不是一些其他编译器系统所使用的通用归档机制。这样做的好处是，可以针对加快链接过程而优化格式。

© 2013 Microchip Technology Inc.

8.2.2 使用库管理器

库文件可以直接使用命令行驱动程序编译，请参见**第 4.8.44 节“--OUTPUT= type：指定输出文件类型”**。在这种情况下，驱动程序将使用相应的选项调用 LIBR，使您不必直接使用库管理器。例如，您可能希望手动执行该步骤，或者可能需要查看库文件的内容。本节介绍如何从命令行上执行库管理器。不能从 IDE （如 MPLAB IDE）中调用库管理器。

库管理器程序的名称为 LIBR，其命令的格式如下：

```
LIBR [options]  k  file.lpp [file1.p1 file2.p1...]
LIBR [options]  k  file.lib [file1.obj file2.obj ...]
```

options 为会影响程序输出的零个或多个库管理器选项。表 8-1 列出了这些选项。

表 8-1： 库管理器命令行选项

选项	作用
-P *width*	指定页面宽度
-W	禁止非致命错误

键字母 *k* 指示库管理器所请求的命令（替换、提取或删除模块，列出模块或符号）。表 8-2 列出了这些命令。

表 8-2： 库管理器键字母命令

键	含义
r	替换模块
d	删除模块
x	提取模块
m	列出模块
s	列出模块与符号
o	重新排序模块

键后面列出的第一个文件名是要使用的库文件的名称。随后的文件（如需要）是指定命令所需的库的模块。

如果要编译一个 p 代码库，列出的模块必须为 p 代码文件。如果要编译一个目标文件库，列出的模块必须为目标文件。

在替换或提取模块时，必须指定要替换或提取的模块的名称。如果未提供任何名称，将替换或提取库中的所有模块。

向库中添加一个文件是通过请求库管理器在库中替换它来执行的。由于它并不存在，所以会将该模块追加到库的末尾。如果使用了 r 键，而库不存在，将会创建它。

使用 d 键字母时，将从库中删除指定的模块。在这种情况下，不提供任何模块名称会产生错误。

m 和 s 键字母会列出指定的模块，并且对于 s 键字母，将会列出其中定义或引用的全局符号。D 或 U 字母用于指示每个符号是在模块中定义的，还是只是引用但未定义。与 r 和 x 键字母一样，空模块列表将意味着库中的所有模块。

© 2013 Microchip Technology Inc.

o 键可接受模块名称的列表，并会对库文件中的匹配模块重新排序，使它们的顺序与命令行上列出的模块顺序相同。未列出的模块将保留其现有顺序，并将出现在重新排序的模块之后。

8.2.2.1 示例

以下给出了一些库管理器用法的示例。以下命令：

```
LIBR s pic-stdlib-d24.lpp ctime.p1
```

会列出模块 ctime.p1 中的全局符号，如下所示：

```
ctime.p1                        D _moninit
                                D _localtime
                                D _gmtime
                                D _asctime
                                D _ctime
```

每个符号之前的 D 字母指示这些符号是由模块定义的。

使用以上命令但未指定模块名称时，将列出库中定义（或未定义）的所有符号。

以下命令将从库 lcd.lib 中删除目标模块 a.obj、 b.obj 和 c.obj：

```
LIBR d lcd.lib a.obj b.obj c.obj
```

8.2.3 提供参数

由于通常需要向 LIBR 提供许多目标文件参数，因而如果未给出命令行参数，将从标准输入中读取参数。如果标准输入连接到控制台， LIBR 将提示用户进行输入。

通过在行尾使用*反斜杠*作为延续字符，可以提供多行输入。如果从一个文件重定向标准输入， LIBR 将从该文件获取输入，而不产生提示。例如：

```
libr
libr> r file.lib 1.obj 2.obj 3.obj \
libr> 4.obj 5.obj 6.obj
```

将执行大致相同的操作，就如同是在命令行上键入了目标文件。libr> 提示由 LIBR 自身打印，文本的其余部分则作为输入键入。

```
libr <lib.cmd
```

LIBR 将从 lib.cmd 读取输入，并执行其中所找到的命令。这实际上支持向 LIBR 发出无限长的命令。

8.2.4 库的排序

库管理器将按模块在命令行上的顺序使用模块来创建库。在更新库时，将会保留模块的顺序。创建库之后添加到库中的所有新模块都会被追加到末尾。

模块在库中的顺序对于链接器是有意义的。如果库中包含的某个模块引用同一个库中另一个模块中定义的符号，定义该符号的模块应位于引用符号的模块之后。

8.2.5 错误消息

LIBR 会发出各种错误消息，其中大多数都表示致命错误，而另外一些则表示无害的情况，除非使用了 -W 选项，否则还是会报告它们。在这种情况下，将禁止所有警告消息。

© 2013 Microchip Technology Inc.

8.3 OBJTOHEX

链接器能够生成目标文件作为输出。如果需要任何其他格式，必须通过运行实用程序 OBJTOHEX 来生成。这可以将链接器生成的目标文件转换为各种不同的格式，包括各种 HEX 格式。该程序的调用方式为：

```
OBJTOHEX [options]  inputfile  outputfile
```

所有参数都是可选的。表 8-3 列出了 OBJTOHEX 的选项。

表 8-3： OBJTOHEX 命令行选项

选项	含义
-8	生成 CP/M-86 输出文件
-A	生成 ATDOS .atx 输出文件
-B*base*	生成偏移量为 *base* 的二进制文件。默认文件名为 l.obj
-C*ckfile*	从 *ckfile* 或标准输入读取校验和规范的列表
-D	生成 COD 文件
-E	生成 MS-DOS .exe 文件
-F*fill*	使用值为 *fill* 的字填充未用存储器——默认值为 0FFh
-I	生成具有线性寻址的扩展记录的 *Intel* HEX 文件
-L	将重定位信息传递到输出文件中 （用于 .exe 文件）
-M	生成 *Motorola* HEX 文件 （S19、 S28 或 S37 格式）
-N	生成用于 Minix 的输出文件
-P*stk*	生成用于 *Atari* ST 的输出文件，具有可选的堆栈大小
-R	在输出文件中包含重定位信息
-S*file*	将符号文件写入 *file*
-T	生成 *Tektronix* HEX 文件
-TE	生成扩展的 TekHEX 文件
-U	生成 COFF 输出文件
-UB	生成 UBROF 格式文件
-V	倒转输出文件中的字和长字的顺序
-*n*,*m*	格式化 Motorola 或 Intel HEX 文件；其中， *n* 是每个记录的最高字节数，*m* 指定记录长度舍入。非舍入记录会使用零填充为 *m* 的倍数。 *m* 本身必须为 2 的倍数。
--EDF	指定消息文件位置
--EMAX	指定最大错误数
--MSGDISABLE	指定禁止的消息
--VER	打印版本号并停止

如果省略 *outputfile*，则它默认设为 l.HEX 或 l.bin，具体取决于是否使用了 -b 选项。 *inputfile* 默认设为 l.obj。

除非另有说明，所有地址都将解释为十进制值。要强制将它解释为十六进制数字，应添加一个 H 或 h 后缀；例如， 765FH 将被视为十六进制数字。

8.3.1 校验和规范

如果要生成 HEX 文件输出，请使用 HEXMATE 的校验和工具，如**第 8.6 节“HEXMATE”**中所述。

对于其他文件格式，OBJTOHEX 校验和规范可用于进行自动校验和计算，它采用多行的形式，每行描述一个校验和。校验和行的语法为：

```
addr1-addr2 where1-where2 +offset
```

所有 *addr1*、*addr2*、*where1*、*where2* 和 *offset* 均为十六进制数字，不带通常的 H 后缀。

该规范表示应对 *addr1*（含）至 *addr2*（含）中的字节进行求和，并将其和放入地址 *where1* 至 *where2*（含）。对于 8 位校验和，这两个地址应相同。对于先存储低字节的校验和， *where1* 应小于 *where2*，反之亦然。

+*offset* 值是可选的，但如果提供，该值将用于初始化校验和。否则它初始化为零。

例如：

```
0005-1FFF 3-4 +1FFF
```

它会对 5（含）至 1FFFH（含）中的字节求和，然后将 1FFFH 与其和相加。 16 位校验和将放置在单元 3 和 4，低字节位于单元 3。校验和使用 1FFFH 初始化，以提供保护，防止存储器中出现全零 ROM 或错误放置的 ROM。该校验和的运行时检查会将进行校验和计算的 ROM 的最后一个地址与校验和相加。对于所考虑的 ROM，它应为 1FFFH。但是，初始化值可以所需的任何方式使用。

© 2013 Microchip Technology Inc.

8.4 CREF

交叉引用列表实用程序 CREF 用于将编译器或汇编器生成的原始交叉引用信息格式化为排序列表。

原始交叉引用文件使用 --CR 命令行驱动程序选项来生成。汇编器将使用 -C 汇编器选项或 XREF 控制行来生成原始交叉引用文件。

CREF 命令的一般格式为：

```
cref [options]  files
```

其中，*options* 为下面介绍的零个或多个选项，*files* 是一个或多个原始交叉引用文件。

CREF 可接受通配符文件名和 I/O 重定向。通过不带任何参数调用 CREF，并在命令行上输入内容来响应 cref> 提示符，可以提供很长的命令行。行尾的*反斜杠*将被解释为表示其后跟随更多命令行。

CREF 可接受**表 8-4：“CREF 命令行选项”**中列出的选项。

表 8-4： CREF 命令行选项

选项	含义
-F*prefix*	排除路径名或文件名以 *prefix* 开头的文件中的符号
-H*heading*	指定列表文件的标题
-L*len*	指定列表文件的页面长度
-O*outfile*	指定列表文件的名称
-P*width*	设置列表宽度
-S*stoplist*	读取文件 *stoplist*，并忽略列出的所有符号
-X*prefix*	排除以 *prefix* 开头的所有符号
--EDF	指定消息文件位置
--EMAX	指定最大错误数
--MSGDISABLE	指定禁止的消息
--VER	打印版本号并停止

以下几节将对每个选项进行详细说明。

8.4.1 -F*prefix*

通常需要从交叉引用列表中排除在系统头文件 （如 <stdio.h>）中定义的所有符号。-F 选项用于指定一个路径名前缀，用于排除路径名以该前缀开头的文件中定义的所有符号。例如， -F\ 将排除路径名以 \ 开头的所有文件中的所有符号。

8.4.2 -H*heading*

-H 选项接受一个字符串作为参数，它将用作列表中的标题。默认标题为所指定的第一个原始交叉引用信息文件的名称。

8.4.3 -L*len*

指定生成列表所用的纸张的长度；例如，如果列表要打印在 55 行的纸张上，则需要使用 -L55 选项。默认值为 66 行。

8.4.4 -O*outfile*

用于指定输出文件名。默认情况下，列表将写入到标准输出，并可按常规方式重定向。或者，也可以指定 *outfile* 作为输出文件名。

8.4.5 -P*width*

该选项用于指定列表要格式化为的宽度，例如， -P132 会将列表针对 132 列打印机进行格式化。默认值为 80 列。

8.4.6 -S*stoplist*

-S 选项应使用一个文件的名称作为其参数，该文件中包含不要在交叉引用中列出的符号列表。符号应在该文件中以每行一个的方式列出。请使用 C 域符号。可以使用多个 -S 选项提供多个 stoplist。

8.4.7 -X*prefix*

-X 选项用于基于作为参数传递给 -X 的前缀从列表中排除符号。例如，如果要排除以字符序列 xyz 开头的所有符号，则需要使用选项 -Xxyz。如果字符序列中出现数字，则它将匹配符号中的任意数字；例如， -XX0 将排除以字母 X 开头并跟随一个数字的所有符号。

8.4.8 --EDF=*message file*：设置消息文件路径

该选项主要由命令行驱动程序 xc8 用于指定消息描述文件的路径。默认文件位于编译器安装目录的 dat 目录中。

关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

8.4.9 --EMAX=*number*：指定最大错误数

该选项主要由命令行驱动程序 xc8 用于指定在 CREF 终止之前可以遇到的最大错误数量。默认数量为 10 个错误。

如果使用命令行驱动程序 xc8 和 --ERROR*S* 驱动程序选项进行编译，则会应用该选项，请参见**第 4.8.29 节 “--ERRORS：最大错误数”**。

关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

8.4.10 --MSGDISABLE=*message numbers*：禁止消息

该选项主要由命令行驱动程序 xc8 用于禁止特定消息编号。它可接受要在编译过程中禁止的消息编号的*逗号*分隔列表。

如果使用命令行驱动程序 xc8 和 --MSGDISABLE 驱动程序选项进行编译，则会应用该选项，请参见**第 4.8.38 节 “--MSGDISABLE：禁止警告消息”**。

关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

© 2013 Microchip Technology Inc.

8.4.11 --VER：打印版本号

选择该选项会打印与 CREF 的版本和内部版本有关的信息。即使命令行上存在其他选项和文件， CREF 也会在处理完该选项之后终止。

8.5 CROMWELL

CROMWELL 实用程序可将代码和符号文件转换为不同的格式。这些文件通常由调试器使用，用于对代码进行源代码级调试。表 8-5 列出了可用的输出格式。

表 8-5： CROMWELL 格式类型

键	格式
cod	*Bytecraft* COD 文件
coff	COFF 文件格式
elf	ELF/DWARF 文件
eomf51	扩展 OMF-51 格式
hitech	HI-TECH 软件格式
icoff	ICOFF 文件格式
ihex	*Intel* HEX 文件格式
mcoff	Microchip COFF 文件格式
omf51	OMF-51 文件格式
pe	P&E 文件格式
s19	*Motorola* HEX 文件格式

CROMWELL 应用程序由命令行驱动程序根据需要自动执行。如果手动运行该应用程序，则需要以下信息。

CROMWELL 命令的一般格式为：

CROMWELL [*options*] *inputFiles* -o*key* [*outputFile*]

其中， *options* 可以为表 8-6 列出的任意选项。

表 8-6： CROMWELL 命令行选项

选项	说明
-P*name[,architecture]*	器件名称和架构
-N	标识代码类
-D	转储输入文件
-C	仅标识输入文件
-F	将局部符号伪装为全局符号
-O*key*	设置输出格式
-I*key*	设置输入格式
-L	列出可用的格式
-E	去除文件扩展名
-B	指定大尾数法字节顺序
-M	去除下划线字符
-V	详细模式
--EDF=*path*	指定消息文件位置
--EMAX=*number*	指定最大错误数
--MSGDISABLE=*list*	指定禁止的消息
--VER	打印版本号并停止

outputFile（可选）是输出文件的名称。 *inputFiles* 通常为 HEX 和 SYM 文件。

CROMWELL 会自动搜索 SDB 文件，并在找到时进行读取。以下段落进一步介绍了这些选项。

© 2013 Microchip Technology Inc.

8.5.1 -Pname[,architecture]

-P 选项可接受一个字符串，它是所使用器件的名称。 CROMWELL 可在生成所选输出格式时使用它。

请注意，要以 COFF 格式生成输出，该选项还需要一个额外的参数，它指定器件的架构。因此，对于该格式，该选项的用法必须为以下形式：-P*name,architecture*。表 8-7 列举了支持生成 COFF 文件的架构。

表 8-7： 架构参数

架构	说明
PIC12	Microchip 低档 PIC® MCU 芯片
PIC14	Microchip 中档 PIC MCU 芯片
PIC14E	Microchip 增强型中档 PIC MCU 芯片
PIC16	Microchip 高档 （17CXXX） PIC MCU 芯片
PIC18	Microchip PIC18 芯片
PIC24	Microchip PIC24F 和 PIC24H 芯片
PIC30	Microchip dsPIC30 和 dsPIC33 芯片

8.5.2 -N

要生成某些输出文件格式（如 COFF），CROMWELL 要求提供程序存储空间 psect 类的名称。类的名称以*逗号*分隔列表的形式指定。要确定链接器使用了哪些类，请参见映射文件 （**第 7.4 节 “映射文件”**）。

例如，中档器件通常需要 -NCODE、 CONST、 ENTRY 和 STRING。

8.5.3 -D

-D 选项用于以人们可阅读的格式显示关于指定输入文件的详细信息。如果需要检查该文件（通常为二进制文件）的内容，该选项非常有用。输入文件可以为如表 8-5 中所示的文件类型之一。

8.5.4 -C

该选项将尝试确定指定的输入文件是否为表 8-5 中所示的格式之一。如果文件可识别，则会显示其类型的确认。

8.5.5 -F

在生成 COD 文件时，该选项可用于强制将所有局部符号表示为全局符号。这对于仿真器无法从 COD 文件中读取局部符号信息的情况会很有用。

8.5.6 -Okey

该选项用于指定输出文件的格式。 *key* 可以为表 8-5 列出的任意类型。

8.5.7 -Ikey

该选项可用于指定默认的输入文件格式。 *key* 可以为表 8-5 列出的任意类型。

8.5.8 -L

使用该选项来显示支持的文件格式类型。将显示类似于表 8-5 所示的列表。

8.5.9 -E

使用该选项来指示 CROMWELL 忽略所提供的所有文件扩展名。将改为使用默认扩展名。

8.5.10 -B

在支持不同尾数法类型的格式中，使用该选项来指定大尾数法字节顺序。

8.5.11 -M

在生成 COD 文件时，该选项将从符号中删除前面的下划线字符。

8.5.12 -V

开启详细模式，该模式将显示关于 CROMWELL 正在执行哪些操作的信息。

8.5.13 --EDF=*message file*：设置消息文件路径

该选项主要由命令行驱动程序 xc8 用于指定消息描述文件的路径。默认文件位于编译器安装目录的 dat 目录中。此外，关于编译器消息系统的完整信息，请参见**第 4.6 节“编译器消息”**。

8.5.14 --EMAX=*number*：指定最大错误数

该选项主要由命令行驱动程序 xc8 用于指定在 CROMWELL 终止之前可以遇到的最大错误数量。默认数量为 10 个错误。

如果使用命令行驱动程序 xc8 和 --ERRORS 驱动程序选项进行编译，则会应用该选项，请参见**第 4.8.29 节“--ERRORS：最大错误数”**。此外，关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

8.5.15 --MSGDISABLE=*message numbers*：禁止消息

该选项主要由命令行驱动程序 xc8 用于禁止特定消息编号。它可接受要在编译过程中禁止的消息编号的*逗号*分隔列表。

如果使用命令行驱动程序 xc8 和 --MSGDISABLE 驱动程序选项进行编译，则会应用该选项，请参见**第 4.8.38 节 “--MSGDISABLE：禁止警告消息”**。此外，关于编译器消息系统的完整信息，请参见**第 4.6 节 “编译器消息”**。

8.5.16 --VER：打印版本号

选择该选项会打印与 CROMWELL 的版本和内部版本有关的信息。即使命令行上存在其他选项和文件， CROMWELL 也会在处理完该选项之后终止。

© 2013 Microchip Technology Inc.

8.6 HEXMATE

HEXMATE 实用程序是设计用来操作 Intel HEX 文件的程序。 HEXMATE 是一个链接阶段后的实用程序，它由编译器驱动程序自动调用，并提供了以下功能：

- 计算并存储可变长度的校验和值
- 使用已知的数据序列填充未用的存储单元
- 将多个 Intel HEX 文件合并为一个输出文件
- 将 INHX32 文件转换为其他 INHX 格式 （如 INHX8M）
- 检测 HEX 文件中的特定或部分操作码序列
- 查找 / 替换特定或部分操作码序列
- 提供 HEX 文件中使用的地址的映射
- 更改或修复 HEX 文件中的数据记录的长度
- 验证 Intel HEX 文件中的校验和

HEXMATE 的典型应用可能包括：

- 在编译时将自举程序或调试模块合并到主应用程序中
- 计算某个程序存储器范围的校验和，并将它的值存储在程序存储器或 EEPROM 中
- 用一条指令填充未用的存储单元以将 PC 发送到某个已知地址 （如 PC 丢失）
- 在某个固定地址处存储序列号
- 在某个固定地址处存储字符串 （如时间标记）
- 在某个特定存储器地址处存储初始值 （如初始化 EEPROM）
- 检测是否使用了错误 / 受限的指令
- 调整 HEX 文件，使之满足特定自举程序的要求

8.6.1 HEXMATE 命令行选项

HEXMATE 由命令行驱动程序 xc8 自动调用。这主要是用于将 HEX 文件与源文件生成的输出进行合并；但是，有一些 xc8 选项可直接映射到 HEXMATE 选项，所以可以请求其他功能，而无需显式地在命令行上运行 HEXMATE。对于其他功能，以下详细介绍了运行该应用程序时可用的选项。

如果要直接运行 HEXMATE，其用法为：

HEXMATE [specs,]*file1.HEX* [[specs,]*file2.HEX* ... [specs,]*fileN.HEX*] [*options*]

其中，*file1.HEX* 至 *fileN.HEX* 构成要使用 HEXMATE 合并的输入 Intel HEX 文件的列表。如果仅指定一个 HEX 文件，则不会发生合并，但通过附加选项指定其他功能。表 8-8 列出了 HEXMATE 可接受的命令行选项。

表 8-8： HEXMATE 命令行选项

选项	作用
-ADDRESSING	将所有 HEXMATE 选项中的地址字段设置为使用字寻址或其他寻址模式
-BREAK	拆分连续数据，从而使新记录在所设置地址处开始
-CK	计算并存储校验和值
-FILL	使用已知值编程未用单元
-FIND	进行搜索，并在检测到特定代码序列时发出通知
-FIND...,DELETE	如果检测到该代码序列则删除它 （慎用）
-FIND...,REPLACE	使用新的代码序列替换代码序列
-FORMAT	指定最大数据记录长度或选择 INHX 形式
-HELP	显示所有选项或显示特定选项的帮助消息
-LOGFILE	将 HEXMATE 输出分析和各种结果保存到文件中
-O*file*	指定输出文件的名称
-SERIAL	在某个固定地址处存储序列号或代码序列
-SIZE	报告生成的 HEX 镜像中包含的数据字节数
-STRING	在某个固定地址处存储 ASCII 字符串
-STRPACK	使用字符串压缩在某个固定地址处存储 ASCII 字符串
-W	调整警告灵敏度
+	用作任意选项的前缀，以覆盖其地址范围中的其他数据 （如需要）

现在将更详细地介绍 HEXMATE 的输入参数。请注意，为 HEXMATE 选项提供的任何整数值应以无0x前缀或h字符后缀的十六进制值的形式输入。此外请注意，除非在-ADDRESSING选项中另外指定，否则这些选项中指定的任何地址字段都需要以字节地址的形式输入。

© 2013 Microchip Technology Inc.

8.6.1.1 规范 , 文件名 .HEX

HEXMATE 可处理的 Intel HEX 文件应为 INHX32 或 INHX8M 格式。可以对每个 HEX 文件应用附加的规范，以指定关于应如何处理该文件的限制或条件。

如果使用了任何规范，则它们必须位于文件名之前。之后通过*逗号*将规范列表与文件名进行分隔。

可以使用规范 r*Start-End* 来应用*范围限制*。范围限制将导致仅使用该范围内的地址数据。例如：

```
r100-1FF,myfile.hex
```

将使用 myfile.hex 作为输入，但仅处理该文件中地址处于范围 100h-1FFh （含）内的数据。

可以通过规范 *sOffset* 来应用地址偏移。如果使用地址偏移，则在生成输出时从该 HEX 文件中读取的数据将移动 （按指定的偏移量）到一个新的地址。偏移量可以为正值或负值。例如：

```
r100-1FFs2000,myfile.HEX
```

会将数据块从 100h-1FFh 移动到新的地址范围 2100h-21FFh。

移动可执行代码段时，需要小心。只有程序代码是位置无关的代码时，才能对它进行移动。

8.6.1.2 + 前缀

当参数或输入文件之前带有 + 操作符时，从该数据源获取的数据会被强制输送到输出文件中，并覆盖该地址范围内的其他现有数据。例如：

```
+input.HEX +-STRING@1000=”My string”
```

通常，如果两个数据源尝试在同一位置存储不同数据， HEXMATE 会发出错误。使用 + 操作符可通知 HEXMATE，如果多个数据源尝试将数据存储到同一地址，使用 + 前缀指定的数据源将具有优先权。

8.6.1.3 -ADDRESSING

默认情况下，HEXMATE 选项中的所有地址参数都预期值将以字节地址的形式输入。在一些器件架构中，本机寻址格式可能不是字节寻址。在这些情况下，如果能够以器件的本机格式输入地址组成部分，操作将会更简单。为方便这一点，可以使用 -ADDRESSING 选项。

该选项接受且仅接受一个参数，该参数配置每个地址单元包含的字节数。例如，如果器件的程序存储器自然使用 16 位 （2 字节）字寻址格式，选项 -ADDRESSING=2 会将 HEXMATE 配置为将所有命令行地址字段解释为字地址。该设置的影响是全局性的，所有 HEXMATE 选项现在都会根据该设置来解释地址。该选项可用于指定从每个地址 1 字节到每个地址 4 字节的寻址模式。

8.6.1.4 -BREAK

该选项需要一个*逗号*分隔的地址列表。如果在 HEX 文件中遇到其中任何地址，当前数据记录将结束，新的数据记录将从指定地址处重新开始。这可用于使用新的数据记录来强制区分程序空间中的不同功能区域。一些 HEX 文件阅读器依赖于它。

8.6.1.5 -CK

-CK 选项用于计算校验和。该选项的用法如下：

```
-CK=start-end@destination [+offset][wWidth][tCode][gAlogithm]
```

其中：

- *start* 和 *end* 指定用于计算校验和的地址范围。
- *destination* 是将存储校验和结果的地址。该值不能处于计算范围内。
- *offset* 是一个与校验和结果相加的可选初始值。
- *Width* 是可选的，它指定校验和结果的字节宽度。计算的结果可以为 1 至 4 字节的字节宽度。如果请求一个正宽度，则结果将以大尾数法字节顺序存储。负宽度将导致结果以小尾数法字节顺序存储。如果宽度未指定，则结果将为 2 字节宽，并以小尾数法字节顺序存储。
- *Code* 是校验和结果中尾随在每个字节之后的十六进制代码。它可以用于将校验和结果的每个字节嵌入一条指令内。
- *Algorithm* 是一个整数，用以选择要用于计算校验和结果的 HEXMATE 算法。表 8-9 列出了可选的算法。如果未指定，则使用的默认校验和算法为 8 位加法（1）。

使用校验选项的典型示例为：

```
-CK=0-1FFF@2FFE+2100w2
```

这将计算范围 0-1FFFh 的校验和，并将校验和结果写入地址 2FFEh。校验和值将偏移 2100h。结果将为 2 字节宽。

表 8-9： HEXMATE 校验和算法选择

选择器	算法说明
-4	初始值减去 32 位值
-3	初始值减去 24 位值
-2	初始值减去 16 位值
-1	初始值减去 8 位值
1	初始值加上 8 位值
2	初始值加上 16 位值
3	初始值加上 24 位值
4	初始值加上 32 位值
7	Fletcher 校验和 （8 位）
8	Fletcher 校验和 （16 位）

© 2013 Microchip Technology Inc.

8.6.1.6 -FILL

-FILL 选项用于以已知值填充未用的存储单元。该选项的用法如下：

-FILL=[const_width:]fill_expr[@address[:end_address]]

其中：

- *const_width* 的形式为 wn，指示 *fill_expr* 中每个常量的宽度 （*n* 字节）。如果未指定 *const_width*，默认值为架构的本机宽度。即， -FILL=w1:1 将使用值 0x01 填充每个字节。
- *fill_expr* 可以使用以下语法 （其中， *const* 和 *increment* 均为 *n* 字节常量）：
  - *const* 将使用重复常量填充存储器；即， -FILL=0xBEEF 将产生 0xBEEF、0xBEEF、 0xBEEF 和 0xBEEF
  - *const*+=*increment* 将使用递增常量填充存储器；即， -FILL=0xBEEF+=1 将产生 0xBEEF、 0xBEF0、 0xBEF1 和 0xBEF2
  - *const*-=*increment* 将使用递减常量填充存储器；即，-FILL=0xBEEF-=0x10 将产生 0xBEEF、 0xBEDF、 0xBECF 和 0xBEBF
  - *const*,*const*,...,*const* 将使用重复常量的列表填充存储器；即，-FILL=0xDEAD,0xBEEF 将产生 0xDEAD、 0xBEEF、 0xDEAD 和 0xBEEF
- fill_expr 之后的选项会产生以下行为：
  - @*address* 将使用 *fill_expr* 填充一个特定地址；即，-FILL=0xBEEF@0x1000 会在地址 1000h 处放入 0xBEEF
  - @*address*:*end_address* 将使用 *fill_expr* 填充存储器范围；即，-FILL=0xBEEF@0:0xFF 会在 0 和 255 之间的未用地址中放入 0xBEEF

所有常量可以按正常 C 语法使用（无符号）二进制、八进制，十进制或十六进制表示，例如， 1234 是一个十进制值、 0xFF00 是十六进制， FF00 是非法的。

8.6.1.7 -FIND

该选项用于检测并记录某个操作码或部分代码序列的出现。该选项的用法如下：

-FIND=Findcode [m*Mask*]@*Start*-*End* [/*Align*][w][t"*Title*"]

其中：

- *Findcode* 是要搜索的十六进制代码序列，并以小尾数法字节顺序输入。
- *Mask* 是可选的。它指定应用于 *Findcode* 值的位掩码，以进行限制较少的搜索。它以小尾数法字节顺序输入。
- *Start* 和 *End* 限制要搜索的地址范围。
- *Align* 是可选的。它指定只有代码序列的起始地址位于该值的倍数地址上时，才会匹配它。
- *w* （如存在）将导致 HEXMATE 在每次检测到代码序列时发出警告。
- *Title* 是可选的。它用于为该代码序列提供标题。定义标题将使日志报告和消息更具描述性、可读性更强。标题不会影响实际的搜索结果。

以下给出了一些示例。

选项 -FIND=3412@0-7FFF/2w 将检测 0h 和 7FFFh 之间，在 2 字节地址边界上对齐的代码序列 1234h。 w 指示将在每次发现该序列时发出一条警告。

在下一个示例 -FIND=3412M0F00@0-7FFF/2wt"ADDXY" 中，选项与上一个示例相同，但所匹配的代码序列使用 000Fh 作为掩码，所以 HEXMATE 将搜索所有 123xh 操作码（其中， x 是任意数字）。如果使用字节掩码，则它的字节宽度必须与应用它的操作码相等。由 HEXMATE 生成的所有消息或报告都会使用名称 *ADDXY* 来指代该操作码，因为它是为该搜索定义的标题。

如果 HEXMATE 生成日志文件，它将包含所有搜索的结果。-FIND 接受长度为 1 至 8 字节的完整 HEX 数据字节。（可选） -FIND 可以与 REPLACE 或 DELETE 配合使用 （如下所述）。

8.6.1.8 -FIND...,DELETE

如果使用了 DELETE 形式的 -FIND 选项，则会删除所有匹配的序列。该功能应慎用，通常建议不要用于删除可执行代码。

8.6.1.9 -FIND...,REPLACE

如果使用了 REPLACE 形式的 -FIND 选项，则会使用新代码替换或部分替换所有匹配的序列。该子选项的用法如下：

-FIND...,REPLACE=*Code* [m*Mask*]

其中：

- *Code* 是用于替换匹配 -FIND 条件的序列的小尾数法十六进制代码。
- *Mask* 是可选的位掩码，指定 *Code* 内的哪些位将替换匹配的代码序列。举例来说，如果只需修改16位指令内的4位，这会很有用。其余12位可以通过掩码而保留不变。

© 2013 Microchip Technology Inc.

8.6.1.10 -FORMAT

-FORMAT 选项可用于指定 INHX 格式的特定形式或调整最大记录长度。该选项的用法如下：

```
-FORMAT=Type [,Length]
```

其中：

- *Type* 指定要生成的特定 INHX 格式。
- *Length* 是可选的，它设置每个数据记录的最大字节数。有效长度介于 1 和 16 之间， 16 为默认值。

假设存在这种情形：某个自举程序尝试下载一个 INHX32 文件，但发生失败，因为它无法处理属于 INHX32 标准一部分的扩展地址记录。您知道该自举程序只能编程地址在 0 至 64k 范围内的数据， HEX 文件中任何超出该范围的数据都可以安全地忽略。在这种情况下，通过以 INHX8M 格式生成 HEX 文件，该操作可能会成功。执行该操作的 HEXMATE 选项为 `-FORMAT=INHX8M`。

现在，假设同一自举程序还要求每个数据记录包含 8 个字节的数据，不多也不少。这可以通过结合使用 `-FORMAT` 与 `-FILL` 选项实现。适当使用 `-FILL` 可以确保要进行编程的地址范围内的数据中不存在空隙。这将满足最小数据长度要求。要将数据记录的最大长度设置为 8 字节，只需将前一个选项修改为 `-FORMAT=INHX8M,8`。

表 8-10 列出了该选项支持的可能类型。请注意， INHX032 并不是实际的 INHX 格式。选择该类型会生成一个 INHX32 文件，但也会将高端地址信息初始化为零。这是一些器件编程器的要求。

表 8-10： -FORMAT 选项中使用的 INHX 类型

类型	说明
INHX8M	无法编程超出 64K 之外的地址
INHX32	可以使用扩展线性地址记录编程超出 64K 之外的地址
INHX032	INHX32，并将高端地址初始化为零

8.6.1.11 -HELP

使用 `-HELP` 将列出所有 HEXMATE 选项。输入另一个 HEXMATE 选项作为 `-HELP` 的参数时，将显示给定选项的详细帮助消息。例如：

```
-HELP=string
```

将显示 `-STRING` HEXMATE 选项的附加帮助。

8.6.1.12 -LOGFILE

`-LOGFILE` 选项会将 HEX 文件统计信息保存到指定文件中。例如：

```
-LOGFILE=output.log
```

将分析 HEXMATE 生成的 HEX 文件，并将报告保存到名为 `output.log` 的文件中。

8.6.1.13 -MASK

该选项可将某个存储器范围与特定位掩码进行逻辑与运算。它用于确保程序字中的未实现位（如果有）保留为空。该选项的用法如下：

```
-MASK=hexcode@start-end
```

其中，*hexcode* 是一个将与 *start* 至 *end* 地址范围内的数据进行与运算的十六进制值。多字节掩码值可以使用小尾数法字节顺序输入。

8.6.1.14 -O*FILE*

将在该文件中创建生成的 Intel HEX 输出。例如：

```
-Oprogram.hex
```

会将产生的输出保存到 program.hex。输出文件可以采用与其输入文件之一相同的名称，但这样做时，它将完全替换该输入文件。

8.6.1.15 -SERIAL

该选项将在某个固定地址处存储特定 HEX 值。该选项的用法如下：

```
-SERIAL=Code [+/-Increment]@Address [+/-Interval][rRepetitions]
```

其中：

- *Code* 是要存储的十六进制值，并以小尾数法字节顺序输入。
- *Increment* 是可选的，并允许在每次重复（如要求）时根据该值更改 *Code* 的值。
- *Address* 是存储该代码的单元，或其中的第一次重复的存储单元。
- *Interval* 是可选的，它指定每次重复该代码时的地址偏移。
- *Repetitions* 是可选的，它指定重复该代码的次数。

例如：

```
-SERIAL=000001@EFFE
```

会将 HEX 代码 00001h 存储到地址 EFFEh。

另一个示例：

```
-SERIAL=0000+2@1000+10r5
```

将从值 0000 开始在地址 1000h 处存储 5 个代码。后续的代码将出现在 +10h 地址间隔处，代码值将按增量 +2h 变化。

8.6.1.16 -SIZE

使用 -SIZE 选项将在标准输出中报告所产生的 HEX 镜像中的数据字节数。如果请求使用日志文件，长度还会被记录在日志文件中。

© 2013 Microchip Technology Inc.

8.6.1.17 -STRING

-STRING 选项将在某个固定地址处嵌入 ASCII 字符串。该选项的用法如下：

-STRING@*Address* [t*Code*]=”*Text*”

其中：

- *Address* 是存储该字符串的存储单元。
- *Code* 是可选的，用于在字符串每个字节之后尾随某个字节序列。这可以用于将字符串字节编码到指令内。
- *Text* 是要转换为 ASCII 并嵌入的字符串。

例如：

```
-STRING@1000=”My favorite string”
```

将在地址1000h处存储字符串My favorite string（包括nul终止符）的ASCII数据。

同样：

```
-STRING@1000t34=”My favorite string”
```

将存储相同的字符串，字符串中的每个字节都尾随 HEX 代码 34h。

8.6.1.18 -STRPACK

该选项执行与 -STRING 相同的功能，但存在两个重要的区别。首先，它仅存储每个字符的低 7 位。然后， 7 位字符对将连接在一起，并以 14 位字而不是独立字节的形式进行存储。这被称为字符串压缩。这通常仅对于程序空间以 14 位字进行寻址的器件（PIC10/12/16 器件）有用。第二个区别是 -STRING 的 t 说明符不适用于 -STRPACK 选项。

注：

© 2013 Microchip Technology Inc.

附录 A 库函数

本章按字母顺序列出标准编译器库内的函数和预处理器宏。

概要将指示可找到函数或宏的声明或定义的头文件。此外，它还会显示函数的函数原型或宏的等效原型。

__CONFIG （低档和中档器件）

概要

```
#include <xc.h>

__CONFIG(data)
```

说明

该宏用于编程将设置器件工作模式的配置熔丝。

该宏假设参数为 16 位值，并使用该参数来编程配置位。

定义了描述在每款器件上可用的每个可编程属性的 16 位掩码。这些掩码可以在通过 <xc.h> 包含的特定于芯片的头文件中找到。

可以通过与运算来选择多个属性。

示例

```
#include <xc.h>

__CONFIG(RC & UNPROTECT)

void
main (void)
{
}
```

另请参见

__EEPROM_DATA()、 __IDLOC()、 __IDLOC7() 和 CONFIG() （PIC18）

__CONFIG （PIC18）

概要

```
#include <xc.h>

__CONFIG(num, data)
```

说明

提供该宏只是为了遗留支持。对于新项目，请使用 `#pragma config`。

该宏用于编程将设置器件工作模式的配置熔丝。

该宏可接受对应于它要编程的配置寄存器的编号，之后是它用于更新配置寄存器的 16 位值。

定义了描述在每款器件上可用的每个可编程属性的 16 位掩码。这些掩码可以在通过 `<xc.h>` 包含的特定于芯片的头文件中找到。

可以通过与运算来选择多个属性。

示例

```
#include <xc.h>

__CONFIG(1,RC & OSCEN)
__CONFIG(2,WDTPS16 & BORV45)
__CONFIG(4, DEBUGEN)

void
main (void)
{
}
```

另请参见

`__EEPROM_DATA()`、`__IDLOC()`、`__IDLOC7()` 和 `CONFIG()`（低档和中档器件）

__DELAY_MS 和 __DELAY_US

概要

```
__delay_ms(x)  // request a delay in milliseconds
__delay_us(x)  // request a delay in microseconds
```

说明

由于以基于时间（而不是周期数）的方式请求延时往往更方便，所以提供了宏 `__delay_ms(x)` 和 `__delay_us(x)`。这些宏只是简单地包装 `_delay(n)` 并将基于时间的请求转换为基于系统频率的指令周期数。为了实现这一点，这些宏需要预先定义预处理器符号 `_XTAL_FREQ`。该符号应定义为由系统使用的振荡器频率（以赫兹为单位）。

如果在未定义振荡器频率符号的情况下使用这些宏或所请求的延时周期太长，将会产生错误。

另请参见

`_delay()`

© 2013 Microchip Technology Inc.

__EEPROM_DATA

概要

```
#include <xc.h>

__EEPROM_DATA(a,b,c,d,e,f,g,h)
```

说明

该宏用于在编程时将初始值存储到器件的 EEPROM 寄存器中。

必须向该宏提供每次调用它时要写入的 8 字节块，并且可以通过重复调用来存储多个块。

在每次使用时， _EEPROM_DATA() 将从 EEPROM 地址零开始写入，并将写入地址自动递增 8。

示例

```
#include <xc.h>

__EEPROM_DATA(0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07)
__EEPROM_DATA(0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F)

void
main (void)
{
}
```

另请参见

__CONFIG()

__IDLOC

概要

```
#include <xc.h>

__IDLOC(x)
```

说明

提供该宏只是为了遗留支持。对于新项目，请使用 #pragma config。

该宏会将数据放入在器件可寻址存储器之外为 ID 保留的特殊存储单元。这对于存储序列号等信息是非常有用的。

该宏将尝试向为 ID 用途保留的 4 个存储单元中写入 4 个半字节数据。

示例

```
#include <xc.h>

/* will store 1, 5, F and 0 in the ID registers */
__IDLOC(15F0);

void
main (void)
{
}
```

另请参见

__IDLOC7() 和 __CONFIG()

__IDLOC7

概要

```
#include <xc.h>

__IDLOC7(a,b,c,d)
```

说明

提供该宏只是为了遗留支持。对于新项目，请使用 #pragma config。

该宏会将数据放入在器件可寻址存储器之外为 ID 保留的特殊存储单元。这对于存储序列号等信息是非常有用的。

该宏将尝试向为 ID 用途保留的 4 个存储单元的每个存储单元中写入 7 位数据。

示例

```
#include <xc.h>

/* will store 7Fh, 70, 1 and 5Ah in the ID registers */
__IDLOC(0x7F,70,1,0x5A);

void
main (void)
{
}
```

注

并不是所有器件都允许对 ID 存储单元进行 7 位编程。要确定是否可以对特定器件使用该宏，请参见器件数据手册。

另请参见

__IDLOC() 和 __CONFIG()

_DELAY()

概要

```
#include <xc.h>

void _delay(unsigned long cycles);
```

说明

这是一个由代码生成器进行扩展的内联函数。在调用时，该程序将扩展为内联汇编延时序列。序列将包含根据参数延迟指定指令周期数的代码。参数必须为常量。

如果请求的延时周期太长（约 50,659,000 个指令周期），将会产生错误。要实现非常大的延时，可以多次调用该函数。

© 2013 Microchip Technology Inc.

示例

```
#include <xc.h>

void
main (void)
{
  control |= 0x80;
  _delay(10);    // delay for 10 cycles
  control &= 0x7F;
}
```

另请参见

_delay3()、__delay_us() 和 __delay_ms()

_DELAY3()

概要

```
#include <xc.h>

void _delay3(unsigned char cycles);
```

说明

这是一个由代码生成器进行扩展的内联函数。在调用时，该程序将扩展为内联汇编延时序列。序列将包含根据参数延迟指定周期数 3 倍时间的代码。参数可以为字节长度的常量或变量。

示例

```
#include <xc.h>

void
main (void)
{
  control |= 0x80;
  _delay(10);    // delay for 30 cycles
  control &= 0x7F;
}
```

另请参见

_delay

ABS

概要

```
#include <stdlib.h>

int abs (int j)
```

说明

abs() 函数将返回 j 的绝对值。

示例

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    int a = -5;

    printf("The absolute value of %d is %d\n", a, abs(a));
}
```

另请参见

labs() 和 fabs()

返回值

j 的绝对值。

ACOS

概要

```
#include <math.h>

double acos (double f)
```

说明

acos() 函数实现 cos() 的逆运算；即，向它传递一个介于范围 -1 至 +1 的值，它将返回以弧度表示的角度，其余弦等于该值。

示例

```
#include <math.h>
#include <stdio.h>

/* Print acos() values for -1 to 1 in degrees. */

void
main (void)
{
    float i, a;

    for(i = -1.0; i < 1.0 ; i += 0.1) {
        a = acos(i)*180.0/3.141592;
        printf("acos(%f) = %f degrees\n", i, a);
    }
}
```

另请参见

sin()、cos()、tan()、asin()、atan() 和 atan2()

返回值

以弧度表示的角度，范围为 0 至 π。

© 2013 Microchip Technology Inc.

ASCTIME

概要

```
#include <time.h>

char * asctime (struct tm * t)
```

说明

asctime() 函数接受 struct tm 结构形式的时间（函数的参数指向该结构），并返回一个描述当前日期和时间的 26 字符的字符串，其格式为：

Sun Sep 16 01:03:52 1973\n\0

请注意字符串末尾的换行符。字符串中每个字段的宽度都是固定的。以下示例将获取当前时间，使用 localtime() 将其转换为 struct tm，然后它将其转换为 ASCII，并进行打印。 time() 函数需要由用户提供 （详情请参见 time()）。

示例

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf("%s", asctime(tp));
}
```

另请参见

ctime()、 gmtime()、 localtime() 和 time()

返回值

指向字符串的指针。

注

示例要求用户提供 time() 程序，因为它无法由编译器提供。更多详细信息，请参见 time()。

ASIN

概要

```
#include <math.h>

double asin (double f)
```

说明

asin() 函数实现 sin() 的逆运算；即，向它传递一个介于范围 -1 至 +1 的值，它将返回以弧度表示的角度，其正弦等于该值。

示例

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    float i, a;

    for(i = -1.0; i < 1.0 ; i += 0.1) {
        a = asin(i)*180.0/3.141592;
        printf("asin(%f) = %f degrees\n", i, a);
    }
}
```

另请参见

sin()、 cos()、 tan()、 acos()、 atan() 和 atan2()

返回值

以弧度表示的角度，介于范围 -π/2 至 π/2。

ASSERT

概要

```
#include <assert.h>

void assert (int e)
```

说明

该宏用于调试目的；基本使用方法是将断言自由地放置在代码中的这种位置：代码的正确操作取决于某些特定条件在初始时为真。 assert() 程序可用于在运行时确保某个假设为真。例如，以下语句断言 tp 不等于 NULL：

```
assert(tp);
```

如果在运行时表达式的求值结果为假，程序将中止并发出一条消息，指示断言的源文件和行号，以及用作其参数的表达式。由于篇幅有限，无法更全面地介绍 assert() 的用法，但它与验证程序正确性的方法是密切相关的。

assert() 宏依赖于函数 _fassert() 的实现。默认情况下，它使用 printf() 打印信息。应对该程序进行检查，以确保它满足应用程序的需求。将该函数的源文件包含到项目中（即使并未修改它），然后重新编译。_fassert()函数不包含在任何库文件中。

示例

```
#include <assert.h>

void
ptrfunc (struct xyz * tp)
{
    assert(tp != 0);
}
```

注

用户需要实现底层程序 _fassert(...)。

© 2013 Microchip Technology Inc.

ATAN

概要

```
#include <math.h>

double atan (double x)
```

说明

该函数返回其参数的反正切；即，它返回范围 -π/2 至 π/2 之间的角度 “e”。

示例

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf(“atan(%f) is %f\n”, 1.5, atan(1.5));
}
```

另请参见

sin()、cos()、tan()、asin()、acos() 和 atan2()

返回值

其参数的反正切。

ATAN2

概要

```
#include <math.h>

double atan2 (double x, double x)
```

说明

该函数返回 y/x 的反正切。

示例

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf(“atan2(%f, %f) is %f\n”, 10.0, -10.0, atan2(10.0, -10.0));
}
```

另请参见

sin()、cos()、tan()、asin()、acos() 和 atan()

返回值

y/x 的反正切。

ATOF

概要

```
#include <stdlib.h>

double atof (const char * s)
```

说明

atof() 函数将扫描传递给它的字符串，并略过前导空格。然后，它会将数字的 ASCII 表示形式转换为双精度值。数字可以采用十进制、正常浮点数或科学记数法表示。

示例

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    double i;

    gets(buf);
    i = atof(buf);
    printf("Read %s: converted to %f\n", buf, i);
}
```

另请参见

atoi()、atol() 和 strtod()

返回值

双精度浮点数。如果未在字符串中找到任何数字，则返回 0.0。

ATOI

概要

```
#include <stdlib.h>

int atoi (const char * s)
```

说明

atoi() 函数将扫描传递给它的字符串，略过前导空格并读取可选的符号。然后，它会将十进制数字的 ASCII 表示形式转换为一个整数。

© 2013 Microchip Technology Inc.

示例

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = atoi(buf);
    printf(“Read %s: converted to %d\n”, buf, i);
}
```

另请参见

xtoi()、atof() 和 atol()

返回值

有符号整型。如果未在字符串中找到任何数字，则返回 0。

ATOL

概要

```
#include <stdlib.h>

long atol (const char * s)
```

说明

atol() 函数将扫描传递给它的字符串，并略过前导空格。然后，它会将十进制数字的 ASCII 表示形式转换为一个长整型值。

示例

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    long i;

    gets(buf);
    i = atol(buf);
    printf(“Read %s: converted to %ld\n”, buf, i);
}
```

另请参见

atoi() 和 atof()

返回值

长整型值。如果未在字符串中找到任何数字，则返回 0。

BSEARCH

概要

```
#include <stdlib.h>

void * bsearch (const void * key, void * base, size_t n_memb,
size_t size, int (*compar)(const void *, const void *))
```

说明

bsearch() 函数用于在有序数组中搜索与某个特定键匹配的元素。它使用二叉树搜索算法，并调用 compar 指向的函数来比较数组中的元素。

示例

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct value {
    char name[40];
    int value;
} values[100];

int
val_cmp (const void * p1, const void * p2)
{
    return strcmp(((const struct value *)p1)->name,
                  ((const struct value *)p2)->name);
}

void
main (void)
{
    char inbuf[80];
    int i;
    struct value * vp;

    i = 0;
    while(gets(inbuf)) {
        sscanf(inbuf,”%s %d”, values[i].name, &values[i].value);
        i++;
    }
    qsort(values, i, sizeof values[0], val_cmp);
    vp = bsearch(“fred”, values, i, sizeof values[0], val_cmp);
    if(!vp)
        printf(“Item ’fred’ was not found\n”);
    else
        printf(“Item ’fred’ has value %d\n”, vp->value);
}
```

另请参见

qsort()

返回值

指向匹配数组元素的指针（如果存在多个匹配元素，则可能返回其中任意一个）。如果找不到匹配项，则返回空值。

注

比较函数必须具有正确的原型。

© 2013 Microchip Technology Inc.

CEIL

概要

```
#include <math.h>

double ceil (double f)
```

说明

该程序返回不小于 f 的最小整数。

示例

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double j;

    scanf(“%lf” &j);
    printf(“The ceiling of %lf is %lf\n”, j, ceil(j));
}
```

CGETS

概要

```
#include <conio.h>

char * cgets (char * s)
```

说明

cgets() 函数将从控制台读取一行输入并放入作为参数传递的缓冲区中。它通过重复调用 getche() 来实现该功能。在读取字符时，将会对它们进行缓冲，并且使用退格键可删除先前键入的字符，使用 ctrl-U 可删除目前为止键入的整行内容。其他字符会被放入缓冲区中，使用回车或换行（换行符）可终止该函数。收集到的字符串使用空值终止。

示例

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, “exit” == 0)
            break;
        cputs(“Type ’exit’ to finish\n”);
    }
}
```

另请参见

getch()、 getche()、 putch() 和 cputs()

返回值

返回值是作为惟一参数传递的字符。

CLRWDT

概要

```
#include <xc.h>

CLRWDT();
```

说明

该宏用于清零器件的内部看门狗定时器。

示例

```
#include <xc.h>

void
main (void)
{
    WDTCON=1;
      /* enable the WDT */

    CLRWDT();
}
```

COS

概要

```
#include <math.h>

double cos (double f)
```

说明

该函数将产生其参数的余弦值，它是一个以弧度表示的角度。余弦值通过展开多项式级数逼近来计算。

示例

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf(“sin(%3.0f) = %f, cos = %f\n”, i, sin(i*C), cos(i*C));
}
```

另请参见

sin()、 tan()、 asin()、 acos()、 atan() 和 atan2()

© 2013 Microchip Technology Inc.

返回值

介于范围 -1 至 +1 的双精度值。

COSH、 SINH 和 TANH

概要

```
#include <math.h>

double cosh (double f)
double sinh (double f)
double tanh (double f)
```

说明

这些函数实现三角函数 cos()、 sin() 和 tan() 的双曲对等形式。

示例

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf(“%f\n”, cosh(1.5));
    printf(“%f\n”, sinh(1.5));
    printf(“%f\n”, tanh(1.5));
}
```

返回值

函数 cosh() 返回双曲余弦值。

函数 sinh() 返回双曲正弦值。

函数 tanh() 返回双曲正切值。

CPUTS

概要

```
#include <conio.h>

void cputs (const char * s)
```

说明

cputs() 函数会将其参数字符串写入控制台，并在字符串中每个换行符之前输出回车符。它会重复调用 putch()。在主机系统上， cputs() 与 puts() 的区别在于它将直接写入控制台，而不是使用文件 I/O。在嵌入式系统中，cputs() 和 puts() 是等效的。

示例

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, “exit” == 0)
            break;
        cputs(“Type ’exit’ to finish\n”;
    }
}
```

另请参见

cputs()、 puts() 和 putch()

CTIME

概要

```
#include <time.h>

char * ctime (time_t * t)
```

说明

ctime() 函数会将其参数指向的以秒表示的时间转换为 asctime() 所描述的相同格式的字符串。因此，示例程序将打印当前时间和日期。

示例

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf(“%s”, ctime(&clock));
}
```

另请参见

gmtime()、 localtime()、 asctime() 和 time()

返回值

指向字符串的指针。

注

示例要求用户提供 time() 程序，因为它无法由编译器提供。更多详细信息，请参见 time()。

© 2013 Microchip Technology Inc.

DEVICE_ID_READ()

概要

```
#include <xc.h>

unsigned int device_id_read(void);
```

说明

该函数返回工厂编程到芯片的器件 ID 代码。该代码可用于识别器件及其版本号。

示例

```
#include <xc.h>

void
main (void)
{
    unsigned int   id_value;
    unsigned int   device_code;
    unsigned char revision_no;

    id_value = device_id_read();
        /* lower 5 bits represent revision number
         * upper 11 bits identify device */
    device_code = (id_value >> 5);
    revision_no = (unsigned char)(id_value & 0x1F);

}
```

另请参见

`flash_read()` 和 `config_read()`

返回值

`device_id_read()`返回工厂编程的用于标识器件类型及其版本号的16位器件ID代码。

注

`device_id_read()` 仅适用于可读取自身程序存储器的那些器件。

DI 和 EI

概要

```
#include <xc.h>

void ei (void)
void di (void)
```

说明

`di()` 和 `ei()` 程序分别用于禁止和重新允许中断。它们都以宏的形式实现。以下示例显示了在某个长整型变量（该变量在中断期间会被修改）的访问代码周围使用 `ei()` 和 `di()`。在不进行这种处理的情况下，如果在访问计数值的连续字之间发生了中断，将有可能返回不正确的值。

永远不要在中断函数中调用 `ei()` 宏，在中断函数中也不需要调用 `di()`。

示例

```
#include <xc.h>

long count;

void
interrupt tick (void)
{
    count++;
}

long
getticks (void)
{
    long val;    /* Disable interrupts around access
                    to count, to ensure consistency.*/
    di();
    val = count;
    ei();
    return val;
}
```

DIV

概要

```
#include <stdlib.h>

div_t div (int numer, int denom)
```

说明

div() 函数计算将分子除以分母得到商和余数。

示例

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    div_t x;

    x = div(12345, 66);
    printf("quotient = %d, remainder = %d\n", x.quot, x.rem);
}
```

另请参见

udiv()、 ldiv() 和 uldiv()

返回值

在 div_t 结构中返回商和余数。

EEPROM 程序

说明

这些函数现在在外设库中提供。关于该库函数的完整信息，请参见外设库文档。

© 2013 Microchip Technology Inc.

EVAL_POLY

概要

```
#include <math.h>

double eval_poly (double x, const double * d, int n)
```

说明

eval_poly() 函数在 x 点对多项式进行求值，其系数包含在数组 d 中，例如：

```
y = x*x*d2 + x*d1 + d0
```

多项式的阶数通过 n 传递。

示例

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double x, y;
    double d[3] = {1.1, 3.5, 2.7};

    x = 2.2;
    y = eval_poly(x, d, 2);
    printf(“The polynomial evaluated at %f is %f\n”, x, y);
}
```

返回值

多项式在 x 点进行求值得到的双精度值。

EXP

概要

```
#include <math.h>

double exp (double f)
```

说明

exp() 程序返回其参数的指数函数；即， e 的 f 次幂。

示例

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 0.0 ; f <= 5 ; f += 1.0)
        printf(“e to %1.0f = %f\n”, f, exp(f));
}
```

另请参见

log()、 log10() 和 pow()

FABS

概要

```
#include <math.h>

double fabs (double f)
```

说明

该程序返回其双精度型参数的绝对值。

示例

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f %f\n", fabs(1.5), fabs(-1.5));
}
```

另请参见

`abs()` 和 `labs()`

闪存程序

说明

这些函数现在在外设库中提供。关于该库函数的完整信息，请参见外设库文档。

FLOOR

概要

```
#include <math.h>

double floor (double f)
```

说明

该程序返回不大于 `f` 的最大整数。

示例

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", floor( 1.5 ));
    printf("%f\n", floor( -1.5));
}
```


© 2013 Microchip Technology Inc.

FMOD

概要

```
#include <math.h>

double fmod (double x, double y)
```

说明

函数 fmod 以浮点数的形式返回 x/y 的余数。

示例

```
#include <math.h>

void
main (void)
{
    double rem, x;

    x = 12.34;
    rem = fmod(x, 2.1);
}
```

返回值

x/y 的浮点型余数。

FREXP

概要

```
#include <math.h>

double frexp (double f, int * p)
```

说明

frexp() 函数将浮点数分成归一化小数部分和 2 的整数次幂。整数存储到 p 指向的 int 对象中。它的返回值 x 处于区间 (0.5, 1.0) 中或为零， f 等于 x 乘以 2 的整数次幂 （整数存储在 *p 中）。如果 f 为零，则结果的两个部分均为零。

示例

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;
    int i;

    f = frexp(23456.34, &i);
    printf("23456.34 = %f * 2^%d\n", f, i);
}
```

另请参见

ldexp()

FTOA

概要

```
#include <stdlib.h>

char * ftoa (float f, int * status)
```

说明

函数 ftoa 将 f 的内容转换为一个字符串，字符串存储到之后返回的缓冲区中。

示例

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char * buf;
    float input = 12.34;
    int status;
    buf = ftoa(input, &status);
    printf("The buffer holds %s\n", buf);
}
```

另请参见

strtol()、itoa()、utoa() 和 ultoa()

返回值

该程序返回对写入结果的缓冲区的引用。

GETCH

概要

```
#include <conio.h>

char getch (void)
```

说明

getch() 函数以空桩的形式提供，它可以在每个项目需要时完成。通常，该函数将从与 stdin 关联的外设中读取一个数据字节，并返回其值。

示例

```
#include <conio.h>

char result;

void
main (void)
{
       result = getch();
}
```

另请参见

getche() 和 getchar()

© 2013 Microchip Technology Inc.

GETCHE

概要

```
#include <conio.h>

char getche (void)
```

说明

getche() 函数以空桩的形式提供，它可以在每个项目需要时完成。通常，该函数将从与 stdin 关联的外设中读取一个数据字节，并返回其值。与 getch() 不同，它会回显接收到的这个字符。

示例

```
#include <conio.h>

char result;

void
main (void)
{
       result = getche();
}
```

另请参见

getch() 和 getchar()

GETCHAR

概要

```
#include <stdio.h>

int getchar (void)
```

说明

getchar() 程序通常从 stdin 进行读取，但它以调用 getche() 的形式实现。

示例

```
#include <stdio.h>

void
main (void)
{
    int c;

    while((c = getchar()) != EOF)
        putchar(c);
}
```

另请参见

getc() 和 getche()

注

该程序会调用 getche()。

GETS

概要

```
#include <stdio.h>

char * gets (char * s)
```

说明

gets() 函数从标准输入读取一行并放入 s 处的缓冲区中，并删除换行符（比较：fgets()）。缓冲区使用空值终止。在嵌入式系统中，gets() 等效于 cgets()，导致重复地调用 getche() 来获取字符。可以进行编辑（使用退格键）。

示例

```
#include <stdio.h>

void
main (void)
{
    char buf[80];

    printf("Type a line: ");
    if(gets(buf))
        puts(buf);
}
```

另请参见

fgets()、freopen() 和 puts()

返回值

它返回其参数，或对于文件结束符返回 NULL。

GET_CAL_DATA

概要

```
#include <xc.h>

double get_cal_data (const unsigned char * code_ptr)
```

说明

该函数返回来自 PIC MCU 14000 校准空间的 32 位浮点型校准数据。仅使用该函数来访问 KREF、KBG、VHTHERM 和 KTC（即，32 位浮点型参数）。FOSC 和 TWDT 均可直接以字节形式访问。

© 2013 Microchip Technology Inc.

示例

```
#include <xc.h>

void
main (void)
{
    double x;
    unsigned char y;

        /* Get the slope reference ratio. */
    x = get_cal_data(KREF);

        /* Get the WDT time-out. */
    y = TWDT;
}
```

返回值

校准参数的值。

注

该函数只能在 PIC14000 上使用。

GMTIME

概要

```
#include <time.h>

struct tm * gmtime (time_t * t)
```

说明

该函数将 t 指向的时间 （自 1970 年 1 月 1 日 00:00:00 以来的秒数）转换为分解的时间，存储在 time.h 中定义的结构中。该结构在 “数据类型”部分中定义。

示例

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = gmtime(&clock);
    printf("It’s %d in London\n", tp->tm_year+1900);
}
```

另请参见

ctime()、 asctime()、 time() 和 localtime()

返回值

返回 tm 类型的结构。

注

示例要求用户提供 `time()` 程序，因为它无法由编译器提供。更多详细信息，请参见 `time()`。

ISALNUM、 ISALPHA、 ISDIGIT 和 ISLOWER 等

概要

```
#include <ctype.h>

int isalnum (char c)
int isalpha (char c)
int isascii (char c)
int iscntrl (char c)
int isdigit (char c)
int islower (char c)
int isprint (char c)
int isgraph (char c)
int ispunct (char c)
int isspace (char c)
int isupper (char c)
int isxdigit(char c)
```

说明

这些宏（在 `ctype.h` 中定义）测试所提供的字符是否属于几种重叠字符组之一的成员。请注意， `isascii()` 除外，所有这些宏都是针对 c 定义的：如果 `isascii(c)` 为 true，或如果 c = EOF。

`isalnum(c)`	c 处于 0-9、 a-z 或 A-Z 中
`isalpha(c)`	c 处于 A-Z 或 a-z 中
`isascii(c)`	c 为 7 位 ascii 字符
`iscntrl(c)`	c 为控制字符
`isdigit(c)`	c 为十进制数字
`islower(c)`	c 处于 a-z 中
`isprint(c)`	c 为打印字符
`isgraph(c)`	c 为非空格可打印字符
`ispunct(c)`	c 不是字母数字
`isspace(c)`	c 是空格、制表符或换行符
`isupper(c)`	c 处于 A-Z 中
`isxdigit(c)`	c 处于 0-9、 a-f 或 A-F 中

示例

```
#include <ctype.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = 0;
    while(isalnum(buf[i]))
        i++;
    buf[i] = 0;
    printf("'%s' is the word\n", buf);
}
```


© 2013 Microchip Technology Inc.

另请参见

toupper()、 tolower() 和 toascii()

ISDIG

概要

```
#include <ctype.h>

int isdig (int c)
```

说明

isdig() 函数测试输入字符 c，确定它是否为十进制数字（0 – 9），如果是则返回 true；否则返回 false。

示例

```
#include <ctype.h>

void
main (void)
{
    char buf[] = "1998a";
    if(isdig(buf[0]))
        printf(" type detected\n");
}
```

另请参见

isdigit()（列在 isalnum() 下）

返回值

如果为十进制数字则返回零，否则返回一个非零值。

ITOA

概要

```
#include <stdlib.h>

char * itoa (char * buf, int val, int base)
```

说明

函数 itoa 将 val 的内容转换为一个字符串，字符串存储到 buf 中。转换将根据 base 中指定的基数执行。假定 buf 引用的缓冲区分配了足够的空间。

示例

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[10];
    itoa(buf, 1234, 16);
    printf("The buffer holds %s\n", buf);
}
```

另请参见

strtol()、 utoa()、 ltoa() 和 ultoa()

返回值

该程序返回写入结果的缓冲区的副本。

LABS

概要

```
#include <stdlib.h>

int labs (long int j)
```

说明

labs() 函数将返回长整型值 j 的绝对值。

示例

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    long int a = -5;

    printf("The absolute value of %ld is %ld\n", a, labs(a));
}
```

另请参见

abs()

返回值

j 的绝对值。

LDEXP

概要

```
#include <math.h>

double ldexp (double f, int i)
```

说明

ldexp() 函数执行 frexp() 的逆运算；整数 i 与浮点数 f 的指数相加，并返回结果。

示例

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    f = ldexp(1.0, 10);
    printf("1.0 * 2^10 = %f\n", f);
}
```

另请参见

frexp()

© 2013 Microchip Technology Inc.

返回值

返回值为整数 i 与浮点值 f 的指数相加获得的结果。

LDIV

概要

```
#include <stdlib.h>

ldiv_t ldiv (long number, long denom)
```

说明

ldiv() 程序将分子除以分母，计算得到商和余数。商的符号与算术商相同。其绝对值是小于算术商绝对值的最大整数。

ldiv() 函数类似于 div() 函数，区别在于参数和返回结构的成员均为 long int 类型。

示例

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    ldiv_t lt;

    lt = ldiv(1234567, 12345);
    printf("Quotient = %ld, remainder = %ld\n", lt.quot, lt.rem);
}
```

另请参见

div()、uldiv() 和 udiv()

返回值

返回 ldiv_t 类型的结构。

LOCALTIME

概要

```
#include <time.h>

struct tm * localtime (time_t * t)
```

说明

localtime() 函数将 t 指向的时间 （自 1970 年 1 月 1 日 00:00:00 以来的秒数）转换为分解的时间，存储在 time.h 中定义的结构中。程序 localtime() 会考虑到全局整型 time_zone 的内容。它应包含本地时区相对于格林威治时间向西的分钟数。在无法预先确定该值的系统上， localtime() 将返回与 gmtime() 相同的结果。

示例

```
#include <stdio.h>
#include <time.h>

    char * wday[] = {
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
};

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf("Today is %s\n", wday[tp->tm_wday]);
}
```

另请参见

ctime()、asctime() 和 time()

返回值

返回 tm 类型的结构。

注

示例要求用户提供 time() 程序，因为它无法由编译器提供。更多详细信息，请参见 time()。

LOG 和 LOG10

概要

```
#include <math.h>

double log (double f)
double log10 (double f)
```

说明

log() 函数返回 f 的自然对数。函数 log10() 返回 f 以 10 为底的对数。

示例

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 1.0 ; f <= 10.0 ; f += 1.0)
        printf("log(%1.0f) = %f\n", f, log(f));
}
```

另请参见

exp() 和 pow()

© 2013 Microchip Technology Inc.

返回值

如果参数为负值，则返回零。

LONGJMP

概要

```
#include <setjmp.h>

void longjmp (jmp_buf buf, int val)
```

说明

longjmp() 函数与 setjmp() 配合使用可提供一种非局部转移的机制。要使用这项功能，应在外层函数中使用jmp_buf参数调用setjmp()。对setjmp()的调用将返回0。

要返回到该执行层级，可使用相同的 jmp_buf 参数从内层执行中调用 longjmp()。但是，在调用 longjmp() 时，调用 setjmp() 的函数必须仍然处于活动状态。违反该规则会导致错误，因为这将会使用包含无效数据的堆栈。 longjmp() 的 val 参数将为 setjmp() 表面上返回的值。它通常不为零，以区别于真正的 setjmp() 调用。

示例

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;

    if(i = setjmp(jb)) {
        printf("setjmp returned %d\n" i);
        exit(0);
    }
    printf("setjmp returned 0 - good\n");
    printf("calling inner...\n");
    inner();
    printf("inner returned - bad!\n");
}
```

另请参见

setjmp()

返回值

longjmp()程序永不返回。

注

在调用 longjmp() 时，调用 setjmp() 的函数必须仍然处于活动状态。违反该规则会导致灾难，因为这将会使用包含无效数据的堆栈。

LTOA

概要

```
#include <stdlib.h>

char * ltoa (char * buf, long val, int base)
```

说明

函数 ltoa 将 val 的内容转换为一个字符串，字符串存储到 buf 中。转换将根据 base 中指定的基数执行。假定 buf 引用的缓冲区分配了足够的空间。

示例

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[10];
    ltoa(buf, 12345678L, 16);
    printf("The buffer holds %s\n", buf);
}
```

另请参见

strtol()、itoa()、utoa() 和 ultoa()

返回值

该程序返回写入结果的缓冲区的副本。

MEMCHR

概要

```
#include <string.h>

void * memchr (const void * block, int val, size_t length)
```

说明

memchr() 函数类似于 strchr()，只是它不是搜索以空值终止的字符串，而是搜索指定的存储器块，在一定长度内查找某个特定字节。其参数为指向要搜索的存储器的指针、要搜索的字节值，以及块的长度。将返回一个指向块中第一个匹配字节的指针。

© 2013 Microchip Technology Inc.

示例

```
#include <string.h>
#include <stdio.h>

unsigned int ary[] = {1, 5, 0x6789, 0x23};

void
main (void)
{
    char * cp;

    cp = memchr(ary, 0x89, sizeof ary);
    if(!cp)
        printf("Not found\n");
    else
        printf("Found at offset %u\n", cp - (char *)ary);
}
```

另请参见

strchr()

返回值

返回与参数匹配的第一个字节的指针 （如存在）； 否则返回 NULL。

MEMCMP

概要

```
#include <string.h>

int memcmp (const void * s1, const void * s2, size_t n)
```

说明

类似于 strncmp()，memcmp() 函数比较两个长度为 n 的存储器块，并返回一个有符号值。与 strncmp() 不同，比较不会在空字符处停止。

示例

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int buf[10], cow[10], i;

    buf[0] = 1;
    buf[2] = 4;
    cow[0] = 1;
    cow[2] = 5;
    buf[1] = 3;
    cow[1] = 3;
    i = memcmp(buf, cow, 3*sizeof(int));
    if(i < 0)
        printf("Less than\n");
    else if(i > 0)
        printf("Greater than\n");
    else
        printf("Equal\n");
}
```

© 2013 Microchip Technology Inc.

另请参见

strncpy()、 strncmp()、 strchr()、 memset() 和 memchr()

返回值

根据 s1 指向的字符串按排序序列是小于、等于还是大于 s2 指向的字符串，返回 -1、0 或 1。

MEMCPY

概要

```
#include <string.h>

void * memcpy (void * d, const void * s, size_t n)
```

说明

memcpy() 函数会从 s 所指向的位置开始，将 n 字节的存储器内容复制到 d 所指向的存储器块。复制重叠块的结果是未定义的。memcpy() 函数与 strcpy() 的区别在于它将复制指定的字节数，而不是复制直到空终止符的所有字节。

示例

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];

    memset(buf, 0, sizeof buf);
    memcpy(buf, "A partial string", 10);
    printf("buf = ’%s’\n", buf);
}
```

另请参见

strncpy()、 strncmp()、 strchr() 和 memset()

返回值

memcpy() 程序将返回它的第一个参数。

MEMMOVE

概要

```
#include <string.h>

void * memmove (void * s1, const void * s2, size_t n)
```

说明

memmove() 函数类似于函数 memcpy()，只是它可以正确地处理重叠块的复制。即，它将根据情况进行正向或反向复制，以正确地将一个块复制到另一个与之重叠的块。

另请参见

strncpy()、 strncmp()、 strchr() 和 memcpy()

© 2013 Microchip Technology Inc.

返回值

函数 memmove() 将返回它的第一个参数。

MEMSET

概要

```
#include <string.h>

void * memset (void * s, int c, size_t n)
```

说明

memset() 函数将使用字节 c，从 s 所指向的位置开始填充 n 字节的存储器。

示例

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char abuf[20];

    strcpy(abuf, "This is a string";
    memset(abuf, 'x', 5);
    printf("buf = '%s'\n", abuf);
}
```

另请参见

strncpy()、 strncmp()、 strchr()、 memcpy() 和 memchr()

MKTIME

概要

```
#include <time.h>

time_t mktime (struct tm * tmptr)
```

说明

mktime() 函数转换由 tm 结构 tmptr 引用的本地日历时间，转换为并返回以自 1970 年 1 月 1 日经过的秒数表示的时间，或如果无法表示该时间，则返回 -1。

示例

```
#include <time.h>
#include <stdio.h>

void
main (void)
{
    struct tm birthday;

    birthday.tm_year = 83;   // the 5th of May 1983
    birthday.tm_mon = 5;
    birthday.tm_mday = 5;
    birthday.tm_hour = birthday.tm_min = birthday.tm_sec = 0;
    printf("you were born approximately %ld seconds after the unix
epoch\n",
 mktime(&birthday));
}
```

另请参见

ctime() 和 asctime()

返回值

tm结构中包含的时间表示自1970年纪元时间以来的秒数或-1（如果无法表示该时间）。

MODF

概要

```
#include <math.h>

double modf (double value, double * iptr)
```

说明

modf() 函数会将参数 value 拆分为整数和小数部分，每个部分的符号都与 value 相同。例如， -3.17 将拆分为整数部分 （-3）和小数部分 （-0.17）。

整数部分以双精度形式存储在 iptr 指向的对象中。

示例

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double i_val, f_val;

    f_val = modf( -3.17, &i_val);
}
```

返回值

value 的有符号小数部分。

© 2013 Microchip Technology Inc.

NOP

概要

```
#include <xc.h>

NOP();
```

说明

此处执行 NOP 指令。它通常用于微调延时或创建断点的句柄。在硬件中的一些敏感序列期间，有时会需要 NOP 指令。

示例

```
#include <xc.h>

void
crude_delay(unsigned char x) {
   while(x--){
        NOP();  /* Do nothing for 3 cycles */
        NOP();
        NOP();
   }
}
```

POW

概要

```
#include <math.h>

double pow (double f, double p)
```

说明

pow() 函数计算第一个参数 f 的 p 次幂。

示例

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 1.0 ; f <= 10.0 ; f += 1.0)
        printf("pow(2, %1.0f) = %f\n", f, pow(2, f));
}
```

另请参见

log()、 log10() 和 exp()

返回值

f 的 p 次幂。

PRINTF 和 VPRINTF

概要

```
#include <stdio.h>

int printf (const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vprintf (const char * fmt, va_list va_arg)
```

说明

printf() 函数是一个格式化输出程序，对 stdout 进行操作。它依靠 putch() 函数来确定 stdout 的目标。putch() 函数必须作为每个项目的一部分进行编写，同时编写用于初始化该程序所用的外设的代码。编译器的 sources 目录中可以找到 putch 函数的桩 （stub）。

向 USART 发送一个字节的典型 putch 程序可能需要编写为以下形式。

```
void putch(char data) {
  while( ! TXIF)
    continue;
  TXREG = data;
}
```

完成之后，将 putch 的源文件包含到项目中。

© 2013 Microchip Technology Inc.

在调用时，将向 printf() 程序传递一个格式字符串，后面跟随零个或多个参数的列表。该格式字符串包含一些转换规范，其中每个规范都用于打印参数列表值中的一个参数值。

每个转换规范均采用 %m.nc 的形式，其中的百分比符号 % 引入一项转换，后面跟随可选的宽度规范 m。n 规范是可选的精度规范（使用*点号*引入），c 是指定转换类型的字母。

m 前的减号 （“-”）指示在字段中对转换值进行向左调整而不是向右调整。当字段宽度大于转换所需的宽度时，将根据规范在左侧或右侧执行空格填充。如果指定对数值转换进行向右调整，并且 m 的第一个数字为 0，则将使用 0 而不是空格执行填充。对于整数格式，精度指示要输出的最低位数，并在需要时插入前导零来补足该数字。

宽度前面的井号字符 （#）指示使用备用格式。下面介绍了备用格式的性质。并不是所有格式都具有备用格式。在这些情况下，井号字符的存在没有任何作用。

如果使用字符 * 代替十进制常量（例如，在格式 %*d 中），则会从列表中获取一个整型参数来提供该值。

转换的类型包括：

f 浮点——m 代表总宽度，n 代表小数点后的位数。如果省略了 n，则它默认设为 6。如果精度为零，则除非指定备用格式，否则将省略小数点。

e 以科学记数法打印相应的参数。否则则类似于 f。

g 使用e或f格式，即两者中宽度最小而精度最高的格式。小数点后尾随的所有0都会被删除，如果小数点后没有任何数字，则它也会被删除。

o x X u d 整型转换——分别以 8、16、16、10 和 10 为基数。对于 d，转换是有符号的；否则为无符号的。精度值代表要打印的总位数，并可用于强制使用前导零。例如，%8.4x 将在 8 位宽的字段中至少打印 4 个十六进制数字。键字母前带有 l 表示值参数为长整型。字母 X 将使用大写字母 *A-F* 而不是 *a-f* 打印十六进制数字，后者将在使用字母 x 时使用。当指定备用格式时，对于八进制格式将提供前导零，对于十六进制格式将提供前导 0x 或 0X。

s 打印字符串——假定值参数是一个字符。在 m 个字符宽的字段中，最多将打印来自字符串的 n 个字符。

c 假定参数为单个字符，并按字面打印。

用作转换规范的所有其他字符都会被打印。因此， % 将产生一个百分号。

vprintf() 函数类似于 printf()，但它接受可变参数列表，而不是参数的列表。关于可变参数列表的更多信息，请参见 va_start() 的说明。以下给出了一个使用 vprintf() 的示例。

示例

```
printf("Total = %4d%", 23);
```

将产生结果 “Total = 23%”

```
printf("Size is %lx", size);
```

其中 size 为长整型，打印长度为十六进制值

```
printf("Name = %.8s", "a1234567890");
```

将产生结果 “Name = a1234567”

```
printf("xx%*d", 3, 4);
```

将产生结果 “xx 4”

```
/* vprintf example */

#include        <stdio.h>

int
error (char * s, ...)
{
    va_list ap;

    va_start(ap, s);
    printf("Error: ");
    vprintf(s, ap);
    putchar('\n');
    va_end(ap);
}

void
main (void)
{
    int i;

    i = 3;
    error("testing 1 2 %d", i);
}
```

另请参见

sprintf()

返回值

printf() 和 vprintf() 函数返回写入到 stdout 的字符数。

PUTCH

概要

```
#include <conio.h>

void putch (char c)
```

说明

putch() 函数以空桩的形式提供，它可以在每个项目需要时完成。该函数通常接受一个字节的数据，并将它发送到与 stdout 关联的外设。

© 2013 Microchip Technology Inc.

示例

```
#include <conio.h>

char * x = "This is a string";

void
main (void)
{
    char * cp;

    cp = x;
    while(*x)
        putch(*x++);
    putch('\n');
}
```

另请参见

printf() 和 putchar()

PUTCHAR

概要

```
#include <stdio.h>

int putchar (int c)
```

说明

putchar() 函数调用 putch() 来向 stdout 打印一个字符，并在 stdio.h 中定义。

示例

```
#include <stdio.h>

char * x = "This is a string";

void
main (void)
{
    char * cp;

    cp = x;
    while(*x)
        putchar(*x++);
    putchar('\n');
}
```

另请参见

putc()、getc()、freopen() 和 fclose()

返回值

作为参数传递的字符；如果发生错误则返回 EOF。

PUTS

概要

```
#include <stdio.h>

int puts (const char * s)
```

说明

puts() 函数将字符串 s 写入 stdout 流，并追加一个换行符。不会复制终止字符串的空字符。

示例

```
#include <stdio.h>

void
main (void)
{
    puts("Hello, world!");
}
```

另请参见

fputs()、 gets()、 freopen() 和 fclose()

返回值

发生错误时返回 EOF ； 否则返回零。

QSORT

概要

```
#include <stdlib.h>

void qsort (void * base, size_t nel, size_t width,
int (*func)(const void *, const void *))
```

说明

qsort() 函数是快速排序算法的一种实现。它对由 nel 个项目组成的数组进行排序，每个项目的长度为width字节，连续位于基址为base的存储器中。func的参数是qsort()用于比较项目的函数的指针。它使用要进行比较的两个项目的指针来调用 func。如果确认第一个项目大于、等于或小于第二个项目， func 应分别返回大于零、等于零或小于零的值。

© 2013 Microchip Technology Inc.

示例

```
#include <stdio.h>
#include <stdlib.h>

int array[] = {
    567, 23, 456, 1024, 17, 567, 66
};

int
sortem (const void * p1, const void * p2)
{
    return *(int *)p1 - *(int *)p2;
}

void
main (void)
{
    register int i;

    qsort(aray, sizeof array/sizeof array[0],
        sizeof array[0], sortem);
    for(i = 0 ; i != sizeof array/sizeof array[0] ; i++)
        printf("%d\t", array[i]);
    putchar('\n');
}
```

注

函数参数必须为指向类似以下函数类型的指针：

```
int func (const void *, const void *)
```

例如，它必须接受两个 `const void*` 参数，并且必须具有原型声明。

RAND

概要

```
#include <stdlib.h>

int rand (void)
```

说明

`rand()` 函数是一个伪随机数发生器。它返回一个介于范围 0 至 32767 的整数，每次调用时都会以伪随机方式变化。如果从相同的起始点开始，该算法将生成一个确定性的序列。起始点使用 `srand()` 调用设置。以下给出了使用 `time()` 函数每次为序列生成不同起始点的示例。

示例

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t toc;
    int i;

    time(&toc);
    srand((int)toc);
    for(i = 0 ; i != 10 ; i++)
        printf("%d\t", rand());
    putchar('\n');
}
```

另请参见

srand()

注

示例要求用户提供 time() 程序，因为它无法由编译器提供。更多详细信息，请参见 time()。

ROUND

概要

```
#include <math.h>

double round (double x)
```

说明

round 函数将参数舍入为最接近的整数值，但采用浮点格式。介于整数值之间的值将进行向上舍入。

示例

```
#include <math.h>

void
main (void)
{
    double input, rounded;
    input = 1234.5678;
    rounded = round(input);
}
```

另请参见

trunc()

© 2013 Microchip Technology Inc.

SETJMP

概要

```
#include <setjmp.h>

int setjmp (jmp_buf buf)
```

说明

setjmp() 函数与 longjmp() 配合使用来实现非局部转移。更多信息，请参见 longjmp()。

示例

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;

    if(i = setjmp(jb)) {
        printf("setjmp returned %d\n", i);
        exit(0);
    }
    printf("setjmp returned 0 - good\n");
    printf("calling inner...\n");
    inner();
    printf("inner returned - bad!\n");
}
```

另请参见

longjmp()

返回值

setjmp() 函数在实际调用后返回零，如果它是在调用 longjmp() 之后显然返回的，则返回非零。

SIN

概要

```
#include <math.h>

double sin (double f)
```

说明

该函数返回其参数的正弦函数。

示例

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("sin(%3.0f) = %f\n", i, sin(i*C));
        printf("cos(%3.0f) = %f\n", i, cos(i*C));
}
```

另请参见

cos()、tan()、asin()、acos()、atan() 和 atan2()

返回值

f 的正弦值。

SLEEP

概要

```
#include <xc.h>

SLEEP();
```

说明

该宏用于将器件置为低功耗待机模式。

示例

```
#include <xc.h>
extern void init(void);

void
main (void)
{
    init(); /* enable peripherals/interrupts */

    while(1)
        SLEEP();  /* save power while nothing happening */
}
```

SPRINTF 和 VSPRINTF

概要

```
#include <stdio.h>

int sprintf (char * buf, const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vsprintf (char * buf, const char * fmt, va_list ap)
```


© 2013 Microchip Technology Inc.

说明

sprintf() 函数的工作方式类似于 printf()，只是它不是将转换的输出放在 stdout 流中，而是将字符放在 buf 处的缓冲区中。结果字符串将以空值终止，并返回缓冲区中的字符数。

vsprintf() 函数类似于 sprintf()，但它接受可变参数列表，而不是参数的列表。关于可变参数列表的更多信息，请参见 va_start() 的说明。

另请参见

printf() 和 sscanf()

返回值

这两个程序均返回放入缓冲区中的字符数。

SQRT

概要

```
#include <math.h>

double sqrt (double f)
```

说明

函数 sqrt() 使用牛顿逼近法实现平方根程序。

示例

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double i;

    for(i = 0 ; i <= 20.0 ; i += 1.0)
        printf("square root of %.1f = %f\n", i, sqrt(i));
}
```

另请参见

exp()

返回值

返回平方根的值。

注

如果参数为负值，则会发生域错误，并且 errno 将被设置为 EDOM。

SRAND

概要

```
#include <stdlib.h>

void srand (unsigned int seed)
```

说明

srand() 函数使用给定的 seed 初始化 rand() 访问的随机数发生器。这提供了一种机制来改变 rand() 所产生的伪随机序列的起始点。

示例

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t toc;
    int i;

    time(&toc);
    srand((int)toc);
    for(i = 0 ; i != 10 ; i++)
        printf("%d\t", rand());
    putchar('\n');
}
```

另请参见

rand()

SSCANF 和 VSSCANF

概要

```
#include <stdio.h>

int sscanf (const char * buf, const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vsscanf (const char * buf, const char * fmt, va_list ap)
```

说明

sscanf() 函数的工作方式类似于 scanf()，只是转换不是从 stdin 获取，而是从 buf 处的字符串中获取。

vsscanf() 函数接受一个参数，而不是参数的列表。关于可变参数列表的更多信息，请参见 va_start() 的说明。

另请参见

scanf()、 fscanf() 和 sprintf()

返回值

如果出现输入故障，则返回 EOF 的值，否则返回输入项的数量。

© 2013 Microchip Technology Inc.

STRCAT

概要

```
#include <string.h>

char * strcat (char * s1, const char * s2)
```

说明

该函数会将字符串 s2 追加（连接）到字符串 s1 的末尾。结果将以空值终止。参数 s1 必须指向一个足以存放结果字符串的字符数组。

示例

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = "... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

另请参见

strcpy()、 strcmp()、 strncat() 和 strlen()

返回值

返回 s1 的值。

STRCHR 和 STRICHR

概要

```
#include <string.h>

char * strchr (const char * s, int c)
char * strichr (const char * s, int c)
```

说明

strchr() 函数搜索字符串 s 中是否存在字符 c。如果找到一个，则返回一个指向该字符的指针，否则返回 NULL。

strichr() 函数是该函数不区分大小写的版本。

示例

```
#include <strings.h>
#include <stdio.h>

void
main (void)
{
    static char temp[] = "Here it is...";
    char c = 's';

    if(strchr(temp, c))
        printf("Character %c was found in string\n", c);
    else
        printf("No character was found in string");
}
```

另请参见

strrchr()、strlen() 和 strcmp()

返回值

返回第一个匹配项的指针，或者如果字符串中不存在该字符，则返回 NULL。

注

虽然函数接受对应于字符的整型参数，但它仅使用值的低 8 位。

STRCMP 和 STRICMP

概要

```
#include <string.h>

int strcmp (const char * s1, const char * s2)
int stricmp (const char * s1, const char * s2)
```

说明

strcmp() 函数将比较它的两个以空值终止的字符串参数，并返回一个有符号整数来指示 s1 是小于、等于还是大于 s2。使用标准排序序列进行比较，即 ASCII 字符集的排序序列。

stricmp() 函数是该函数不区分大小写的版本。

示例

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    int i;

    if((i = strcmp("ABC", "ABc")) < 0)
        printf("ABC is less than ABc\n");
    else if(i > 0)
        printf("ABC is greater than ABc\n");
    else
        printf("ABC is equal to ABc\n");
}
```


© 2013 Microchip Technology Inc.

另请参见

strlen()、 strncmp()、 strcpy() 和 strcat()

返回值

小于、等于或大于零的有符号整数。

注

其他 C 实现可能使用不同的排序序列；返回值为负值、零或正值；即，不要明确地检查返回值是否为 -1 或 1。

STRCPY

概要

```
#include <string.h>

char * strcpy (char * s1, const char * s2)
```

说明

该函数会将以空值终止的字符串 s2 复制到 s1 指向的字符数组。目标数组必须足够大，足以容纳包括空终止符在内的整个字符串。

示例

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = "... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

另请参见

strncpy()、 strlen()、 strcat() 和 strlen()

返回值

返回目标缓冲区 s1。

STRCSPN

概要

```
#include <string.h>

size_t strcspn (const char * s1, const char * s2)
```

说明

strcspn() 函数返回在 s1 指向的字符串中，不包含来自 s2 所指向字符串的字符的起始部分的长度。

示例

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    static char set[] = "xyz";

    printf("%d\n", strcspn("abcdevwxyz", set));
    printf("%d\n", strcspn("xxxbcadefs", set));
    printf("%d\n", strcspn("1234567890", set));
}
```

另请参见

strspn()

返回值

返回该部分的长度。

STRLEN

概要

```
#include <string.h>

size_t strlen (const char * s)
```

说明

strlen() 函数返回字符串 s 中的字符数 （不包括空终止符）。

示例

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = "... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

返回值

空终止符前的字符数。

© 2013 Microchip Technology Inc.

STRNCAT

概要

```
#include <string.h>

char * strncat (char * s1, const char * s2, size_t n)
```

说明

该函数会将字符串 s2 追加（连接）到字符串 s1 的末尾。最多复制 n 个字符，结果将以空值终止。 s1 必须指向一个足以容纳结果字符串的字符数组。

示例

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = "... end of line";
    strncat(s1, s2, 5);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

另请参见

strcpy()、 strcmp()、 strcat() 和 strlen()

返回值

返回 s1 的值。

STRNCMP 和 STRNICMP

概要

```
#include <string.h>

int strncmp (const char * s1, const char * s2, size_t n)
int strnicmp (const char * s1, const char * s2, size_t n)
```

说明

strncmp() 函数将比较它的两个以空值终止的字符串参数（最多 n 个字符），并返回一个有符号整数来指示 s1 是小于、等于还是大于 s2。使用标准排序序列进行比较，即 ASCII 字符集的排序序列。

strnicmp() 函数是该函数不区分大小写的版本。

示例

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int i;

    i = strncmp("abcxyz", "abcxyz", 6);
    if(i == 0)
        printf("The strings are equal\n");
    else if(i > 0)
        printf("String 2 less than string 1\n");
    else
        printf("String 2 is greater than string 1\n");
}
```

另请参见

strlen()、strcmp()、strcpy() 和 strcat()

返回值

小于、等于或大于零的有符号整数。

注

其他 C 实现可能使用不同的排序序列；返回值为负值、零或正值；即，不要明确地测试返回值是否为 -1 或 1。

STRNCPY

概要

```
#include <string.h>

char * strncpy (char * s1, const char * s2, size_t n)
```

说明

该函数会将以空值终止的字符串 s2 复制到 s1 指向的字符数组。最多复制 n 个字符。如果字符串 s2 的长度大于 n，目标字符串将不是以空值终止。目标数组必须足够大，足以容纳包括空终止符在内的整个字符串。

示例

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strncpy(buffer, "Start of line", 6);
    s1 = buffer;
    s2 = "... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```


© 2013 Microchip Technology Inc.

另请参见

strcpy()、 strcat()、 strlen() 和 strcmp()

返回值

返回目标缓冲区 s1。

STRPBRK

概要

```
#include <string.h>

char * strpbrk (const char * s1, const char * s2)
```

说明

strpbrk() 函数返回一个指针 （指向字符串 s1 中第一次出现字符串 s2 中任意字符的位置）或空值 （如果 s1 中不存在 s2 中的任何字符）。

示例

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = "This is a string.";

    while(str != NULL) {
        printf("%s\n", str);
        str = strpbrk(str+1, "aeiou");
    }
}
```

返回值

指向第一个匹配字符的指针，或如果未找到任何字符，则返回 NULL。

STRRCHR 和 STRRICHR

概要

```
#include <string.h>

char * strrchr (char * s, int c)
char * strrichr (char * s, int c)
```

说明

strrchr() 函数类似于 strchr() 函数，但它从字符串末尾处开始搜索，而不是从起始处开始；即，它查找以空值终止的字符串 s 中的*最后一处*字符 c。如果成功，它将返回指向匹配位置的指针，否则将返回 NULL。

strrichr() 函数是该函数不区分大小写的版本。

示例

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = "This is a string.";

    while(str != NULL) {
        printf("%s\n", str);
        str = strrchr(str+1, 's');
    }
}
```

另请参见

strchr()、strlen()、strcmp()、strcpy() 和 strcat()

返回值

指向字符的指针，如果未找到则返回 NULL。

STRSPN

概要

```
#include <string.h>

size_t strspn (const char * s1, const char * s2)
```

说明

strspn() 函数返回在 s1 指向的字符串中，完全包含来自 s2 所指向字符串的字符的起始部分的长度。

示例

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("%d\n", strspn("This is a string", "This"));
    printf("%d\n", strspn("This is a string", "this"));
}
```

另请参见

strcspn()

返回值

该部分的长度。

STRSTR 和 STRISTR

概要

```
#include <string.h>

char * strstr (const char * s1, const char * s2)
char * stristr (const char * s1, const char * s2)
```


© 2013 Microchip Technology Inc.

说明

strstr() 函数在 s1 所指向的字符串中查找第一次出现 s2 所指向字符串的位置。

stristr() 程序是该函数不区分大小写的版本。

示例

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("%d\n", strstr("This is a string", "str"));
}
```

返回值

指向所找到字符串的指针，或如果未找到该字符串，则返回空值。

STRTOD

概要

```
#include <stdlib.h>

double strtod (const char * s, const char ** res)
```

说明

解析字符串 s，将它转换为双精度浮点型。该函数会在输入中查找第一个由所期望形式的字符组成的子字符串，并在略过前导空格字符之后对它进行转换。如果 res 不为 NULL，它将被设为指向转换后的子字符串的第一个字符。

示例

```
#include <stdio.h>
#include <strlib.h>

void
main (void)
{
    char buf[] = "35.7  23.27";
    char * end;
    double in1, in2;

    in1 = strtod(buf, &end);
    in2 = strtod(end, NULL);
    printf("in comps: %f, %f\n", in1, in2);
}
```

另请参见

atof()

返回值

返回一个双精度值，它代表输入字符串转换后的浮点值。

STRTOL

概要

```
#include <stdlib.h>

double strtol (const char * s, const char ** res, int base)
```

说明

解析字符串 s，将它转换为长整型。该函数会在输入中查找第一个由所期望形式的字符组成的子字符串，并在略过前导空格字符之后对它进行转换。输入的基数将通过 base 确定。如果它为零，则基数默认设为 10。如果 res 不为 NULL，它将被设为指向转换后的子字符串的第一个字符。

示例

```
#include <stdio.h>
#include <strlib.h>

void
main (void)
{
    char buf[] = "0X299 0x792";
    char * end;
    long in1, in2;

    in1 = strtol(buf, &end, 16);
    in2 = strtol(end, NULL, 16);
    printf("in (decimal): %ld, %ld\n", in1, in2);
}
```

另请参见

strtod()

返回值

返回表示输入字符串使用指定基数转换后的长整型值。

STRTOK

概要

```
#include <string.h>

char * strtok (char * s1, const char * s2)
```

说明

对 strtok() 进行多次调用会将字符串 s1（它包含由零个或多个文本标记组成的序列，这些标记使用来自分隔符字符串 s2 的一个或多个字符分隔）拆分为单独的标记。

第一次调用必须提供字符串 s1。该调用会返回指向第一个标记的第一个字符的指针，或如果未找到标记，则返回 NULL。标记间的分隔符将被一个空字符覆盖，它将终止当前标记。

对于 strtok() 的后续调用，s1 应设置为 NULL。这些调用会从所找到的上一个标记的末尾开始搜索，并且再次返回指向下一个标记的第一个字符的指针，或如果未找到更多标记，则返回 NULL。

© 2013 Microchip Technology Inc.

示例

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * ptr;
    char buf[] = "This is a string of words.";
    char * sep_tok = ",?! "

    ptr = strtok(buf, sep_tok);
    while(ptr != NULL) {
        printf("%s\n", ptr);
        ptr = strtok(NULL, sep_tok);
    }
}
```

返回值

返回指向标记第一个字符的指针，或如果未找到标记，则返回空值。

注

分隔符字符串 s2 在调用之间可能会不同。

TAN

概要

```
#include <math.h>

double tan (double f)
```

说明

tan() 函数计算 f 的正切值。

示例

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("tan(%3.0f) = %f\n", i, tan(i*C));
}
```

另请参见

sin()、cos()、asin()、acos()、atan() 和 atan2()

返回值

f 的正切值。

TIME

概要

```
#include <time.h>

time_t time (time_t * t)
```

说明

该函数没有提供，因为它依赖于提供当前时间的目标系统。该函数由用户实现。在实现时，该函数应返回以自 1970 年 1 月 1 日 00:00:00 以来的秒数表示的当前时间。如果参数 t 不等于 NULL，同一个值将存储到 t 所指向的对象。

示例

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf("%s", ctime(&clock));
}
```

另请参见

ctime()、 gmtime()、 localtime() 和 asctime()

返回值

在实现时，该程序将返回以自 1970 年 1 月 1 日 00:00:00 以来的秒数表示的当前时间。

注

time() 程序没有提供，如果需要，用户必须按照上述规范实现该程序。

TOLOWER、 TOUPPER 和 TOASCII

概要

```
#include <ctype.h>

char toupper (int c)
char tolower (int c)
char toascii (int c)
```

说明

toupper() 函数将其小写形式的字母参数转换为大写形式，tolower() 程序执行反向转换， toascii() 宏返回确保处于范围 0-0177 中的结果。如果其参数不是字母字符，函数 toupper() 和 tolower() 将返回其参数。

© 2013 Microchip Technology Inc.

示例

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void
main (void)
{
    char * array1 = "aBcDE";
    int i;

    for(i=0;i < strlen(array1); ++i) {
        printf("%c", tolower(array1[i]));
    }
    printf("n");
}
```

另请参见

islower()、 isupper() 和 isascii() 等

TRUNC

概要

```
#include <math.h>

double trunc (double x)
```

说明

trunc 函数会采用浮点格式将参数舍入为最接近的整数值，即不大于该参数的幅值。

示例

```
#include <math.h>

void
main (void)
{
    double input, rounded;
    input = 1234.5678;
    rounded = trunc(input);
}
```

另请参见

round()

UDIV

概要

```
#include <stdlib.h>

int udiv (unsigned num, unsigned denom)
```

说明

udiv() 函数计算 number 和 denom 相除得到的商和余数，并将结果存储到返回的 udiv_t 结构中。

示例

```
#include <stdlib.h>

void
main (void)
{
    udiv_t result;
    unsigned  num = 1234, den = 7;

    result = udiv(num, den);
}
```

另请参见

uldiv()、div() 和 ldiv()

返回值

以 udiv_t 结构的形式返回商和余数。

ULDIV

概要

```
#include <stdlib.h>

int uldiv (unsigned long num, unsigned long denom)
```

说明

uldiv() 函数计算 number 和 denom 相除得到的商和余数，并将结果存储到返回的 uldiv_t 结构中。

示例

```
#include <stdlib.h>

void
main (void)
{
    uldiv_t result;
    unsigned long num = 1234, den = 7;

    result = uldiv(num, den);
}
```

另请参见

ldiv()、udiv() 和 div()

返回值

以 uldiv_t 结构的形式返回商和余数。

UTOA

概要

```
#include <stdlib.h>

char * utoa (char * buf, unsigned val, int base)
```


© 2013 Microchip Technology Inc.

说明

函数 utoa() 将 val 的无符号内容转换为一个字符串，字符串存储到 buf 中。转换将根据 base 中指定的基数执行。假定 buf 引用的缓冲区分配了足够的空间。

示例

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[10];
    utoa(buf, 1234, 16);
    printf("The buffer holds %s\n", buf);
}
```

另请参见

strtol()、itoa()、ltoa() 和 ultoa()

返回值

该程序返回写入结果的缓冲区的副本。

VA_START、VA_ARG 和 VA_END

概要

```
#include <stdarg.h>

void va_start (va_list ap, parmN)
type  va_arg  (ap, type)
void va_end  (va_list ap)
```

说明

这些宏用于通过一种可移植的方式访问函数的参数，这些参数在原型中使用省略号（...）表示，而提供给函数的参数的类型和数量在编译时是未知的。

函数最右侧的参数（显示为 *parmN*）在这些宏中发挥着重要的作用，因为它是访问更多参数的起始点。在接受可变参数数量的函数中，应声明 va_list 类型的变量，然后用该变量和 *parmN* 的名称调用宏 va_start()。这将初始化该变量，以便随后调用宏 va_arg() 来访问后续的参数。

每次调用 va_arg() 都需要两个参数；先前定义的变量，以及下一个参数的预期类型的类型名称。请注意，通过这种方式访问的所有参数都会被默认约定拓宽为 int、unsigned int 或 double 类型。例如，如果传递了某个字符参数，则应该通过 va_arg(ap, int) 访问它，因为 char 类型会被拓宽为 int 类型。

下面给出了一个接受一个整型参数，并跟随一些其他参数的函数示例。在该示例中，函数要求后续参数为 char 类型，但请注意编译器并不知道这一点，编程人员需要负责确保提供正确的参数。

示例

```
#include <stdio.h>
#include <stdarg.h>

void
pf (int a, ...)
{
    va_list ap;

    va_start(ap, a);
    while(a--)
        puts(va_arg(ap, char *));
    va_end(ap);
}

void
main (void)
{
    pf(3, "Line 1", "Line 2", "Line 3");
}
```

XTOI

概要

```
#include <stdlib.h>

unsigned xtoi (const char * s)
```

说明

xtoi() 函数会扫描传递给它的字符串，略过前导空格并读取可选的符号，然后将十六进制数字的 ASCII 表示形式转换为整型。

示例

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = xtoi(buf);
    printf("Read %s: converted to %x\n", buf, i);
}
```

另请参见

atoi()

返回值

无符号整型。如果未在字符串中找到任何数字，则返回 0。

© 2013 Microchip Technology Inc.

附录 B 错误和警告消息

本章列出了 MPLAB XC8 C 编译器的错误、警告和建议性消息，以及每条消息的说明。这是完整的历史消息集合，涵盖了所有先前的 HI-TECH C 编译器和所有编译器版本。并不是此处显示的所有消息都与您使用的编译器版本有关。

大多数消息都指定了一个惟一编号，它显示在每条消息说明之前的括号中，在发出消息时编译器也会打印该编号。此处显示的消息按它们的编号进行排序。无编号的消息在结尾列出，并按字母顺序排序。

在错误消息对面的括号中会列出可能产生该消息的应用程序的名称。在某些情况下，会给出可能触发该错误的代码或选项的示例。在错误消息中，使用 * 来表示编译器将替换的、特定于该特定错误的字符串。

请注意，C 代码或汇编源代码中的一个问题可能会触发多条错误消息。您应尝试按错误或警告的产生顺序来解决它们。

消息 1-249

(1) too many errors (*)

（所有应用程序）

正在执行的编译器应用程序遇到太多错误，将立即退出。将会处理其他未编译的源文件，但不会运行通常在适当时候执行的编译器应用程序。可以接受的错误数量可以使用 --ERROR5 选项控制，请参见**第 4.8.29 节 “--ERRORS：最大错误数”**。

(2) error/warning (*) generated, but no description available

（所有应用程序）

正在执行的编译器应用程序发出了一条消息（建议性 / 警告 / 错误），但消息说明文件（MDF）中没有可供打印的说明。这可能是因为 MDF 已过时，或消息问题尚未被译为选定的语言。

(3) malformed error information on line *, in file *

（所有应用程序）

编译器尝试加载选定语言的消息，但消息说明文件（MDF）已损坏，无法正确读取。

(100) unterminated #if[n][def] block from line *

（预处理器）

#if 或类似代码块未使用匹配的 #endif 终止，例如：

```
#if INPUT        /* error flagged here */
void main(void)
{
  run();
}                /* no #endif was found in this module */
```

(101) #* may not follow #else *（预处理器）*

在与某个 #else 相同的条件块中使用了 #else 或 #elif。它们只能跟随在 #if 之后，例如：

```
#ifdef FOO
  result = foo;
#else
  result = bar;
#elif defined(NEXT)   /* the #else above terminated the #if */
  result = next(0);
#endif
```

(102) #* must be in an #if *（预处理器）*

#elif、#else 或 #endif 伪指令之前必须具有匹配的 #if 行。如果显然存在对应的 #if 行，则检查是否存在诸如额外的 #endif 或未正确终止的注释之类的内容，例如：

```
#ifdef FOO
  result = foo;
#endif
  result = bar;
#elif defined(NEXT)   /* the #endif above terminated the #if */
  result = next(0);
#endif
```

(103) #error: * *（预处理器）*

这是一个编程人员产生的错误；有一条伪指令导致了一个有意的错误。它通常用于检查编译时定义等方面。删除该伪指令可去除错误，但请先检查该伪指令为什么出现在此处。

(104) preprocessor #assert failure *（预处理器）*

预处理器 #assert 伪指令的参数求值为零。这是一个编程人员引起的错误。

```
#assert SIZE == 4  /* size should never be 4 */
```

(105) no #asm before #endasm *（预处理器）*

遇到了一个 #endasm 操作符，但前面没有匹配的 #asm，例如：

```
void cleardog(void)
{
  clrwdt
#endasm  /* in-line assembler ends here,
            only where did it begin? */
}
```

(106) nested #asm directives *（预处理器）*

嵌套 #asm 伪指令是不合法的。请检查是否缺少或拼错了 #endasm 伪指令，例如：

```
#asm
   MOVE   r0, #0aah
#asm      ; previous #asm must be closed before opening another
   SLEEP
#endasm
```


© 2013 Microchip Technology Inc.

(107) illegal # directive "*" *（预处理器和解析器）*

编译器不理解 # 伪指令。它可能是错误拼写的预处理器 # 伪指令，例如：

```
#indef DEBUG  /* oops -- that should be #undef DEBUG */
```

(108) #if[n][def] without an argument *（预处理器）*

预处理器伪指令 #if、#ifdef 和 #ifndef 必须有一个参数。#if 的参数应为一个表达式，而 #ifdef 或 #ifndef 的参数应为单个名称，例如：

```
#if            /* oops -- no argument to check */
  output = 10;
#else
  output = 20;
#endif
```

(109) #include syntax error *（预处理器）*

#include 的文件名参数的语法无效。#include 的参数必须为有效的文件名，包含在双引号 "" 或尖括号 < > 中。其中不应包含空格，并且必须存在右引号或右括号。在行中不应存在注释之外的其他内容，例如：

```
#include stdio.h  /* oops -- should be: #include <stdio.h> */
```

(110) too many file arguments; usage: cpp [input [output]] *（预处理器）*

CPP 应最多使用两个文件参数调用。如果预处理器是由编译器驱动程序执行的，请联系 Microchip 技术支持。

(111) redefining preprocessor macro "*" *（预处理器）*

指定的宏被重新定义为不同于原始定义的内容。如果是要有意地重新定义宏，请先使用 #undef 取消原始定义，例如：

```
#define ONE 1
/* elsewhere: */
/* Is this correct? It will overwrite the first definition. */
#define ONE one
```

(112) #define syntax error *（预处理器）*

宏定义具有语法错误。这可能是由于宏或形参名称不是以字母开始或缺少*右括号*)，例如：

```
#define FOO(a, 2b)  bar(a, 2b)  /* 2b is not to be! */
```

(113) unterminated string in preprocessor macro body *（预处理器和汇编器）*

宏定义包含缺少右引号的字符串。

(114) illegal #undef argument *（预处理器）*

#undef 的参数必须为一个有效的名称。它必须以字母开始，例如：

```
#undef 6YYY  /* this isn't a valid symbol name */
```

(115) recursive preprocessor macro definition of "*" defined by "*" *（预处理器）*

指定宏的定义方式使得在扩展它时导致递归扩展自身。

(116) end of file within preprocessor macro argument from line * *（预处理器）*

宏参数未进行终止。这可能意味着宏调用中省略了右括号。给出的行号代表宏参数的起始行，例如：

```
#define FUNC(a, b) func(a+b)
FUNC(5, 6;    /* oops -- where is the closing bracket? */
```

(117) misplaced constant in #if *（预处理器）*

#if 表达式中的常量应仅出现在语法上正确的位置。该错误很有可能是由于遗漏操作符而引起的，例如：

```
#if FOO BAR  /* oops -- did you mean: #if FOO == BAR ? */
```

(118) stack overflow processing #if expression *（预处理器）*

在某个 #if 表达式中，预处理器填满了表达式求值堆栈。请简化表达式——它可能包含太多带括号的子表达式。

(119) invalid expression in #if line *（预处理器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(120) operator "*" in incorrect context *（预处理器）*

#if 表达式中遇到了未正确放置的操作符 （两个二元操作符未使用值分隔），例如：

```
#if FOO * % BAR == 4  /* what is "* %" ? */
  #define BIG
#endif
```

(121) expression stack overflow at operator "*" *（预处理器）*

#if 行中的表达式使用大小为 128 的堆栈进行求值。极复杂的表达式有可能使它发生溢出。请简化该表达式。

(122) unbalanced parenthesis at operator "*" *（预处理器）*

#if 表达式的求值发现不匹配的括号。请检查表达式中的括号使用是否正确，例如：

```
#if ((A) + (B)  /* oops -- a missing ), I think */
  #define ADDED
#endif
```

(123) misplaced "?" or ":"; previous operator is "*" *（预处理器）*

在 #if 表达式中遇到了未与相应 ? 操作符匹配的冒号操作符，例如：

```
#if XXX : YYY  /* did you mean:  #if COND ? XXX : YYY */
```

(124) illegal character "*" in #if *（预处理器）*

在 #if 表达式中存在一个无关字符。有效字符包括字母、数字和构成可接受操作符的字符，例如：

```
#if YYY  /* what are these characters doing here? */
  int m;
#endif
```


© 2013 Microchip Technology Inc.

(125) illegal character (* decimal) in #if *（预处理器）*

在 #if 表达式中存在一个无关的不可打印字符。有效字符包括字母、数字和构成可接受操作符的字符，例如：

```
#if ^S YYY  /* what is this control characters doing here? */
  int m;
#endif
```

(126) strings can't be used in #if *（预处理器）*

预处理器不允许在 #if 表达式中使用字符串，例如：

```
/* no string operations allowed by the preprocessor */
#if MESSAGE > "hello"
#define DEBUG
#endif
```

(127) bad syntax for defined() in #[el]if *（预处理器）*

预处理器表达式中的 defined() 伪函数要求其参数必须为单个名称。名称必须以字母开始，并应括在括号中，例如：

```
/* oops -- defined expects a name, not an expression */
#if defined(a&b)
  input = read();
#endif
```

(128) illegal operator in #if *（预处理器）*

#if 表达式具有非法的操作符。请检查语法是否正确，例如：

```
#if FOO = 6  /* oops -- should that be: #if FOO == 5 ? */
```

(129) unexpected "\" in #if *（预处理器）*

*反斜杠*错误地处于 #if 语句中，例如：

```
#if FOO == \34
  #define BIG
#endif
```

(130) unknown type "*" in #[el]if sizeof() *（预处理器）*

预处理器 sizeof() 中使用了未知的类型。预处理器只能对基本类型或指向基本类型的指针进行 sizeof() 求值，例如：

```
#if sizeof(unt) == 2  /* should be: #if sizeof(int) == 2 */
  i = 0xFFFF;
#endif
```

(131) illegal type combination in #[el]if sizeof() *（预处理器）*

预处理器在 #if 表达式中的 sizeof() 的参数中发现非法的类型组合，例如：

```
/* To sign, or not to sign, that is the error. */
#if sizeof(signed unsigned int) == 2
  i = 0xFFFF;
#endif
```

(132) no type specified in #[el]if sizeof() *（预处理器）*

在预处理器 #if 表达式中使用了 sizeof()，但未指定任何类型。在预处理器表达式中， sizeof() 的参数必须为有效的简单类型或指向简单类型的指针，例如：

```
#if sizeof()  /* oops -- size of what? */
  i = 0;
#endif
```

(133) unknown type code (0x*) in #[el]if sizeof() *（预处理器）*

预处理器在sizeof()表达式求值中发生了内部错误。请检查是否存在格式错误的类型说明符。这是内部错误。详情请联系 Microchip 技术支持。

(134) syntax error in #[el]if sizeof() *（预处理器）*

预处理器在 #if 表达式中的 sizeof 的参数中发现语法错误，例如：可能原因为括号不匹配或类似错误，例如：

```
#if sizeof(int == 2)  // oops - should be: #if sizeof(int) == 2
  i = 0xFFFF;
#endif
```

(135) unknown operator (*) in #if *（预处理器）*

预处理器试图对具有自己不理解的操作符的表达式进行求值。这是内部错误。详情请联系 Microchip 技术支持。

(137) strange character "*" after ## *（预处理器）*

在标记连接操作符 ## 之后发现一个既不是字母也不是数字的字符。由于该操作符的结果必须为合法标记，所以操作数必须为仅包含字母和数字的标记，例如：

```
/* the ’ character will not lead to a valid token */
#define cc(a, b) a ## ’b
```

(138) strange character (*) after ## *（预处理器）*

在标记连接操作符 ## 之后发现一个既不是字母也不是数字的不可打印字符。由于该操作符的结果必须为合法标记，所以操作数必须为仅包含字母和数字的标记，例如：

```
/* the ’ character will not lead to a valid token */
#define cc(a, b) a ## ’b
```

(139) end of file in comment *（预处理器）*

在注释内遇到了文件结束符。请检查是否缺少注释结束标志，例如：

```
   /* Here the comment begins. I諱 not sure where I end, though
}
```

(140) can’t open * file "*": * *（驱动程序、预处理器、代码生成器和汇编器）*

无法打开指定的命令文件进行读取。请确认在命令行上指定的文件的拼写和路径，例如：

```
xc8 @communds
```

应为：

```
xc8 @commands
```


© 2013 Microchip Technology Inc.

(141) can’t open * file "*": * *（任意）*

无法创建输出文件。请确认在命令行上指定的文件的拼写和路径。

(144) too many nested #if blocks *（预处理器）*

`#if` 和 `#ifdef` 等条件块最多只能嵌套 32 级。

(146) #include filename too long *（预处理器）*

在查找包含文件时，所构造的文件名超出了内部缓冲区的长度。由于该缓冲区的长度为 4096 字节，通常不太可能发生这种情况。

(147) too many #include directories specified *（预处理器）*

最多可以为预处理器指定 7 个目录来搜索包含文件。使用驱动程序指定的目录数太大。

(148) too many arguments for preprocessor macro *（预处理器）*

根据 C 标准，宏最多只能具有 31 个参数。

(149) preprocessor macro work area overflow *（预处理器）*

宏扩展的总长度超出了内部表的长度。该表的长度通常为 32768 字节。因此任何宏扩展都不能扩展为总长度超出 32 KB。

(150) illegal "__" preprocessor macro "*" *（预处理器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(151) too many arguments in preprocessor macro expansion *（预处理器）*

在宏调用中提供了太多的参数。允许的最大数量为 31。

(152) bad dp/nargs in openpar(): c = * *（预处理器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(153) out of space in preprocessor macro * argument expansion *（预处理器）*

宏参数超出了内部缓冲区的长度。该缓冲区的长度通常为 4096 字节。

(155) work buffer overflow concatenating "*" *（预处理器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(156) work buffer "*" overflow *（预处理器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(157) can’t allocate * bytes of memory *（代码生成器和汇编器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(158) invalid disable in preprocessor macro "*" *（预处理器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(159) too many calls to unget() *（预处理器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(161) control line "*" within preprocessor macro expansion *（预处理器）*

在扩展宏时遇到了预处理器控制行 （以 # 开始的行）。不应发生这种情况。

(162) #warning: * *（预处理器和驱动程序）*

该警告或者是用户定义的#warning预处理器伪指令产生的，或者是驱动程序在读取映射文件时遇到了问题。如果是后者，请联系 Microchip 技术支持来了解详情。

(163) unexpected text in control line ignored *（预处理器）*

在控制行末尾出现额外字符时，将出现该警告。额外文本将被忽略，但会发出警告。最好 （根据标准 C）将文本括起来作为注释，例如：

```
#if defined(END)
  #define NEXT
#endif END     /* END would be better in a comment here */
```

(164) #include filename "*" was converted to lower case *（预处理器）*

#include 文件名必须先转换为小写形式，然后才能打开，例如：

```
#include <STDIO.H>  /* oops -- should be: #include <stdio.h> */
```

(165) #include filename "*" does not match actual name (check upper/lower case) *（预处理器）*

在 Windows 版本中，这表示要包含的文件实际上存在，并且拼写方式与 #include 文件名相同；但是，每个字符的大小写并不完全匹配。例如，指定 #include "code.c" 时，将会包含 Code.c （如果找到该文件）。在 Linux 版本中，如果未找到该文件，则会出现该警告。

(166) too few values specified with option "*" *（预处理器）*

预处理器 （CPP） -S 选项的值列表是不完整的。如果预处理器是由编译器驱动程序调用的，则不应发生这种情况。传递给该选项的值代表 char、 short、 int、 long、 float 和 double 类型的长度。

(167) too many values specified with -S option; "*" unused *（预处理器）*

提供给 -S 预处理器选项的值太多。请参见消息 166。

(168) unknown option "*" *（任意）*

提供给引起错误的组件的选项无法识别。

(169) strange character (*) after ## *（预处理器）*

之后存在意外的字符。

(170) symbol "*" in undef was never defined *（预处理器）*

作为参数提供给 #undef 的符号尚未定义。对于某些编译器，可以禁止该警告。该警告可以通过以下形式的代码来避免：

```
#ifdef SYM
  #undef SYM  /* only undefine if defined */
#endif
```


© 2013 Microchip Technology Inc.

(171) wrong number of preprocessor macro arguments for "*" (* instead of *) *（预处理器）*

调用宏时使用的参数数量不正确，例如：

```
#define ADD(a, b) (a+b)
ADD(1, 2, 3)        /* oops -- only two arguments required */
```

(172) formal parameter expected after # *（预处理器）*

字符串化操作符 #（不应与用于预处理器控制行的前导 # 混淆）必须跟随形式宏参数，例如：

```
#define str(x) #y  /* oops -- did you mean x instead of y? */
```

如果需要将某个标记字符串化，则需要定义一个特殊的宏来执行该操作，例如：

```
#define __mkstr__(x) #x
```

然后，在每次需要将标记转换为字符串时使用 `__mkstr__(token)`。

(173) undefined symbol "*" in #if, 0 used *（预处理器）*

`#if` 表达式中的某个符号不是已定义的预处理器宏。针对该表达式的目的，它的值被视为零。对于某些编译器，可以禁止该警告。示例：

```
#if FOO+BAR    /* e.g. FOO was never #defined */
  #define GOOD
#endif
```

(174) multi-byte constant "*" isn't portable *（预处理器）*

多字节常量是不可移植的，实际上它会被编译器的后面阶段拒绝，例如：

```
#if CHAR == 'ab'
  #define MULTI
#endif
```

(175) division by zero in #if; zero result assumed *（预处理器）*

在 `#if` 表达式内存在被零除，它已被视为产生零，例如：

```
#if foo/0  /* divide by 0: was this what you were intending? */
  int a;
#endif
```

(176) missing newline *（预处理器）*

行尾缺少换行符。每一行（包括最后一行）末尾都必须具有换行符。该问题通常是由编辑器引入的。

(177) symbol "*" in -U option was never defined *（预处理器）*

预处理器的 `-U` 选项中指定的宏名并未初始定义，因此无法取消定义。

(179) nested comments *（预处理器）*

在发现嵌套注释时，将发出该警告。嵌套注释可能表明前一个注释结束标记缺失或格式不正确，例如：

```
output = 0;  /* a comment that was left unterminated
flag = TRUE; /* next comment:
                hey, where did this line go? */
```

(180) unterminated comment in included file *（预处理器）*

在包含文件内开始的注释必须在包含文件内结束。

(181) non-scalar types can't be converted to other types *（解析器）*

不能将结构、联合或数组转换为其他类型，例如：

```
struct TEST test;
struct TEST * sp;
sp = test;       /* oops -- did you mean: sp = &test; ? */
```

(182) illegal conversion between types *（解析器）*

该表达式暗含不兼容类型之间的转换，如将结构类型转换为整型，例如：

```
struct LAYOUT layout;
int i;
layout = i;          /* int cannot be converted to struct */
```

请注意，即使结构仅包含一个 `int`（举例来说），也不能将它赋予 `int` 变量，反之亦然。

(183) function or function pointer required *（解析器）*

只有函数或函数指针可以作为函数调用的主体，例如：

```
int a, b, c, d;
a = b(c+d);   /* b is not a function --
                 did you mean a = b*(c+d) ? */
```

(184) calling an interrupt function is illegal *（解析器）*

不能从其他函数中调用使用 `interrupt` 限定的函数。它只能由硬件（或软件）中断调用。这是因为中断函数具有特殊的函数进入和退出代码，仅适用于从中断中进行调用。中断函数可以调用其他非中断函数。

(185) function does not take arguments *（解析器和代码生成器）*

该函数没有参数，但此处使用了一个或多个参数来调用它，例如：

```
int get_value(void);
void main(void)
{
  int input;
  input = get_value(6);  /* oops --
                            parameter should not be here */
}
```

(186) too many function arguments *（解析器）*

此处的参数数量多于该函数可接受的数量。

```
void add(int a, int b);
add(5, 7, input);      /* call has too many arguments */
```

(187) too few function arguments *（解析器）*

该函数需要的参数数量多于该调用中提供的参数数量，例如：

```
void add(int a, int b);
add(5);                  /* this call needs more arguments */
```


© 2013 Microchip Technology Inc.

(188) constant expression required *（解析器）*

在该上下文中，需要一个可以在编译时求值为常量的表达式，例如：

```
int a;
switch(input) {
  case a:  /* oops!
              can't use variable as part of a case label */
    input++;
}
```

(189) illegal type for array dimension *（解析器）*

数组维度必须为整型或枚举值。

```
int array[12.5];  /* oops -- twelve and a half elements, eh? */
```

(190) illegal type for index expression *（解析器）*

索引表达式必须为整型或枚举值，例如：

```
int i, array[10];
i = array[3.5];   /* oops --
                     exactly which element do you mean? */
```

(191) cast type must be scalar or void *（解析器）*

类型强制转换（括在括号中的抽象类型声明符）必须指示标量（即，不是数组或结构）类型或 `void` 类型，例如：

```
lip = (long [])input;  /* oops -- maybe: lip = (long *)input */
```

(192) undefined identifier "*" *（解析器）*

程序中使用了该符号，但它尚未定义或声明。如果认为它已定义，请检查是否存在拼写错误。

(193) not a variable identifier "*" *（解析器）*

该标识符不是一个变量；它可能是某种其他类型的对象，例如一个标号。

(194) ")" expected *（解析器）*

此处要求出现一个*右括号*)。这可能表明在表达式中遗漏了该字符，或存在某种其他语法错误。错误将标记在代码开始变为无意义的行上。它可能是不完整表达式之后的语句，例如：

```
if(a == b  /* the closing parenthesis is missing here */
  b = 0;   /* the error is flagged here */
```

(195) expression syntax *（解析器）*

该表达式的格式不正确，无法由编译器进行解析，例如：

```
a /=% b;  /* oops -- maybe that should be: a /= b; */
```

(196) struct/union required *（解析器）*

点号 "." 之前需要一个结构或联合标识符，例如：

```
int a;
a.b = 9;  /* oops -- a is not a structure */
```

(197) struct/union member expected *（解析器）*

结构或联合成员名称必须跟随在*点号* "." 或箭头 （"->"）之后。

(198) undefined struct/union "*" *（解析器）*

指定的结构或联合标记是未定义的，例如：

```
struct WHAT what;  /* a definition for WHAT was never seen */
```

(199) logical type required *（解析器）*

用作 if、 while 语句或布尔操作符 （如！和 &&）操作数的表达式必须为标量整型，例如：

```
struct FORMAT format;
if(format)            /* this operand must be a scaler type */
  format.a = 0;
```

(200) taking the address of a register variable is illegal *（解析器）*

使用 register 声明的变量不会在存储器中分配存储单元，因此尝试通过应用 & 操作符获取它的地址是非法的，例如：

```
int * proc(register int in)
{
  int * ip = &in;
  /* oops -- in may not have an address to take */
  return ip;
}
```

(201) taking the address of this object is illegal *（解析器）*

用作 & 操作符操作数的表达式不是一个代表存储器存储单元的表达式 （“左值”），因此无法定义其地址，例如：

```
ip = &8;  /* oops -- you can’t take the address of a literal */
```

(202) only lvalues may be assigned to or modified *（解析器）*

只有左值（即，直接表示可寻址存储单元的标识符或表达式）才能进行赋值或修改，例如：

```
int array[10];
int * ip;
char c;
array = ip;   /* array isn’t a variable,
                 it can’t be written to */
```

类型强制转换并不会产生一个左值，例如：

```
/* the contents of c cast to int
   is only a intermediate value */
(int)c = 1;
```

但是，可以使用指针编写以下代码：

```
*(int *)&c = 1
```

(203) illegal operation on bit variable *（解析器）*

并不是对位变量的所有操作都受支持。该操作就是其中之一，例如：

```
bit   b;
int * ip;
ip = &b;  /* oops --
             cannot take the address of a bit object */
```


© 2013 Microchip Technology Inc.

(204) void function can't return a value *（解析器）*

`void` 函数不能返回值。任何 `return` 语句都不应跟随表达式，例如：

```
void run(void)
{
  step();
  return 1;
  /* either run should not be void, or remove the 1 */
}
```

(205) integral type required *（解析器）*

该操作符要求只能为整型的操作数。

(206) illegal use of void expression *（解析器）*

`void` 表达式不具有任何值，因此不能在任何需要带值表达式的位置使用它，例如，用作算术操作符的操作数。

(207) simple type required for "*" *（解析器）*

需要一个简单类型 （即，不是数组或结构）作为该操作符的操作数。

(208) operands of "*" not same type *（解析器）*

该操作符的操作数属于不同的指针，例如：

```
int * ip;
char * cp, * cp2;
cp = flag ? ip : cp2;
/* result of ? : will be int * or char * */
```

也许您的本意是以下形式的代码：

```
cp = flag ? (char *)ip : cp2;
```

(209) type conflict *（解析器）*

该操作符的操作数属于不兼容的类型。

(210) bad size list *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(211) taking sizeof bit is illegal *（解析器）*

对 C `bit` 类型使用 `sizeof` 操作符是非法的。对某个类型使用 `sizeof` 操作符时，将给出存储该类型对象所需的字节数。因此对 `bit` 类型使用它毫无意义，是非法操作。

(212) missing number after pragma "pack" *（解析器）*

pragma `pack` 需要一个十进制数作为参数。它用于指定结构中每个成员的对齐方式。使用它时需要谨慎，因为一些处理器会强制对齐，如果对奇编号边界进行字取操作，将无法正常工作，例如：

```
#pragma pack  /* what is the alignment value */
```

也许您的本意是以下形式的代码：

```
#pragma pack 2
```

(214) missing number after pragma "interrupt_level" *（解析器）*

pragma `interrupt_level` 需要一个指示中断优先级的参数。对于中档器件，值将为 1；对于 PIC18 器件，值为 1 或 2。

(215) missing argument to pragma "switch" *（解析器）*

pragma switch 需要使用 auto、 direct 或 simple 作为参数，例如：

```
#pragma switch  /* oops -- this requires a switch mode */
```

也许您的本意是以下形式的代码：

```
#pragma switch simple
```

(216) missing argument to pragma "psect" *（解析器）*

pragma psect 需要 *oldname* = *newname* 形式的参数；其中，*oldname* 是编译器已知的现有 psect 名称， *newname* 是所需的新名称，例如：

```
#pragma psect  /* oops -- this requires an psect to redirect */
```

也许您的本意是以下形式的代码：

```
#pragma psect text=specialtext
```

(218) missing name after pragma "inline" *（解析器）*

inline pragma 要求其后跟随一个函数的名称。只有代码生成器可以识别函数名时，才会对它进行扩展；其他函数不会被更改，例如：

```
#pragma inline  /* what is the function name? */
```

也许您的本意是以下形式的代码：

```
#pragma inline memcpy
```

(219) missing name after pragma "printf_check" *（解析器）*

printf_check pragma 要求其后跟随一个函数的名称。它指定对函数进行 printf 样式的格式字符串检查，例如：

```
#pragma printf_check  /* what function is to be checked? */
```

也许您的本意是以下形式的代码：

```
#pragma printf_check sprintf
```

<stdio.h> 中包含了所有标准的 printf 样式函数的 pragma 伪指令。

(220) exponent expected *（解析器）*

浮点常量在 e 或 E 之后必须具有至少一个数字，例如：

```
float f;
f = 1.234e;  /* oops -- what is the exponent? */
```

(221) hexadecimal digit expected *（解析器）*

0x 之后应至少跟随一个 HEX 数字 0-9 和 A-F 或 a-f，例如：

```
a = 0xg6;  /* oops -- was that meant to be a = 0xf6 ? */
```

(222) binary digit expected *（解析器）*

0b 格式说明符之后要求跟随二进制数字，例如：

```
i = 0bf000;  /* oops -- f000 is not a base two value */
```

(223) digit out of range *（解析器和汇编器）*

该数值中的某个位超出了该数值的基数的范围；例如，在八进制数值中使用数字 8，或在十进制数值中使用 HEX 数字 A-F。八进制数值通过以 0 开头的数字字符串表示，而 HEX 数值则以 “0X” 或 “0x” 开头。例如：

```
int a = 058;
/* leading 0 implies octal which has digits 0 - 7 */
```


© 2013 Microchip Technology Inc.

(224) illegal "#" directive *（解析器）*

检测到非法的 # 预处理器伪指令。代码某处的某条伪指令可能拼写错误。

(225) missing character in character constant （解析器）

单引号中缺少字符，例如：

```
char c = ";  /* the character value of what? */
```

(226) char const too long *（解析器）*

括在单引号中的字符常量不能包含多个字符，例如：

```
c = '12';  /* oops -- only one character may be specified */
```

(227) "." expected after ".." *（解析器）*

可以出现两个连续点号的惟一情形是作为*省略号*的一部分，而该符号必须具有三个点号。（*省略号*用于在函数原型中指示可变数量的参数。）

`..` 的本意或者是表示*省略号*，这将需要添加一个额外的*点号*；或者是表示*结构成员操作符*，这将需要删除一个*点号*。

(228) illegal character (*) *（解析器）*

该字符在 C 代码中是非法的。有效字符包括字母、数字和构成可接受操作符的字符，例如：

```
c = a;  /* oops -- did you mean c = 'a'; ? */
```

(229) unknown qualifier "*" given to -A *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(230) missing argument to -A *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(231) unknown qualifier "*" given to -I *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(232) missing argument to -I *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(233) bad -Q option "*" *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(234) close error *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(236) simple integer expression required *（解析器）*

操作符 @ 之后需要一个简单的整型表达式，用于将绝对地址与变量相关联，例如：

```
int address;
char LOCK @ address;
```

(237) function "*" redefined *（解析器）*

在该模块中遇到了一个函数的多个定义。函数重载是非法的，例如：

```
int twice(int a)
{
  return a*2;
}
/* only one prototype & definition of rv can exist */
long twice(long a)
{
  return a*2;
}
```

(238) illegal initialization *（解析器）*

无法初始化 typedef 声明，因为它不会保留任何可以进行初始化的存储单元，例如：

```
/* oops -- uint is a type, not a variable */
typedef unsigned int uint = 99;
```

(239) identifier "*" redefined (from line *) *（解析器）*

在同一作用域中已定义了该标识符。不能重新定义它，例如：

```
  int a;  /* a filescope variable called "a" */
  int a;  /* attempting to define another of the same name */
```

请注意，同名但定义不同作用域的变量是合法的，但不建议这样做。

(240) too many initializers *（解析器）*

该对象具有太多的初始值设定项。检查对象（数组或结构）定义中初始值设定项的数量，例如：

```
/* three elements, but four initializers */
int ivals[3] = { 2, 4, 6, 8};
```

(241) initialization syntax *（解析器）*

该对象的初始化语法不正确。请检查大括号和逗号的位置和数量是否正确，例如：

```
int iarray[10] = {{'a', 'b', 'c'};
/* oops -- one two many {s */
```

(242) illegal type for switch expression *（解析器）*

switch 操作必须具有为整型或枚举值的表达式，例如：

```
double d;
switch(d) {  /* oops -- this must be integral */
  case '1.0':
    d = 0;
}
```


© 2013 Microchip Technology Inc.

(243) inappropriate break/continue *（解析器）*

发现某个 break 或 continue 语句未包围在相应的控制结构中。 continue 只能在 while、for 或 do while 循环中使用，而 break 只能在这些循环或 switch 语句内使用，例如：

```
switch(input) {
  case 0:
    if(output == 0)
      input = 0xff;
    } /* oops! this shouldn't be here and closed the switch */
    break;     /* this should be inside the switch */
```

(244) "default" case redefined *（解析器）*

在 switch 语句中只允许一个 default 标号。此处具有多个，例如：

```
switch(a) {
default:     /* if this is the default case... */
  b = 9;
  break;
default:     /* then what is this? */
  b = 10;
  break;
```

(245) "default" case not in switch *（解析器）*

遇到了一个名为 default 的标号，但它不是包围在 switch 语句中。default 标号仅在 switch 语句主体内是合法的。

如果在该 default 标号之前具有一条 switch 语句，则可能说明 switch 代码中的右大括号太多，导致 switch 语句提前终止。请参见消息 246。

(246) case label not in switch *（解析器）*

遇到了一个 case 标号，但不存在外围的 switch 语句。case 标号只能出现在 switch 语句主体内。

如果在该 case 标号之前具有一条 switch 语句，则可能说明 switch 代码中的右大括号太多，导致 switch 语句提前终止，例如：

```
switch(input) {
  case '0':
    count++;
    break;
  case '1':
    if(count>MAX)
      count= 0;
    }          /* oops -- this shouldn't be here */
    break;
  case '2':    /* error flagged here */
```

(247) duplicate label "*" *（解析器）*

在该函数中对于标号多次使用了相同的名称。请注意，标号的作用域是整个函数，而不只是包围标号的块，例如：

```
start:
  if(a > 256)
    goto end;
start:           /* error flagged here */
  if(a == 0)
    goto start;  /* which start label do I jump to? */
```

(248) inappropriate "else" *（解析器）*

遇到了无法与 `if` 语句关联的 `else` 关键字。这可能意味着缺少大括号或其他语法错误，例如：

```
/* here is a comment which I have forgotten to close...
if(a > b) {
  c = 0;
/* ... that will be closed here, thus removing the "if" */
else        /* my "if" has been lost */
  c = 0xff;
```

(249) probable missing "}" in previous block *（解析器）*

编译器遇到了类似函数或其他声明的内容，但前面的函数尚未使用右大括号结束。这可能意味着前一个函数的某个地方遗漏了右大括号，虽然它可能不是最后一个，例如：

```
void set(char a)
{
  PORTA = a;
                 /* the closing brace was left out here */
void clear(void)  /* error flagged here */
{
  PORTA = 0;
}
```

消息 250-499

(251) array dimension redeclared *（解析器）*

声明的数组维度与其先前声明具有不同的非零值。可以接受重新声明先前使用零维度声明的数组的大小，但不接受其他形式，例如：

```
extern int array[5];
int array[10];        /* oops -- has it 5 or 10 elements? */
```

(252) argument * conflicts with prototype *（解析器）*

该函数定义的指定参数（参数 0 是最左边的参数）与该函数先前的原型不一致，例如：

```
/* this is supposedly calc's prototype */
extern int calc(int, int);
int calc(int a, long int b)  /* hmmm -- which is right? */
{                            /* error flagged here */
  return sin(b/a);
}
```


© 2013 Microchip Technology Inc.

(253) argument list conflicts with prototype *（解析器）*

函数定义中的参数列表与该函数的先前原型不同。请检查参数的数量和类型是否都相同。

```
extern int calc(int);   /* this is supposedly calc's prototype */
int calc(int a, int b)  /* hmmm -- which is right? */
{                       /* error flagged here */
  return a + b;
}
```

(254) undefined *: "*" *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(255) not a member of the struct/union "*" *（解析器）*

该标识符不是此处使用的结构或联合类型的成员，例如：

```
struct {
  int a, b, c;
} data;
if(data.d)    /* oops --
                 there is no member d in this structure */
  return;
```

(256) too much indirection *（解析器）*

指针声明只能具有 16 级的间接引用。

(257) only "register" storage class allowed *（解析器）*

函数参数惟一允许的存储类别为 `register`，例如：

```
void process(static int input)
```

(258) duplicate qualifier *（解析器）*

在该类型规范中出现了两次相同的限定符。这种情况可能是由于直接使用或通过使用 `typedef` 而导致。删除多余的限定符。例如：

```
typedef volatile int vint;
/* oops -- this results in two volatile qualifiers */
volatile vint very_vol;
```

(259) can't be qualified both far and near *（解析器）*

同时将某个类型限定为 `far` 和 `near` 是非法的，例如：

```
far near int spooky;  /* oops -- choose far or near, not both */
```

(260) undefined enum tag "*" *（解析器）*

该枚举标记尚未定义，例如：

```
enum WHAT what;  /* a definition for WHAT was never seen */
```

(261) struct/union member "*" redefined *（解析器）*

在该结构或联合中已经使用了该名称的结构或联合成员，例如：

```
struct {
  int a;
  int b;
  int a;  /* oops -- a different name is required here */
} input;
```

© 2013 Microchip Technology Inc.

(262) struct/union "*" redefined *（解析器）*

多次定义了某个结构或联合，例如：

```
struct {
  int a;
} ms;
struct {
  int a;
} ms;    /* was this meant to be the same name as above? */
```

(263) members can’t be functions *（解析器）*

结构或联合的成员不能为函数。它可以为指向函数的指针，例如：

```
struct {
  int a;
  int get(int);  /* should be a pointer: int (*get)(int); */
} object;
```

(264) bad bitfield type *（解析器）*

位域的类型只能为 `int`（或 `unsigned`），例如：

```
struct FREG {
  char b0:1;   /* these must be part of an int, not char */
  char   :6;
  char b7:1;
} freg;
```

(265) integer constant expected *（解析器）*

结构声明中的成员名称之后出现的*冒号*表示该成员是位域。*冒号*之后必须为一个整型常量，用以定义位域中的位数，例如：

```
struct {
  unsigned first:  /* oops -- should be: unsigned first; */
  unsigned second;
} my_struct;
```

如果它的本意是表示一个带有位域的结构，则以下给出了一个示例：

```
struct {
  unsigned first : 4;  /* 4 bits wide */
  unsigned second: 4;  /* another 4 bits */
} my_struct;
```

(266) storage class illegal *（解析器）*

不能为结构或联合的成员指定存储类别。其存储类别取决于结构的存储类别，例如：

```
struct {
  /* no additional qualifiers may be present with members */
  static int first;
} ;
```


© 2013 Microchip Technology Inc.

(267) bad storage class *（代码生成器）*

代码生成器遇到了存储类别无效的变量定义，例如：

```
auto int foo; /* auto not permitted with global variables */
int power(static int a)  /* parameters may not be static */
{
  return foo * a;
}
```

(268) inconsistent storage class *（解析器）*

声明具有冲突的存储类别。一个声明中应仅出现一个存储类别，例如：

```
extern static int where;  /* so is it static or extern? */
```

(269) inconsistent type *（解析器）*

一个声明中应仅出现一个基本类型，例如：

```
int float if;  /* is it int or float? */
```

(270) variable can’t have storage class "register" *（解析器）*

只有函数参数或自动变量可以使用 `register` 限定符声明，例如：

```
register int gi;      /* this cannot be qualified register */
int process(register int input)  /* this is okay */
{
  return input + gi;
}
```

(271) type can’t be long *（解析器）*

只有 `int` 和 `float` 类型可以使用 `long` 限定。

```
long char lc;  /* what? */
```

(272) type can’t be short *（解析器）*

只有 `int` 类型可以使用 `short` 修饰，例如：

```
short float sf;  /* what? */
```

(273) type can’t be both signed and unsigned *（解析器）*

不能在同一声明中一起使用 `signed` 和 `unsigned` 类型修饰符，因为它们具有相反的含义，例如：

```
signed unsigned int confused;  /* which is it? */
```

(274) type can’t be unsigned *（解析器）*

浮点型不能设为 `unsigned`，例如：

```
unsigned float uf;  /* what? */
```

(275) "..." illegal in non-prototype argument list *（解析器）*

*省略号*只能作为原型参数列表中的最后一项出现。它不能独自出现，也不可能出现在不具有类型的参数名之后；即， K&R 样式的非原型函数定义。例如：

```
/* K&R-style non-prototyped function definition */
int kandr(a, b, ...)
  int a, b;
{
```

(276) type specifier required for prototyped argument *（解析器）*

原型参数需要类型说明符。不能只有一个标识符。

(277) can’t mix prototyped and non-prototyped arguments *（解析器）*

函数声明只能具有全原型参数（即，括号内具有类型）或全 K&R 样式的参数（即，括号内只有名称，声明中的参数类型列在函数主体起始处之前），例如：

```
int plus(int a, b)  /* oops -- a is prototyped, b is not */
int b;
{
  return a + b;
}
```

(278) argument "*" redeclared *（解析器）*

指定参数在同一参数列表中声明了多次，例如：

```
/* can’t have two parameters called "a" */
int calc(int a, int a)
```

(279) initialization of function arguments is illegal *（解析器）*

函数参数不能在声明中进行初始化。参数的初始化在调用函数时发生，并由调用函数为参数提供一个值，例如：

```
/* oops -- a is initialized when proc is called */
extern int proc(int a = 9);
```

(280) arrays of functions are illegal *（解析器）*

不能定义一个函数数组。但是，可以定义指向函数的指针的数组，例如：

```
int * farray[]();  /* oops -- should be: int (* farray[])(); */
```

(281) functions can’t return functions *（解析器）*

函数不能返回一个函数。它可以返回一个函数指针。返回函数指针的函数可以声明如下：`int (* (name()))()`。请注意正确绑定声明各个部分所需的许多括号。

(282) functions can’t return arrays *（解析器）*

函数只能返回标量（简单）类型或结构。它不能返回数组。

(283) dimension required *（解析器）*

在多维数组中，只有最高（即第一个）维度可以不赋值。之后的所有维度都必须以常量表达式表示，例如：

```
/* This should be, for example: int arr[][7] */
int get_element(int arr[2][])
{
  return array[1][6];
}
```

(284) invalid dimension *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

© 2013 Microchip Technology Inc.

(285) no identifier in declaration *（解析器）*

该声明中缺少标识符。当编译器由于缺少右大括号之类错误而产生混淆时，也可能发生该错误，例如：

```
void interrupt(void)  /* what is the name of this function? */
{
}
```

(286) declarator too complex *（解析器）*

该声明符过于复杂，编译器无法进行处理。请检查声明，并寻找一种方法来简化它。如果编译器发现它太复杂，那么任何维护代码的人员也会如此。

(287) arrays of bits or pointers to bit are illegal *（解析器）*

位的数组或指向位变量的指针是非法的，例如：

```
bit barray[10];  /* wrong -- no bit arrays */
bit * bp;        /* wrong -- no pointers to bit variables */
```

(288) only functions may be void *（解析器）*

变量不能为 `void` 类型。只有函数可以为 `void` 类型，例如：

```
int a;
void b;  /* this makes no sense */
```

(289) only functions may be qualified "interrupt" *（解析器）*

除了函数，限定符 `interrupt` 不能应用于任何其他对象，例如：

```
/* variables cannot be qualified interrupt */
interrupt int input;
```

(290) illegal function qualifier(s) *（解析器）*

对函数应用了在该上下文中没有意义的限定符。一些限定符只有在用于左值时才有意义（即 `const` 或 `volatile`）。这可能表明您忘记了指示函数应返回限定对象指针的星号 `*`，例如：

```
const char ccrv(void) /* const * char ccrv(void) perhaps? */
{                     /* error flagged here */
  return ccip;
}
```

(291) K&R identifier "*" not an argument *（解析器）*

该标识符出现在 K&R 样式的参数声明符中，但未列在函数名后的括号内，例如：

```
int process(input)
int unput;        /* oops -- that should be int input; */
{
}
```

(292) function parameter may not be a function *（解析器）*

函数参数不能为一个函数。它可以为指向函数的指针，因此，或许声明中遗漏了“*”。

(293) bad size in index_type() *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(294) can't allocate * bytes of memory *（代码生成器和 Hexmate）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(295) expression too complex *（解析器）*

该表达式导致编译器的内部堆栈发生溢出，应重新组合或拆分为两个表达式。

(296) out of memory *（Objtohex）*

这可能是内部编译器错误。详情请联系 Microchip 技术支持。

(297) bad argument (*) to tysize() *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(298) end of file in #asm *（预处理器）*

在 #asm 块内遇到文件结束符。这可能意味着 #endasm 缺失或拼写错误，例如：

```
#asm
  MOV  r0, #55
  MOV  [r1], r0
}                /* oops -- where is the #endasm */
```

(300) unexpected end of file *（解析器）*

在 C 模块中意外遇到文件结束符，例如：

```
void main(void)
{
  init();
  run();    /* is that it? What about the close brace */
```

(301) end of file on string file *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(302) can't reopen "*": * *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(303) can't allocate * bytes of memory (line *) *（解析器）*

在尝试排序和合并字符串时，解析器无法为所遇到的最长字符串分配存储器。请尝试减少该模块中的字符串的数量或长度。

(306) can't allocate * bytes of memory for * *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(307) too many qualifier names *（解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(308) too many case labels in switch *（代码生成器）*

该 switch 语句中具有太多的 case 标号。任何一个 switch 语句中允许的最大 case 标号数量为 511。

(309) too many symbols *（汇编器）*

汇编器的符号表具有太多的符号。请减少程序中的符号数量。

© 2013 Microchip Technology Inc.

(310) "]" expected *（解析器）*

在数组声明或使用数组索引的表达式中要求存在一个右方括号，例如：

```
process(carray[idx);  /* oops --
                         should be: process(carray[idx]); */
```

(311) closing quote expected *（解析器）*

指定字符串要求存在一个右引号。

(312) "*" expected *（解析器）*

解析器要求出现所指示的标记。

(313) function body expected *（解析器）*

遇到使用 K&R 样式参数（即，括号内包含参数名，但未包含类型）的函数声明时，要求其后跟随函数主体，例如：

```
/* the function block must follow, not a semicolon */
int get_value(a, b);
```

(314) ";" expected *（解析器）*

语句中缺少*分号*。发现右大括号或关键字跟随在未使用*分号*终止的语句后面，例如：

```
while(a) {
  b = a--  /* oops -- where is the semicolon? */
}          /* error is flagged here */
```

注：在不处于右大括号或关键字之前的语句中遗漏分号时，通常会导致对于随后的代码（分析器假定它是原始语句的一部分）发出某种其他错误。

(315) "{" expected *（解析器）*

此处要求出现一个*左大括号*。该错误可能是由于函数定义缺少*左大括号*而产生的，例如：

```
/* oops! no opening brace after the prototype */
void process(char c)
  return max(c, 10) * 2; /* error flagged here */
}
```

(316) "}" expected *（解析器）*

此处要求出现一个*右大括号*。该错误可能是由于某个已初始化数组缺少*右大括号*而产生的，例如：

```
char carray[4] = { 1, 2, 3, 4;  /* oops -- no closing brace */
```

(317) "(" expected *（解析器）*

此处要求出现一个*左括号* (。它必须是 `while`、 `for`、 `if`、 `do` 或 `asm` 关键字之后的第一个标记，例如：

```
if a == b  /* should be: if(a == b) */
  b = 0;
```

(318) string expected *（解析器）*

`asm` 语句的操作数必须为括在括号内的字符串，例如：

```
asm(nop);  /* that should be asm("nop");
```

(319) while expected *（解析器）*

在 do 语句结束处要求出现一个 while 关键字，例如：

```
do {
  func(i++);
}             /* do the block while what condition is true? */
if(i > 5)     /* error flagged here */
  end();
```

(320) ":" expected *（解析器）*

case 标号或 default 关键字之后缺少*冒号*。发生这种情况的原因通常是意外地键入*分号*而不是*冒号*，例如：

```
switch(input) {
  case 0;         /* oops -- that should have been: case 0: */
    state = NEW;
```

(321) label identifier expected *（解析器）*

goto 之后必须出现一个表示标号的标识符，例如：

```
if(a)
  goto 20;
/* this is not BASIC -- a valid C label must follow a goto */
```

(322) enum tag or "{" expected *（解析器）*

关键字 enum 之后必须跟随属于或将定义为 enum 标记的标识符，或者跟随左大括号，例如：

```
enum 1, 2;  /* should be, for example: enum {one=1, two }; */
```

(323) struct/union tag or "{" expected *（解析器）*

在 struct 或 union 关键字之后必须跟随代表一个结构或联合的标识符或者左大括号，例如：

```
struct int a;  /* this is not how you define a structure */
```

也许您的本意是以下形式的代码：

```
struct {
  int a;
} my_struct;
```

(324) too many arguments for printf-style format string *（解析器）*

该格式字符串的参数太多。这是无害的，但可能表示格式字符串存在错误，例如：

```
/* oops -- missed a placeholder? */
printf("%d - %d", low, high, median);
```

(325) error in printf-style format string *（解析器）*

此处的格式字符串中存在错误。该字符串被解释为 printf() 样式的格式字符串，但它的语法不正确。如果不更正，它会在运行时导致意外的行为，例如：

```
printf("%l", lll);  /* oops -- maybe: printf("%ld", lll); */
```

(326) long int argument required in printf-style format string *（解析器）*

该格式说明符需要一个长整型参数。请检查格式说明符和相应参数的数量和顺序，例如：

```
printf("%lx", 2);  // maybe you meant: printf("%lx", 2L);
```


© 2013 Microchip Technology Inc.

(327) long long int argument required in printf-style format string *（解析器）*

该格式说明符需要一个 long long 类型的参数。请检查格式说明符和相应参数的数量和顺序，例如：

```
printf("%llx", 2);  // maybe you meant: printf("%llx", 2LL);
```

请注意， MPLAB XC8 并不直接支持 long long 整型。

(328) int argument required in printf-style format string *（解析器）*

该 printf 样式的格式说明符需要一个整型参数。请检查格式说明符和相应参数的数量和顺序，例如：

```
printf("%d", 1.23); /* wrong number or wrong placeholder */
```

(329) double argument required in printf-style format string *（解析器）*

对应于该参数的 printf 格式说明符为 %f 或类似形式，并需要一个浮点型表达式。请检查 printf 是否缺少或存在额外的格式说明符或参数。

```
printf("%f", 44);  /* should be: printf("%f", 44.0); */
```

(330) pointer to * argument required in printf-style format string *（解析器）*

该格式说明符需要一个指针参数。请检查格式说明符和相应参数的数量和顺序。

(331) too few arguments for printf-style format string *（解析器）*

该格式字符串的参数太少。这将导致在运行时打印或转换一个垃圾值，例如：

```
printf("%d - %d", low);
  /* oops! where is the other value to print? */
```

(332) "interrupt_level" should be 0 to 7 *（解析器）*

pragma interrupt_level 的参数必须为 0 至 7；但是，中档器件仅使用中断优先级 1； PIC18 器件可以使用优先级 1 或 2。例如：

```
#pragma interrupt_level  9 /* oops -- the level is too high */
void interrupt isr(void)
{
  /* isr code goes here */
}
```

(333) unrecognized qualifier name after "strings" *（解析器）*

向 pragma strings 传递了无法识别的限定符，例如：

```
/* oops -- should that be #pragma strings const ? */
#pragma strings cinst
```

(334) unrecognized qualifier name after "printf_check" *（解析器）*

向 #pragma printf_check 传递了无法识别的限定符，例如：

```
/* oops -- should that be const not cinst? */
#pragma printf_check(printf) cinst
```

(335) unknown pragma "*" *（解析器）*

遇到未知的 pragma 伪指令，例如：

```
#pragma rugsused myFunc w  /* I think you meant regsused */
```

© 2013 Microchip Technology Inc.

(336) string concatenation across lines *（解析器）*

将连接位于两行上的字符串。请检查这是否是所需的结果，例如：

```
char * cp = "hi"
  "there";    /* this is okay,
                 but is it what you had intended? */
```

(337) line does not have a newline on the end *（解析器）*

文件中最后一行的行尾缺少*换行符*（取决于操作系统的字符）。某些编辑器会创建此类文件，这会对于包含文件导致一些问题。 ANSI C 标准要求所有源文件都仅包含完整的行。

(338) can’t create * file "*" *（任意）*

应用程序尝试创建或打开指定的文件，但无法创建它。请检查是否所有文件路径名均正确。

(339) initializer in extern declaration *（解析器）*

包含关键字 extern 的声明包含初始化。这将覆盖 extern 存储类别，因为要初始化一个对象，就需要定义它 （即，为它分配存储单元），例如：

```
extern int other = 99;  /* if it’s extern and not allocated
                           storage, how can it be initialized? */
```

(340) string not terminated by null character *（解析器）*

char 数组使用长度大于数组的字符串字面值进行初始化。因此，数组中没有足够的空间来安全地追加空终止符，例如：

```
char foo[5] = "12345"; /* the string stored in foo won’t have
                          a null terminating, i.e.
                          foo = [’1’, ’2’, ’3’, ’4’, ’5’] */
```

(343) implicit return at end of non-void function *（解析器）*

在声明为返回一个值的函数中，存在使它能够到达函数主体结束而不返回一个值的执行路径。请插入一个返回某个值的 return 语句，或者如果该函数不返回值，则将它声明为 void 类型，例如：

```
int mydiv(double a, int b)
{
  if(b != 0)
    return a/b;   /* what about when b is 0? */
}                 /* warning flagged here */
```

(344) non-void function returns no value *（解析器）*

某个函数声明为返回一个值，但其 return 语句未指定返回值，例如：

```
int get_value(void)
{
  if(flag)
    return val++;
  return;
  /* what is the return value in this instance? */
}
```


© 2013 Microchip Technology Inc.

(345) unreachable code *（解析器）*

永远不会执行该代码段，因为没有任何执行路径可以到达该代码段，例如：

```
while(1)          /* how does this loop finish? */
  process();
flag = FINISHED;  /* how do we get here? */
```

(346) declaration of "*" hides outer declaration *（解析器）*

声明的某个对象与某个外部的声明（即，处于外部并位于当前函数或块之前的声明）同名。这是合法的，但可能导致在预期使用外部的变量时意外使用另一个变量，例如：

```
int input;         /* input has filescope */
void process(int a)
{
  int input;       /* local blockscope input */
  a = input;       /* this will use the local variable.
                      Is this right? */
```

(347) external declaration inside function *（解析器）*

函数包含 `extern` 声明。这是合法的，但通常是不可取的，因为它会将函数声明的作用域限制在函数主体内。这意味着如果编译器稍后在同一文件中遇到该extern对象的另一处声明、使用或定义，它将不再具有先前的声明，因而无法检查这些声明是否一致。这会在链接时导致奇怪的程序行为或签名错误。它还会隐藏相同对象的任何先前声明，再次破坏编译器的类型检查。一般来说，请总是在所有其他函数之外声明 `extern` 变量和函数。例如：

```
int process(int a)
{
  /* this would be better outside the function */
  extern int away;
  return away + a;
}
```

(348) auto variable "*" should not be qualified *（解析器）*

自动变量不应具有与之关联的 `near` 或 `far` 限定符。其存储类别由堆栈结构隐式定义。自动变量可以使用 `static` 限定，但之后它将不再是自动变量。

(349) non-prototyped function declaration for "*" *（解析器）*

函数使用旧式 （K&R）参数进行声明。最好对于所有函数都使用原型声明，例如：

```
int process(input)
int input;         /* warning flagged here */
{
}
```

最好编写为以下形式：

```
int process(int input)
{
}
```

(350) unused * "*" (from line *) *（解析器）*

指示的对象从未在所编译的函数或模块中使用。该对象或者是多余的，或者原先要使用它的代码已在编译中被排除或对象名称存在拼写错误。请注意，符号 rcsid 和 sccsid 永远不会被报告为未使用。

(352) float parameter coerced to double *（解析器）*

如果非原型函数具有声明为 float 的参数，编译器会将它转换为 double float 类型。这是因为默认的 C 类型转换约定规定，在浮点数传递给非原型函数时，需要将它转换为 double 类型。函数声明与该约定保持一致是非常重要的，例如：

```
double inc_flt(f)  /* f will be converted to double */
float f;           /* warning flagged here */
{
  return f * 2;
}
```

(353) sizeof external array "*" is zero *（解析器）*

外部数组大小的求值结果为零。这可能是由于数组在 extern 声明中不具有显式的维度。

(354) possible pointer truncation *（解析器）*

限定为 far 的指针被赋予默认指针或限定为 near 的指针，或默认指针被赋予一个限定为 near 的指针。这可能会导致指针截断和信息丢失，具体取决于所使用的存储器模型。

(355) implicit signed to unsigned conversion *（解析器）*

有符号数字被赋予或以其他形式转换为较长的无符号类型。根据 ANSI C 的 “值保留” 规则，这将导致有符号值先通过符号扩展而扩展为一个具有目标类型长度的有符号数字，然后再转换为无符号值（这不会涉及到任何位模式变化）。因此，可能会发生意外的符号扩展。为确保不发生这种情况，需要先将有符号值转换为等效的无符号值，例如：

```
signed char sc;
unsigned int ui;
ui = sc;    /* if sc contains 0xff,
               ui will contain 0xffff for example */
```

将对 char 变量执行符号扩展，转换为更长的类型。如果不希望发生这种情况，请使用强制类型转换，例如：

```
ui = (unsigned char)sc;
```

(356) implicit conversion of float to integer *（解析器）*

浮点值被赋予或以其他形式转换为整型。这可能导致浮点值发生截断。强制类型转换可以消除该警告。

```
double dd;
int i;
i = dd;    /* is this really what you meant? */
```

如果希望使用这种形式的表达式，则通过强制类型转换来指示这一点：

```
i = (int)dd;
```


© 2013 Microchip Technology Inc.

(357) illegal conversion of integer to pointer *（解析器）*

整型被赋予或以其他形式转换为指针类型。这通常意味着使用了错误的变量，但如果这确实是您希望执行的操作，请使用强制类型转换来指示编译器您希望进行该转换，警告将会被消除。这也可能意味着您忘记了 & 地址操作符，例如：

```
int * ip;
int i;
ip = i;    /* oops -- did you mean ip = &i ? */
```

如果希望使用这种形式的表达式，则通过强制类型转换来指示这一点：

```
ip = (int *)i;
```

(358) illegal conversion of pointer to integer *（解析器）*

指针被赋予或以其他形式转换为整型。这通常意味着使用了错误的变量，但如果这确实是您希望执行的操作，请使用强制类型转换来指示编译器您希望进行该转换，警告将会被消除。这也可能意味着您忘记了 * 解引用操作符，例如：

```
int * ip;
int i;
i = ip;    /* oops -- did you mean i = *ip ? */
```

如果希望使用这种形式的表达式，则通过强制类型转换来指示您的本意：

```
i = (int)ip;
```

(359) illegal conversion between pointer types *（解析器）*

一种类型的指针（即指向特定类型的对象）被转换为不同类型的指针。这通常意味着使用了错误的变量，但如果这确实是您期望执行的操作，请使用强制类型转换来指示编译器您希望进行该转换，警告将会被消除，例如：

```
long input;
char * cp;
cp = &input;  /* is this correct? */
```

这是访问多字节变量内字节的常用方法。要指示这是程序的预定操作，请使用强制类型转换：

```
cp = (char *)&input;  /* that's better */
```

如果指针指向具有相同类型的对象，但对象具有不同的限定符，则在指针之间进行转换也可能会出现该警告，例如：

```
char * cp;
/* yes, but what sort of characters? */
cp = "I am a string of characters";
```

如果字符串字面值的默认类型为 `const char *`，那么该警告是有效的。应将它编写为：

```
const char * cp;
cp = "I am a string of characters";  /* that's better */
```

从指针类型中省略限定符往往是灾难性的，但几乎可以肯定不是您所期望的。

(360) array index out of bounds *（解析器）*

使用小于零或大于等于数组中元素数量的常量值作为索引访问数组。通过指针变量访问数组元素时，不会发出该警告，例如：

```
int i, * ip, input[10];
i = input[-2];           /* oops -- this element doesn't exist */
ip = &input[5];
i = ip[-2];              /* this is okay */
```

(361) function declared implicit int *（解析器）*

在编译器遇到对于名称当前未定义的函数的函数调用时，编译器会自动将函数声明为 int 类型，具有未指定 （K&R 样式）的参数。如果随后遇到了函数的定义，其类型和参数有可能不同于先前的隐式声明，从而导致编译器错误。解决方案是确保所有函数都已定义，或至少在使用之前声明，最好是带有原型参数。如果需要提前声明一个函数，应根据情况在它前面加上关键字 extern 或 static。例如：

```
/* I may prevent an error arising from calls below */
void set(long a, int b);
void main(void)
{
  /* by here a prototype for set should have seen */
  set(10L, 6);
}
```

(362) redundant "&" applied to array *（解析器）*

地址操作符 & 被应用于一个数组。由于使用数组名总是会给出其地址，所以这是不必要的，已被忽略，例如：

```
int array[5];
int * ip;
/* array is a constant, not a variable; the & is redundant. */
ip = &array;
```

(363) redundant "&" or "*" applied to function address *（解析器）*

地址操作符 “&” 被应用于一个函数。由于使用函数名总是会给出其地址，所以这是不必要的，已被忽略，例如：

```
extern void foo(void);
void main(void)
{
    void(*bar)(void);
    /* both assignments are equivalent */
    bar = &foo;
    bar = foo;  /* the & is redundant */
}
```

(364) attempt to modify object qualified * *（解析器）*

程序不能对声明为 const 或 code 的对象进行赋值或以其他任何方式修改。尝试修改此类对象的影响取决于特定的编译器。

```
const int out = 1234;  /* "out" is read only */
out = 0;               /* oops --
                          writing to a read-only object */
```


© 2013 Microchip Technology Inc.

(365) pointer to non-static object returned *（解析器）*

该函数返回指向非静态（如自动）变量的指针。这很可能是一个错误，因为在函数返回时，与自动变量关联的存储单元会变为无效，例如：

```
char * get_addr(void)
{
  char c;
  /* returning this is dangerous;
     the pointer could be dereferenced */
  return &c;
}
```

(366) operands of "*" not same pointer type *（解析器）*

该操作符的操作数属于不同的指针类型。这可能意味着使用了错误的指针，但如果代码实际上是您所期望的，请使用强制类型转换来取消该错误消息。

(367) identifier is already extern; can't be static *（解析器）*

该函数已声明为 `extern` 类型，可能是通过隐式声明。它现在被重新声明为 `static` 类型，但该重新声明是无效的。

```
void main(void)
{
  /* at this point the compiler assumes set is extern... */
  set(10L, 6);
}
/* now it finds out otherwise */
static void set(long a, int b)
{
  PORTA = a + b;
}
```

(368) array dimension on "*[]" ignored *（预处理器）*

函数参数中的数组维度已被忽略，因为参数在传递时实际上会被转换为指针。因此，可以传递任意大小的数组。或者从参数中删除维度，或者使用指针语法定义参数，例如：

```
/* param should be: "int array[]" or "int *" */
int get_first(int array[10])
{                               /* warning flagged here */
  return array[0];
}
```

(369) signed bitfields not supported *（解析器）*

仅支持无符号位域。如果位域声明为 `int` 类型，编译器仍会将它视为无符号，例如：

```
struct {
  signed int sign: 1;    /* oops -- this must be unsigned */
  signed int value: 7;
} ;
```

(370) illegal basic type; int assumed *（解析器）*

无法识别要转换为限定基本类型的强制类型转换的基本类型，基本类型被假定为 `int`，例如：

```
/* here ling is assumed to be int */
unsigned char bar = (unsigned ling) 'a';
```

(371) missing basic type; int assumed *（解析器）*

该声明中未包含基本类型，所以假定为 int。该声明不是非法的，但最好包含基本类型，以明确其意图，例如：

```
char c;
i;      /* don't let the compiler make assumptions, use : int i */
func(); /* ditto, use: extern int func(int); */
```

(372) "," expected *（解析器）*

此处要求出现一个*逗号*。这可能意味着在声明列表中遗漏了两个标识符之间的*逗号*。这也可能意味着前面紧邻的类型名称存在拼写错误，并因而被解释为标识符，例如：

```
unsigned char a;
/* thinks: chat & b are unsigned, but where is the comma? */
unsigned chat b;
```

(373) implicit signed to unsigned conversion *（解析器）*

在要求无符号类型的位置提供了有符号类型，它已被隐式强制转换为无符号类型，例如：

```
unsigned int foo = -1;
/* the above initialization is implicitly treated as:
   unsigned int foo = (unsigned) -1; */
```

(374) missing basic type; int assumed *（解析器）*

在转换为限定基本类型的强制类型转换中缺少基本类型，并假定为 int，例如：

```
int i = (signed) 2; /* (signed) assumed to be (signed int) */
```

(375) unknown FNREC type "*" *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(376) bad non-zero node in call graph *（链接器）*

链接器遇到了调用图中的顶层节点被调用图中的低层节点引用的情况。这可能意味着程序存在间接递归，这在使用编译堆栈时是不允许的。

(378) can't create * file "*" *（Hexmate）*

无法创建该类型的文件。是否该文件或具有该名称的文件已在使用？

(379) bad record type "*" *（链接器）*

这是内部编译器错误。确保目标文件是有效的目标文件。详情请联系 Microchip 技术支持。

(380) unknown record type (*) *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(381) record "*" too long (*) *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(382) incomplete record: type = *, length = * *（Dump 和 Xstrip）*

该消息由 DUMP 或 XSTRIP 实用程序生成，指示目标文件不是有效的目标文件，或者它已被截断。详情请联系 Microchip 技术支持。

© 2013 Microchip Technology Inc.

(383) text record has length (*) too small *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(384) assertion failed: file *, line *, expression * *（链接器和解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(387) illegal or too many -G options *（链接器）*

存在多个链接器 -g 选项，或者 -g 选项之后未跟随任何参数。这些参数用于指定如何计算段地址。

(388) duplicate -M option *（链接器）*

第二次向链接器指定了映射文件名。如果使用的是编译器驱动程序，则不应发生该问题。如果是手动调用链接器，请确保命令行上仅存在该选项的一个实例。关于该选项的正确语法的信息，请参见**第 4.8.8 节 “-M：生成映射文件”**。

(389) illegal or too many -O options *（链接器）*

该链接器 -o 标志是非法的，或遇到了另一个 -o 选项。链接器的 -o 选项必须紧跟一个文件名，不能有任何中间空格。

(390) missing argument to -P *（链接器）*

传递给链接器的 -p 选项太多，或者 -p 选项未跟随任何参数。各个 -p 选项的参数可以使用*逗号*进行组合和分隔。

(391) missing argument to -Q *（链接器）*

-Q 链接器选项需要使用机器类型作为参数。

(392) missing argument to -U *（链接器）*

-U （取消定义）选项需要一个参数。

(393) missing argument to -W *（链接器）*

-W 选项 （列表宽度）需要一个数值参数。

(394) duplicate -D or -H option *（链接器）*

第二次向链接器指定了符号文件名。如果使用的是编译器驱动程序，则不应发生该问题。如果是手动调用链接器，请确保命令行上仅存在这两个选项之一的一个实例。

(395) missing argument to -J *（链接器）*

必须在 -j 链接器选项之后指定中止之前的最大错误数量。

(397) usage: hlink [-options] files.obj files.lib *（链接器）*

命令行链接器的用法不正确。如果要直接调用链接器，请参见**第 7.2 节 “操作”**了解更多详细信息。或者，这可能是内部编译器错误，详情请联系 Microchip 技术支持。

(398) output file can’t be also an input file *（链接器）*

链接器检测到尝试将其输出文件覆盖写入其输入文件之一。无法执行该操作，因为它需要同时读取和写入输入和输出文件。

(400) bad object code format *（链接器）*

这是内部编译器错误。目标文件的目标代码格式无效。请确保它是有效的目标文件。详情请联系 Microchip 技术支持。

(402) bad argument to -F *（Objtohex）*

为 objtohex 的 -F 选项提供了无效的参数。如果要直接调用该命令行工具，请参见**第 8.3 节 “OBJTOHEX”**了解更多详细信息。或者，这可能是内部编译器错误，详情请联系 Microchip 技术支持。

(403) bad -E option: "*" *（Objtohex）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(404) bad maximum length value to -<digits> *（Objtohex）*

OBJTOHEX -n,m HEX 长度 / 舍入选项的第一个值无效。

(405) bad record size rounding value to -<digits> *（Objtohex）*

OBJTOHEX -n,m HEX 长度 / 舍入选项的第二个值无效。

(406) bad argument to -A *（Objtohex）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(407) bad argument to -U *（Objtohex）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(408) bad argument to -B *（Objtohex）*

该选项需要以 8、 10 或 16 为基数的整型参数。如果要直接调用 objtohex，请参见**第 8.3 节 “OBJTOHEX”**了解更多详细信息。或者，这可能是内部编译器错误，详情请联系 Microchip 技术支持。

(409) bad argument to -P *（Objtohex）*

该选项需要以 8、 10 或 16 为基数的整型参数。如果要直接调用 objtohex，请参见**第 8.3 节 “OBJTOHEX”**了解更多详细信息。这可能是内部编译器错误，详情请联系 Microchip 技术支持。

(410) bad combination of options *（Objtohex）*

提供给 OBJTOHEX 的选项组合是无效的。

(412) text does not start at 0 *（Objtohex）*

某些内容中的代码必须从零开始。此处不是如此。

(413) write error on "*" *（汇编器、链接器和 Cromwell）*

指定文件发生写入错误。这可能意味着磁盘空间已用尽。

(414) read error on "*" *（链接器）*

链接器尝试读取该文件时遇到一个错误。

(415) text offset too low in COFF file *（Objtohex）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(416) bad character (*) in extended TEKHEX line *（Objtohex）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

© 2013 Microchip Technology Inc.

(417) seek error in "*" *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(418) image too big *（Objtohex）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(419) object file is not absolute *（Objtohex）*

传递给 OBJTOHEX 的目标文件中具有重定位项。这可能表明它是错误的目标文件，或者为链接器或 OBJTOHEX 提供了无效的选项。来自汇编器的目标输出文件是可重定位的，不是绝对的。链接器的目标文件输出是绝对的。

(420) too many relocation items *（Objtohex）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(421) too many segments *（Objtohex）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(422) no end record *（链接器）*

该目标文件没有结束记录。这可能意味着它不是一个目标文件。如果目标文件是由编译器生成的，请联系 Microchip 技术支持。

(423) illegal record type *（链接器）*

目标文件中存在错误。这说明目标文件无效或链接器中发生了内部错误。如果目标文件是由编译器创建的，请联系 Microchip 技术支持来了解详情。

(424) record too long *（Objtohex）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(425) incomplete record *（Objtohex 和库管理器）*

传递给 OBJTOHEX 或库管理器的目标文件已损坏。详情请联系 Microchip 技术支持。

(427) syntax error in checksum list *（Objtohex）*

由 OBJTOHEX 读取的校验和列表中存在语法错误。校验和列表是为了响应某个选项从标准输入中读取的。

(428) too many segment fixups *（Objtohex）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(429) bad segment fixups *（Objtohex）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(430) bad checksum specification *（Objtohex）*

提供给 OBJTOHEX 的校验和列表的语法不正确。

(431) bad argument to -E *（Objtoexe）*

该选项需要以 8、10 或 16 为基数的整型参数。如果要直接调用 objtoexe，请检查该参数。或者，这可能是内部编译器错误，详情请联系 Microchip 技术支持。

(432) usage: objtohex [-ssymfile] [object-file [exe-file]] *（Objtohex）*

命令行工具 objtohex 的用法不正确。如果要直接调用 objtohex，请参见**第 8.3 节“OBJTOHEX”**了解更多详细信息。或者，这可能是内部编译器错误，详情请联系 Microchip 技术支持。

(434) too many symbols (*) *（链接器）*

符号表中存在太多的符号，其限制为 * 个符号。请将一些全局符号更改为局部符号，以减少符号数量。

(435) bad segment selector "*" *（链接器）*

链接器的段规范选项 （-G）是无效的，例如：

```
-GA/f0+10
```

是否忘记了基数？

```
-GA/f0h+10
```

(436) psect "*" re-orged *（链接器）*

该 psect 多次指定了其起始地址。

(437) missing "=" in class spec *（链接器）*

类规范需要一个 = 符号，例如，Ctext=ROM。更多信息，请参见**第 7.2.3 节“-Cpsect=class”**。

(438) bad size in -S option *（链接器）*

-S 规范中给出的地址无效：它应为有效的数字，使用十进制、八进制或十六进制基数。基数通过后缀 O （八进制）或 H （十六进制）来指定。也可以使用前缀 0x 来指示十六进制。大小写对于任何数字或基数都不重要。十进制是默认设置，例如：

```
-SCODE=f000
```

是否忘记了基数？

```
-SCODE=f000h
```

(439) bad -D spec: "*" *（链接器）*

-D 规范 （为类提供 *delta* 值）的格式是无效的，例如：

```
-DCODE
```

该类的 *delta* 值是什么？也许您的本意是以下形式的代码：

```
-DCODE=2
```

(440) bad delta value in -D spec *（链接器）*

为 -D 规范提供的 *delta* 值是无效的。该值应为以 8、 10 或 16 为基数的整数。

(441) bad -A spec: "*" *（链接器）*

-A 规范 （为链接器提供地址范围）的格式是无效的，例如：

```
-ACODE
```

该类的范围是什么？也许您的本意是：

```
-ACODE=0h-1fffh
```


© 2013 Microchip Technology Inc.

(442) missing address in -A spec *（链接器）*

-A 规范 （为链接器提供地址范围）的格式是无效的，例如：

```
-ACODE=
```

该类的范围是什么？也许您的本意是：

```
-ACODE=0h-1fffh
```

(443) bad low address "*" in -A spec *（链接器）*

-A 规范中给出的低地址是无效的：它应为有效的数字，使用十进制、八进制或十六进制基数。基数通过后缀 O （八进制）或 H （十六进制）来指定。也可以使用前缀 0x 来指示十六进制。大小写对于任何数字或基数都不重要。十进制是默认设置，例如：

```
-ACODE=1fff-3fffh
```

是否忘记了基数？

```
-ACODE=1fffh-3fffh
```

(444) expected "-" in -A spec *（链接器）*

链接器选项 -A 中的高地址和低地址之间需要一个减号 -，例如：

```
-AROM=1000h
```

也许您的本意是：

```
-AROM=1000h-1fffh
```

(445) bad high address "*" in -A spec *（链接器）*

-A 规范中给出的高地址是无效的：它应为有效的数字，使用十进制、八进制或十六进制基数。基数通过后缀 O （八进制）或 H （十六进制）来指定。也可以使用前缀 0x 来指示十六进制。大小写对于任何数字或基数都不重要。十进制是默认设置，例如：

```
-ACODE=0h-ffff
```

是否忘记了基数？

```
-ACODE=0h-ffffh
```

更多信息，请参见**第 7.2.1 节 “-Aclass =low-high,...”**。

(446) bad overrun address "*" in -A spec *（链接器）*

-A 规范中给出的溢出地址是无效的：它应为有效的数字，使用十进制、八进制或十六进制基数。基数通过后缀 O （八进制）或 H （十六进制）来指定。也可以使用前缀 0x 来指示十六进制。大小写对于任何数字或基数都不重要。十进制是默认设置，例如：

```
-AENTRY=0-0FFh-1FF
```

是否忘记了基数？

```
-AENTRY=0-0FFh-1FFh
```

(447) bad load address "*" in -A spec *（链接器）*

-A 规范中给出的装入地址是无效的：它应为有效的数字，使用十进制、八进制或十六进制基数。基数通过后缀 O （八进制）或 H （十六进制）来指定。也可以使用前缀 0x 来指示十六进制。大小写对于任何数字或基数都不重要。十进制是默认设置，例如：

```
-ACODE=0h-3fffh/a000
```

是否忘记了基数？

```
-ACODE=0h-3fffh/a000h
```

(448) bad repeat count "*" in -A spec *（链接器）*

-A 规范中给出的重复计数是无效的，例如：

```
-AENTRY=0-0FFhxf
```

是否忘记了基数？

```
-AENTRY=0-0FFhxfh
```

(449) syntax error in -A spec: * *（链接器）*

-A 规范是无效的。有效的 -A 规范应为以下形式：

-AROM=1000h-1FFFh

(450) psect "*" was never defined *（链接器）*

-P 选项中列出了该 psect，但它未在程序内的任何模块中定义。

(451) bad psect origin format in -P option *（链接器）*

-P 选项中的起点格式不是格式有效的十进制、八进制或十六进制数字，它也不是现有 psect 的名称。十六进制数字必须跟随后缀 H，例如：

```
-pbss=f000
```

是否忘记了基数？

```
-pbss=f000h
```

(452) bad "+" (minimum address) format in -P option *（链接器）*

链接器 -p 选项中的最低地址规范存在格式错误，例如：

```
-pbss=data+f000
```

是否忘记了基数？

```
-pbss=data+f000h
```

(453) missing number after "%" in -P option *（链接器）*

-p 选项中的 % 操作符 （用于舍入边界）必须跟随一个数字。

© 2013 Microchip Technology Inc.

(454) link and load address can't both be set to "." in -P option *（链接器）*

psect 的链接和装入地址都使用*点号*字符指定。这两个地址中只有一个地址可以使用这种方式指定，例如：

```
-Pmypsect=1000h/.
-Pmypsect=./1000h
```

这两个选项都是有效的，并且是等效的。但是，以下用法具有歧义：

```
-Pmypsect=./.
```

该 psect 的链接或装入地址是什么？

(455) psect "*" not relocated on 0x* byte boundary *（链接器）*

该 psect 未被重定位到所需的边界上。检查 psect 是否可重定位，并根据需要更正 -p 选项。

(456) psect "*" not loaded on 0x* boundary *（链接器）*

该 psect 具有重定位要求，但 -p 选项给出的装入地址无法满足它。例如，如果某个 psect 必须处于 4 KB 边界上，则不能将它设为在 100H 处开始。

(459) remove failed, error: *, * *（xstrip）*

输出文件的创建在删除中间文件时发生失败。

(460) rename failed, error: *, * *（xstrip）*

输出文件的创建在重命名中间文件时发生失败。

(461) can't create * file "*" *（汇编器和代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(464) missing key in avmap file *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(465) undefined symbol "*" in FNBREAK record *（链接器）*

链接器在某个不可重入函数的 FNBREAK 记录中发现未定义的符号。如果它不是手写的汇编代码，请联系 Microchip 技术支持。

(466) undefined symbol "*" in FNINDIR record *（链接器）*

链接器在某个不可重入函数的 FNINDIR 记录中发现未定义的符号。如果它不是手写的汇编代码，请联系 Microchip 技术支持。

(467) undefined symbol "*" in FNADDR record *（链接器）*

链接器在某个不可重入函数的 FNADDR 记录中发现未定义的符号。如果它不是手写的汇编代码，请联系 Microchip 技术支持。

(468) undefined symbol "*" in FNCALL record *（链接器）*

链接器在某个不可重入函数的 FNCALL 记录中发现未定义的符号。如果它不是手写的汇编代码，请联系 Microchip 技术支持。

(469) undefined symbol "*" in FNROOT record *（链接器）*

链接器在某个不可重入函数的 FNROOT 记录中发现未定义的符号。如果它不是手写的汇编代码，请联系 Microchip 技术支持。

(470) undefined symbol "*" in FNSIZE record *（链接器）*

链接器在某个不可重入函数的 FNSIZE 记录中发现未定义的符号。如果它不是手写的汇编代码，请联系 Microchip 技术支持。

(471) recursive function calls: *（链接器）*

这些函数（或这个函数）相互递归调用。这些函数中的一个或多个具有静态分配的局部变量（编译堆栈）。请使用 reentrant 关键字（如果该编译器支持它）或重新进行编码以避免递归，例如：

```
int test(int a)
{
   if(a == 5) {
      /* recursion may not be supported by some compilers */
      return test(a++);
   }
   return 0;
}
```

(472) non-reentrant function "*" appears in multiple call graphs: rooted at "*" and "*" *（链接器）*

主干代码和中断代码都会调用该函数。请使用 reentrant 关键字（如果该编译器支持它）或重新进行编码以避免使用局部变量或参数，或者复制该函数，例如：

```
void interrupt my_isr(void)
{
  scan(6);     /* scan is called from an interrupt function */
}
void process(int a)
{
  scan(a);     /* scan is also called from main-line code */
}
```

(473) function "*" is not called from specified interrupt_level *（链接器）*

永远不会从相同中断级别的中断函数中调用所指示的函数，例如：

```
#pragma interrupt_level 1
void foo(void)
{
    ...
}
#pragma interrupt_level 1
void interrupt bar(void)
{
    // this function never calls foo()
}
```


© 2013 Microchip Technology Inc.

(474) no psect specified for function variable/argument allocation *（链接器）*

从未发现 FNCONF 汇编器伪指令（向链接器指定关于自动 / 参数块的信息）。它在标准运行时文件中提供（如需要）。该错误可能意味着未链接正确的运行时启动模块。如果运行时启动模块是手写的，请确保已使用 FNCONF 伪指令。

(475) conflicting FNCONF records *（链接器）*

链接器发现两条冲突的 FNCONF 伪指令。该伪指令应仅指定一次，并包含在通常链接到每个程序中的标准运行时启动代码中。

(476) fixup overflow referencing * * (location 0x* (0x*+*), size *, value 0x*) *（链接器）*

链接器被要求重定位（fixup）某个项目，但它在重定位之后将无法装回到空间中。更多信息，请参见下面的错误消息 (477)。

(477) fixup overflow in expression (location 0x* (0x*+*), size *, value 0x*) *（链接器）*

fixup 是一个由链接器执行的过程，该过程会将汇编指令中对变量等的符号引用替换为一个绝对值。它是在将 psect（程序段或块）定位到目标器件上的可用存储器中之后进行的。如果对于某个符号确定的值太大，无法装入汇编指令分配的空间内，将会发生 fixup 溢出。例如，如果汇编指令使用 8 位字段来存放地址，而链接器确定用来表示该地址的符号具有值 0x110，则显然无法将该值插入指令。

该错误的原因有很多，但手写汇编器代码总是第一嫌疑对象。写得很差的 C 代码也可能会生成最终产生 fixup 溢出错误的汇编代码。假设具有以下错误消息。

```
main.obj: 8: Fixup overflow in expression (loc 0x7FD (0x7FC+1),
    size 1, value 0x1FC)
```

这表明导致问题的文件为 main.obj。它通常为编译 main.c 或 main.as 产生的输出。它指示您应查看的文件。下一个数字（在该示例中为 8）是导致问题的目标文件中的记录编号。如果使用 DUMP 实用程序来检查目标文件，可以确定该记录；但是，通常不需要这么做。

指令所在的存储单元（loc 0x7FD）、指令中对应于值的字段的长度（size 1，以字节为单位），以及符号代表的实际值 value（0x1FC），通常是追踪错误原因惟一需要的信息。请注意，如果长度不是 8 位的倍数，将会向上舍入为最接近的字节长度；例如，指令中的 7 位空间将显示为 1 字节。

生成相应模块的汇编列表文件。查找错误消息中指定的地址。

```
7   07FC   0E21   MOVLW 33
8   07FD   6FFC   MOVWF _foo
9   07FE   0012   RETURN
```

并且为了进行确认，在同一文件底部的符号表中查找该地址处的汇编指令引用的符号。

```
Symbol Table                    Fri Aug 12 13:17:37 2004
_foo 01FC   _main 07FF
```

在该示例中，导致问题的指令可接受存储区的 8 位偏移量，但显然地址 0x1FC 超出了这个长度。也许应将该指令编写为：

```
MOVWF  (_foo&0ffh)
```

它会屏蔽掉包含存储区信息的地址高位。

如果导致该错误的汇编指令是由编译器生成的，则在汇编列表文件根据该指令查找导致错误的文件，以确定是哪条 C 语句生成了该指令。然后，您需要通过检查 C 代码来确定可能的错误。未正确限定的指针是常见的触发原因。

(478) * range check failed (location 0x* (0x*+*), value 0x* > limit 0x*) *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(479) circular indirect definition of symbol "*" *（链接器）*

指定符号已被设为等同于某个外部符号，后者又反过来被设为等同于第一个符号。

(480) function signatures do not match: * (*): 0x*/0x* *（链接器）*

指定函数在不同模块中具有不同的签名。这意味着它被以不同的形式声明；例如，它可能在一个模块中声明了原型，但未在另一个中声明。在两个指定模块中，检查函数的可见声明，并确保它们是兼容的，例如：

```
extern int get_value(int in);
/* and in another module: */
/* this is different to the declaration */
int get_value(int in, char type)
{
```

(481) common symbol "*" psect conflict *（链接器）*

某个公共符号被定义为处于多个 psect 中。

(482) symbol "*" is defined more than once in "*" *（汇编器）*

在多个位置定义了该符号。如果某个符号在同一模块中多次定义，则汇编器会发出该错误，例如：

```
_next:
  MOVE r0, #55
  MOVE [r1], r0
_next:           ; oops -- choose a different name
```

如果符号（C 或汇编）在不同模块中多次定义，则链接器会发出该警告。错误消息中会给出模块的名称。请注意，在编译之后， C 标识符通常会在其名称前加上 *下划线*。

(483) symbol "*" can’t be global *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(484) psect "*" can’t be in classes "*" and "*" *（链接器）*

一个 psect 不能处于多个类中。这或者是由于汇编模块对于 PSECT 伪指令使用了冲突的 `class=` 选项，或者是对于链接器使用了冲突的 `-C` 选项而导致的，例如：

```
psect final,class=CODE
finish:
/* elsewhere: */
psect final,class=ENTRY
```


© 2013 Microchip Technology Inc.

(485) unknown "with" psect referenced by psect "*" *（链接器）*

指定 psect 使用 psect `with` 标志与某个 psect 一起放置。用于放置它的 psect 不存在，例如：

```
psect starttext,class=CODE,with=rext
    ; was that meant to be with text?
```

(486) psect "*" selector value redefined *（链接器）*

该 psect 的选择器值定义了多次。

(487) psect "*" type redefined: */* *（链接器）*

不同模块为该 psect 定义了不同的类型。这可能意味着您试图链接不兼容的目标模块；例如，将 386 平面模型代码与 8086 实模式代码进行链接。

(488) psect "*" memory space redefined: */* *（链接器）*

某个全局 psect 被定义为处于两个不同的存储空间。请重命名其中一个 psect，或者如果它们是同一 psect，则使用 `space` psect 标志将它们放在同一存储空间中，例如：

```
psect spdata,class=RAM,space=0
  ds 6
; elsewhere:
psect spdata,class=RAM,space=1
```

(489) psect "*" memory delta redefined: */* *（链接器）*

某个全局 psect 被定义了两个不同的 delta 值，例如：

```
psect final,class=CODE,delta=2
finish:
; elsewhere:
psect final,class=CODE,delta=1
```

(490) class "*" memory space redefined: */* *（链接器）*

某个类被定义为处于两个不同的存储空间。请重命名其中一个类，或者如果它们是同一个类，则将它们放在同一存储空间中。

(491) can’t find 0x* words for psect "*" in segment "*" *（链接器）*

链接器执行的主要任务之一是将它从程序生成的代码和数据块（或 psect）定位到目标器件的可用存储器中。该错误指示链接器无法找到足以容纳其中一个 psect 的可用存储区。该错误消息会指示链接器尝试定位的 psect 的名称和段名，后者通常是使用链接器 `-A` 选项定义的类的名称。**第 5.15.2 节 “编译器生成的 psect”** 列出了编译器生成的每个 psect，以及它所包含的内容。通常，名称为 `text` 或包含 `text` 的 psect 与程序代码相关。 `bss` 或 `data` 之类的名称指代变量块。该错误的可能原因有两个。

第一个原因是，程序或程序数据的长度超出了选定器件上的空间总量。即，器件存储器的某些部分已完全填满。如果是这种情况，则必须减小所指定 psect 的长度。

该消息的第二个原因是要定位的 psect 所需的存储器总量是足够的，但该存储器被碎片化，最大的连续块太小而无法容纳该 psect。链接器无法在这种情况下拆分 psect。即，链接器无法将 psect 的一部分放在一个位置，将其他部分放在别的位置。因此，链接器必须能够为每个 psect 寻找足够大的连续存储器块。如果这是错误的原因，则必须将 psect 拆分为较小的 psect （如可能）。

要确定哪些存储器仍然可用，可以生成映射文件并在其中进行查找；关于如何生成映射文件的信息，请参见**第4.8.8节“-M：生成映射文件”**。搜索字符串UNUSED ADDRESS RANGES。在该标题下，查找错误消息中指定的段的名称。如果该名称不存在，则说明可用于该 psect 的所有存储器都已分配。如果它存在，则在该段下对于每个可用存储器块都会指定一个地址范围。确定每个块的大小，并将它与错误消息中指定的字数进行比较。

通过使用所有编译器优化或重新构造程序，可以减小包含代码的 psect。如果必须将某个代码 psect 拆分为两个或更多的小 psect，就会需要将一个函数拆分为两个或多个更小的函数 （它们可以相互调用）。这些函数可能需要放在新的模块中。

在调用编译器优化时，包含数据的 psect 可能会减小，但效果不会那么显著。程序可能需要重新编写，从而使它需要的变量减少。如果必须更改默认的链接器选项，可以使用驱动程序 -L- 选项间接通过驱动程序实现，请参见**第 4.8.7 节 “-L-：直接调整链接器选项”**。如果您使用的编译器使用编译堆栈，则可以通过**第 4.8.8 节 “-M：生成映射文件”**了解关于解释映射文件调用图的信息。（如果映射文件中不存在字符串 `Call graph:`，则说明编译的代码使用硬件堆栈。）如果数据 psect 需要拆分成更小的 psect，变量的定义将需要移动到新的模块中，或更均匀地分布在现有模块中。自动变量的存储器分配完全由编译器进行处理。除了减少使用的这些变量的数量之外，编程人员几乎无法控制它们的操作。无论编译的代码使用的是硬件堆栈还是编译堆栈，都适用这一点。

例如，在接收到以下消息之后：

```
Can’t find 0x34 words (0x34 withtotal) for psect text
 in segment CODE (error)
```

在映射文件中查找未用存储器的范围。

```
UNUSED ADDRESS RANGES
        CODE             00000244-0000025F
                         00001000-0000102f
        RAM              00300014-00301FFB
```

在 CODE 段中，有一个块具有 0x1c（0x25f-0x244+1）字节的可用空间，另一个块具有 0x30 的可用空间。它们都不够大，无法容纳 0x34 字节长的 psect `text`。但请注意，可用的存储器总量大于 0x34 字节。

(492) attempt to position absolute psect "*" is illegal *（链接器）*

该 psect 是绝对的，不应在 `-P` 选项中指定它的地址。删除 `abs` psect 标志或删除 `-P` 链接器选项。

(493) origin of psect "*" is defined more than once *（链接器）*

该 psect 的起点定义了多次。很可能存在多个指定了该 psect 的 `-p` 链接器选项。

© 2013 Microchip Technology Inc.

(494) bad -P format "*/*" *（链接器）*

传递给链接器的 `-P` 选项的格式不正确。该选项用于指定 psect 的位置，例如：

```
-Ptext=10g0h
```

也许您的本意是：

```
-Ptext=10f0h
```

(495) use of both "with=" and "INCLASS/INCLASS" allocation is illegal *（链接器）*

如果在定义某个 psect 时，还使用了 `with` psect 标志，则在类中指定该 psect 的链接和位置是不合法的。

(497) psect "*" exceeds max size: *h > *h *（链接器）*

psect 的字节数多于使用 `size` psect 标志指定的最大允许值。

(498) psect "*" exceeds address limit: *h > *h *（链接器）*

psect 的最高地址超出了使用 `limit` psect 标志对它设置的限制。该 psect 需要链接到不同位置或 psect 中存在太多的代码 / 数据。

(499) undefined symbol: *（汇编器和链接器）*

随后的符号在链接时是未定义的。这可能是由于拼写错误或未能链接相应模块而导致的。

消息 500-749

(500) undefined symbols: *（链接器）*

随后的符号列表在链接时是未定义的。这些错误可能是由于拼写错误或未能链接相应模块而导致的。

(501) program entry point is defined more than once *（链接器）*

传递给链接器的目标文件中定义了多个入口点。结束入口点在 `END` 伪指令之后指定。运行时启动代码定义入口点，例如：

```
powerup:
  goto start
  END  powerup  ; end of file and define entry point
; other files that use END should not define another entry point
```

(502) incomplete * record body: length = * *（链接器）*

一个目标文件包含具有非法长度的记录。这可能意味着该文件已被截断或不是一个目标文件。详情请联系 Microchip 技术支持。

(503) ident records do not match *（链接器）*

传递给链接器的目标文件不具有匹配的 ident 记录。这意味着它们针对的是不同的器件类型。

(504) object code version is greater than *.* *（链接器）*

目标模块的目标代码版本高于已知链接器可处理的最高版本。请检查是否使用了正确的链接器。如果未对链接器应用补丁，请联系 Microchip 技术支持。

(505) no end record found inobject file *（链接器）*

目标文件中未包含结束记录。这可能意味着该文件已损坏或不是一个目标文件。如果目标文件是由编译器生成的，请联系 Microchip 技术支持。

(506) object file record too long: *+* *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(507) unexpected end of file in object file *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(508) relocation offset (*) out of range 0..*-*-1 *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(509) illegal relocation size: * *（链接器）*

链接器读取的目标代码格式中存在错误。这或者意味着您使用的链接器已过时，或者汇编器或链接器中发生了内部错误。如果目标文件是由编译器创建的，请联系 Microchip 技术支持来了解详情。

(510) complex relocation not supported for -R or -L options *（链接器）*

向链接器传递了 -R 或 -L 选项，但文件包含复杂的重定位。

(511) bad complex range check *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(512) unknown complex operator 0x* *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(513) bad complex relocation *（链接器）*

链接器被要求执行语法不正确的复杂重定位。可能意味着目标文件已损坏。

(514) illegal relocation type: * *（链接器）*

一个目标文件包含具有非法重定位类型的重定位记录。这可能意味着该文件已损坏或不是一个目标文件。如果目标文件是由编译器创建的，请联系 Microchip 技术支持来了解详情。

(515) unknown symbol type * *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(516) text record has bad length: *-*-(*+1) < 0 *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(520) function "*" is never called *（链接器）*

该函数永远不会被调用。这可能并不表示存在问题，但可以通过删除它来节省空间。如果您认为应调用该函数，请检查您的源代码。一些汇编库程序永远不会被调用，虽然它们实际上会执行。在这种情况下，这些程序将按特殊顺序进行链接，从而使程序执行从一个程序直落到下一个。

(521) call depth exceeded by function "*" *（链接器）*

调用图显示函数的嵌套深度大于指定的深度。

© 2013 Microchip Technology Inc.

(522) library "*" is badly ordered *（链接器）*

该库中的顺序很差。它仍然可以正确链接，但如果更好地安排其中的顺序，链接速度会更快。

(523) argument to -W option (*) illegal and ignored *（链接器）*

链接器选项 `-w` 的参数超出范围。该选项控制两个功能。对于警告级别，其范围为 -9 至 9。对于映射文件宽度，其范围为大于等于 10。

(524) unable to open list file "*": * *（链接器）*

无法打开指定的列表文件。链接器将尝试 fixup 列表文件，从而使它将包含绝对地址。请确保在编译阶段生成汇编列表文件。或者，从链接步骤中删除汇编列表文件生成选项。

(525) too many address (memory) spaces; space (*) ignored *（链接器）*

地址空间数量的限制 （使用 PSECT 汇编器伪指令指定）当前为 16。

(526) psect "*" not specified in -P option (first appears in "*") *（链接器）*

该 psect 未在链接器的 `-P` 或 `-A` 选项中指定。它已被链接到程序末尾，这可能不是您所希望的位置。

(528) no start record; entry point defaults to zero *（链接器）*

传递给链接器的所有目标文件均不包含起始记录。程序的起始地址已被设置为零。这可能是无害的，但建议在启动模块中通过使用 END 伪指令定义起始地址。

(529) usage: objtohex [-Ssymfile] [object-file [HEX-file]] *（Objtohex）*

命令行工具 `objtohex` 的用法不正确。如果要直接调用 `objtohex`，请参见**第 8.3 节 “OBJTOHEX”**了解更多详细信息。或者，这可能是内部编译器错误，详情请联系 Microchip 技术支持。

(593) can’t find 0x* words (0x* withtotal) for psect "*" in segment "*" *（链接器）*

请参见消息 (491)。

(594) undefined symbol: *（链接器）*

随后的符号在链接时是未定义的。这可能是由于拼写错误或未能链接正确的模块而导致的。

(595) undefined symbols: *（链接器）*

随后的符号列表在链接时是未定义的。这些错误可能是由于拼写错误或未能链接正确的模块而导致的。

(596) segment "*" (*-*) overlaps segment "*" (*-*) *（链接器）*

指定的段具有重叠的代码或数据。请检查由 `-P` 链接器选项分配的地址。

(599) No psect classes given for COFF write *（Cromwell）*

要生成 COFF 文件，`CROMWELL` 要求指定程序存储器 psect 类。请确保按照**第 8.5.2 节 “-N”**的说明使用 `-N` 选项。

(600) No chip arch given for COFF write *（Cromwell）*

要生成 COFF 文件， CROMWELL 要求指定芯片架构。请确保按照表 8-7 的说明使用 -P 选项。

(601) Unknown chip arch "*" for COFF write *（Cromwell）*

CROMWELL 无法识别为生成 COFF 文件而指定的芯片架构。请确保按照**第 8.5.1 节 “-Pname[,architecture]”**的说明使用 -P选项，并且指定的架构与表8-7中的一种匹配。

(602) null file format name *（Cromwell）*

传递给 CROMWELL 的 -I 或 -O 选项必须指定文件格式。

(603) ambiguous file format name "*" *（Cromwell）*

向 CROMWELL 指定的输入或输出格式具有歧义。这些格式分别使用 -i *键*和 -o *键*选项指定。

(604) unknown file format name "*" *（Cromwell）*

向 CROMWELL 指定的输出格式是未知的，例如：

```
cromwell -m -P16F877 main.HEX main.sym -ocot
```

输出文件类型为 cot，您的本意是否是 cof？

(605) did not recognize format of input file *（Cromwell）*

传递给 CROMWELL 的输入文件必须为 COD、 Intel HEX、 Motorola HEX、 COFF、 OMF51、 P&E 或 HI-TECH。

(606) inconsistent symbol tables *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(607) inconsistent line number tables *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(608) bad path specification *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(609) missing device spec after -P *（Cromwell）*

传递给 CROMWELL 的 -p 选项必须指定器件名称。

(610) missing psect classes after -N *（Cromwell）*

CROMWELL 要求为 -N 选项提供 psect 类名的列表。

(611) too many input files *（Cromwell）*

指定要由 CROMWELL 转换的输入文件太多。

(612) too many output files *（Cromwell）*

向 CROMWELL 指定的输出文件格式太多。

(613) no output file format specified *（Cromwell）*

必须向 CROMWELL 指定输出格式。

(614) no input files specified *（Cromwell）*

CROMWELL 必须具有要转换的输入文件。

© 2013 Microchip Technology Inc.

(616) option -Cbaseaddr is illegal with options -R or -L *（链接器）*

链接器选项 `-Cbaseaddr` 不能与 `-R` 或 `-L` 链接器选项一起使用。

(618) error reading COD file data *（Cromwell）*

读取输入 COD 文件时出错。请确认在命令行上指定的文件的拼写和路径。

(619) I/O error reading symbol table *（Cromwell）*

COD 文件在指定记录中具有无效的格式。

(620) filename index out of range in line number record *（Cromwell）*

COD 文件在指定记录中具有无效的值。

(621) error writing ELF/DWARF section "*" on "*" *（Cromwell）*

将指示的段写入给定文件时出错。请确认在命令行上指定的文件的拼写和路径。

(622) too many type entries *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(623) bad class in type hashing *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(624) bad class in type compare *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(625) too many files in COFF file *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(626) string lookup failed in COFF: get_string() *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(627) missing "*" in SDB file "*" line * column * *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(629) bad storage class "*" in SDB file "*" line * column * *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(630) invalid syntax for prefix list in SDB file "*" *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(631) syntax error at token "*" in SDB file "*" line * column * *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(632) can’t handle address size (*) *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(633) unknown symbol class (*) *（Cromwell）*

`CROMWELL` 在 COFF、Microchip COFF 或 ICOFF 文件的符号表中遇到它无法识别的符号类。

(634) error dumping "*" *（Cromwell）*

CROMWELL 的输入文件为不受支持的类型，或该文件无法被转储到屏幕上。

(635) invalid HEX file "*" on line * *（Cromwell）*

指定的 HEX 文件包含无效的行。如果 HEX 文件是由编译器生成的，请联系 Microchip 技术支持。

(636) checksum error in Intel HEX file "*" on line * *（Cromwell 和 Hexmate）*

在指定 Intel HEX 文件的指定行发现校验和错误。该 HEX 文件可能已损坏。

(637) unknown prefix "*" in SDB file "*" *（Cromwell）*

这是内部编译器警告。详情请联系 Microchip 技术支持。

(638) version mismatch: 0x* expected *（Cromwell）*

输入 Microchip COFF 文件不是使用 CROMWELL 生成的。

(639) zero bit width in Microchip optional header *（Cromwell）*

输入 Microchip COFF 文件中的可选头部指示程序或数据存储空间为零位宽。

(668) prefix list did not match any SDB types *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(669) prefix list matched more than one SDB type *（Cromwell）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(670) bad argument to -T *（Clist）*

-T 选项的指定制表符长度的参数不存在或格式不正确。该选项要求一个十进制整型参数。

(671) argument to -T should be in range 1 to 64 *（Clist）*

-T 选项的指定制表符长度的参数不处于预期范围。该选项要求一个介于 1（含）至 64（含）范围的十进制整型参数。

(673) missing filename after * option *（Objtohex）*

指示的选项需要一个有效的文件名。请确保提供给该选项的文件名参数存在，并且拼写正确。

(674) too many references to "*" *（Cref）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(677) set_fact_bit on pic17! *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(678) case 55 on pic17! *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(679) unknown extraspecial: * *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(680) bad format for -P option *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

© 2013 Microchip Technology Inc.

(681) bad common spec in -P option *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(682) this architecture is not supported by the PICC™ Lite compiler *（代码生成器）*

指定了低档、中档或高档器件之外的目标器件。该编译器仅支持来自这些架构系列的器件。

(683) bank 1 variables are not supported by the PICC Lite compiler *（代码生成器）*

检测到绝对地址位于存储区 1 中的变量。该编译器不支持对处于该存储区中的变量进行代码生成。

(684) bank 2 and 3 variables are not supported by the PICC Lite compiler *（代码生成器）*

检测到绝对地址位于存储区 2 或 3 中的变量。该编译器不支持对处于这些存储区中的变量进行代码生成。

(685) bad putwsize() *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(686) bad switch size (*) *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(687) bad pushreg "*" *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(688) bad popreg "*" *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(689) unknown predicate "*" *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(690) interrupt function requires address *（代码生成器）*

高档 PIC 器件支持多个中断。中断定义需要使用 *@ address* 来指示该程序所关联的向量，例如：

```
void interrupt isr(void) @ 0x10
{
  /* isr code goes here */
}
```

中档 PIC 器件不需要该构造。

(691) interrupt functions not implemented for 12 bit PIC MCU *（代码生成器）*

12 位的 PIC MCU 处理器系列不支持中断。

(692) interrupt function "*" may only have one interrupt level *（代码生成器）*

只能有一个中断优先级与中断函数相关联。请进行检查，以确保对于指定的函数仅使用了一条 `interrupt_level` pragma 伪指令。对于从中断函数中调用的主干函数，可以多次使用该 pragma 伪指令。例如：

```
#pragma interrupt_level 0
#pragma interrupt_level 1  /* which is it to be: 0 or 1? */
void interrupt isr(void)
{
```

(693) interrupt level may only be 0 (default) or 1 *（代码生成器）*

惟一可能的中断优先级为 0 或 1。请进行检查，以确保所有 interrupt_level pragma 伪指令都使用这些优先级。

```
#pragma interrupt_level 2  /* oops -- only 0 or 1 */
void interrupt isr(void)
{
  /* isr code goes here */
}
```

(694) no interrupt strategy available *（代码生成器）*

器件不支持在中断服务程序期间保存并随后恢复寄存器。

(695) duplicate case label (*) *（代码生成器）*

在该 switch 语句中，有两个 case 标号具有相同的值，例如：

```
switch(in) {
case '0':  /* if this is case '0'... */
  b++;
  break;
case '0':  /* then what is this case? */
  b--;
  break;
}
```

(696) out-of-range case label (*) *（代码生成器）*

该 case 标号不是控制表达式可以产生的值，因而永远不会选择该标号。

(697) non-constant case label *（代码生成器）*

该 switch 语句中的 case 标号具有不是常量的值。

(698) bit variables must be global or static *（代码生成器）*

位变量不能为自动类型。如果需要一个作用域为代码或函数块内部的位变量，请使用 static 限定它，例如：

```
bit proc(int a)
{
  bit bb;        /* oops --  this should be: static bit bb; */
  bb = (a > 66);
  return bb;
}
```

(699) no case labels in switch *（代码生成器）*

该 switch 语句中不具有任何 case 标号，例如：

```
switch(input) {
}              /* there is nothing to match the value of input */
```

(700) truncation of enumerated value *（代码生成器）*

检测到大于该编译器支持的最大值的枚举值，它已被截断，例如：

```
enum { ZERO, ONE, BIG=0x99999999 } test_case;
```

(701) unreasonable matching depth *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

© 2013 Microchip Technology Inc.

(702) regused(): bad arg to G *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(703) bad GN *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(704) bad RET_MASK *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(705) bad which (*) after I *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(706) bad which in expand() *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(707) bad SX *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(708) bad mod "+" for how = "*" *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(709) metaregister "*" can’t be used directly *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(710) bad U usage *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(711) bad how in expand() *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(712) can’t generate code for this expression *（代码生成器）*

该错误指示 C 表达式太难，代码生成器无法实际进行编译。要成功进行代码生成，代码生成器必须知道如何编译表达式，必须有足够的资源（即，寄存器或临时存储单元）可用。简化该表达式（例如，使用一个临时变量来存放中间结果）可能可以避开该消息。关于该消息的详细信息，请联系 Microchip 技术支持。

如果所编译的代码不正常，也可能会发出该错误。例如，对使用 const 限定的对象执行写操作的代码是非法的，将会导致警告消息，但代码生成器可能会试图生成执行写操作的代码，但不会成功。

(713) bad initialization list *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(714) bad intermediate code *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(715) bad pragma "*" *（代码生成器）*

向代码生成器传递了它无法理解的 `pragma` 伪指令。这意味着您使用的 pragma 伪指令并未对于目标器件实现。

(716) bad argument to -M option "*" *（代码生成器）*

向代码生成器传递了它无法理解的 -M 选项。如果它是由标准编译器驱动程序调用的，则不应发生这种情况。

(718) incompatible intermediate code version; should be *.* *（代码生成器）*

由 P1 生成的中间代码文件不是可用于该代码生成器的正确版本。这或者是由于在同一目录中安装了一个或多个编译器的这种不兼容版本，或者发生了导致临时文件损坏的临时文件错误。请检查 TEMP 环境变量的设置。如果它引用很长的路径名，则将其更改为较短的路径。如果需要，请联系 Microchip 技术支持来了解详情。

(720) multiple free: * *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(721) element count must be constant expression *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(722) bad variable syntax in intermediate code *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(723) function definitions nested too deep *（代码生成器）*

C 代码不大可能发生该错误，因为 C 不能具有嵌套函数！详情请联系 Microchip 技术支持。

(724) bad op (*) in revlog() *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(726) bad op "*" in uconval() *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(727) bad op "*" in bconfloat() *（代码生成器）*

这是内部代码生成器错误。详情请联系 Microchip 技术支持。

(728) bad op "*" in confloat() *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(729) bad op "*" in conval() *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(730) bad op "*" *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(731) expression error with reserved word *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(732) initialization of bit types is illegal *（代码生成器）*

不能初始化 bit 类型的变量，例如：

```
bit b1 = 1; /* oops!
               b1 must be assigned after its definition */
```


© 2013 Microchip Technology Inc.

(733) bad string "*" in pragma "psect" *（代码生成器）*

向代码生成器传递了格式不正确的字符串的 pragma psect 伪指令，例如：

```
#pragma psect text  /* redirect text psect into what? */
```

也许您的本意是以下形式的代码：

```
#pragma psect text=special_text
```

(734) too many "psect" pragmas *（代码生成器）*

使用了太多的 #pragma psect 伪指令。

(735) bad string "*" in pragma "stack_size" *（代码生成器）*

stack_size pragma 的参数格式不正确。该 pragma 伪指令必须后跟一个数字，表示允许的最大堆栈大小。

(737) unknown argument "*" to pragma "switch" *（代码生成器）*

对于 #pragma switch 伪指令使用了无效的 switch 代码生成方法。可能的参数包括：auto、 simple 和 direct。

(739) error closing output file *（代码生成器）*

在关闭文件时，编译器检测到一个错误。详情请联系 Microchip 技术支持。

(740) zero dimension array is illegal *（代码生成器）*

向代码生成器传递了产生零维度数组的声明。

(741) bitfield too large (* bits) *（代码生成器）*

位域中的最大位数为 8，长度与存储单元宽度相同。

```
struct {
  unsigned flag  : 1;
  unsigned value : 12;  /* oops -- that's larger than 8 bits wide */
  unsigned cont  : 6;
} object;
```

(742) function "*" argument evaluation overlapped *（链接器）*

函数调用涉及到在两个函数之间重叠的参数。以下形式的调用可能会发生这种情况：

```
void fn1(void)
{
  fn3( 7, fn2(3), fn2(9));    /* Offending call */
}
char fn2(char fred)
{
  return fred + fn3(5,1,0);
}
char fn3(char one, char two, char three)
{
  return one+two+three;
}
```

其中， fn1 调用 fn3，并通过调用 fn2 对两个参数求值，该函数又反过来调用 fn3。应修改程序结构，以防止这种类型的调用序列。

(743) divide by zero *（代码生成器）*

在代码中检测到涉及被零除的表达式。

(744) static object "*" has zero size *（代码生成器）*

声明了静态对象，但其长度为零。

(745) nodecount = * *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(746) object "*" qualified const, but not initialized *（代码生成器）*

某个对象被限定为 `const` 类型，但在定义时未提供初始值。由于该对象不能由 C 程序写入，这可能意味着无意中遗漏了初始值。

(747) unrecognized option "*" to -Z *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(748) variable "*" may be used before set *（代码生成器）*

该变量可能在赋值之前被使用。由于它是一个自动变量，这将导致它具有一个随机值，例如：

```
void main(void)
{
  int a;
  if(a)         /* oops -- a has never been assigned a value */
    process();
}
```

(749) unknown register name "*" used with pragma *（链接器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

© 2013 Microchip Technology Inc.

消息 750-999

(750) constant operand to || or &&

（代码生成器）

逻辑操作符 || 或 && 的一个操作数为常量。请检查表达式是否缺少或具有错误放置的括号。如果使能了全局优化器，并且其中一个操作数是值会被代码生成器跟踪的自动或静态局部变量，则也可能出现该消息，例如：

```
{
int a;
a = 6;
if(a || b)  /* a is 6, therefore this is always true */
    b++;
```

(751) arithmetic overflow in constant expression

（代码生成器）

代码生成器对某个常量表达式进行求值，产生的值对于表达式类型来说太大。触发该警告最常见的代码是对有符号数据类型进行赋值。例如：

```
signed char c;
c = 0xFF;
```

作为一个有符号的 8 位量，c 只能赋值为 -128 至 127。常量等于 255，超出了该范围。如果您的本意是将该变量中的所有位置 1，则使用以下形式之一：

```
c = ~0x0;
c = -1;
```

它会将变量中的所有位置 1，无论变量的长度如何，并且不会发出警告。

该警告也可能由于中间值溢出而触发。例如：

```
unsigned int i;  /* assume ints are 16 bits wide */
i = 240 * 137;   /* this should be okay, right? */
```

使用计算器快速检查发现 240 * 137 等于 32880，它可以轻松地存储在一个 unsigned int 中，但会产生警告。这是什么原因？因为 240 和 137 均为 signed int 值。因此，相乘的结果也必须为 signed int 值，但 signed int 无法容纳值 32880。（两个操作数均为常量值，因此代码生成器可以在编译时对该表达式求值，但它必须遵循所有 ANSI C 规则来执行该操作。）以下代码会强制乘法运算产生无符号结果：

```
i = 240u * 137;  /* force at least one operand
                    to be unsigned */
```

(752) conversion to shorter data type

（代码生成器）

在该表达式中可能会发生截断，因为左值的类型比右值的类型短，例如：

```
char a;
int b, c;
a = b + c;  /* int to char conversion
               may result in truncation */
```

(753) undefined shift (* bits)

（代码生成器）

尝试对值执行的移位位数大于等于该数据类型中的位数。在许多处理器上，这将产生未定义的结果。这是不可移植的代码， C 标准将其标记为具有未定义的结果，例如：

```
int input;
input <<= 33;  /* oops -- that shifts the entire value out */
```

© 2013 Microchip Technology Inc.

(754) bitfield comparison out of range *（代码生成器）*

这是将位域与超出位域范围的值进行比较产生的结果。即，将 2 位位域与值 5 进行比较将永远不会为真，因为 2 位位域的范围为 0 至 3。例如：

```
struct {
  unsigned mask : 2;  /* mask can hold values 0 to 3 */
} value;
int compare(void)
{
  return (value.mask == 6);  /* test can
}
```

(755) divide by zero *（代码生成器）*

进行求值的常量表达式涉及到被零除，例如：

```
a /= 0;  /* divide by 0: was this what you were intending */
```

(757) constant conditional branch *（代码生成器）*

条件分支（由 `if`、`for` 和 `while` 等语句生成）总是遵循相同的路径。它是某种涉及变量和常量表达式的比较。要让代码生成器发出该消息，变量必须具有局部作用域（自动或静态局部变量），并且必须使能全局优化器 （级别可能高于 1），并且警告级别阈值可能需要低于默认级别 0。

全局优化器会尽可能长时间地跟踪局部变量在函数期间的内容。对于将这些变量与常量进行比较的 C 代码，可以在编译时推断出比较结果，并通过对输出代码进行硬编码来避免比较，例如：

```
{
  int a, b;
  a = 5;
  /* this can never be false;
     always perform the true statement */
  if(a == 4)
    b = 6;
```

会生成将 a 设置为 5，然后立即将 b 设置为 6 的代码。

对于比较 `if(a==4)` 将不会生成任何代码。如果 a 是全局变量，其他函数 （特别是中断函数）可能会对其进行修改，因而无法执行变量跟踪。

该警告可能表明编译器执行了多项优化。它可能表明，表达式缺少括号或具有错误放置的括号，导致求值产生不同于预期的值。

如果编写了诸如 `while(1)` 形式的代码，也可能会发出该警告。要生成无限循环，请使用 `for(;;)`。

© 2013 Microchip Technology Inc.

对于 for 循环会发生类似的情况，例如：

```
{
  int a, b;
  /* this loop must iterate at least once */
  for(a=0; a!=10; a++)
    b = func(a);
```

在此例中，代码生成器可以同样确定 a 被赋值为 0，然后立即检查以确定它是否等于 10。由于 a 在 for 循环期间会被修改，所以不能删除比较代码，但代码生成器可以调整代码，使之在循环的第一轮迭代中不执行比较；仅在随后几轮中执行比较。这可能不会减小代码的长度，但它可以加快程序执行速度。

(758) constant conditional branch: possible use of "=" instead of "==" *（代码生成器）*

if 或其他条件构造中存在一个将常量赋予变量的表达式。这可能意味着您不小心使用了赋值 = 而不是比较 ==，例如：

```
  int a, b;
  /* this can never be false;
     always perform the true statement */
  if(a = 4)
    b = 6;
```

会将值 4 赋予 a，由于赋值的结果总是为 true，比较可以忽略，并总是对 b 进行赋值。您的本意是否是：

```
/* this can never be false;
   always perform the true statement */
if(a == 4)
    b = 6;
```

它会检查 a 是否等于 4。

(759) expression generates no code *（代码生成器）*

该表达式不生成任何输出代码。请检查是否存在在函数调用中遗漏了括号之类的情况，例如：

```
int fred;
fred;     /* this is valid, but has no effect at all */
```

某些器件需要通过读取特殊功能寄存器来清零硬件标志。为了支持这种情形，在某些情况下，代码生成器会对于仅包含变量 ID 的语句生成代码。对于限定为 volatile 的变量可能会出现这种情形。输出代码通常将读取该变量，但不对读取的值执行任何操作。

(760) portion of expression has no effect *（代码生成器）*

该表达式的一部分没有作用，对表达式的值不会产生任何影响，例如：

```
int a, b, c;
a = b,c;  /* "b" has no effect,
             was that meant to be a comma? */
```

(761) sizeof yields 0 *（代码生成器）*

代码生成器获取了某个对象的长度，发现它为零。几乎可以肯定这表明在指针的声明中存在错误；例如，您可能声明了一个指向零长度数组的指针。通常，指向数组的指针几乎没有什么用处。如果需要一个指向未知长度的对象数组的指针，则只需要一个指向单个对象的指针，然后可以进行索引或递增。

(762) constant truncated when assigned to bitfield *（代码生成器）*

常量值对于赋予它的位域结构成员太大，例如：

```
struct INPUT {
  unsigned a : 3;
  unsigned b : 5;
} input_grp;
input_grp.a = 0x12;  /* oops -- 0x12 cannot fit into a 3-bit wide
object */
```

(763) constant left operand to "? :" operator *（代码生成器）*

条件操作符 ? 的左操作数为常量，因此三元操作符 ?: 的结果将总是相同的，例如：

```
a = 8 ? b : c;  /* this is the same as saying a = b; */
```

(764) mismatched comparison *（代码生成器）*

在某个变量或表达式和某个常量值之间进行了比较，但常量值不处于该表达式的可能值范围内，例如：

```
unsigned char c;
if(c > 300)      /* oops -- how can this be true? */
  close();
```

(765) degenerate unsigned comparison *（代码生成器）*

无符号值与零进行比较，它的结果将总是为 true 或 false，例如：

```
unsigned char c;
if(c >= 0)
```

将总是为 true，因为无符号值永远不会小于零。

(766) degenerate signed comparison *（代码生成器）*

有符号值与该类型可能的最小负值进行比较，从而比较结果将总是为 true 或 false，例如：

```
char c;
if(c >= -128)
```

将总是为 true，因为 8 位有符号 char 类型的最小负值为 -128。

(767) constant truncated to bitfield width *（代码生成器）*

常量值对于它进行操作的位域结构成员太大，例如：

```
struct INPUT {
  unsigned a : 3;
  unsigned b : 5;
} input_grp;
input_grp.a |= 0x13;   /* oops -- 0x13 to large for 3-bit wide object
*/
```


© 2013 Microchip Technology Inc.

(768) constant relational expression *（代码生成器）*

存在结果总是为 true 或 false 的关系表达式。举例来说，它可能是将无符号数与负值进行比较的结果；或将某个变量与大于它可表示的最大数字的值进行比较，例如：

```
unsigned int a;
if(a == -10)    /* if a is unsigned, how can it be -10? */
  b = 9;
```

(769) no space for macro definition *（汇编器）*

汇编器已用尽存储器。

(772) include files nested too deep *（汇编器）*

宏扩展和包含文件处理填满了汇编器的内部堆栈。打开的宏和包含文件的最大数量为 30。

(773) macro expansions nested too deep *（汇编器）*

汇编程序中的宏扩展嵌套太深。其限制为每次嵌套 30 个宏和包含文件。

(774) too many macro parameters *（汇编器）*

该宏定义中存在太多的宏参数。

(776) can’t allocate space for object "*" (offs: *) *（汇编器）*

汇编器已用尽存储器。

(777) can’t allocate space for opnd structure within object "*", (offs: *) *（汇编器）*

汇编器已用尽存储器。

(780) too many psects defined *（汇编器）*

定义了太多的 psect！好家伙，这是什么程序！

(781) can’t enter abs psect *（汇编器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(782) REMSYM error *（汇编器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(783) "with" psects are cyclic *（汇编器）*

如果 Psect A 要与 （with） Psect B 一起放置， Psect B 要与 Psect A 一起放置，则没有层次结构。 `with` 标志是 psect 的一个属性，它指示该 psect 必须放在与指定 psect 相同的存储器页。

从其中一个 psect 声明中删除 `with` 标志。这种汇编器声明可能为以下形式：

```
psect my_text,local,class=CODE,with=basecode
```

它将定义名为 `my_text` 的 psect，并将它放在与 psect `basecode` 相同的页中。

(784) overfreed *（汇编器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(785) too many temporary labels *（汇编器）*

该汇编文件中存在太多的临时标号。汇编器最多允许 2000 个临时标号。

(787) can't handle "v_rtype" of * in copyexpr *（汇编器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(788) invalid character "*" in number *（汇编器）*

数字包含不属于范围 0-9 或 0-F 的字符。

(790) end of file inside conditional *（汇编器）*

扫描 “endif”以匹配先前 “if”时遇到文件结束符。

(793) unterminated macro argument *（汇编器）*

宏的参数未进行终止。请注意，尖括号 （“< >”）用于括起宏参数。

(794) invalid number syntax *（汇编器）*

数字的语法无效。举例来说，它可能表示在八进制数中使用了 8 或 9，或其他格式不正确的数字。

(796) use of LOCAL outside macros is illegal *（汇编器）*

LOCAL 伪指令仅在宏内是合法的。它用于定义将对于每次宏调用都是惟一的局部标号。

(797) syntax error in LOCAL argument *（汇编器）*

在汇编器宏中使用 LOCAL 汇编器伪指令定义的某个符号的语法不正确。请确保所有符号和所有其他汇编标识符都符合目标器件的汇编语言。

(798) macro argument may not appear after LOCAL *（汇编器）*

伪指令 LOCAL 之后的标号列表不能包含该宏的任何形参，例如：

```
mmm MACRO a1
   MOVE   r0, #a1
   LOCAL  a1        ; oops --
; the macro parameter cannot be used with local
ENDM
```

(799) REPT argument must be >= 0 *（汇编器）*

REPT 伪指令的参数必须大于零，例如：

```
REPT -2             ; -2 copies of this code? */
  MOVE    r0, [r1]++
ENDM
```

(800) undefined symbol "*" *（汇编器）*

指定符号未在该模块中定义，并且未指定 GLOBAL。

(801) range check too complex *（汇编器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(802) invalid address after END directive *（汇编器）*

在汇编器 END 伪指令之后指定的程序起始地址必须为当前文件中的一个标号。

(803) undefined temporary label *（汇编器）*

引用了一个未定义的临时标号。请注意，临时标号必须具有一个 >= 0 的数字。

© 2013 Microchip Technology Inc.

(804) write error on object file *（汇编器）*

汇编器无法写入一个目标文件。这可能是内部编译器错误。详情请联系 Microchip 技术支持。

(806) attempted to get an undefined object (*) *（汇编器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(807) attempted to set an undefined object (*) *（汇编器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(808) bad size in add_reloc() *（汇编器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(809) unknown addressing mode (*) *（汇编器）*

汇编文件中使用了未知的寻址模式。

(811) "cnt" too large (*) in display() *（汇编器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(814) device type not defined *（汇编器）*

器件必须从命令行上（如 -16c84）通过器件汇编器伪指令或通过 LIST 汇编器伪指令定义。

(815) syntax error in chipinfo file at line * *（汇编器）*

chipinfo 文件的指定行中包含非标准语法。

(816) duplicate ARCH specification in chipinfo file "*" at line * *（汇编器和驱动程序）*

chipinfo 文件的器件段具有多个 ARCH 值。仅允许使用一个 ARCH 值。如果您并未手动编辑 chipinfo 文件，请联系 Microchip 技术支持来了解详情。

(817) unknown architecture in chipinfo file at line * *（汇编器和驱动程序）*

读取芯片 INI 文件时遇到未知的芯片架构 （系列）。

(818) duplicate BANKS for "*" in chipinfo file at line * *（汇编器）*

chipinfo 文件的器件段具有多个 BANKS 值。仅允许使用一个 BANKS 值。如果您并未手动编辑 chipinfo 文件，请联系 Microchip 技术支持来了解详情。

(819) duplicate ZEROREG for "*" in chipinfo file at line * *（汇编器）*

chipinfo 文件的器件段具有多个 ZEROREG 值。仅允许使用一个 ZEROREG 值。如果您并未手动编辑 chipinfo 文件，请联系 Microchip 技术支持来了解详情。

(820) duplicate SPAREBIT for "*" in chipinfo file at line * *（汇编器）*

chipinfo 文件的器件段具有多个 SPAREBIT 值。仅允许使用一个 SPAREBIT 值。如果您并未手动编辑 chipinfo 文件，请联系 Microchip 技术支持来了解详情。

(821) duplicate INTSAVE for "*" in chipinfo file at line * *（汇编器）*

chipinfo 文件的器件段具有多个 INTSAVE 值。仅允许使用一个 INTSAVE 值。如果您并未手动编辑 chipinfo 文件，请联系 Microchip 技术支持来了解详情。

(822) duplicate ROMSIZE for "*" in chipinfo file at line * *（汇编器）*

chipinfo 文件的器件段具有多个 ROMSIZE 值。仅允许使用一个 ROMSIZE 值。如果您并未手动编辑 chipinfo 文件，请联系 Microchip 技术支持来了解详情。

(823) duplicate START for "*" in chipinfo file at line * *（汇编器）*

chipinfo 文件的器件段具有多个 START 值。仅允许使用一个 START 值。如果您并未手动编辑 chipinfo 文件，请联系 Microchip 技术支持来了解详情。

(824) duplicate LIB for "*" in chipinfo file at line * *（汇编器）*

chipinfo 文件的器件段具有多个 LIB 值。仅允许使用一个 LIB 值。如果您并未手动编辑 chipinfo 文件，请联系 Microchip 技术支持来了解详情。

(825) too many RAMBANK lines in chipinfo file for "*" *（汇编器）*

chipinfo 文件包含的器件段具有太多的 RAMBANK 字段。请减少值的数量。

(826) inverted ram bank in chipinfo file at line * *（汇编器和驱动程序）*

在 chipinfo 文件的 RAM 字段中指定的第二个 HEX 数必须大于第一个的值。

(827) too many COMMON lines in chipinfo file for "*" *（汇编器）*

芯片配置文件中存在太多指定公共 （快速操作存储区）存储器的行。

(828) inverted common bank in chipinfo file at line * *（汇编器和驱动程序）*

在 chipinfo 文件的 COMMON 字段中指定的第二个 HEX 数必须大于第一个的值。如果您并未修改 chipinfo INI 文件，请联系 Microchip 技术支持。

(829) unrecognized line in chipinfo file at line * *（汇编器）*

chipinfo 文件包含的器件段具有无法识别的行。如果并未编辑 INI，请联系 Microchip 技术支持。

(830) missing ARCH specification for "*" in chipinfo file *（汇编器）*

chipinfo 文件的器件段不具有 ARCH 值。必须指定器件的架构。如果并未修改 chipinfo 文件，请联系 Microchip 技术支持。

(832) empty chip info file "*" *（汇编器）*

chipinfo 文件不包含任何数据。如果您并未手动编辑 chipinfo 文件，请联系 Microchip 技术支持来了解详情。

(833) no valid entries in chipinfo file *（汇编器）*

chipinfo 文件不包含任何有效的器件说明。

(834) page width must be >= 60 *（汇编器）*

列表页面宽度必须至少为 60 个字符。如果小于该宽度，将无法生成格式正确的列表，例如：

```
LIST C=10  ; the page width will need to be wider than this
```


© 2013 Microchip Technology Inc.

(835) form length must be >= 15 *（汇编器）*

使用 `-F` *length* 选项指定的页面长度必须至少为 15 行。允许将该长度设置为零，此时会完全关闭分页。默认值为零 （不分页）。

(836) no file arguments *（汇编器）*

汇编器在不带任何文件参数的情况下被调用。它无法汇编任何对象。

(839) relocation too complex *（汇编器）*

该表达式中的复杂重定位太大，无法插入目标文件。

(840) phase error *（汇编器）*

汇编器在两个不同阶段中对于某个符号计算得到不同的值。这可能是由于宏或条件汇编的诡异用法而导致的。

(841) bad source/destination for movfp/movpf instruction *（汇编器）*

使用 MOVFP/MOVPF 指令指定的绝对地址太大。

(842) bad bit number *（汇编器）*

位编号必须为介于范围 0-7 的绝对表达式。

(843) a macro name can't also be an EQU/SET symbol *（汇编器）*

发现某个 EQU 或 SET 符号与某个宏同名。这是不允许的。例如：

```
getval MACRO
  MOV  r0, r1
ENDM
getval EQU 55h  ; oops -- choose a different name to the macro
```

(844) lexical error *（汇编器）*

在输入中发现无法识别的字符或标记。

(845) symbol "*" defined more than once *（汇编器）*

在多个位置定义了该符号。如果某个符号在同一模块中多次定义，则汇编器会发出该错误，例如：

```
_next:
   MOVE   r0, #55
   MOVE   [r1], r0
_next:           ; oops -- choose a different name
```

如果符号（C 或汇编）在不同模块中多次定义，则链接器会发出该警告。错误消息中会给出模块的名称。请注意，在编译之后， C 标识符通常会在其名称前加上 *下划线*。

(846) relocation error *（汇编器）*

无法将两个可重定位量进行相加。可以将常量与一个可重定位值相加，而同一 psect 中的两个可重定位地址可进行相减。对于汇编器在汇编时必须知道其值的各个位置，必须使用绝对值。

(847) operand error *（汇编器）*

该操作码的操作数是无效的。关于该指令的操作数的正确形式，请查看汇编器参考手册。

(848) symbol has been declared EXTERN *（汇编器）*

某个汇编标号与一个已声明为 EXTERN 的符号使用了相同的名称。

(849) illegal instruction for this device *（汇编器）*

该器件不支持该指令。

(850) PAGESEL not usable with this device *（汇编器）*

PAGESEL 伪指令不能用于选定的器件。

(851) illegal destination *（汇编器）*

该指令的目标 （,f 或 ,w）不正确。

(852) radix must be from 2 - 16 *（汇编器）*

使用 RADIX 汇编器伪指令指定的基数必须介于 2（二进制）至 16（十六进制）的范围。

(853) invalid size for FNSIZE directive *（汇编器）*

FNSIZE 汇编器伪指令参数必须为正的常量。

(855) ORG argument must be a positive constant *（汇编器）*

ORG 汇编器伪指令的参数必须为正的常量，或者为某个设为等于正的常量的符号，例如：

```
ORG -10  /* this must a positive offset to the current psect */
```

(856) ALIGN argument must be a positive constant *（汇编器）*

align 汇编器伪指令需要一个非零的正整型参数。

(857) psect may not be local and global *（链接器）*

局部 psect 不能与全局 psect 同名，例如：

```
psect text,class=CODE       ; text is implicitly global
   MOVE   r0, r1
; elsewhere:
psect text,local,class=CODE
   MOVE   r2, r4
```

如果未显式说明作用域， global 标志是 psect 的默认值。

(859) argument to C option must specify a positive constant *（汇编器）*

LIST 汇编器控制的 C= 选项的参数 （它设置列表输出的列宽度）必须为正的十进制常量数字，例如：

```
LIST C=a0h  ; constant must be decimal and positive,
              try: LIST C=80
```

(860) page width must be >= 49 *（汇编器）*

LIST 汇编器伪指令的页面宽度子选项必须指定至少为 49 的宽度。

(861) argument to N option must specify a positive constant *（汇编器）*

LIST 汇编器控制的 N 选项的参数 （它设置列表输出的页面长度）必须为正的常量数字，例如：

```
LIST N=-3  ; page length must be positive
```


© 2013 Microchip Technology Inc.

(862) symbol is not external *（汇编器）*

某个符号已声明为 EXTERN，但也在当前模块中进行了定义。

(863) symbol can't be both extern and public *（汇编器）*

如果符号声明为外部符号，则需要导入它。如果它声明为公共符号，则需要从当前模块导出它。符号不能同时为这两种类型。

(864) argument to "size" psect flag must specify a positive constant *（汇编器）*

PSECT 汇编器伪指令的 size 选项的参数必须为正的常量数字，例如：

```
PSECT text,class=CODE,size=-200  ; a negative size?
```

(865) psect flag "size" redefined *（汇编器）*

PSECT 汇编器伪指令的 size 标志与先前 PSECT 伪指令不同，例如：

```
psect spdata,class=RAM,size=400
; elsewhere:
psect spdata,class=RAM,size=500
```

(866) argument to "reloc" psect flag must specify a positive constant *（汇编器）*

PSECT 汇编器伪指令的 reloc 选项的参数必须为正的常量数字，例如：

```
psect test,class=CODE,reloc=-4  ; the reloc must be positive
```

(867) psect flag "reloc" redefined *（汇编器）*

PSECT 汇编器伪指令的 reloc 标志与先前 PSECT 伪指令不同，例如：

```
psect spdata,class=RAM,reloc=4
; elsewhere:
psect spdata,class=RAM,reloc=8
```

(868) argument to "delta" psect flag must specify a positive constant *（汇编器）*

PSECT 汇编器伪指令的 DELTA 选项的参数必须为正的常量数字，例如：

```
PSECT text,class=CODE,delta=-2  ; negative delta value doesn't make
sense
```

(869) psect flag "delta" redefined *（汇编器）*

psect 的 “DELTA”选项在同一模块中多次重新定义。

(870) argument to "pad" psect flag must specify a positive constant *（汇编器）*

PSECT 汇编器伪指令的 “PAD”选项的参数必须为一个非零的正整数。

(871) argument to "space" psect flag must specify a positive constant *（汇编器）*

PSECT 汇编器伪指令的 space 选项的参数必须为正的常量数字，例如：

```
PSECT text,class=CODE,space=-1  ; space values start at zero
```

(872) psect flag "space" redefined *（汇编器）*

PSECT 汇编器伪指令的 space 标志与先前 PSECT 伪指令不同，例如：

```
psect spdata,class=RAM,space=0
; elsewhere:
psect spdata,class=RAM,space=1
```

(873) a psect may only be in one class *（汇编器）*

不能将某个 psect 分配给多个类。该 psect 在此处的定义与其他位置的定义不同。psect 的类通过一个标志使用以下形式指定：

```
psect text,class=CODE
```

请查找指定了一个不同类名的其他 psect 定义。

(874) a psect may only have one "with" option *（汇编器）*

psect 只能与（with）一个其他 psect 一起放置。psect 的 with 选项通过一个标志使用以下形式指定：

```
psect bss,with=data
```

请查找指定了一个不同 with psect 名称的其他 psect 定义。

(875) bad character constant in expression *（汇编器）*

字符常量要求仅包含一个字符，但发现它大于一个字符或根本无字符。特定于汇编器的示例：

```
MOV  r0, #’12’    ; ’12’ specifies two characters
```

(876) syntax error *（汇编器）*

检测到一个语法错误。它可能是由于很多原因导致的。

(877) yacc stack overflow *（汇编器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(878) -S option used: "*" ignored *（驱动程序）*

指定的汇编文件与 -S 选项一起被传递给驱动程序。驱动程序实际上不需要执行任何操作，因为该文件已经是一个汇编文件。

(880) invalid number of parameters. Use "* –HELP" for help *（驱动程序）*

编译器驱动程序的命令行用法不正确。

(881) setup succeeded *（驱动程序）*

编译器已使用 --setup 驱动程序选项成功安装。

(883) setup failed *（驱动程序）*

编译器未使用 --setup 驱动程序选项成功安装。请确保该选项的目录参数拼写正确、对于主机操作系统语法正确，并且它存在。

© 2013 Microchip Technology Inc.

(884) please ensure you have write permissions to the configuration file *（驱动程序）*

由于驱动程序无法访问 XML 配置文件，编译器未使用 `--setup` 驱动程序选项成功安装。请确保对该文件具有写权限。驱动程序将按顺序搜索以下配置文件：

- 环境变量 `XC_XML` 指定的文件
- 文件 `/etc/xc.xml`（如果目录 “`/etc`” 是可写的，并且主目录中没有 `.xc.xml` 文件）
- 主目录中的 `.xc.xml` 文件

如果无法找到以上任何文件，则会出现以上错误。

(889) this * compiler has expired *（驱动程序）*

该编译器的演示版使用期已结束。

(890) contact Microchip to purchase and re-activate this compiler *（驱动程序）*

该编译器演示版安装的评估期已到期。您需要购买编译器来重新激活它。但如果您确信评估期提早结束，请联系 Microchip 技术支持。

(891) can’t open psect usage map file "*": * *（驱动程序）*

驱动程序无法打开指定的文件。在使用驱动程序选项 `--summary=file` 时，驱动程序将生成 psect 使用情况映射文件。请确保该文件未在另一个应用程序中打开。

(892) can’t open memory usage map file "*": * *（驱动程序）*

驱动程序无法打开指定的文件。在使用驱动程序选项 `--summary=file` 时，驱动程序将生成存储器使用情况映射文件。请确保该文件未在另一个应用程序中打开。

(893) can’t open HEX usage map file "*": * *（驱动程序）*

驱动程序无法打开指定的文件。在使用驱动程序选项 `--summary=file` 时，驱动程序将生成 HEX 使用情况映射文件。请确保该文件未在另一个应用程序中打开。

(894) unknown source file type "*" *（驱动程序）*

无法确定指定输入文件的扩展名。驱动程序只能识别扩展名为 `as`、 `c`、 `obj`、 `usb`、 `p1`、 `lib` 或 `HEX` 的文件。

(895) can’t request and specify options in the one command *（驱动程序）*

驱动程序选项 `--getoption` 和 `--setoption` 的用法是互斥的。

(896) no memory ranges specified for data space *（驱动程序）*

未为指定器件的数据存储空间指定任何片上或外部存储器范围。

(897) no memory ranges specified for program space *（驱动程序）*

未为指定器件的程序存储空间指定任何片上或外部存储器范围。

© 2013 Microchip Technology Inc.

(899) can't open option file "*" for application "*": * *（驱动程序）*

无法打开 `--getoption` 或 `--setoption` 驱动程序选项所指定的选项文件。如果使用 `--setoption` 选项，请确保文件的名称拼写正确，并且文件存在。如果使用 `--getoption` 选项，请确保可以在给定位置创建该文件，或者它不由任何其他应用程序使用。

(900) exec failed: * *（驱动程序）*

无法执行所列出的子组件。文件是否存在？请尝试重新安装编译器。

(902) no chip name specified; use "* –CHIPINFO" to see available chip names *（驱动程序）*

驱动程序在未选择编译所针对的芯片的情况下被调用。使用 **–**CHIPINFO 选项运行驱动程序将显示所有芯片的列表，可以从中选择编译所针对的芯片。

(904) illegal format specified in "*" option *（驱动程序）*

该选项的用法不正确。请使用 **–**HELP 确认正确的用法，或参见介绍该选项的手册部分。

(905) illegal application specified in "*" option *（驱动程序）*

无法理解传递给该选项的应用程序或它不属于编译器。

(907) unknown memory space tag "*" in "*" option specification *（驱动程序）*

该存储器选项的参数是一个字符串，但它不匹配任何有效的*标记*。要确定哪些标记（如果有）对于该器件是有效的，请参见本手册介绍该选项的章节。

(908) exit status = * *（驱动程序）*

正在执行的子组件之一遇到问题，并返回了错误代码。该子组件应已报告了其他消息，以说明所遇到的问题。

(913) "*" option may cause compiler errors in some standard header files *（驱动程序）*

使用该选项将会使标准头文件中使用的一些限定符变为无效，从而导致错误。本手册中专门介绍该选项的章节中详细说明了该问题及其解决方案。

(915) no room for arguments *（预处理器、解析器、代码生成器、链接器和 Objtohex）*

代码生成器无法再分配更多的存储器。

(917) argument too long *（预处理器和解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(918) *: no match *（预处理器和解析器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(919) * in chipinfo file "*" at line * *（驱动程序）*

芯片配置文件中的指定参数是非法的。

(920) empty chipinfo file *（驱动程序和汇编器）*

芯片配置文件可以打开，但它为空。请尝试重新安装编译器。

© 2013 Microchip Technology Inc.

(922) chip "*" not present in chipinfo file "*" *（驱动程序）*

编译器的芯片配置文件中未出现选定的芯片。请联系 Microchip，以了解是否提供对于该器件的支持，或是否需要升级编译器的版本。

(923) unknown suboption "*" *（驱动程序）*

该选项可以接受子选项，但无法理解该子选项。这可能只是简单的拼写错误。如果不是，请使用 –HELP 来确定此处允许的子选项。

(924) missing argument to "*" option *（驱动程序）*

该选项要求更多的数据，但未提供任何数据。请检查该选项的用法。

(925) extraneous argument to "*" option *（驱动程序）*

该选项不接受附加的数据，但提供了附加的数据。请检查该选项的用法。

(926) duplicate "*" option *（驱动程序）*

该选项只能出现一次，但出现了多次。

(928) bad "*" option value *（驱动程序和汇编器）*

指定的选项要求一个有效的十六进制整型参数。

(929) bad "*" option ranges *（驱动程序）*

该选项要求一个采用范围格式 （*范围起始值* - *范围结束值*）的参数，但参数不符合该语法。

(930) bad "*" option specification *（驱动程序）*

该选项的参数未正确指定。使用 –HELP 运行驱动程序，或参见本手册中的驱动程序章节，以验证该选项的正确用法。

(931) command file not specified *（驱动程序）*

在命令行上未找到该应用程序的命令文件 （要求位于 “@”或 “<”之后）。

(939) no file arguments *（驱动程序）*

驱动程序在未在其命令行上列出输入文件的情况下被调用。如果在通过第三方 IDE 进行编译时获得该消息，则可能由于该 IDE 无法验证要编译的源文件或要链接的目标文件，因而未在命令行上提供它们。

(940) *-bit checksum * placed at * *（Objtohex）*

显示所请求的校验和计算的结果。

(941) bad "*" assignment; USAGE: ** *（Hexmate）*

HEXMATE 的一个选项的用法不正确或不完整。按照消息给出的用法操作，并确保该选项的格式正确且完整。

(942) unexpected character on line * of file "*" *（Hexmate）*

文件包含对于此类文件无效的字符，该文件可能已损坏。例如， Intel HEX 文件要求仅包含十六进制数字的 ASCII 表示形式、冒号 （:）和行格式。存在任何其他字符都会导致该错误。

(944) data conflict at address *h between * and * *（Hexmate）*

Hexmate数据源请求将不同的数据存储到同一地址。要强制一个数据源覆盖另一个数据源，请使用 “+”说明符。如果发生冲突的两个指定数据源是相同数据源，则说明数据源可能包含错误。

(945) checksum range (*h to *h) contained an indeterminate value *（Hexmate）*

该校验和计算的范围包含一个无法解析的值。如果校验和结果要存储在校验和计算的地址范围内，则会发生这种情况。

(948) checksum result width must be between 1 and 4 bytes *（Hexmate）*

所请求的校验和字节长度是非法的。校验和结果必须为1至4个字节宽。请检查-CKSUM选项的参数。

(949) start of checksum range must be less than end of range *（Hexmate）*

在传递给 -CKSUM 选项的范围中，起始值大于结束值。参数可能不完整或输入顺序不对。

(951) start of fill range must be less than end of range *（Hexmate）*

在传递给 -FILL 选项的范围中，起始值大于结束值。参数可能不完整或输入顺序不对。

(953) unknown -HELP sub-option: * *（Hexmate）*

传递给 -HELP 的子选项是无效的。请检查子选项的拼写或使用不带子选项的 -HELP 列出所有选项。

(956) -SERIAL value must be between 1 and * bytes long *（Hexmate）*

所存储的序列号超出范围。请确保序列号可以存储在该选项允许的字节数内。

(958) too many input files specified; * file maximum *（Hexmate）*

使用的文件参数太多。请尝试分几个阶段而不是在一条命令中合并这些文件。

(960) unexpected record type (*) on line * of "*" *（Hexmate）*

Intel HEX文件包含无效的记录类型。关于有效的记录类型，请参见Intel HEX格式规范。

(962) forced data conflict at address *h between * and * *（Hexmate）*

HEXMATE 数据源强制将不同的数据存储到同一地址。多个使用“+”说明符的数据源在相同地址处存储数据。存储在其中的实际数据可能不是您预期的数据。

(963) checksum range includes voids or unspecified memory locations *（Hexmate）*

校验和范围内的数据内容中存在间隙。由于要进行校验和计算的地址范围内存在间隙 / 未用字节，运行时计算的校验和可能会与编译时校验和不同。使用已知值填充未用单元可以更正该问题。

(964) unpaired nibble in -FILL value will be truncated *（Hexmate）*

为 FILL 选项提供的十六进制代码包含不完整的字节。将忽略不完整的字节（半字节）。

© 2013 Microchip Technology Inc.

(965) -STRPACK option not yet implemented, option will be ignored *（Hexmate）*

该选项目前不可用，将被忽略。

(966) no END record for HEX file "*" *（Hexmate）*

Intel HEX 文件中不包含 END 类型的记录。该 HEX 文件可能不完整。

(967) unused function definition "*" (from line *) *（解析器）*

正在编译的模块中永远不会调用所指示的静态函数。由于为静态类型，不能从其他模块调用该函数，所以该警告意味着永远不会使用该函数。该函数或者是多余的，或者原先要调用它的代码已在编译中被排除或函数名存在拼写错误。

(968) unterminated string *（汇编器）*

字符串常量似乎缺少右引号。

(969) end of string in format specifier *（解析器）*

`printf()` 样式函数的格式说明符的格式不正确。

(970) character not valid at this point in format specifier *（解析器）*

`printf()` 样式的格式说明符中具有非法字符。

(971) type modifiers not valid with this format *（解析器）*

类型修饰符不能用于该格式。

(972) only modifiers "h" and "l" valid with this format *（解析器）*

只有修饰符 `h`（`short`）和 `l`（`long`）对于该 `printf` 格式说明符是合法的。

(973) only modifier "l" valid with this format *（解析器）*

对于该格式惟一合法的修饰符为 `l`（long）。

(974) type modifier already specified *（解析器）*

该类型中已经指定了该类型修饰符。

(975) invalid format specifier or type modifier *（解析器）*

`printf` 样式字符串中的格式说明符或修饰符对于该特定格式是非法的。

(976) field width not valid at this point *（解析器）*

在 `printf()` 类型格式说明符中的此处不能出现字段宽度。

(978) this identifier is already an enum tag *（解析器）*

跟随在 `struct` 或 `union` 关键字后面的该标识符已经是某个枚举类型的标记，因此只能跟随在关键字 `enum` 之后，例如：

```
enum IN {ONE=1, TWO};
struct IN {             /* oops -- IN is already defined */
  int a, b;
};
```

(979) this identifier is already a struct tag *（解析器）*

跟随在 union 或 enum 关键字后面的该标识符已经是某个结构的标记，因此只能跟随在关键字 struct 之后，例如：

```
struct IN {
  int a, b;
};
enum IN {ONE=1, TWO};  /* oops -- IN is already defined */
```

(980) this identifier is already a union tag *（解析器）*

跟随在 struct 或 enum 关键字后面的该标识符已经是某个联合的标记，因此只能跟随在关键字 union 之后，例如：

```
union IN {
  int a, b;
};
enum IN {ONE=1, TWO};  /* oops -- IN is already defined */
```

(981) pointer required *（解析器）*

此处需要一个指针，例如：

```
struct DATA data;
data->a = 9;      /* data is a structure,
                     not a pointer to a structure */
```

(982) unknown op "*" in nxtuse() *（汇编器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(983) storage class redeclared *（解析器）*

某个变量先前声明为 *static* 类型，现在被重新声明为 *extern* 类型。

(984) type redeclared *（解析器）*

重新声明了该函数或对象的类型。出现该错误可能是因为存在两个不兼容的声明，或因为隐式声明后面跟随一个不兼容的声明，例如：

```
int a;
char a;  /* oops -- what is the correct type? */
```

(985) qualifiers redeclared *（解析器）*

该函数或变量在不同声明中具有不同的限定符。

(986) enum member redeclared *（解析器）*

枚举的成员使用不同的值定义了两次或多次。该成员是否在同一个列表中出现两次，或者成员的名称是否在枚举列表中出现多次？

(987) arguments redeclared *（解析器）*

传递给该函数的参数的数据类型与其原型不匹配。

(988) number of arguments redeclared *（解析器）*

该函数声明中的参数数量与相同函数的先前声明不一致。

© 2013 Microchip Technology Inc.

(989) module has code below file base of *h *（链接器）*

该模块具有地址低于给定地址的代码，但使用 -C 选项指定要创建一个会映射到该地址的二进制输出文件。这意味着来自该模块的代码必须放置在该文件起始处之前！请检查汇编文件中是否缺少 psect 伪指令。

(990) modulus by zero in #if; zero result assumed *（预处理器）*

#if 表达式中的模运算具有零除数。结果已被假定为零，例如：

```
#define ZERO 0
#if FOO%ZERO    /* this will have an assumed result of 0 */
  #define INTERESTING
#endif
```

(991) integer expression required *（解析器）*

在 enum 声明中，可以为成员赋值，但表达式必须求值为 int 类型的常量，例如：

```
enum {one = 1, two, about_three = 3.12};
  /* no non-int values allowed */
```

(992) can’t find op *（汇编器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(993) some command-line options are disabled *（驱动程序）*

编译器正在演示模式下工作。一些命令行选项会被禁止。

(994) some command-line options are disabled and compilation is delayed *（驱动程序）*

编译器正在演示模式下工作。一些命令行选项会被禁止，编译速度会较慢。

(995) some command-line options are disabled, code size is limited to 16kB, compilation is delayed *（驱动程序）*

编译器正在演示模式下工作。一些命令行选项会被禁止；编译速度会较慢，并且允许的最大代码长度被限制为 16 KB。

消息 1000-1249

(1015) missing "*" specification in chipinfo file "*" at line * *（驱动程序）*

该属性要求至少出现一次，但未为该芯片定义。

(1016) missing argument* to "*" specification in chipinfo file "*" at line * *（驱动程序）*

在芯片配置文件中，该属性的该值为空。

(1017) extraneous argument* to "*" specification in chipinfo file "*" at line * *（驱动程序）*

在芯片配置文件中，所列出的规范具有太多的属性。

(1018) illegal number of "*" specification* (* found; * expected) in chipinfo file "*" at line * *（驱动程序）*

该属性要求出现一定次数；但是，对于该芯片它未出现。

(1019) duplicate "*" specification in chipinfo file "*" at line * *（驱动程序）*

该属性只能定义一次，但对于该芯片定义了多次。

(1020) unknown attribute "*" in chipinfo file "*" at line * *（驱动程序）*

芯片配置文件包含一个该版本的编译器无法理解的属性。是否芯片配置文件或驱动程序已被替换为该编译器另一个版本的等效组件？

(1021) syntax error reading "*" value in chipinfo file "*" at line * *（驱动程序）*

芯片配置文件错误地为该器件定义了指定的值。如果要自己修改该文件，请小心操作，并参见该文件开始处的注释来了解关于此处要求哪种类型的值的说明。

(1022) syntax error reading "*" range in chipinfo file "*" at line * *（驱动程序）*

芯片配置文件错误地为该器件定义了指定的范围。如果要自己修改该文件，请小心操作，并参见该文件开始处的注释来了解关于此处要求哪种类型的值的说明。

(1024) syntax error in chipinfo file "*" at line * *（驱动程序）*

芯片配置文件中的指定行处包含语法错误。

(1025) unknown architecture in chipinfo file "*" at line * *（驱动程序）*

所指示行中的属性定义了对于该编译器未知的架构。

(1026) missing architecture in chipinfo file "*" at line * *（汇编器）*

chipinfo 文件的器件段不具有 ARCH 值。必须指定器件的架构。如果并未修改 chipinfo 文件，请联系 Microchip 技术支持。

(1027) activation was successful *（驱动程序）*

编译器已成功激活。

(1028) activation was not successful - error code (*) *（驱动程序）*

编译器未成功激活。

© 2013 Microchip Technology Inc.

(1029) compiler not installed correctly - error code (*) *（驱动程序）*

该编译器未找到任何激活信息，无法继续执行。编译器可能未正确或未完全安装。引用的错误代码可以帮助诊断该失败的原因。如果联系 Microchip 获取有关该问题的帮助，可能会要求您提供该故障代码。

(1030) Hexmate - Intel HEX editing utility (Build 1.%i) *（Hexmate）*

指示正在执行的 HEXMATE 的版本号。

(1031) USAGE: * [input1.HEX] [input2.HEX]... [inputN.HEX] [options] *（Hexmate）*

HEXMATE 的建议用法。

(1032) use –HELP=<option> for usage of these command line options *（Hexmate）*

通过向 HELP 选项传递某个特定选项，可以获取关于该选项的更详细信息。

(1033) available command-line options: *（Hexmate）*

这是出现在该应用程序的可用选项列表之前的简单标题。

(1034) type "*" for available options *（Hexmate）*

您似乎需要帮助。该建议性消息建议如何获取关于该应用程序可用选项或这些选项用法的更多信息。

(1035) bad argument count (*) *（解析器）*

某个函数的参数数量是不合理的。这是内部编译器错误。详情请联系Microchip技术支持。

(1036) bad "*" optional header length (0x* expected) *（Cromwell）*

该 COFF 文件中的可选头部的长度不正确。

(1037) short read on * *（Cromwell）*

在读取该消息中所指示类型的数据时，它在达到其指定长度之前终止。

(1038) string table length too short *（Cromwell）*

COFF 字符串表的指定长度小于最小值。

(1039) inconsistent symbol count *（Cromwell）*

符号表中的符号数量超出了 COFF 头部中指示的数量。

(1040) bad checksum: record 0x*, checksum 0x* *（Cromwell）*

指定类型的记录无法与它自己的校验和值匹配。

(1041) short record *（Cromwell）*

在读取文件时，文件中的一个记录在其指定长度之前结束。

(1042) unknown * record type 0x* *（Cromwell）*

该记录的类型指示符与该文件格式的所有有效类型都不匹配。

(1043) unknown optional header *（Cromwell）*

读取该 Microchip COFF 文件时，文件头部中的可选头部的长度不正确。

(1044) end of file encountered *（Cromwell 和链接器）*

在需要更多数据时发现文件结束。该输入文件是否已被截断？

(1045) short read on block of * bytes *（Cromwell）*

从 UBROF 记录中读取字节数据块时，块在预期长度之前结束。

(1046) short string read *（Cromwell）*

从 UBROF 记录中读取字符串时，字符串在指定长度之前结束。

(1047) bad type byte for UBROF file *（Cromwell）*

该 UBROF 文件未以正确的记录开始。

(1048) bad time/date stamp *（Cromwell）*

该 UBROF 文件具有错误的时间 / 日期标记。

(1049) wrong CRC on 0x* bytes; should be * *（Cromwell）*

该 UBROF 文件中的一个结束记录具有一个不匹配的 CRC 值。

(1050) bad date in 0x52 record *（Cromwell）*

该 UBROF 文件中的一个调试记录具有错误的日期组成部分。

(1051) bad date in 0x01 record *（Cromwell）*

该 UBROF 文件中的程序起始记录或段记录具有错误的日期组成部分。

(1052) unknown record type *（Cromwell）*

读取该 UBROF 文件时，无法确定某个记录类型。

(1053) additional RAM ranges larger than bank size *（驱动程序）*

所请求的附加 RAM 块超出存储区的大小。请尝试将块拆分为不跨越存储区边界的多个范围。

(1054) additional RAM range out of bounds *（驱动程序）*

通过定制 RAM 配置定义的 RAM 存储器范围超出范围。

(1055) RAM range out of bounds (*) *（驱动程序）*

在芯片配置文件中或通过定制配置定义的 RAM 存储器范围超出范围。

(1056) unknown chip architecture *（驱动程序）*

编译器试图针对属于不受支持或已被禁止的架构的器件进行编译。

(1057) fast double option only available on 17 series processors *（驱动程序）*

不能为该器件选择快速双精度库。这些程序仅对于 PIC17 器件可用。

(1058) assertion *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

© 2013 Microchip Technology Inc.

(1059) rewrite loop *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(1081) static initialization of persistent variable "*" *（解析器和代码生成器）*

持久性变量被赋予初始值。这是有些矛盾的，因为初始值将在编译器启动代码执行过程中赋予变量，但 *persistent* 限定符要求编译器启动代码不应更改该变量。

(1082) size of initialized array element is zero *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(1088) function pointer "*" is used but never assigned a value *（代码生成器）*

进行了涉及到函数指针的函数调用，但该指针从未被赋予目标地址，例如：

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to? */
```

(1089) recursive function call to "*" *（代码生成器）*

发现对于指定函数的递归调用。调用可能是直接或间接的（使用函数指针），可能是函数调用它自身，或者是函数调用另一个函数，但另一个函数的调用图中包含指定的函数。

(1090) variable "*" is not used *（代码生成器）*

声明了该变量，但程序未使用它。可以考虑从程序中删除它。

(1091) main function "*" not defined *（代码生成器）*

未定义 *main* 函数。每个 C 程序都必须具有名为 *main* 的函数。

(1094) bad derived type *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(1095) bad call to typeSub() *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(1096) type should be unqualified *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(1097) unknown type string "*" *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(1098) conflicting declarations for variable "*" (*:*) *（解析器和代码生成器）*

在某个变量的声明中或在某个变量的声明和定义之间检测到不同的类型信息，例如：

```
extern long int test;
int test;    /* oops -- which is right? int or long int ? */
```

(1104) unqualified error *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(1118) bad string "*" in getexpr(J) *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(1119) bad string "*" in getexpr(LRN) *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(1121) expression error *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(1137) match() error: * *（代码生成器）*

这是内部编译器错误。详情请联系 Microchip 技术支持。

(1157) W register must be W9 *（汇编器）*

此处所需的工作寄存器必须为 W9，但选择了其他工作寄存器。

(1159) W register must be W11 *（汇编器）*

此处所需的工作寄存器必须为 W11，但选择了其他工作寄存器。

(1178) the "*" option has been removed and has no effect *（驱动程序）*

该选项在该版本的编译器中不再存在，已被忽略。使用编译器的 *–help* 选项或参见手册来查找替换选项。

(1179) interrupt level for function "*" may not exceed * *（代码生成器）*

指定函数的中断优先级太高。每个中断函数都会被分配一个惟一的中断优先级。在分析调用图和可重入调用函数时，将会考虑该优先级。如果使用 `interrupt_level` pragma，请检查指定的值。

(1180) directory "*" does not exist *（驱动程序）*

在安装选项中指定的目录不存在。创建目录并重试。

(1182) near variables must be global or static *（代码生成器）*

使用 *near* 限定的变量必须同时使用 *static* 限定或设为全局变量。自动变量不能限定为 *near*。

(1183) invalid version number *（激活）*

在激活期间，未在 Microchip 激活服务器数据库中找到指定序列号的匹配版本号。

(1184) activation limit reached *（激活）*

指定序列号的激活数量已超出许可证所允许的最大数量。

(1185) invalid serial number *（激活）*

在激活期间，未在 Microchip 激活服务器数据库中找到匹配的序列号。

(1186) license has expired *（驱动程序）*

该编译器的限时许可证已到期。

© 2013 Microchip Technology Inc.

(1187) invalid activation request *（驱动程序）*

编译器尚未正确激活。

(1188) network error * *（激活）*

编译器激活软件无法通过网络连接到 Microchip 激活服务器。

(1190) FAE license only - not for use in commercial applications *（驱动程序）*

指示该编译器已使用 FAE 许可证激活。该许可证不允许将产品用于开发商业应用程序。

(1191) licensed for educational use only *（驱动程序）*

指示该编译器已使用教育许可证激活。教育许可证仅供教育机构使用，不允许将产品用于开发商业应用程序。

(1192) licensed for evaluation purposes only *（驱动程序）*

指示该编译器已使用评估许可证激活。

(1193) this license will expire on * *（驱动程序）*

编译器以限时试用的形式安装。该试用将在指定日期结束。

(1195) invalid syntax for "*" option *（驱动程序）*

为可接受额外参数的命令行选项提供了不适当的数据或数据不足。例如，某个选项可能要求两个均为整型的参数。将字符串作为其中一个参数传递或仅提供一个参数都可能导致该错误。

(1198) too many "*" specifications; * maximum *（Hexmate）*

指定该选项的次数太多。如果可能，请尝试通过几个命令行来执行这些操作。

(1199) compiler has not been activated *（驱动程序）*

该编译器的试用期已到期。编译器现在无法工作，直到使用有效序列号激活为止。请联系 Microchip 购买本软件，并获取序列号。

(1200) Found %0*lXh at address *h *（Hexmate）*

在该地址发现在 -FIND 选项中指定的代码序列。

(1201) all FIND/REPLACE code specifications must be of equal width *（Hexmate）*

该选项中的所有查找、替换和掩码属性必须为相同的字节宽度。请检查提供给该选项的参数。例如，使用 FFh （1 字节）作为掩码来查找 1234h （2 字节）会产生错误；但是，使用 00FFh （2 字节）作为掩码则可以工作。

(1202) unknown format requested in -FORMAT: * *（Hexmate）*

请求了未知或不受支持的 INHX 格式。关于受支持的 INHX 格式，请参见文档。

(1203) unpaired nibble in * value will be truncated *（Hexmate）*

该选项的数据未以完整字节的形式输入。数据可能不完整或遗漏了前导零。例如，值 Fh 仅包含 4 个有效数据位，不是一个完整字节。值 0Fh 包含 8 个有效数据位，它是完整字节。

(1204) * value must be between 1 and * bytes long *（Hexmate）*

为该选项提供了非法长度的数据。为该选项提供的值超出了该选项所需的最大值或最小值界限。

(1205) using the configuration file *; you may override this with the environment variable HTC_XML *（驱动程序）*

这是在编译器安装期间选择的编译器配置文件。它可以通过 HTC_XML 环境变量更改。该文件用于确定编译器的安装路径。

(1207) some of the command line options you are using are now obsolete *（驱动程序）*

传递给驱动程序的一些命令行选项在该版本的编译器中现在已停用；但在宽限期内，驱动程序仍然会处理这些旧选项。

(1208) use –help option or refer to the user manual for option details *（驱动程序）*

检测到一个过时的选项。使用 –help 或参见手册来查找不会导致该建议性消息的替换选项。

(1209) An old MPLAB tool suite plug-in was detected. *（驱动程序）*

传递给驱动程序的选项类似于 Microchip MPLAB IDE 将传递给该编译器的先前版本的那些选项。其中某些选项现在已过时；但是，仍然会解释它们。建议您安装针对 MPLAB IDE 的更新 Microchip 选项插件。

(1210) Visit the Microchip website (www.microchip.com) for a possible upgrade *（驱动程序）*

请访问我们的网站，查看是否有可用于解决先前编译器消息中列出的问题的升级。导航至 MPLAB XC8 C 编译器页面，并查找版本升级可下载文件。如果您的版本是最新的，请联系 Microchip 技术支持来了解进一步的信息。

(1212) Found * (%0*lXh) at address *h *（Hexmate）*

在该地址发现在 -FIND 选项中指定的代码序列。

(1213) duplicate ARCH for * in chipinfo file at line * *（汇编器和驱动程序）*

chipinfo 文件的器件段具有多个 ARCH 值。仅允许使用一个 ARCH 值。如果您并未手动编辑 chipinfo 文件，请联系 Microchip 技术支持来了解详情。

(1218) can’t create cross reference file * *（汇编器）*

汇编器尝试创建交叉引用文件；但是，无法创建它。请检查该文件的路径名是否正确。

(1228) unable to locate installation directory *（驱动程序）*

编译器无法确定它的安装目录。

© 2013 Microchip Technology Inc.

(1230) dereferencing uninitialized pointer "*" *（代码生成器）*

对尚未赋值的指针进行解引用。这会在运行时导致错误的行为。

(1235) unknown keyword * *（驱动程序）*

无法识别 USB 描述符文件中包含的标记。

(1236) invalid argument to *: * *（驱动程序）*

为可以接受额外参数的选项提供了无效的参数值。请检查该选项的用法或预期参数的语法或范围。

(1237) endpoint 0 is pre-defined *（驱动程序）*

试图在 USB 文件中定义端点 0。

(1238) FNALIGN failure on * *（链接器）*

两个函数使用 FNALIGN 伪指令对齐它们的自动 / 参数块，但一个函数会调用另一个，这意味着它们不能进行对齐。如果某个函数指针被赋予每个函数的地址，但一个函数会调用另一个函数，则会发生该错误。例如：

```
int one(int a) { return a; }
int two(int a) { return two(a)+2; }  /* ! */
int (*ip)(int);
ip = one;
ip(23);
ip = two;      /* ip references one and two; two calls one */
ip(67);
```

(1239) pointer * has no valid targets *（代码生成器）*

进行了涉及到函数指针的函数调用，但该指针从未被赋予目标地址，例如：

```
void (*fp)(int);
fp(23);     /* oops -- what function does fp point to? */
```

(1240) unknown checksum algorithm type (%i) *（驱动程序）*

无法打开在 -Efile 或 -E+file 选项之后指定的错误文件。请进行检查，以确保文件或目录是有效的，并具有只读访问权。

(1241) bad start address in * *（驱动程序）*

无法读取 --CHECKSUM 选项的范围起始地址。该值必须为一个十六进制数。

(1242) bad end address in * *（驱动程序）*

无法读取 --CHECKSUM 选项的范围结束地址。该值必须为一个十六进制数。

(1243) bad destination address in * *（驱动程序）*

无法读取 --CHECKSUM 选项的目标地址。该值必须为一个十六进制数。

(1245) value greater than zero required for * *（Hexmate）*

HEXMATE -FIND 选项的 *align* 操作数必须为正数。

(1246) no RAM defined for variable placement *（代码生成器）*

未指定任何涵盖分区 RAM 存储器的存储器。

(1247) no access RAM defined for variable placement *（代码生成器）*

未指定任何涵盖快速操作存储区的存储器。

(1248) symbol (*) encountered with undefined type size *（代码生成器）*

代码生成器被要求定位一个变量，但该变量的长度是未知的。这是内部编译器错误。详情请联系 Microchip 技术支持。

消息 1250-1499

(1250) could not find space (* byte*) for variable * *（代码生成器）*

代码生成器无法在分区 RAM 中找到用于指定变量的空间。

(1253) could not find space (* byte*) for auto/param block *（代码生成器）*

代码生成器无法在 RAM 中找到用于存放自动和参数变量的 psect 的空间。

(1254) could not find space (* byte*) for data block *（代码生成器）*

代码生成器无法在 RAM 中找到用于存放已初始化变量的 data psect 的空间。

(1255) conflicting paths for output directory *（驱动程序）*

通过 `-O` 或 `--OUTDIR` 选项为编译器的输出目录提供了相互矛盾的路径，例如：

```
--outdir=../../   -o../main.HEX
```

(1256) undefined symbol "*" treated as HEX constant *（汇编器）*

可解释为符号或十六进制值的标记与先前定义的所有符号都不匹配，所以将被解释为后者。使用前导零来避免歧义，或使用备用的基数说明符，例如 `0x`。例如：

```
MOV  a, F7h  ; is this the symbol F7h, or the HEX number 0xF7?
```

(1257) local variable "*" is used but never given a value *（代码生成器）*

定义了某个自动变量并将其用在表达式中，但它未在第一次使用之前在 C 代码中赋值。自动变量不会在启动时清零，其初始值是未定义的。例如：

```
void main(void) {
  double src, out;
  out = sin(src);   /* oops -- what value was in src? */
```

(1258) possible stack overflow when calling function "*" *（代码生成器）*

代码生成器进行调用树分析表明硬件堆栈可能发生溢出。这应仅视为一种指示信息。中断、汇编器优化器和程序结构都可能会影响堆栈的使用情况。堆栈使用情况基于 C 程序，并不包括基于汇编代码产生的任何调用树。

(1259) can’t optimize for both speed and space *（驱动程序）*

为驱动程序提供了针对速度编译和针对空间编译的相互矛盾的选项，例如：

```
--opt=speed,space
```


© 2013 Microchip Technology Inc.

(1260) macro "*" redefined *（汇编器）*

遇到了多个同名的宏定义，例如：

```
MACRO fin
  ret
ENDM
MACRO fin   ; oops -- was this meant to be a different macro?
  reti
ENDM
```

(1261) string constant required *（汇编器）*

DS 或 DSU 伪指令需要一个字符串参数，例如：

```
DS ONE     ; oops -- did you mean DS "ONE"?
```

(1262) object "*" lies outside available * space *（代码生成器）*

某个绝对变量被定位到某个存储单元，但该存储单元不处于针对目标器件定义的存储器范围内，例如：

```
int data @ 0x800     /* oops -- is this the correct address? */
```

(1264) unsafe pointer conversion *（代码生成器）*

指向一种结构的指针被转换为另一种结构的指针，但两种结构并不具有类似的定义，例如：

```
struct ONE {
  unsigned a;
  long b;          /* ! */
} one;
struct TWO {
  unsigned a;
  unsigned b;      /* ! */
} two;
struct ONE * oneptr;
oneptr = & two;    /* oops --
                    was ONE meant to be same struct as TWO? */
```

(1267) fixup overflow referencing * into * bytes at 0x* *（链接器）*

更多信息，请参见错误消息 (477)。

(1268) fixup overflow storing 0x* in * bytes at * *（链接器）*

更多信息，请参见错误消息 (477)。

(1273) Omniscient Code Generation not available in Free mode *（驱动程序）*

该消息指示在该免费模式编译器下不会使能编译器的高级功能。

(1275) only functions may be qualified "*" *（解析器）*

仅在函数定义中使用时才有意义的限定符被用于变量定义。

```
interrupt int dacResult;  /* oops --
            the interrupt qualifier can only be used with functions */
```

(1276) buffer overflow in DWARF location list *（Cromwell）*

与 ELF/DWARF 调试文件关联的缓冲区发生溢出。详情请联系 Microchip 技术支持。

(1278) omitting "*" which does not have a location *（Cromwell）*

变量未列出存储单元，将从调试输出中省略。详情请联系 Microchip 技术支持。

(1284) malformed mapfile while generating summary: CLASS expected but not found *（驱动程序）*

读取以生成存储器摘要的映射文件的格式不正确。该文件已被编辑或已损坏，或者这是编译器错误——详情请联系 Microchip 技术支持。

(1285) malformed mapfile while generating summary: no name at position * *（驱动程序）*

读取以生成存储器摘要的映射文件的格式不正确。该文件已被编辑或已损坏，或者这是编译器错误——详情请联系 Microchip 技术支持。

(1286) malformed mapfile while generating summary: no link address at position * *（驱动程序）*

读取以生成存储器摘要的映射文件的格式不正确。该文件已被编辑或已损坏，或者这是编译器错误——详情请联系 Microchip 技术支持。

(1287) malformed mapfile while generating summary: no load address at position * *（驱动程序）*

读取以生成存储器摘要的映射文件的格式不正确。该文件已被编辑或已损坏，或者这是编译器错误——详情请联系 Microchip 技术支持。

(1288) malformed mapfile while generating summary: no length at position * *（驱动程序）*

读取以生成存储器摘要的映射文件的格式不正确。该文件已被编辑或已损坏，或者这是编译器错误——详情请联系 Microchip 技术支持。

(1289) line range limit exceeded, debugging may be affected *（Cromwell）*

某个 C 语句产生的汇编代码输出的长度超出了预设的限制。这意味着 CROMWELL 生成的调试信息可能不准确。该警告并不表明任何潜在的代码失败。

(1290) buffer overflow in DWARF debugging information entry *（Cromwell）*

与 ELF/DWARF 调试文件关联的缓冲区发生溢出。详情请联系 Microchip 技术支持。

(1291) bad ELF string table index *（Cromwell）*

传递给 CROMWELL 的 ELF 文件的格式不正确，无法使用。

(1292) malformed define in .SDB file * *（Cromwell）*

传递给 CROMWELL 的指定 SDB 文件的格式不正确，无法使用。

(1293) couldn’t find type for "*" in DWARF debugging information entry *（Cromwell）*

无法通过传递给 CROMWELL 的 SDB 文件确定符号的类型。该文件已被编辑或已损坏，或者这是编译器错误——详情请联系 Microchip 技术支持。

(1294) there is only one day left until this license expires *（驱动程序）*

编译器正在以演示模式运行，评估许可证在不足一天后将到期，到期后将无法再以专业模式运行。到期之后，编译器可以在免费模式下无限期地工作，但会产生较大的二进制输出。

© 2013 Microchip Technology Inc.

(1295) there are * days left until this license will expire *（驱动程序）*

编译器正在以演示模式运行，在评估许可证在指定时间内到期之后，将无法再以专业模式运行。到期之后，编译器可以在免费模式下无限期地工作，但会产生较大的二进制输出。

(1296) source file "*" conflicts with "*" *（驱动程序）*

编译器遇到多个具有相同基本名称的源文件。只有这些文件包含在不同目录中时，才会出现这种情况。由于编译器和 IDE 基于源文件的基本名称来设定中间文件的名称，而中间文件总是存储在相同位置，所以这种情况是非法的。请确保所有源文件的基本名称是惟一的。

(1297) option * not available in Free mode *（驱动程序）*

当编译器在免费模式下工作时，某些选项不可用。被禁止的选项通常与编译器的执行方式相关（如 `--GETOPTION` 和 `--SETOPTION`），它们不会控制与代码生成相关的编译器功能。

(1298) use of * outside macros is illegal *（汇编器）*

某些汇编器伪指令（如 `EXITM`）只能在宏定义内使用。

(1299) non-standard modifier "*" - use "*" instead *（解析器）*

使用了非标准的 `printf` 占位符修饰符。请改为使用指定的修饰符。例如，应使用标准 `hh` 修饰符而不是 `b` 来指示值应打印为 `char` 类型。

(1300) maximum number of program classes reached. List may be truncated *（Cromwell）*

在命令行上向 `CROMWELL` 传递了类名的列表。如果传入的类名的数量太大，将不会使用全部类名，调试信息可能会受到影响。

(1301) invalid ELF section header. Skipping *（Cromwell）*

`CROMWELL` 在 ELF 段头部中发现无效的段。将跳过该段。

(1302) could not find valid ELF output extension for this device *（Cromwell）*

扩展名不是针对该目标器件系列的。

(1303) invalid variable location detected: * - * *（Cromwell）*

无法通过 SDB 文件确定符号的位置。

(1304) unknown register name: "*" *（Cromwell）*

在 SDB 文件中，所指示符号的位置代表一个寄存器，但无法识别寄存器名称。

(1305) inconsistent storage class for variable: "*" *（Cromwell）*

无法识别 SDB 文件中所指示符号的存储类别。

(1306) inconsistent size (* vs *) for variable: "*" *（Cromwell）*

SDB 文件中所指示符号的长度与其类型的长度不匹配。

(1307) psect * truncated to * bytes *（驱动程序）*

无法将代表堆栈或堆的psect设为所请求的大小，它将被截断，以装入可用的存储空间。

(1308) missing/conflicting interrupts sub-option, defaulting to "*" *（驱动程序）*

--INTERRUPT 选项的子选项缺失或格式不正确，例如：

```
--INTERRUPTS=single,multi
```

哎呀，您的本意是单向量还是多向量中断？

(1309) ignoring invalid runtime * sub-option (*) using default *（驱动程序）*

--RUNTIME 选项的指定子选项的格式不正确，例如：

```
--RUNTIME=default,speed:0y1234
```

哎呀，它应该为 0x1234。

(1310) specified speed (*Hz) exceeds max operating frequency (*Hz), defaulting to *Hz *（驱动程序）*

为 --RUNTIME 选项的 perform 子选项指定的频率对于选定器件太大。

```
--RUNTIME=default,speed:0xffffffff
```

哎呀，该值太大。

(1311) missing configuration setting for config word *, using default *（驱动程序）*

源代码中未提供指定字的配置设置，将使用默认值。

(1312) conflicting runtime perform sub-option and configuration word settings, assuming *Hz *（驱动程序）*

配置设置与使用 --RUNTIME 选项的 perform 子选项指定的值发生冲突，已选择了默认频率。

(1313) * sub-options ("*") ignored *（驱动程序）*

子选项的参数是不需要的，将被忽略。

```
--OUTPUT=intel:8
```

哎呀， :8 是不需要的。

(1314) illegal action in memory allocation *（代码生成器）*

这是内部错误。详情请联系 Microchip 技术支持。

(1315) undefined or empty class used to link psect * *（链接器）*

链接器被要求将某个 psect 放置在某个类指定的地址范围内，但该类从未定义或不包含任何存储器范围。

(1316) attribute "*" ignored *（解析器）*

遇到了一个有效但解析器未实现的属性。它将被解析器忽略，该属性将没有任何作用。详情请联系 Microchip 技术支持。

(1317) missing argument to attribute "*" *（解析器）*

遇到了一个需要参数的属性，但该参数不存在。详情请联系 Microchip 技术支持。

(1318) invalid argument to attribute "*" *（解析器）*

遇到了属性的参数，但它的格式不正确。详情请联系 Microchip 技术支持。

© 2013 Microchip Technology Inc.

(1319) invalid type "*" for attribute "*" *（解析器）*

这表示一个错误的选项被传递给解析器。详情请联系 Microchip 技术支持。

(1320) attribute "*" already exists *（解析器）*

这表示同一属性选项被多次传递给解析器。详情请联系 Microchip 技术支持。

(1321) bad attribute -T option "%s" *（解析器）*

传递给解析器的属性选项的格式不正确。详情请联系 Microchip 技术支持。

(1322) unknown qualifier "%s" given to -T *（解析器）*

在属性选项中指定的限定符是未知的。详情请联系 Microchip 技术支持。

(1323) attribute expected *（解析器）*

使用了 `__attribute__` 伪指令，但未指定属性类型。

```
int rv (int a) __attribute__(())  /* oops -- what is the attribute? */
```

(1324) qualifier "*" ignored *（解析器）*

一些限定符是有效的，但在一些编译器或目标器件上可能未实现。该警告指示将忽略该限定符。

(1325) no such CP* register: ($*), select (*) *（代码生成器）*

某个变量使用 cp0 限定，但在使用该变量指定的地址处不存在相应的协处理器寄存器。

```
cp0 volatile unsigned int mycpvar @ 0x7000;    /* oops --
                                          did you mean 0x700, try... */
cp0 volatile unsigned int mycpvar @ __REGADDR(7, 0);
```

(1326) "*" qualified variable (*) missing address *（代码生成器）*

某个变量使用 cp0 限定，但未指定协处理器寄存器地址。

```
cp0 volatile unsigned int mycpvar;  /* oops -- what address ? */
```

(1327) interrupt function "*" redefined by "*" *（代码生成器）*

所编写的中断函数被链接到已链接了中断函数的向量位置。

```
void interrupt timer1_isr(void) @ TIMER_1_VCTR { ... }
void interrupt timer2_isr(void) @ TIMER_1_VCTR { ... }    /* oops --
                                 did you mean that to be TIMER_2_VCTR */
```

(1328) coprocessor * registers can’t be accessed from * code *（代码生成器）*

所指示指令集中的代码非法地尝试访问协处理器寄存器。请确保使用正确的指令集对相应的函数进行编码。

(1329) can only modify RAM type interrupt vectors *（代码生成器）*

SETVECTOR( ) 宏被用于尝试更改中断向量表，但该表处于 ROM 中，不能在运行时更改。

(1330) only functions or function pointers may have an instruction set architecture qualifier *（代码生成器）*

指令集限定符被用于不代表可执行代码的内容。

```
mips16e int input;   /* oops -- you can’t qualify a variable with an
instruction set type */
```

(1331) interrupt functions may not be qualified "*" *（代码生成器）*

对中断函数使用了非法的函数限定符。

```
mips16e void interrupt tisr(void) @ CORE_TIMER_VCTR;   /* oops --
                  you can’t use mips16e with interrupt functions */
```

(1332) invalid qualifier (*) and type combination on "*" *（代码生成器）*

一些限定变量必须具有特定的类型或长度。检测到不允许的组合。

```
volatile cp0 int mycpvar @ __REGADDR(7,0);   /* oops --
                you must use unsigned types with the cp0 qualifier */
```

(1333) can’t extend instruction *（汇编器）*

尝试在指令不可扩展的位置扩展 MIPS16E 指令。这是内部错误。详情请联系 Microchip 技术支持。

(1334) invalid * register operand *（汇编器）*

汇编指令使用了非法的寄存器。这是内部错误或是由手写汇编代码导致的。

```
        psect my_text,isa=mips16e,reloc=4
        move t0,t1  /* oops -- these registers can’t be used in the
16-bit instruction set */
```

(1335) instruction "*" is deprecated *（汇编器）*

使用了已被弃用的汇编指令。

```
beql t0,t1,12  /* oops -- this instruction is no longer supported */
```

(1336) a psect may belong to only one ISA *（汇编器）*

psect具有一个定义所允许的指令集架构的标志。所定义的psect的ISA标志与相同psect的另一个定义的标志发生冲突。

```
mytext,global,isa=mips32r2,reloc=4,delta=1
mytext,global,isa=mips16e,reloc=4,delta=1   /* oops --
               is this the right psect name or the wrong ISA value */
```

(1337) instruction/macro "*" is not part of psect ISA *（汇编器）*

某个psect的ISA标志指定了一种架构类型，但在其中发现来自不同指令集架构的指令。

```
psect my_text,isa=mips16e,reloc=4
mtc0  t0,t1   /* oops -- this is a 32-bit instruction */
```

(1338) operand must be a * bit value *（汇编器）*

指令的常量操作数太大，无法放入指令字段宽度。

```
psect my_text,isa=mips32r2,reloc=4
li  t0,0x123456789  /* oops -- this constant is too large */
```


© 2013 Microchip Technology Inc.

(1339) operand must be a * bit * value *（汇编器）*

指令的常量操作数太大，无法放入指令字段宽度，并且必须具有指示的类型。

```
addiu   a3, a3, 0x123456   /* oops --
      the constant operand to this MIPS16E instruction is too large */
```

(1340) operand must be >= * and <= * *（汇编器）*

操作数必须处于指定的范围内。

```
ext t0,t1,50,3   /* oops -- third operand is too large */
```

(1341) pos+size must be > 0 and <= 32 *（汇编器）*

位域指令的长度和位置操作数之和必须为指定范围内的值。

```
ext t0,t1,50,3   /* oops -- 50 + 3 is too large */
```

(1342) whitespace after "\" *（预处理器）*

在*反斜杠*和*换行符*之间发现空格字符，它将被忽略。

(1343) hexfile data at address 0x* (0x*) overwritten with 0x* *（Objtohex）*

指定的地址将被附加的数据覆盖。这表明有多个代码段占用相同的地址。

(1346) can't find 0x* words for psect "*" in segment "*" (largest unused contiguous range 0x%lX) *（链接器）*

另请参见消息 (491)。新的消息形式还会指示链接器可以找到的最大可用块。除非存在一个足以容纳该psect的单独空间，否则链接器将发出该消息。在涉及到分区或分页时，最大的可用空间通常会远小于剩余的总空间。

(1347) can't find 0x* words (0x* withtotal) for psect "*" in segment "*" (largest unused contiguous range 0x%lX) *（链接器）*

另请参见消息 (593)。新的消息形式还会指示链接器可以找到的最大可用块。除非存在一个足以容纳该psect的单独空间，否则链接器将发出该消息。在涉及到分区或分页时，最大的可用空间通常会远小于剩余的总空间。

(1348) enum tag "*" redefined (from *:*) *（解析器）*

定义了多个具有相同名称的枚举标记，消息中会指示前一个定义。

```
enum VALS { ONE=1, TWO, THREE };
enum VALS { NINE=9, TEN };  /* oops -- is INPUT the right tag name? */
```

(1350) pointer operands to "-" must reference the same array *（代码生成器）*

如果要将两个地址相减，地址必须属于符合 ANSI 标准的相同对象。

```
int * ip;
int fred, buf[20];
ip = &buf[0] - &fred;  /* oops --
          second operand must be an address of a "buf" element */
```

(1352) truncation of operand value (0x*) to * bits *（汇编器）*

汇编指令的操作数太大，已被截断。

```
movlw  0x321  ; oops -- is this the right value?
```

(1354) ignoring configuration setting for unimplemented word * *（驱动程序）*

对于目标器件上不存在的字指定了配置字设置。

```
__CONFIG(3, 0x1234);  /* config word 3 does not exist on an 18C801 */
```

(1355) inline delay argument too large *（代码生成器）*

使用了内联延时序列 _delay，但所请求的指令周期数太大。请通过多次使用该程序来实现所需的延时长度。

```
#include <xc.h>
void main(void) {
  delay(0x400000); /* oops -- can’t delay by this number of cycles */
}
```

(1356) fixup overflow referencing * * (0x*) into * byte* at 0x*/0x* -> 0x* (* */0x*)** *（链接器）*

另请参见消息 (477)。该消息形式会预先计算引起问题的指令的地址，并将包含指令的 psect 的 delta 值考虑在内。

(1357) fixup overflow storing 0x* in * byte* at 0x*/0x* -> 0x* (* */0x*)** *（链接器）*

另请参见消息 (477)。该消息形式会预先计算引起问题的指令的地址，并将包含指令的 psect 的 delta 值考虑在内。

(1358) no space for * temps (*) *（代码生成器）*

代码生成器无法找到足以容纳该程序的临时变量 （中间结果暂存变量）的空间。

(1359) no space for * parameters *（代码生成器）*

代码生成器无法找到足以容纳特定函数的参数变量的空间。

(1360) no space for auto/param * *（代码生成器）*

代码生成器无法找到足以容纳特定函数的自动变量的空间。一些在寄存器中传递的参数可能也需要在该自动区域中分配空间。

(1361) syntax error in configuration argument *（解析器）*

#pragma config 的参数的格式不正确。

```
#pragma config WDT     /* oops -- is WDT on or off? */
```

(1362) configuration setting *=* redefined *（代码生成器）*

相同的 config pragma 设置被使用不同的值多次发出。

```
#pragma config WDT=OFF
#pragma config WDT=ON     /* oops -- is WDT on or off? */
```


© 2013 Microchip Technology Inc.

(1363) unknown configuration setting (* = *) used *（驱动程序）*

配置值和设置对于目标器件是未知的。

```
#pragma config WDR=ON     /* oops -- did you mean WDT? */
```

(1364) can’t open configuration registers data file * *（驱动程序）*

无法找到包含值配置设置的文件。

(1365) missing argument to pragma "varlocate" *（解析器）*

#pragma varlocate 的参数的格式不正确。

```
#pragma varlocate    /* oops -- what do you want to locate & where? */
```

(1366) syntax error in pragma "varlocate" *（解析器）*

#pragma varlocate 的参数的格式不正确。

```
#pragma varlocate fred     /* oops -- which bank for fred? */
```

(1367) end of file in _asm *（解析器）*

在 _asm _endasm 块内遇到了文件结束标记。

(1368) assembler message: * *（汇编器）*

所显示的是由汇编源文件中包含的 MESSG 伪指令产生的汇编器建议性消息。

(1369) can’t open proc file * *（驱动程序）*

无法打开选定器件的 proc 文件。

(1370) peripheral library support is not available for the * *（驱动程序）*

外设库对于选定器件不可用。

(1371) float type can’t be bigger than double type; double has been changed to * bits *（驱动程序）*

使用 --float 和 --double 选项导致 double 类型的长度小于 float 类型的长度。这是 C 标准不允许的。 double 类型的长度已被增大为所指示的长度。

(1372) interrupt level cannot be greater than * *（代码生成器）*

指定 interrupt_level 对于选定器件太高了。

```
#pragma interrupt_level 4
// oops - there aren't that many interrupts on this device
```

(1374) the compiler feature "*" is no longer supported, * *（驱动程序）*

编译器不再支持所指示的功能。

(1375) multiple interrupt functions (* and *) defined for device with only one interrupt vector *（代码生成器）*

所指定的两个函数都被限定为中断函数，但目标器件仅支持一个中断向量，因而仅支持一个中断函数。

```
interrupt void isr_lo(void) {
  // ...
}
interrupt void isr_hi(void) {   // oops, can’t define two ISRs
  // ...
}
```

(1376) initial value (*) too large for bitfield width (*) *（代码生成器）*

带位域的结构被定义为使用值进行初始化。指示的值太大，无法放入相应的位域宽度。

```
struct {
  unsigned flag :1;
  unsigned mode :3;
} foobar = { 1, 100 };   // oops, 100 is too large for a 3 bit object
```

(1377) no suitable strategy for this switch *（代码生成器）*

编译器无法基于代码和您使用#pragma switch伪指令作出的选择，确定用于对C switch语句进行编码的 switch 策略。您可能需要选择不同的策略。

(1378) syntax error in pragma "*" *（解析器）*

所指定 pragma 伪指令的参数是无效的。

```
#pragma addrqual ingore  // oops -- did you mean ignore?
```

(1379) no suitable strategy for this switch *（代码生成器）*

编译器根据一系列策略之一来编码 switch() 语句。 case 分支值的具体数量和值，以及 switch 表达式和 switch pragma 将决定所选择的策略。该错误指示没有任何策略可用来编码 switch() 语句。请联系 Microchip 支持来了解程序详情。

(1380) unable to use switch strategy "*" *（代码生成器）*

编译器根据一系列策略之一来编码 switch() 语句。 case 分支值的具体数量和值，以及 switch 表达式和 switch pragma 将决定所选择的策略。该错误指示无法使用所请求的策略来编码 switch() 语句。请联系 Microchip 支持来了解程序详情。

(1381) invalid case label range *（解析器）*

为 case 范围提供的值不正确。它们必须构成一个升序范围，并为整型常量。

```
case 0 ... -2:  // oops -- do you mean -2 ... 0  ?
```

(1385) * "*" is deprecated (declared at *:*) *（解析器）*

代码使用的某个变量或函数已被使用属性标记为弃用。

```
char __attribute__((deprecated)) foobar;
foobar = 9;     // oops -- this variable is near end-of-life
```

(1386) unable to determine the semantics of the configuration setting "*" for register "*" *（解析器和代码生成器）*

提供给某个配置位设置的数值与数据手册中指定的设置不存在直接关联。编译器将尝试遵从您的请求，但请查看您所使用器件的数据手册。

```
#pragma config OSC=11
// oops -- there is no direct association for that value on an 18F2520
// either use OSC=3 or OSC=RC
```


© 2013 Microchip Technology Inc.

(1387) inline delay argument must be constant *（代码生成器）*

__delay 内联函数只能接受一个常量表达式作为其参数。

```
int delay_val = 99;
__delay(delay_val);   // oops, argument must be a constant expression
```

(1388) configuration setting/register of "*" with 0x* will be truncated by 0x* *（解析器和代码生成器）*

配置位被编程的值对于该设置太大或不是规定值之一。

```
#pragma config WDTPS=138  // oops -- do you mean 128?
```

(1389) attempt to reprogram configuration * "*" with * (is *) *（解析器和代码生成器）*

某个已编程的配置位被再次使用与原始设置冲突的设置进行编程。

```
#pragma config WDT=ON
#pragma config WDT=OFF  // oops -- watchdog on or off?
```

(1390) identifier specifies insignificant characters beyond maximum identifier length *（解析器）*

使用的某个标识符太长，超出所设置的标识符长度。这可能意味着可能无法正确识别这些长标识符，代码将发生失败。使用 -N 选项可调整最大标识符长度。

```
int theValueOfThePortAfterTheModeBitsHaveBeenSet;
        // oops, make your symbol shorter or increase the maximum
        // identifier length
```

(1391) constant object size of * exceeds the maximum of * for this chip *（代码生成器）*

定义的 const 对象对于目标器件太大。

```
const int array[200] = { ... };  // oops -- not on a Baseline part!
```

(1392) function "*" is called indirectly from both mainline and interrupt code *（代码生成器）*

主干（非中断）和中断代码都调用了某个函数。如果发出该警告，它说明此类代码当前违反了选定器件的编译器限制。

(1393) possible hardware stack overflow detected, estimated stack depth: * *（代码生成器）*

编译器检测到某个程序的调用图使用的堆栈空间可能多于在目标器件上分配的空间。如果存在这种情况，代码可能会失败。编译器只能对涉及到中断时的堆栈使用情况作出假定，它们会产生堆栈使用的最坏情况估计。如果发出该警告，请确认是否存在函数调用嵌套。

(1394) attempting to create memory range (* - *) larger than page size, * *（驱动程序）*

编译器驱动程序检测到存储器设置包含大于器件页大小的程序存储器“页”。如果使用 --ROM 选项更改了默认存储器设置，很有可能会发生这种情况。请参见器件数据手册来确定所使用的器件的页大小，并确保使用 --ROM 选项指定的所有连续存储器范围具有的边界都对应于器件页边界。

```
--ROM=100-1fff
```

以上可能需要进行分页。如果页大小为 800h，以上可指定为

```
--ROM=100-7ff,800-fff,1000-17ff,1800-1fff
```

(1395) notable code sequence candidate suitable for compiler validation suite detected (*) *（代码生成器）*

编译器具有内置的检查功能，可以确定是否遇到了内部代码模板组合。如果在编译代码时发现了独特的组合，将发出该消息。该消息并不是错误或警告，显示它并不表示可能发生代码失败，但如果您愿意参与，可以将您正在编译的代码发送给技术支持，以协助进行编译器测试过程。

(1396) "*" positioned in the * memory region (0x* - 0x*) reserved by the compiler *（代码生成器）*

一些存储区是保留供编译器使用的。这些区域通常不用于分配在您的代码中定义的变量。但是，通过将变量设为绝对变量，可以将变量放入这些区域，并避免链接器通常情况下会发出的错误。（绝对变量可以放在任何位置，甚至放置在其他对象之上。）来自代码生成器的该警告指示已检测到一个会被定位到编译器的保留存储器中的绝对变量。您必须将该绝对变量定位到其他位置。该消息通常在将变量放置在公共存储空间中时发出。

```
char shared @ 0x7;   // oops, this memory is required by the compiler
```

(1397) unable to implement non-stack call to "*"; possible hardware stack overflow *（代码生成器）*

编译器必须在不使用 CALL 汇编指令和硬件堆栈（即，使用查找表）的情况下对某个 C 函数调用进行编码，但它无法实现。如果通过指针间接调用该函数，则可能需要一条调用指令，但如果硬件堆栈已满，则额外的调用会导致堆栈溢出。

(1401) eeprom qualified variables can’t be accessed from both interrupt and mainline code *（代码生成器）*

所有 eeprom 变量都通过不可重入的程序进行访问。如果尝试从中断和主干代码中访问使用 `eeprom` 限定的变量，代码可能会发生失败。请避免在中断函数中访问 eeprom 变量。

(1402) a pointer to eeprom can’t also point to other data types *（代码生成器）*

指针不能同时具有处于 eeprom 空间和普通数据空间中的目标。

(1403) pragma "*" ignored *（解析器）*

您指定的 pragma 伪指令没有任何作用，将被编译器忽略。该消息只会在 C18 兼容模式下发出。

```
#pragma varlocate "mySection" fred  // oops -- not accepted
```

(1404) unsupported: * *（解析器）*

`unsupported __attribute__` 被用于指示某个代码功能不受支持。

打印的消息将指示不受支持的功能，应避免使用它。

(1405) storage class specifier "*" ignored *（解析器）*

您指定的存储类别是不需要的，将被编译器忽略。该消息只会在 C18 兼容模式下发出。

```
int procInput(auto int inValue)  // oops -- no need for auto
{ ...
```


© 2013 Microchip Technology Inc.

(1406) auto eeprom variables are not supported *（代码生成器）*

限定为 eeprom 的变量不能为自动类型。如果需要，可以定义限定为 eeprom 的静态局部对象。

```
void main(void) {
    eeprom int mode;  // oops -- make this static or global
```

(1407) bit eeprom variables are not supported *（代码生成器）*

限定为 eeprom 的变量不能为 bit 类型。

```
eeprom bit myEEbit;    // oops -- you can’t define bits in EEPROM
```

(1408) ignoring initialization of far variables *（代码生成器）*

限定为 far 类型的变量不能赋予初始值。请在后面的代码中进行赋值。

```
far int chan = 0x1234; // oops -- you can’t assign a value here
```

(1409) warning number used with pragma "warning" is invalid *（解析器）*

用于警告 pragma 伪指令的消息编号小于零或大于可用的最高消息编号。

```
#pragma warning disable 1316 13350   // oops -- maybe number 1335?
```

(1410) can’t assign the result of an invalid function pointer *（代码生成器）*

编译器允许一些函数通过一个强制转换为函数指针的常量进行调用，但并不是所有函数。指定的地址对于该器件无效。

```
foobar += ((int (*)(int))0x0)(77);
        // oops -- you can’t call a function with a NULL pointer
```

(1411) Additional ROM range out of bounds *（驱动程序）*

使用 --ROM 选项指定的程序存储器位于该器件所支持的片上或外部存储器范围之外。

```
--ROM=default,+2000-2ffff
```

哎呀——存储地址太高了，是否应为 2fff?

(1412) missing argument to pragma "warning disable" *（解析器）*

#pragma warning disable 之后应为要禁止的消息编号的逗号分隔列表。

```
#pragma warning disable  // oops -- what messages are to be disabled?
```

请尝试以下形式的代码。

```
#pragma warning disable 1362
```

(1413) "*" is positioned at address 0x0 and has had its address taken; pointer comparisons may be invalid *（代码生成器）*

某个绝对对象被放置在地址 0 处，并已获取其地址。根据定义，这是一个 NULL 指针，检查指针是否为 NULL （即，检查以确定地址是否有效）的代码可能会失败。

```
int foobar @ 0x00;
int  * ip;

void
main(void)
{
   ip = &foobar;  // oops -- 0 is not a valid address
```

(1414) option * is defunct and has no effect *（驱动程序）*

不再支持所使用的选项。它将被忽略。

```
xc8 --chip=18f452 --cp=24 main.c
```

哎呀——不再需要 --cp 选项。

(1415) argument to "merge" psect flag must be 0 or 1 *（汇编器）*

该 psect 标志必须赋值为 0 或 1。

```
PSECT myTxt,class=CODE,merge=true  ; oops -- I think you mean merge=1
```

(1416) psect flag "merge" redefined *（汇编器）*

与先前 psect 同名的 psect 指定的 merge 标志值与先前的不同。

```
psect mytext,class=CODE,merge=1
; and later
psect mytext,class=CODE,merge=0
Oops, can mytext be merged or not?
```

(1417) argument to "split" psect flag must be 0 or 1 *（汇编器）*

该 psect 标志必须赋值为 0 或 1。

```
psect mytext,class=CODE,split=5
```

哎呀， split 标志参数必须为 0 或 1。

(1418) Attempt to read "control" qualified object which is Write-Only *（代码生成器）*

试图读取只写寄存器。

```
state = OPTION;  // oops -- you cannot read this register
```

(1419) using the configuration file *; you may override this with the environment variable XC_XML *（驱动程序）*

这是在编译器安装期间选择的编译器配置文件。它可以通过 XC_XML 环境变量更改。该文件用于确定编译器的安装路径。另请参见消息 1205。

(1420) ignoring suboption "*" *（驱动程序）*

您指定的子选项在该实现中无效，将被忽略。

```
--RUNTIME=default,+ramtest
```

哎呀——什么是 ramtest?

(1421) the qualifier __xdata is not supported by this architecture *（解析器）*

您指定的限定符在该实现中无效，将被忽略。

```
__xdata int coeff[2];  // that has no meaning for this target
```

(1422) the qualifier __ydata is not supported by this architecture *（解析器）*

您指定的限定符在该实现中无效，将被忽略。

```
__ydata int coeff[2];  // that has no meaning for this target
```

(1423) case ranges are not supported *（驱动程序）*

在 case 分支值中使用 GCC 样式的数值范围不符合 CCI 标准。请使用单独的 case 标号和值，以符合标准。

```
switch(input) {
case 0 ... 5:    // oops -- ranges of values are not supported
    low();
```


© 2013 Microchip Technology Inc.

(1424) short long integer types are not supported *（解析器）*

使用 short long 类型不符合 CCI 标准。请改为使用相应的 long 类型。

```
short long typeMod;    // oops -- not a valid type for CCI
```

(1425) __pack qualifier only applies to structures and structure members *（解析器）*

您指定的限定符仅在用于结构或结构成员时才有意义。它将被忽略。

```
__pack int c;  // oops -- there aren't inter-member spaces to pack in
an int
```

(1426) 24-bit floating point types are not supported; * have been changed to 32-bits *（驱动程序）*

浮点型必须为 32 位宽，以符合 CCI 标准。这些类型将被编译为 32 位宽的量。

```
--DOUBLE=24
```

哎呀——无法设置该双精度类型长度。

(1427) machine-dependent path specified in name of included file, use -I instead *（预处理器）*

头文件规范不能包含目录分隔符，以符合 CCI 标准。

```
#inlcude <inc\lcd.h>    // oops -- do not indicate directories here
```

请删除路径信息，并使用 -I 选项来指示它，例如：

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

并发出 -Ilcd 选项。

(1429) attribute "*" is not understood by the compiler; this attribute will be ignored *（解析器）*

您使用的所指示的属性对于该实现是无效的。它将被忽略。

```
int x __attribute__ ((deprecate)) = 0;
```

哎呀——您的本意是否是 deprecated?

(1430) section redefined from "*" to "*" *（解析器）*

您试图将一个对象放在多个段中。

```
int __section("foo") __section("bar") myvar;  // oops -- which section
should it be in?
```

(1431) only variable and function definitions at file-scope may be located using __section() *（解析器）*

不能尝试使用 __section() 说明符定位局部对象。

```
int main(void) {
    int __section("myData") counter;  // oops -- you cannot specify a
section for autos
```

(1432) "*" is not a valid section name *（解析器）*

使用 __section() 指定的段名不是有效的段名。段名必须符合正常的 C 标识符规则。

```
    int __section("28data") counter;  // oops -- name cannot start
with digits
```

(1433) function "*" could not be inlined *（汇编器）*

无法内联指定函数。将以正常方式调用该函数。

```
int inline getData(int port)   // sorry -- no luck inlining this
{   //...
```

(1434) missing name after pragma "intrinsic" *（解析器）*

instrinsic pragma 需要一个函数名。大多数情况下都不需要该 pragma 伪指令。如果您的本意是对函数进行内联，请参见 `inline` 关键字或 pragma 伪指令。

```
#pragma intrinsic    // oops -- what function is intrinsically called?
```

(0) delete what ? *（库管理器）*

使用 d 键时，库管理器要求列出要删除的一个或多个模块，例如：

```
libr d c:\ht-pic\lib\pic704-c.lib
```

未指示要删除的模块。请尝试使用以下形式：

```
libr d c:\ht-pic\lib\pic704-c.lib wdiv.obj
```

(0) incomplete ident record *（库管理器）*

目标文件中的 IDENT 记录不完整。详情请联系 Microchip 技术支持。

(0) incomplete symbol record *（库管理器）*

目标文件中的 SYM 记录不完整。详情请联系 Microchip 技术支持。

(0) library file names should have.lib extension: * *（库管理器）*

指定库文件名时，请使用 `.lib` 扩展名。

(0) module * defines no symbols *（库管理器）*

在模块的目标文件中未找到任何符号。这可能是所期望的，或者它可能意味着一部分代码被意外删除或被注释掉。

(0) replace what ? *（库管理器）*

使用 `r` 键时，库管理器要求列出要替换的一个或多个模块，例如：

```
libr r lcd.lib
```

该命令需要在库名称之后指定模块（`.obj` 文件）的名称。

© 2013 Microchip Technology Inc.

附录 C 实现定义的行为

本章介绍 MPLAB XC8 C 编译器的该实现中实现定义的行为。一些 C 代码的确切行为会因编译器而不同，C 语言的 ANSI 标准要求供应商用文档说明语言的实现定义特性的具体细节。

每一项之后的括号内的数字指示标准中与该项有关的章节编号。

C.1 翻译 （G.3.1）

C.1.1 如何标识一条诊断消息 （5.1.1.3）

诊断的格式可由用户完全控制。默认情况下，在命令行上进行编译时，将使用以下格式。在显示中总是会指示惟一的消息 ID 号。只有在消息是警告时，才会显示字符串 (warning)。

```
filename: function()
linenumber:source line
   ^ (ID) message (warning)
```

或

```
filename: linenumber: (ID) message (warning)
```

其中，*filename* 是包含代码的文件的名称（或如果没有特定的相关文件，则为空）；*linenumber* 是代码的行号（或如果没有相关的行号，则为 0）； *ID* 是标识该消息的惟一编号； *message* 是诊断消息本身。

C.2 环境 （G.3.2）

C.2.1 main 函数的参数的语义 （5.1.2.2.1）

函数 main 不具有任何参数，也不具有返回值。它遵循以下原型：

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

C.3 标识符 （G.3.3）

C.3.1 不带外部链接的标识符中的有效初始字符 （第 31 个字符之后）的数量 （6.1.2）

默认情况下，前 31 个字符是有效的。用户最高可调整为 255 个。

C.3.2 带外部链接的标识符中的有效初始字符 （第 6 个字符之后）的数量 （6.1.2）

默认情况下，前 31 个字符是有效的。用户最高可调整为 255 个。

C.3.3 在带有外部链接的标识符中，大小写区分是否有意义 （6.1.2）

所有标识符中的所有字符都区分大小写。

C.4 字符 （G.3.4）

C.4.1 标准中明确规定的除外，源代码字符集和执行字符集的成员 （5.2.1）

这两个字符集都与 ASCII 字符集相同。

C.4.2 用于多字节字符编码的移位状态 （5.2.1.2）

无移位状态。

C.4.3 执行字符集中字符的位数 （5.2.4.2.1）

字符具有 8 位。

C.4.4 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的映射 （6.1.3.4）

映射是恒等函数。

C.4.5 包含一个字符的整型字符常量的值或者不以基本执行字符集或宽字符常量的扩展字符集表示的转义序列的值 （6.1.3.4）

它是最右侧字符的数值。

C.4.6 包含多个字符的整型字符常量的值或包含多个多字节字符的宽字节常量的值 （3.1.3.4）

不支持。

C.4.7 普通 char 是与 signed char 还是与 unsigned char 具有相同的值范围 （6.2.1.1）

普通 `char` 被视为 `unsigned char`。

© 2013 Microchip Technology Inc.

C.5 整型 （G.3.5）

C.5.1 各种类型的整型的表示和值集 （6.1.2.5）

请参见**第 5.4.2 节 “整型数据类型”**。

C.5.2 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，或者无符号整型转换为同等长度的有符号整型的结果 （6.2.1.2）

原始值的低位将被复制到有符号整型中；或者，原始值所有低位被复制到有符号整型中。

C.5.3 有符号整型的按位运算的结果 （6.3）

按位运算将操作数视为无符号进行操作。

C.5.4 整型除法的余数的符号 （6.3.5）

余数的符号与被除数相同。表 C-1 列出了所有被除数和除数符号组合的除法运算结果的预期符号。

第二个操作数为零 （被零除）时，结果将总是为零。

表 C-1： 整型除法

被除数	除数	商	余数
+	+	+	+
-	+	-	-
+	-	-	+
-	-	+	-

C.5.5 负值有符号整型的右移操作的结果 （6.3.7）

右移操作符会对有符号值进行符号扩展。因此，具有 `signed int` 值 0x0124 的对象右移一位将产生值 0x0092，值 0x8024 右移一位将产生值 0xC012。

右移无符号整型值将总是清零结果的 MSb。

左移 （<< 操作符）有符号或无符号值将总是清零结果的 LSb。

C.6 浮点型 （G.3.6）

C.6.1 各种类型的浮点数的表示和值集 （6.1.2.5）

请参见**第 5.4.3 节 “浮点型数据类型”**。

C.6.2 当整数转换为不能精确地表示原始值的浮点数时的截断方向（6.2.1.3）

整数值将舍入为可以表示的最接近的浮点值。

C.6.3 当浮点数转换为较短的浮点数时的截断或舍入方向 （6.2.1.4）

浮点数将被截断为较短的浮点数。

C.7 数组和指针 （G.3.7）

C.7.1 存放数组的最大长度所需的整型类型；即，sizeof 操作符 size_t 的类型 （6.3.3.4 和 7.1.1）

`size_t` 的类型为 `unsigned int`。

C.7.2 将指针强制转换为整型或反之的结果 （6.3.4）

将某个整型强制转换为某个指针变量时，如果该指针变量在整个程序中仅被赋予数据存储器中对象的地址或仅被赋予程序存储器中对象的地址，则将不进行修改而将整型地址复制到指针变量中。如果某个指针变量在整个程序中既会被赋予数据存储器中对象的地址，也会被赋予程序存储器中对象的地址，则如果整型的地址被强制转换为指向 `const` 类型的指针，则它的 MSb 会被置 1，否则 MSb 不会被置 1。整型的剩余位将不进行修改而赋给指针变量。

将某个指针强制转换为某个整型时，指针存放的值将不进行修改而赋给整型，前提是整型的长度不小于指针的长度。

C.7.3 存放同一数组的成员的两个指针之差 ptrdiff_t 所需的整型类型（6.3.6 和 7.1.1）

`ptrdiff_t` 的类型为 `unsigned int`。

C.8 寄存器 （G.3.8）

C.8.1 可使用register存储类别说明符实际放入寄存器中的对象的范围（6.5.1）

该说明符不起作用。

© 2013 Microchip Technology Inc.

C.9 结构、联合、枚举和位域 （G.3.9）

C.9.1 使用不同类型的成员访问的联合对象的成员 （6.3.2.3）

存储在联合成员中的值将根据访问时所用的成员类型进行访问和解释。

C.9.2 结构成员的填充和对齐 （6.5.2.1）

不对结构成员进行填充或对齐。

C.9.3 普通 int 位域是视为有符号整型位域还是无符号整型位域 （6.5.2.1）

它将被视为 `unsigned int` 类型。不支持有符号位域。

C.9.4 位域在 int 中的分配顺序 （6.5.2.1）

结构中定义的第一个位域将分配存储单元中的 LSb 位置。随后的位域将分配到高位。

C.9.5 位域是否可以跨越存储单元边界 （6.5.2.1）

位域不可能跨越存储单元。任何可能跨越存储单元的位域都将分配新存储单元中的 LSb 位置。

C.9.6 选择用于表示枚举类型的值的整型类型 （6.5.2.2）

选择用于表示枚举类型的类型取决于枚举值。如果任意一个值为负值，则选择有符号类型；否则选择无符号类型。如果 `char` 类型足以容纳值的范围，则选择它；否则，选择 `int` 类型。枚举值必须装入 `int` 类型内，如果不是如此，它将被截断。

C.10 限定符 （G.3.10）

C.10.1 什么构成对 volatile 限定类型的对象的访问 （6.5.5.3）

每次引用 `volatile` 限定对象的名称都构成对该对象的一次访问。

C.11 声明符 （G.3.11）

C.11.1 可修改算术、结构或联合类型的声明符的最大数量 （6.5.4）

编译器未施加任何限制。

C.12 语句 （G.3.12）

C.12.1 switch 语句中 case 分支值的最大数量 （6.6.4.2）

对于 `switch` 语句中 `case` 标号的数量，不存在实际的限制。

C.13 预处理伪指令 （G.3.13）

C.13.1 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 （6.8.1）

这两个环境中的字符常量的求值结果为相同的值。

C.13.2 此类字符常量是否可以具有负值 （6.8.1）

它不能为负值。

C.13.3 定位可包含源文件的方法 （6.8.2）

对于在尖括号 < > 中指定的文件，将先在 -I 选项指定的目录中进行搜索，然后在标准编译器目录 （这是在编译器安装位置中找到的 include 目录）中进行搜索。

对于在引号 " " 中指定的文件，编译器将先搜索当前工作目录，再搜索 -I 选项指定的目录，然后在标准编译器目录中搜索。

如果文件名的第一个字符为 /，则假定指定了文件的完整或相对路径。在 Windows 编译器上，路径还通过 \ 或 DOS 盘符后跟一个冒号的方式指定，例如，在文件名的最前面出现 C:。

C.13.4 对于可包含的源文件的带引号名称的支持 （6.8.2）

支持带引号的名称。

C.13.5 源文件字符序列的映射 （6.8.2）

源文件字符映射到其对应的 ASCII 值。

C.13.6 对每条识别的 #pragma 伪指令的行为 （6.8.6）

请参见**第 5.14.4 节 “pragma 伪指令”**。

C.13.7 编译的日期和时间不可用时，__DATE__和__TIME__的定义（6.8.8）

这些宏总是在环境中可用。

© 2013 Microchip Technology Inc.

C.14 库函数 （G.3.14）

C.14.1 宏 NULL 扩展为的空常量 （7.1.6）

宏 NULL 扩展为 0。

C.14.2 assert 函数打印的诊断和终止行为 （7.2）

该函数将打印为 stderr "Assertion failed: %s line %d: \"%s\"\n"，其中的占位符将分别替换为文件名、行号和消息字符串。程序不会终止。

C.14.3 由 isalnum、isalpha、iscntrl、islower、isprint 和 isupper 函数测试的字符集 （7.3.1）

isalnum：ASCII 字符 a-z、 A-Z 和 0-9

isalpha：ASCII 字符 a-z 和 A-Z

iscntrl：小于 32 的 ASCII 值

islower：ASCII 字符 a-z

isprint：32 和 126 之间的 ASCII 值

isupper：ASCII 字符 A-Z

C.14.4 在发生域错误时数学函数所返回的值 （7.5.1）

acos(x) |x|>1.0 pi/2

asin(x) |x|>1.0 0.0

atan2(x,y) x=0,y=0 0.0

log(x) x<0 0.0

pow(0,0) 0.0

pow(0, y) y<0 0.0

pow(x,y) x<0 y 为非整型 0.0

sqrt(x) x<0 0.0

fmod(x, 0) x

C.14.5 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 （7.5.1）

在发生下溢时， exp()、 frexp() 和 log() 函数会将 errno 设置为 ERANGE。

C.14.6 当fmod函数的第二个参数为零时，是发生域错误还是返回零（7.5.6.4）

它将返回第一个参数，不产生任何域错误。

C.14.7 signal 函数的信号集 （7.7.1.1）

未实现 signal() 函数。

C.14.8 fprintf 函数中 %p 转换的输出 （7.9.6.1）

地址打印为 unsigned long 类型。

C.14.9 本地时区和夏令时 （7.12.1）

默认时区为 GMT。可以通过设置变量 time_zone 指定。夏令时未实现。

© 2013 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 Watch （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，变量将在其中静态地分配空间。当无法在目标器件上高效地实现软件或硬件堆栈时，它可以替代软件或硬件堆栈这种机制。

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 具有内置组件和接插模块，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——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断点和跟踪逻辑，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）

确定与指定对象相关联存储区的存在时间。

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 PIC18 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。

© 2013 Microchip Technology Inc.

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。

导入 （Import）

从外面的源 （如 hex 文件）将数据送入 MPLAB 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 （Section）

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，变量将在其中静态地分配空间。当无法在目标器件上高效地实现软件堆栈时，它可以替代软件堆栈这种机制。它会阻止重入。

堆栈，软件 （Stack, Software）

用来存储返回地址、函数参数和局部变量的存储区。该存储器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。

© 2013 Microchip Technology Inc.

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>`。

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 运行在软件模拟器模式。

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不同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 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 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。

工具条 （Tool Bar）

一行或一列图标，单击这些图标时将执行 MPLAB IDE 功能。

工作簿 （Workbook）

MPLAB SIM 激励器用于生成 SCL 激励的一种设置。

观察变量 （Watch Variable）

调试会话期间可在 Watch 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。

© 2013 Microchip Technology Inc.

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。使用串行通信和最低数量的器件引脚对 Microchip 嵌入式器件进行编程的方法。

IDE

集成开发环境，如 MPLAB 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）

数字基，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列表。如果定义了符号，列出的第一个文件是定义的位置。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 项目的组件。

警告 （Warning）

MPLAB 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长度的时间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）

一个目标文件，它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。

快速存取存储区 （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。

© 2013 Microchip Technology Inc.

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。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值。电压值会受线路长度影响 （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，而不是电压值。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。

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 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 配合使用。*参见* ICE/ICD。

MPLAB IDE

Microchip 的集成开发环境。 MPLAB 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 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 配合使用。*参见* ICE/ICD。

MPLAB SIM

Microchip 的软件模拟器，与 MPLAB 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 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 链接器设计为在 MPLAB IDE 中使用，尽管它也可独立于 MPLAB IDE 使用。

MRU

最近使用的。指可从 MPLAB 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。

© 2013 Microchip Technology Inc.

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）链接来生成完全可执行的程序。

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。

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

可一次编程。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* 访问 **Memory** 选项卡可打开 Off-Chip Memory Selection 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所有内容。查看工作中的应用程序，然后调试和编程您自己的更改。

***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，就选择用于 PIC18 MCU 的 MPLAB C 编译器。

嵌套深度 （Nesting Depth）

宏可包含其他宏的最大深度。

© 2013 Microchip Technology Inc.

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 （Simulator）

模仿器件操作的软件程序。

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 更新寄存器窗口、通过观察变量及状态显示，可分析和调试指令。也可单步执行 C 编译器源代码，但不是每次执行一条指令， MPLAB IDE 将执行一行高级 C 语句生成的所有汇编指令。

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）

在应用开始上电时，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 指令舒缓 （RELAX）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。

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。

© 2013 Microchip Technology Inc.

属性，段 （Attribute, Section）

段的 GCC 特性，可以在汇编器 `.section` 伪指令中指定为标志，例如 “executable”、“readonly”或 “data”。

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 Breakpoint）

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。

条件汇编 （Conditional Assembly）

基于指定表达式在编译时的值包含或忽略的汇编语言代码。

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

一个变量限定符，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

Watch 窗口 （Watch 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头线 （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。

© 2013 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

系统窗口控制 （System Window Control）

系统窗口控制位于窗口和某些对话框的左上角。单击这一控制通常会弹出包含 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）

指用户目标应用或目标板的一个术语。

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）

为模拟传递到软件模拟器件的外部输入而生成的数据。

应用 （Application）

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 “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）

Watch （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。

致命错误 （Fatal Error）

引起编译立即停止的错误。不产生其他消息。

© 2013 Microchip Technology Inc.

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——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， IQR）

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 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：

© 2013 Microchip Technology Inc.

索引

符号

预处理器操作符 210
#asm 伪指令 202
#define 伪指令 94
#endasm 伪指令 202
#include 伪指令 74, 95, 114
#pragma
    addrqual 215
    inline 215
    interrupt_level 215
    intrinsic 215
    printf_check 216
    psect 216
    regsused 219
    switch 219
#undef 伪指令 98
预处理器操作符 210
$ 汇编标号字符 251
$ 位置计数器符号 251
%%u psect pragma 占位符 216
% 宏参数前缀 262
% 消息格式占位符字符 88
&& 按位与操作符 249
& 宏连接字符 249, 262
- 子选项 91
+ 子选项 91
.asm 文件，参见汇编文件
.as 文件，参见汇编文件
.cmd 文件，参见命令文件
.dep 文件，参见相依性关系文件
.d 文件，参见相依性关系文件
.hex 文件，参见 hex 文件
.h 文件，参见头文件
.lib 文件，参见库
.lpp 文件，参见库和 p 代码
.lst 文件，参见汇编列表文件
.map 文件，参见映射文件
.pl 文件，参见 p 代码文件
.pre 文件，参见预处理文件
.pro 文件，参见原型文件
.sdb 文件，参见调试信息
.sym 文件，参见符号文件
. （点号）链接器装入地址字符 282
/ psect 地址符号 282
;; 宏注释禁止字符 262
??nnnn 类型的符号 251, 263
? 汇编标号字符 251
@ address 构造，参见绝对变量 / 函数
@ 命令文件说明符 72, 278
\ 命令文件字符 72
__align 限定符 30
__bank 限定符 29
__Bool 类型 145
__Bxxxx 类型的符号 239
__CONFIG 宏 135, 313, 314
__DATABANK 宏 212
__DATE__ 宏 212
__DEBUG 宏 284
__delay_ms 宏 314
__delay_us 宏 314
__deprecate 限定符 34
__EEPROM_DATA 宏 176, 315
__eeprom 限定符 31
__far 限定符 26
__FILE__ 宏 212
__FLASHTYPE 宏 212
__Hxxxx 类型的符号 239
__IDLOC7 宏 139, 316
__IDLOC 宏 139, 315
__interrupt 限定符 31
__J_PART 宏 212
__LINE__ 宏 212
__LOG 宏 （REAL ICE） 142
__Lxxxx 类型的符号 239
__MPLAB_ICD__ 宏 214
__MPLAB_ICDx__ 宏 212
__MPLAB_PICKITx__ 宏 212
__MPLAB_REALICE__ 宏 212
__near 限定符 27
__pack 限定符 33
__persisten 限定符 28
__powerdown 变量 197
__RESETBITS_ADDR 宏 212
__resetbits 变量 197
__section 限定符 35
__serial0 标号 114
__TIME__ 宏 212
__timeout 变量 197
__TRACE 宏 （REAL ICE） 142
__XC8_VERSION 宏 212
__XC8 宏 212
__XC 宏 212
__xdata 限定符 29
__ydata 限定符 29
_16Fxxx 类型的宏 212
BANKBITS 宏 212
_BANKCOUNT 宏 212
COMMON 宏 212
_delay 函数 59, 215, 316, 317
_EEPROMSIZE 宏 177, 213
_ERRATA_TYPES 宏 213
_FLASH_ERASE_SIZE 宏 213

© 2013 Microchip Technology Inc.

_FLASH_WRITE_SIZE 宏 213
GPRBITS 宏 213
_HAS_OSCVAL 宏 213
_HTC_EDITION_ 宏 213
_HTC_VER_MAJOR_ 宏 213
_HTC_VER_MINOR_ 宏 213
_HTC_VER_PATCH_ 宏 213
_HTC_VER_PLVL_ 宏 213
MPC 宏 213
_OMNI_CODE_ 宏 213
_PIC12E 宏 213
_PIC12 宏 213
_PIC14E 宏 213
_PIC14 宏 213
_PIC18 宏 213
PROGMEM 宏 213
_RAMSIZE 宏 213
_READ_OSCCAL_DATA 宏 141
_ROMSIZE 宏 213
_ 汇编标号字符 204, 251
<> 宏参数字符 249, 262

数字

0b 二进制基数说明符 158

A

abs PSECT 标志 206, 256
ABS1 类 237
abs 函数 317
acos 函数 318
addrqual pragma 伪指令 215
all 子选项 91
ANSI C 标准 14
  不符合 131
  符合标准 115, 131
  实现定义的行为 131, 479-486
ASCII 字符 144, 249
  扩展 160
asctime 函数 319
asin 函数 319
asm C 语句 38, 202
ASMOPT 控制 266
aspic.h 头文件 205
assert 函数 320
atan2 函数 321
atan 函数 321
atof 函数 322
atoi 函数 322
atol 函数 323
安静模式 97
安装，参见编译器安装与激活
按位取反操作符 179

B

BANKMASK 宏 55, 202
BANKSEL 伪指令 54, 202
BANKx 类 237
bankx 限定符 163
BIGRAM 类 237
bin 目录 76
bit PSECT 标志 256
bitclr 宏 140
bitset 宏 140
bsearch 函数 324
bss psect 166, 195, 235
BTEMP 变量 191
版本号 115
版权声明 97
帮助！ 41, 105
包含文件，参见头文件
保护代码 58, 114
保留存储器 65, 101, 111, 112
保修登记 8
被跟踪的对象 188
被零除 481
本机跟踪 142
比例值 256
编译
  make 文件，参见 makefile
  汇编文件 80
  混合文件类型 78
  时间 115
  为汇编文件 98
  为目标文件 93
  为预处理文件 109
  为中间文件 109
  序列 75-79
  增量编译 79
编译堆栈 169-172
  基值 171
编译器安装与激活 41-42
编译器错误
  格式 88
  列表 377-478
编译器工作模式 11, 60, 106
编译器生成的 psect 216, 232-235
编译器生成的代码 64
编译器选项，参见驱动程序选项
编译器选择 44
编译器应用程序 75
编译所用时间 115
编译项目 79
变量
  auto 169
  长度 143, 147
  程序存储器 50
  初始化 195
  从汇编代码访问 204
  存储持续时间 165
  放置在特定地址处 216
  静态 167
  绝对 25, 175
  位置 65
  在程序存储器中 173-174
  在汇编代码中 260
  在寄存器中 178
  最大大小 63
变量的位访问 51
变量名，参见标识符
标号，汇编 249, 252
标识符

© 2013 Microchip Technology Inc.

C 143
汇编 251
惟一长度 19, 51, 97
标准库文件
标准模式，参见编译器工作模式

C

caspic.h 头文件 206
CCI 15, 132
ceil 函数 325
cgets 函数 325
char 数据类型 20, 144
checksum psect 233
chipinfo 文件 111, 112, 242
cinit psect 233
class PSECT 标志 257
CLRWDT 宏 326
CLRWDT 指令 59
CODE 类 236
COMMON 类 237
COND 控制 266
config psect 233
config pragma 伪指令 135
CONFIG 类 237
const psect 233
const 对象
初始化 161
存储单元 173
CONST 类 236
const 限定符 161, 174
control 限定符 140
cosh 函数 327
cos 函数 326
cputs 函数 327
CREF 选项 297-299
CREF 应用程序 267, 297-299
CROMWELL 选项 300, 301-302
CROMWELL 应用程序 84, 300-302
cstack psect 169, 235
ctime 函数 328
C 标识符 143
C 标准函数库 313-376
C 标准库 81
参数，参见函数和参数
操作符，强制类型转换 46
插装跟踪 142
长度限制
const 变量 174
非自动变量 167
自动变量 172
常量
C 说明符 158
汇编 250
字符 160
字符串，参见字符串字面值
程序存储器 173, 257
绝对变量 161, 176
程序计数器 251
程序入口点 197
重叠的 psect 258
重叠的存储区 281
重新编译项目 79
处理器选择 100, 283
存储持续时间 165
存储器
保留 50, 65, 101, 111, 112
存储区，参见分区存储器
公共 162, 165, 191
剩余 66
数据 165, 175, 235, 257, 258
未分区 165
未用 104
线性数据 165
页 52, 247, 248, 258, 264
摘要 66, 115
指定范围 279
存储器分配 165
程序存储器 173
动态 178
非自动变量 166
函数代码 183
静态变量 167
数据存储器 165
存储器模型 178
错误计数器 86
错误文件 279
错误消息 67, 86-90
格式 88, 104
列表 377-478
位置 67
语言 106
最大数量 104

D

DABS 伪指令 261
data psect 166, 216, 235
dat 目录 86
DB 伪指令 260
DDW 伪指令 260
default 子选项 91
delta PSECT 标志 200, 257, 279
device_id_read() 函数 329
div 函数 330
DI 宏 329
doprnt.c 源文件 198
doprnt.pre 文件 109, 199
DS 伪指令 260
DW 伪指令 260
大尾数格式 306
大小限制 63
单步编译 77, 78
等于汇编符号 259
调用深度，参见堆栈深度
调用图 134, 170, 193, 273-275
动态存储器分配 178
读 - 修改 - 写问题 69
读取定时器寄存器 138
读物，推荐 8
端口上的毛刺 56
段，参见 psect
段选择器 280
堆栈 134

深度 274
溢出 69, 134
硬件 169
对齐
HEX 文件记录 113
psect，参见 reloc PSECT 标志
多字节 SFR 138

E

EEDATA 类 237
EEPROM
数据 176
eeprom psect 233
eeprom_data psect 233
eeprom_read 函数 177, 330
eeprom_write 函数 177, 330
EEPROM 存储器
初始化 176
读取 177
写入 177
eeprom 限定符 164
EI 宏 329
ELSE 伪指令 261
ELSIF 伪指令 261
ENDIF 伪指令 261
ENDM 伪指令 262
END 伪指令 255
entry__ 类型的符号 204
ENTRY 类 236
EQU 伪指令 249, 252, 259
ERRATA_4000_BOUNDARY 宏 213
ERRATA_TYPES 宏 103
eval_poly 函数 331
EXPAND 控制 262, 266
exp 函数 331
二进制常量
C 代码 158
汇编 250

F

fabs 函数 332
far 限定符 163
fcall 伪指令 55, 248
fixup 溢出错误 70, 284, 285
Fletcher 校验和算法 306
floor 函数 332
fmod 函数 333
fpbase 符号 156
frexp 函数 333
ftoa 函数 334
F 常量后缀 159
翻译单元 74, 109
返回值，参见函数和返回值
非易失性 RAM 161
分区存储器 63, 165, 258
链接器类 237
限定符 163
线性寻址 64, 167, 175
在汇编代码中选择 54, 202, 247, 264
分页存储器 52
在汇编代码中选择 55
分页的存储器 247, 248, 258, 264
浮点舍入 50
浮点型 102, 105, 146-148
偏移的指数 147
舍入 147
指数 147
浮点型常量后缀 159
浮点型数据类型 105, 146
符号
汇编器生成 251
链接器定义 239
未定义 283
符号表 283
排序 281
符号位 146
符号文件 95, 280, 283, 300
复位 162
确定原因 197
向量 101
之后执行的代码 57, 83, 194, 197

G

get_cal_data 函数 336
getchar 函数 335
getche 函数 335
getch 函数 334
gets 函数 336
global PSECT 标志 257
GLOBAL 伪指令 54, 201, 253, 255
gmtime 函数 337
跟踪功能 142
公共存储器 162, 165, 191
关键字，参见限定符
硅片勘误 103

H

HEXMATE 选项 305-311
HEXMATE 应用程序 76, 303
hex 文件 73, 76, 303
HI-TECH universal toolsuite 117
HI_TECH_C 宏 214
HLINK 应用程序 277
HTC_ERR_FORMAT 环境变量 88
HTC_MSG_FORMAT 环境变量 88
HTC_WARN_FORMAT 环境变量 88
HTML 文件 106
重命名 97
地址 257
地址对齐 113
多个 280
格式 309
合并 57, 304
记录长度 113, 309
扩展地址记录 309
嵌入序列号 310
嵌入字符串 311
数据记录 113, 305
填充未用的存储器 104
统计 309
函数
参数 169, 185, 200

© 2013 Microchip Technology Inc.

长度限制 185
创建原型 110
大小 52, 64
调用约定 187
堆栈使用 187
返回存储区 188
返回值 187
放置在特定地址处 216
复制 192-193
静态 182
绝对 25, 184, 218
内联 182
签名 238, 265
说明符 182
外部 183
位置 65
以汇编代码编写 200
原型 238, 265
指针 156
中断，参见中断函数
合并 hex 文件 305
宏连接 210
互联网地址 9
环境变量 73
汇编变量
C 等效符号 204
global 255
保留存储器 260
标识符 251
初始化 260
定义 260
类型 251
汇编常量 250
汇编代码
编写 53-55
堆栈溢出 69
基于 C 生成 98
优化 108
由 C 调用 200
与 C 混合使用 53, 54, 200
与 C 交互 203
预处理 97, 210
汇编列表文件 64, 85, 99, 243, 269-275
包含条件代码 266
标题和子标题 267
格式 243, 244, 266
宏 243, 266, 267
空行 267
内容 269
排除条件代码 267
十六进制常量 243
新的页面 267
汇编器宏 262
重复 263
禁止注释 262
使用参数重复 264
未命名 263
在列表中禁止 267
在列表中扩展 243, 266
汇编器控制 265-267
汇编器生成的符号 251
汇编器伪指令 254-265
汇编器选项 242-245
汇编器应用程序 241-268
禁止消息 244
用法 241
最大错误数 245
汇编器优化 268
汇编器优化器
堆栈深度注意事项 274
和列表文件 269
使能 108, 243, 266
选择性禁止 266
汇编文件 73, 200
汇编语言
volatile 对象 250
包含文件 266
保留存储器 206, 261
标号 249, 252
标号作用域 253, 255
表达式 253
操作符 253
差异 246
常见错误 55, 70
重复指令 263
初始化单元 260
访问 C 对象 204
分隔符 249
行号 244
寄存器 55, 205, 252
绝对变量 261
数据类型 251
条件 261
伪操作 254
位置计数器 251
相对跳转 251
语句格式 249
注释 249
字符串 250
字符集 249
汇编源文件 74

I

i1 符号前缀 192
ICD，参见调试器
idata psect 216, 233
idloc psect 233
IDLOC 类 237
ID 存储单元 139
IEEE 浮点型格式，参见浮点型 146
IF 伪指令 261
INCLUDE 控制 266
INHX32 hex 文件 305, 309
INHX8M hex 文件 305, 309
init psect 233
inline pragma 伪指令 215
intcode psect 233
Intel HEX 文件，参见 hex 文件
intentry psect 234
interrupt_level pragma 伪指令 215
intrinsic pragma 伪指令 215

int 类型 143
interrupt 限定符 189
IRPC 伪指令 264
IRP 伪指令 264
isalnum 函数 338
isalpha 函数 338
isatty 函数 339
isdigit 函数 338
islower 函数 338
itoa 函数 339

J

jmp_tab psect 234
基数说明符
    C 代码 158
    汇编 250
基数说明符，参见基数说明符
基值 171
激活，参见编译器安装与激活
寄存器
    分配到 178
    特殊函数 252
    由函数使用 65, 181
    在汇编代码中 252
建议性消息 87, 107
交叉引用 267
交叉引用列表 297
交叉引用文件 297
校验和 100, 306
阶段错误 257
结构 149
    位域 51, 150
    最大长度 167
结构位域 150
结构限定符 149
仅包含符号的目标文件 280
禁止中断 58, 192
精简模式，参见编译器工作模式
警告级别 90
    设置 283
警告消息 67, 87
    格式 116
    禁止 68, 107, 220, 283
    位置 67
    显示的级别 116
    阈值级别 116
静态变量 167, 195, 204
静态函数 182, 204
绝对 psect 206, 256, 258
绝对变量 25, 137, 174, 175-176, 261
绝对函数 25, 184, 218
绝对目标文件 281

K

勘误 103
看门狗定时器 68, 494
可重定位目标文件 285
可重入函数 169, 192
可寻址单元，参见 delta PSECT 标志
可字节寻址的存储器 257
可字寻址的存储器 257
客户通知服务 9
客户支持 10
库 81
    p 代码 76, 81, 291
    创建 44, 292
    从项目中排除 113
    格式 291
    更换模块 82
    混淆 114
    链接 277, 283
    列出模块与符号 292
    模块顺序 293
    目标 76, 285, 287, 291
    扫描额外 95
    删除模块 292
    搜索顺序 72
    替换模块 237
    添加文件 292
    用户定义的 82
库管理器 82, 237, 291-294
库函数 313-376
扩展名 73
扩展指令集 133
扩展字符集 160

L

l.obj 文件 281
labs 函数 340
ldexp 函数 340
ldiv 函数 341
LED，闪烁 68
LIBR 应用程序，参见库管理器
lib 目录 81, 95, 198
limit PSECT 标志 257, 283
limits.h 头文件 144
LIST 控制 266
ljmp 伪指令 55, 248
local PSECT 标志 257
localtime 函数 341
LOCAL 伪指令 251
log10 函数 342
log 函数 342
long double 类型 146
long int 类型 143
longjmp 函数 343
ltoa 函数 344
L 常量后缀 158
类
    链接器 236
类，参见链接器类
类型，参见数据类型
类型的长度 143, 147
类型的大小 102
类型之间的转换 179
类型转换 46, 179
联合
    匿名 151
    限定符 149
链接地址 281
链接描述文件 222
链接器定义的符号 239

© 2013 Microchip Technology Inc.

链接器类 111, 112, 236, 257, 282
边界参数 283
地址范围 279
地址限制 283
调整 111, 112
链接器选项 277, 277-284, 287
查看 287
调整 96
链接项目 222
临时变量 169

M

MACRO 伪指令 249, 262
maintext psect 184, 234
main 函数 17, 82, 194
make 文件 72, 77, 79
MDF 86
mediumconst psect 234
memchr 函数 344
memcmp 函数 345
memcpy 函数 346
memmove 函数 346
memset 函数 347
merge PSECT 标志 257, 258
Microchip COF 文件 109
Microchip 互联网网站 9
mktime 函数 347
modf 函数 348
MPLAB IDE
编译器工作模式 44
编译器选择 44
编译选项 45, 96, 117
插件 117
搜索路径 95
调试编译 45, 284
MPLAB X IDE
编译选项 124
命令，参见编译项目和命令行
命令行驱动程序，参见驱动程序
命令文件 72, 293
链接器 278
模块 74
生成 109
默认 psect 254
目标操作数 246
目标代码版本号 286
目标器件，参见器件和选择
目标文件 76, 80, 93, 244, 291
仅符号 280
绝对 281
可重定位 285
内容 285
目标文件库 285

N

near 限定符 162
NOCOND 控制 267
NOEXPAND 控制 267
NOLIST 控制 243, 267
none 子选项 91
NOP 宏 349
NOXREF 控制 267
NULL 宏 24
NULL 指针 156, 157
nul 宏操作符 262
nv psect 166, 235
内联函数 182
匿名联合 151

O

OBJTOHEX 选项 295
OBJTOHEX 应用程序 295
option 指令 140
OPT 控制伪指令 265
ORG 伪指令 206, 259
ovrld PSECT 标志 206, 258

P

PAGESEL 伪指令 55, 247, 264
PAGE 控制 267
PATH 环境变量 73
persistent 限定符 162, 195, 196
picc.ini 文件 111, 112
powerup psect 197, 234
powerup.as 197
powerup 标号 194
pow 函数 350
pragma 伪指令 214
printf_check pragma 伪指令 216
printf 函数 58, 80, 81, 198, 350
格式检查 216
预处理 109
PROCESSOR 伪指令 244
psect 285
bss 166, 195, 235
checksum 233
cinit 233
config 233
const 233
cstack 169, 235
data 166, 235
delta 值 279
eeprom 233
eeprom_data 233
idata 233
idloc 233
init 233
intcode 233
intentry 234
jmp_tab 234
maintext 184, 234
mediumconst 234
powerup 197, 234
reset_vec 234
reset_wrap 234
smallconst 234
strings 234
stringtext 235
textn 184, 200, 216, 235
xxx_text 185, 235
编译器生成 232-235
拆分 184

重叠 258
对齐，参见 reloc PSECT 标志
放置在存储器中 257, 281
函数 184
绝对 206, 258
类 279, 283
链接 285
列表 115
默认 254
区分 ROM 和 RAM 258
页边界和 258
与其他一起放置 258
指定地址 279, 281
指定地址范围 282
组合 257
最大长度 258
最高地址 257
psect pragma 伪指令 216
PSECT 标志 256-258
psect 关联 257
PSECT 伪指令 55, 256
pure PSECT 标志 258
putchar 函数 352, 353
putch 函数 58
puts 函数 354
p 代码库，参见库和 p 代码
p 代码文件 72, 76, 77, 109
混淆 114
跑车 251
配置位 135
偏移的指数 147

Q

qsort 函数 354
启动模块 82, 113
起始记录 255
器件宏 212
器件系列宏 213
器件选择 100, 283
器件支持 64, 100, 133
签名 238, 265
强制类型转换 46, 179
清零变量 195
驱动程序 71-116
长命令行 72
单步编译 78
关于选项的帮助 105
命令格式 72
命令文件 72
输入文件 72
驱动程序选项 45, 72, 91-116
- 91
+ 91
ADDRQUAL 99, 120, 163, 164
all 91
ASMLIST 85, 98, 99, 243, 269
C 93
CCI 39, 99
CHECKSUM 100, 121
CHIP 100, 244, 286
CHIPINFO 64, 100
CODEOFFSET 56, 101, 121
D 94, 98, 119
DEBUGGER 101, 121
default 91
DOUBLE 102, 122, 146
E 88, 94
ECHO 102
EMI 102
ERRATA 103
ERRFORMAT 88, 104
ERRORS 86, 104, 245, 286
FILL 100, 104, 121
FLOAT 105
G 95
GETOPTION 105
HELP 105
HTML 106
I 95, 118
L 95
LANG 87, 106
L （链接器选项） 96, 285, 287
M 85, 96, 286
MEMMAP 106
MODE 106, 120
MSGDISABLE 90, 107, 244
MSGFORMAT 88, 104, 107
N 51, 97, 119
NODEL 78, 107
none 91
O 97
OBJDIR 107
OPT 108, 119, 243
OUTDIR 107, 108, 118
OUTPUT 84, 109, 121
P 97, 119, 205, 210
PASS1 109
PRE 109, 210
PROTO 110
Q 97
RAM 111, 122
ROM 112, 123
RUNTIME 82, 112, 121, 141, 187
S 98
SCANDEP 114
SERIAL 114
SETOPTION 241
SETUP 87, 106
SHROUD 114
STRICT 115
SUMMARY 115, 121
TIME 115
U 98, 119
V 98, 120
WARN 90, 116, 120
WARNFORMAT 88, 89, 104, 116
取消定义宏 98

R

RAM 存储区，参见分区存储器
RAM 类 237
rand 函数 355

© 2013 Microchip Technology Inc.

REAL ICE 调试器 142
REALICE，参见调试器
regsused pragma 伪指令 219
reloc PSECT 标志 231, 258
REPT 伪指令 263
reset_vec psect 234
reset_vec 标号 194
reset_wrap psect 234
RETLW 指令 173
round 函数 356
入口点 255
软件堆栈 134

S

setjmp 函数 357
SET 伪指令 249, 259
SFR 137
    在行内汇编代码中访问 206
    在汇编代码中访问 205
SFRx 类 237
short int 类型 143
SIGNAT 伪指令 201, 238, 265
sinh 函数 327
sin 函数 357
size PSECT 标志 258
SLEEP 宏 358
smallconst psect 234
sources 目录 197
space PSECT 标志 258
SPACE 控制 267
sprintf 函数 358
sqrt 函数 359
srand 函数 360
sscanf 函数 360
start_initialization 194
startup.as，参见启动模块
start 标号 194, 197
STATUS 寄存器，保存 197
STRCODE 类 236
strings psect 234
stringtext psect 235
STRING 类 236
struct 类型，参见结构
SUBTITLE 控制 267
switch pragma 伪指令 219
switch 语句 180, 219
闪存函数 332
闪烁 LED 68
上电程序 83, 194, 197
十六进制常量
    C 代码 158
    汇编 243, 250
实现定义的行为 131, 479-486
输出文件 97, 108
    重命名 97
    名称 84
    目录 108
    指定名称 97
输出文件格式
    二进制 109
    库 109
    指定 109, 295
输入文件 72
数据存储器 165, 175, 235, 257, 258
数据类型
    长度 143, 147
    大小 19
    浮点型 146-148
    汇编 251
    整型 143-145
数据指针 153
数组 167
    和指针长度 153
    虚拟指针目标 157
    最大长度 167
双精度浮点型数据类型 102, 146
搜索路径，参见头文件

T

tanh 函数 327
tan 函数 371
textn psect 184, 200, 216, 235
time 函数 372
TITLE 控制 267
toascii 函数 372
tolower 函数 372
toupper 函数 372
tris 指令 140
trunc 函数 373
特殊的类型限定符 162
特殊功能寄存器，参见 SFR
替换库模块 237
填充未用的存储器 58, 100, 104
调试器 57, 101, 300
调试信息 95
    汇编器 244
    文件格式 300
    优化器和 243
条件汇编 261
通用编译器接口 15, 132
头文件 17, 198
    器件 134, 137
    搜索路径 18, 95

U

ungetc 函数 373, 374
universal toolsuite 插件 117
USB 501
utoa 函数 374
U 常量后缀 158

V

va_arg 函数 375
va_end 函数 375
va_start 函数 375
volatile 限定符 56, 58, 161, 207, 250
vprintf 函数 350
vsprintf 函数 358
vsscanf 函数 360

W

Windows 注册表 73

with PSECT 标志 258
withtotal 258
WWW 地址 9
外部存储器 163
外部函数 183
伪操作，参见汇编器伪指令
伪指令，参见汇编器伪指令
尾数 146
尾数法 143, 146
未初始化变量 195
未定义的符号 207
    增加 283
未分区的存储器，参见公共存储器
未命名的 psect 254
未命名结构成员 150
未命名位域 150
未用变量 207
    删除 161
未用存储器 66
    填充 100
位数据类型 143, 145
位域 22, 23, 51, 150-151
位指令 140, 145
位置计数器 251, 259
文档
    编排 7
    约定 8
文件扩展名 73
文件类型
    符号 280
    汇编列表，参见汇编列表文件
    交叉引用 297
    库，参见库
    命令 72, 278, 293
    目标，参见目标文件
    输入 72
    相依性关系 114
    预处理 109
    原型 110
    中间 107
    中间，参见中间文件

X

xc.h 头文件 134
xc.xml XML 文件 73
XC_XML 环境变量 73
XC8 应用程序 72
XML 文件 73
XREF 控制 267, 297
xtoi 函数 376
xxxx@yyyy 类型的符号 205
现场切换代码 53, 63, 185, 191, 219
线性数据存储器 64, 165, 175
限定符 161-164
    __align 30
    __bank 29
    __deprecate 34
    __eeprom 31
    __far 26
    __interrupt 31
    __near 27
    __pack 33
    __persistent 28
    __section 35
    __xdata 29
    __ydata 29
    auto 169
    bankx 163
    const 161, 174
    control 140
    eeprom 164
    far 163
    near 162
    persistent 162, 195, 196
    volatile 56, 58, 161, 207, 250
    和结构 149
    和自动变量 169
    禁止非 ANSI C 115
    特殊 162
    中断 189
相对跳转 251
相依性关系文件 114
详细输出 98
项目 79
    汇编文件 80
项目名称 84
消息 86-90
    ID 号 86
    错误，参见错误消息
    格式 88
    含义 67
    建议性 87, 107
    禁止 86, 90, 107, 220
    警告，参见警告消息
    警告级别 90
    类型 87
    列表 377-478
    语言 87, 106
    占位符 88
    致命错误 87
消息说明文件 86
消息系统 86-90
    环境变量 88
    将消息重定向到文件 94
    将消息追加到文件 94
    默认警告级别 90
    默认语言 87
    消息计数 86
    支持的语言 87
小尾数法格式 143, 146, 306
    规范 296
序列号 114, 310
    嵌入 310
选项，参见驱动程序选项
循环移位操作符 59, 180

Y

延时程序 59, 314
移位操作符 481
已初始化变量 113, 195
影子寄存器 191
映射文件 85, 281, 286-290

© 2013 Microchip Technology Inc.

硬件堆栈 134
硬件勘误 103
优化 106, 108, 208
    代码长度 61
    导致损坏 58
    更快的代码 62
    汇编器 108, 245
    数据大小 62
    速度与空间 108
    调试 108, 245
    中断函数 63
语言支持 87
预处理 210
    汇编器文件 97
预处理器
    宏连接 210
    类型转换 211
    搜索路径 95
预处理器宏
    包含字符串 94
    定义 94
    取消定义 98
    惟一长度 97
    预定义 36, 212
预处理器伪指令 210-211
    在汇编文件中 97, 249
预处理文件 76, 109, 210
原型文件 110
源代码级调试 300
源文件 74
允许中断 192
运行时环境 112
运行时启动代码 82, 194, 233, 285
    保存变量 162
    变量初始化 195
    汇编列表 99
运行时启动模块 113

Z

增量编译 79
诊断文件 85
振荡器校准常量 59, 141, 196
    保留 141
整型常量 158
整型后缀 158
整型提升 179
支持的器件，参见器件和支持
只读变量 161
指令集 133
指数 146
指针 152-157, 165
    比较 157
    编码 155
    定义 152
    赋予虚拟目标 156
    赋予整型 156
    函数 156
    类型 152
    目标 153
    数据 153
    限定符 152
指针引用图 154, 272
致命错误消息 87
中断 53
    禁止 58, 192
    现场切换 53, 63, 185
    向量 101, 189
    源 189
    允许 192
中断函数 189
    现场切换 191, 219
    移动 101, 185
    优化 63
中间文件 72, 75, 76, 107, 109
    汇编 80
主干代码 170, 189
注册表 73
专业模式，参见编译器工作模式
装入地址 281
字边界 258
字符常量
    汇编 250
    在 C 中 160
字符串 （strxxx）函数 361-371
字符串字面值 160
    存储单元 160, 311
    汇编 250
    类型 160
    连接 160
    压缩 311
自动变量 165, 169, 273
    编译堆栈 169
    长度限制 172
    初始化 195
    存储器分配 169-172
    汇编符号 169
自举程序 56, 112, 113, 309
自述文件 9
最大 （惟一）标识符长度 97

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

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
Farmington Hills, MI
Tel: 1-248-538-2250
Fax: 1-248-538-2260

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

圣克拉拉 Santa Clara
Santa Clara, CA
Tel: 1-408-961-6444
Fax: 1-408-961-6445

加拿大多伦多 Toronto
Mississauga, Ontario, Canada
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: 852-2401-1200
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 – 杭州
Tel: 86-571-2819-3187
Fax: 86-571-2819-3189

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
Fax: 886-7-330-9305

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: 91-20-2566-1512
Fax: 91-20-2566-1513

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 - 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

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

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

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

11/29/12

© 2013 Microchip Technology Inc.