适用于 PIC32C/SAM MCU 的 MPLAB® XC32 C/C++编译器用户指南

开发工具客户须知

重要：
开发工具手册如同所有其他文档一样具有时效性。我们不断改进工具和文档以满足客户的需求，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/）获取最新版本的 PDF 文档。

文档每页的底部均标有 DS 编号。DS 格式为 DS<文档编号><版本>_CN，其中<文档编号>为 8 位数字，<版本>为大写字母。

有关最新信息，请访问 onlinedocs.microchip.com/查看您所使用的工具的帮助信息。

目录

1. 前言

本用户指南探讨了适用于 PIC32C/SAM 的 MPLAB XC32 C/C++编译器和支持信息。

1.1 本指南使用的约定

本指南采用以下文档约定。在大多数情况下，格式符合 *OASIS Darwin Information Typing Architecture (DITA) Version 1.3 Part 3: All-Inclusive Edition*（2018 年 6 月 19 日）。

表 1-1. 文档约定

说明	实现	示例
参考书目	DITA：引用	*适用于 PIC32C/SAM MCU 的 MPLAB® XC32 C/C++编译器用户指南*
需强调的文字	斜体字	...为*仅有*的编译器...
窗口、窗格或对话框名称	DITA：窗口标题	**Output** 窗口。 **New Watch** 对话框。
窗口或对话框中的字段名	DITA：用户界面控件	选择 **Optimizations** 选项类别。
菜单名称或菜单项	DITA：用户界面控件	选择 **File** 菜单，然后选择 **Save**。
菜单路径	DITA：级联菜单，用户界面控件	**File** > **Save**
选项卡	DITA：用户界面控件	单击 **Power** 选项卡。
软件按钮	DITA：用户界面控件	单击 **OK** 按钮。
键盘上的按键	DITA：用户界面控件	按下 **F1** 键。
文件名和路径	DITA：文件路径	`C:\Users\User1\Projects`
源代码：嵌入	DITA：代码片段	切记要在代码的开头添加`#define START`。

源代码：代码块	DITA：代码块	示例如下：
		`#include <xc.h>`
		`main(void) {`
		`  while(1);`
		`}`

用户输入的数据	DITA：用户输入	键入器件名称，例如 `PIC18F47Q10`。
关键字	DITA：代码片段	`static, auto, extern`
命令行选项	DITA：代码片段	`-Opa+, -Opa-`
二进制位值	DITA：代码片段	`0, 1`
常量	DITA：代码片段	`0xFF, 'A'`
可变参数	DITA：代码片段 + 选项	*`file`*`.o`，其中 *`file`* 可以为任何有效的文件名。
可选参数	方括号[ ]	`xc8 [`*`options`*`] `*`files`*
选择互斥参数；“或”选择	花括号和竖线：{ | }	`errorlevel {0|1}`
代替重复文字	省略号...	*`var_name`* `[, `*`var_name`*`...]`

表示由用户提供的代码	省略号...	`void main (void)`
		`{ ...`
		`}`

verilog 格式的数字，其中 N 为总位数，R 为基数，n 为其中一位	N'Rnnnn	4'b0010，2'hF1
器件相关标志。指定并非所有器件上均支持的特性。支持的器件将在标题或文本中列出	[DD]	`xmemory` 属性

1.2 推荐读物

适用于 PIC32 MCU 的 MPLAB XC32 语言工具套件包含 C 编译驱动程序（`xc32-gcc`）、C++编译驱动程序（`xc32-g++`）、汇编器（`xc32-as`）、链接器（`xc32-ld`）和归档器/库管理器（`xc32-ar`）。本文档介绍了如何使用 MPLAB XC32 C/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材料。

发行说明（自述文件）

关于 Microchip 工具的最新信息，请阅读软件附带的相关发行说明（HTML 文件）。可以在 Microchip 网站上找到发行说明的在线副本。

《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）

关于使用 32 位汇编器、目标链接器、目标归档器/库管理器和各种实用程序的指南。

Microchip Unified Standard Library Reference Guide（DS50003209）

本指南详细介绍了编译器随附的标准库提供的函数、类型和预处理器宏。

针对器件的文档

Microchip 网站上提供了许多描述 32 位器件功能和特性的文档，其中包含：

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

C 标准方面的信息

International Standardization Organization (ISO) and International Electrotechnical Commission (IEC) – *ISO/IEC 9899:1999 Programming languages — C*. ISO Central Secretariat, Chemin de Blandonnet 8, CP 401, 1214 Vernier, Geneva, Switzerland.

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

C++标准方面的信息

Stroustrup, Bjarne, *C++ Programming Language: Special Edition,* 3rd Edition. Addison-Wesley Professional; Indianapolis, Indiana, 46240.

ISO/IEC 14882:2014 C++标准。ISO C++标准是由 ISO（国际标准化组织）与 ANSI（美国国家标准研究所）、BSI（英国标准研究所）和 DIN（德国国家标准组织）合作制定的。

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

C 语言参考手册

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

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

Kochan, Steven G., *Programming In ANSI C*，修订版。Hayden Books, Indianapolis, Indiana 46268.

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

Van Sickle, Ted., *Programming Microcontrollers in C*，第一版。LLH Technology Publishing, Eagle Rock, Virginia 24085.

GNU 编译器集合（GNU Compiler Collection，GCC）文档

重要：

1. 以下 GCC 和 Binutils 文档中介绍的功能和选项可能不受 MPLAB XC32 官方支持。建议仅使用 Microchip XC32 文档中介绍的功能。

gcc.gnu.org/onlinedocs/

sourceware.org/binutils/

Arm®参考手册

Arm C 语言扩展，版本 1.1，文档编号 IHI 0053B，发行日期 2013 年 11 月 12 日。

本文档指定了 Arm C 语言扩展，使 C/C++语言编程人员能够在对源代码可移植性限制最小的情况下利用 Arm 架构。

2. 编译器概述

本章定义并介绍了适用于 PIC32C/SAM MCU 的 MPLAB® XC32 C/C++编译器。

2.1 器件支持

MPLAB XC32 C/C++编译器完全支持大多数 Microchip PIC32C、SAM、CEC17、MEC15 和 MEC17 器件。

2.2 编译器说明和文档

MPLAB XC32 C/C++编译器是一款全功能的优化编译器，可将标准 C 程序转换为 32 位器件汇编语言代码。工具链支持基于各种 Arm Cortex®-M*x* 和 A*x* 内核的 PIC32C/SAM 单片机系列。该编译器还支持多种命令行选项和语言扩展，以充分利用 32 位器件的硬件功能，同时满足编译器代码生成器较高的控制要求。

该编译器基于自由软件基金会的 GNU 编译器集合（GCC）。

该编译器可用于几种流行的操作系统，包括 Windows®、Linux®和 macOS®。

该编译器可以在免费（Free）或专业（PRO）工作模式下运行：专业工作模式是许可模式，需要使用激活密钥和互联网连接来使能。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对于所有模式是相同的。这些模式之间的差别仅在于编译器采用的优化级别。

2.2.1 约定

在本手册中，将经常使用“编译器”这一术语。它可以指代构成 MPLAB XC32 C/C++编译器的应用程序集合的全集或子集。通常，举例来说，知道操作是由解析器还是代码生成器应用程序执行的并不重要，说它是由“编译器”执行的就足够了。

使用“编译器”指代命令行驱动程序（或只是驱动程序）也是合理的，因为总是会执行该应用程序来调用编译过程。MPLAB XC32 C/C++编译器软件包的 C/汇编语言驱动程序名为 `xc32-gcc`。C/C++/汇编语言驱动程序称为 `xc32-g++`。驱动程序选项说明介绍了驱动程序及其选项。按照这种角度，应将“编译器选项”视为驱动程序命令行选项，除非在本手册中另有指定。

类似地，“编译”指代在将源代码生成为可执行二进制映像中涉及的步骤的全部或某些部分。

2.2.2 语言标准

MPLAB XC32 编译器是经过充分验证的工具，符合适用于编程语言的 ISO/IEC 9899:1990 C 编程语言标准（本文档中称为 C90）以及 ISO/IEC 9899:1999 C 编程语言标准（C99）。此外，它还符合 ISO/IEC 14882:2014 C++编程语言标准（C++14）。该编译器还支持多种面向 PIC32 MCU 的语言扩展，以充分利用 32 位器件的硬件功能，同时满足编译器代码生成器较高的控制要求。

2.2.3 优化

该编译器使用了一组复杂的优化轮次，这些优化轮次采用许多先进的技术，基于 C/C++源代码生成高效紧凑的代码。优化轮次包括适用于所有 C/C++代码的高级优化，以及可利用器件架构特定功能的特定于 PIC32C/SAM MCU 的优化。

关于优化的更多信息，请参见优化。

2.2.4 ISO/IEC C 库支持

MPLAB XC32 C/C++编译器提供可辅助代码开发的函数库、宏、类型和对象。

Microchip 统一标准库用于 C 项目。该库符合 C99 标准。编译 C++项目时，在标准 C++库（`libstdc++`）中使用与 C 库相同的库实现（`libc`）。

提供了多个数学库来处理带和不带硬件浮点单元的器件。

有关标准 C 库的描述，另请参见 *Microchip Unified Standard Library Reference Guide*，其内容适用于所有 MPLAB XC C 编译器。

2.2.5 ISO/IEC C++库支持

MPLAB XC32 C/C++编译器附带基于 GNU C++库且符合 ISO/IEC 14882:2011 的标准 C++库（`libstdc++`）。编译 C++项目时，Microchip 统一标准库在标准 C++库中用作 C 库（`libc`）。

有关标准 C 库的描述，另请参见 *Microchip Unified Standard Library Reference Guide*，其内容适用于所有 MPLAB XC C 编译器。

2.2.6 编译器驱动程序

该编译器包含了功能强大的命令行驱动程序。使用该驱动程序，可以在单个步骤中编译、汇编和链接应用程序。

2.3 编译器和其他开发工具

该编译器可与许多其他 Microchip 工具配合工作，包括：

- MPLAB XC32 汇编器和链接器——请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）。
- MPLAB X IDE（v5.50 或更高版本）。
- MPLAB 软件模拟器。
- 所有 Microchip 调试工具和编程器。
- 支持 32 位器件的演示板和入门工具包。

3. 操作指南

本章包含关于在为 Microchip 32 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

3.1 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？

3.1.1 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《安装 MPLAB® XC C 编译器并获取许可证》指南（DS50002059L_CN）。其中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。

3.1.2 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 X 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。

从终端或命令行提示符下，使用选项`-status`运行许可证管理器 xclm。该选项会指示许可证管理器打印在系统上安装的所有 MPLAB XC 许可证并退出。例如，在 32 位 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\xc32\v1.00\bin\xclm" -status
```

许可证管理器应运行，打印计算机上可用的所有 MPLAB XC 编译器许可证，并退出。确认您的许可证列为已激活（如 Product:swxc32-pro）。注：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。

3.1.3 是否可以安装同一编译器的多个版本？

可以，编译器和安装过程设计为允许您安装相同编译器的多个版本。对于 MPLAB X IDE，可以通过在 IDE 中更改选项来简便地在版本之间切换（见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，MPLAB XC32 编译器 v1.00 和 v1.10 通常放置在独立的目录中。

```
C:\Program Files\Microchip\xc32\v1.00\
```

```
C:\Program Files\Microchip\xc32\v1.10\
```

3.2 调用编译器

本节讨论如何在命令行上和从 IDE 中运行编译器。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？
- 如何编译库？
- 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 X IDE 中的编译选项的功能？
- MPLAB X IDE 调试编译有何差别？
- 另请参见 如何阻止编译器使用某些特定的存储单元？
- 另请参见 在编译为使用调试器时需要做些什么？
- 另请参见 如何在项目中使用库文件？
- 另请参见 代码将被应用哪些优化？

3.2.1 如何从 MPLAB X IDE 中进行编译？

关于如何设置项目的信息，请参见以下文档：

- 项目设置 ——MPLAB X IDE

3.2.2 如何在命令行上进行编译？

对于所有 32 位器件，编译器驱动程序均名为 `xc32-gcc`；例如，在 Windows 中，它名为 `xc32-gcc.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。

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。驱动程序选项说明详细介绍了驱动程序的命令行选项。输入文件类型列出并介绍了可以传递给驱动程序的文件。

3.2.3 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。多步 C 编译对此进行了介绍。

3.2.4 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。对于 MPLAB X IDE，您可以创建一个项目，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。

在 MPLAB X IDE 中，您可以通过打开 Project Properties（项目属性）窗口（*File（文件） Project Properties*），然后选择 Configuration（配置）类别（`Conf: [default]`）来选择要在编译项目时使用的编译器。最右侧的 Compiler Toolchain（编译器工具链）中将会显示 MPLAB XC32 编译器版本的列表。选择您需要的 MPLAB XC32 编译器。

选定之后，通过选择 XC32（Global Options）（XC32（全局选项））、XC32 编译器和 XC32 链接器类别，将显示对应于该编译器的所有控件。它们会在右侧显示一个选项窗格；每个类别均具有几个窗格，可以从靠近窗格顶部的下拉菜单中进行选择。

3.2.5 如何更改编译器的工作模式？

编译器的工作模式（免费、评估或专业）完全决定了允许的优化级别（见优化），它可以作为一个命令行选项指定（见用于控制优化的选项）。如果在 MPLAB X IDE 下进行编译，则转到 Project Properties 窗

口，单击编译器名称（对于 C 语言项目为 **xc32-gcc**，对于 C++语言项目为 **xc32-g++**），并选择 Optimization（优化）选项类别来设置优化级别——请参见 xc32-g++（32 位 C++编译器）。

在编译项目时，如果您选择了对于许可的工作模式不可用的选项，编译器会发出一条警告消息。编译器将在禁止该选项的情况下继续编译。

3.2.6 如何编译库？

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，也可以将这些模块编译为目标文件，并将它们打包为库归档文件，然后可以与随附的头文件一起编译到应用程序中。

库会较为方便，因为需要管理的文件较少。但是，库需要进行维护。MPLAB XC32 使用*.a 库归档文件。将您的项目移植到新的编译器工具链版本时，请确保重新编译您的库对象。

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。

关于如何创建自己的库的信息，请参见库文件“用户定义的库”小节。

3.2.7 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。另外，本用户指南的驱动程序选项说明中列出了所有选项。如果在 MPLAB X IDE 中进行编译，请参见项目设置。

3.2.8 如何确定 MPLAB X IDE 中的编译选项的功能？

MPLAB X IDE 的 Project Properties 窗口中的大多数窗口小部件和控件以及 XC32 选项，均直接映射到一个命令行驱动程序选项或子选项。关于这些选项和所有相应命令行选项的列表，请参见项目设置。

3.2.9 MPLAB X IDE 调试编译有何差别？

命令行调试编译和 MPLAB X IDE 调试编译之间的主要差别是，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` 等条件编译伪指令（见用于控制预处理器的选项），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。请参见在编译为使用调试器时需要做些什么？。

3.3 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

3.3.1 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？
- 如何将代码移植到不同的器件架构？

3.3.1.1 何时应对表达式进行强制类型转换？

可以使用强制类型转换操作符（类型包含在圆括号中，如（int））对表达式进行显式的强制类型转换。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 short s;

s = 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（short）。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 short，则在以上示例中并不需要、也不应强制转换为 short 类型。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：

```
s = (short)l;
```

该代码以相同的方式工作；但如果在将来，举例来说，s 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。

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。

3.3.1.2 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（见整型提升）。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 何时应对表达式进行强制类型转换？

3.3.1.3 如何在程序中输入非英语字符？

ANSI 标准和 MPLAB XC C 编译器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，请参见常量类型和格式。

3.3.1.4 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`（见静态变量）或 `auto`（见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，您喜欢阅读的 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（见 pragma 伪指令）。

3.3.1.5 如何将代码移植到不同的器件架构？

Microchip 器件具有 3 种基本架构：8 位，它是哈佛架构，具有独立的程序和数据存储器总线；16 位，它是改进型哈佛架构，也具有独立的程序和数据存储器总线；以及 32 位，它是 MIPS 架构。将代码移植到某种架构系列内的不同器件只需对应用程序代码进行最低限度的更新。但是，在架构系列之间进行移植会需要大量的重写。

3.3.2 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？
- 如何访问特殊功能寄存器（SFR）？
- 如何阻止编译器使用某些特定的存储单元？

另请参见链接到其他章节中的如下信息。

在编译为使用调试器时需要做些什么？

3.3.2.1 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 X 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。配置位在源代码中使用 config pragma 伪指令进行设置。关于 config pragma 伪指令的更多信息，请参见配置位访问。

3.3.2.2 如何访问特殊功能寄存器（SFR）？

编译器附带了一些头文件，这些文件定义了一些映射到存储器映射特殊功能寄存器（Special Function Register，SFR）顶部的变量。有些器件文档将它们称为外设寄存器。由于这些映射变量是常规 C 变量，所以可以像任何其他对象一样使用它们，访问基本寄存器无需任何新语法。

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，请参见如何确定用于表示 SFR 和位的名称？。

3.3.2.3 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（如何访问特殊功能寄存器（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。

请查看特定于器件的头文件，通过它们将可以对这些特殊变量进行访问。先搜索数据手册 SFR 名称。如果未找到，则搜索 SFR 表示的对象，因为头文件中的注释通常会说明注释涉及到的宏所执行的操作。

3.3.3 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？

3.3.3.1 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` 属性将变量设为绝对变量（见变量属性）。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。

关于如何移动自动变量的信息，另请参见自动变量分配和访问；关于移动非自动变量的信息，另请参见非自动变量分配；关于移动程序空间变量的信息，另请参见程序存储器中的变量。

3.3.3.2 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` 属性将函数设为绝对函数（见函数属性）。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。

3.3.3.3 如何将变量放入程序存储器？

`const` 限定符意味着所限定的对象是只读的。使用 const 限定的对象也可能放入程序存储器，但其确切位置将取决于以下选项的使用：`-fdata-sections` 和`-mpure-code`、任何链接描述文件定制，以及变量是否已初始化。要确保对象已放入程序存储器，除了 `const` 限定符外，还应使用 `space(prog)`。

3.3.3.4 如何阻止编译器使用某些特定的存储单元？

组合使用 address 属性与 `noload` 属性相连接可用于将存储器的某些段保留。关于变量属性和选项的更多信息，请参见本用户指南中的以下几节：

变量属性

特定于 PIC32C/SAM 器件的选项

关于链接描述文件的详细信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）。

3.3.4 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？
- 如何才能循环移位一个变量？
- 如何确定变量和函数定位到何处？

3.3.4.1 为什么浮点结果与所预期的结果不太相同？

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。在 MPLAB XC32 中，`float` 类型是 32 位浮点型，而 `double` 和 `long double` 类型是 64 位浮点型。

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。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。

3.3.4.2 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` 是否为 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体组成的联合体（见结构体中的位域），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`）。

3.3.4.3 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 MPLAB XC32，没有限制，但要考虑使用其他编译器时的可移植性。

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。

3.3.5 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？

3.3.5.1 函数的最佳大小是多大？

一般来说，函数的源代码长度应尽量保持较小，这有助于可读性和调试。对于执行少量任务的函数，描述和调试函数的操作会容易得多。此外，长度较小的函数具有的副作用通常较少，而副作用可能会成为编码错误的来源。另一方面，在嵌入式编程环境中，大量的小函数和执行它们所需的调用可能会导致使用过多的存储器和堆栈。因此，通常需要做出折衷。

函数大小会导致存储器分页方面的问题，如函数长度限制所述。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。

3.3.5.2 如何防止删除未用函数？

可以对函数应用__attribute__((keep,used))。keep 属性指示编译器将函数放入不会被链接器删除的段，即使使用了--gc-sections 选项来执行垃圾回收。used 属性会阻止编译器应用将删除该函数的优化。此外，它会还禁止与未使用函数相关的警告。

3.3.5.3 如何将函数设为内联函数？

不进行优化时，XC32 编译器不会内联任何函数。

通过将某个函数声明为内联，可以指示 XC32 编译器使对该函数的调用变得更快速。XC32 实现该目标的一种方法是将该函数的代码合并到其调用方的代码中。这可以消除函数调用开销，使执行速度更快；此外，如果任何实际参数值为常量，其已知值可能允许在编译时进行简化，从而不需要包含内联函数的所有代码。对代码长度的影响较难预测；在函数进行内联的情况下，目标代码可能较长也可能较短，这取决于具体情况。

要将某个函数声明为内联，请在其声明中使用 inline 关键字，如下：

```
static inline int
inc (int *a)
{
   return (*a)++;
}
```

当某个函数同时使用 inline 和 static 进行限定时，如果对该函数的所有调用都被合并到调用方中，并且从不使用该函数的地址，则永远不会引用该函数自己的汇编代码。在这种情况下，XC32 不会实际输出该函数的汇编代码。一些调用会由于各种原因而无法进行合并（特别是，无法合并函数的定义之前的调用，也无法合并定义中的递归调用）。如果存在无法合并的调用，则该函数将像正常情况下一样编译为汇编代码。如果程序引用了该函数的地址，则该函数也必须像正常情况下一样进行编译，因为无法对这种情况进行内联。使能优化级别-O1 或更高级别来使能函数内联。

3.3.6 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？

3.3.6.1 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。32 位器件实现了几个独立的中断向量单元，并使用优先级方案。更多信息，请参见中断操作。

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。关于详细信息，请参见允许/禁止中断。

关于所有其他中断相关的任务，包括指定中断向量、现场保护、嵌套和其他注意事项，请参见中断。

3.3.7 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？

3.3.7.1 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，如混合使用汇编语言与 C 变量及函数所述。

如果需要，可以通过使用两种形式的 `asm` 指令将汇编代码内嵌添加到 C 代码中；简单或扩展。使用行内汇编语言给出了这两种形式的说明以及一些示例。

提供了一些宏来内嵌几条简单的指令，如预定义的宏所述。应谨慎使用会更改寄存器内容和器件状态的更复杂行内汇编代码。

关于编译器使用的寄存器，请参见寄存器使用。

3.3.7.2 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。《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）介绍了所有伪指令的操作。下面将介绍两条常用的伪指令。

所有汇编代码必须通过使用 `.section` 伪指令放在段中，以便链接器可以将它作为一个整体处理和放入存储器。更多信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）的“链接器处理”一章。

另一条常用的伪指令是 `.global`，它用于将符号设为可以从多个源文件进行访问。关于该伪指令的更多信息，请参见上述用户指南。

3.3.7.3 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，混合使用汇编语言与 C 变量及函数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。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`（见 volatile 类型限定符）限定。在 C 代码中指定 `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。

3.3.7.4 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。对于 XC32 编译器，可以从预处理的汇编代码或 C/C++代码中使用 xc.h 包含文件。

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。

3.3.7.5 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。

- 您必须将编写的所有汇编代码放入某个段中。更多信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）的“链接器处理”一章。

  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段，不应将它放入单独的段。
- 您必须确保编译器生成的代码未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（见寄存器使用中的寄存器）。行内汇编代码不存在任何假定（见混合使用汇编语言与 C 变量及函数），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。

3.4 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？
- 如何在代码中实现延时？
- 如何才能循环移位一个变量？

3.4.1 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。请参见 volatile 类型限定符。

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它们发生更改时才发生更改，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。编译器会尝试在一次操作中写入 volatile 变量。

3.4.2 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。

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。

通过在 MPLAB X IDE 中使用可装入项目，可以将自举程序的 Hex 文件的内容与应用程序的代码进行合并（关于详细信息，请参见 MPLAB X IDE 文档）。这将产生一个在单个映像中包含自举程序和应用程序代码的 Hex 文件。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。

3.4.3 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 PICkit™ 5 在线调试器或 MPLAB ICD 5 在线调试器）来调试使用 MPLAB XC32 编译器编译的代码。当请求调试运行时，调试执行程序会随程序映像自动下载。该调试执行程序可能会将器件的一些存储器留给自己使用，即您的程序无法使用这些存储器。如果您未在进行调试，只是执行常规的运行、编译项目或清除并编译，则不会使用这些资源。

通常，当调试器连接到 PIC32C/SAM 器件时，调试执行程序不需要使用任何存储器（但连接到 PIC32M 器件时需要）。您的程序不能使用调试器使用的存储器，这一点很重要。

硬件工具在调试时使用的任何存储单元均为 MPLAB X IDE 的属性，其中包含并下载在目标器件上执行的调试执行程序。IDE 在针对调试进行编译时，使用 `-mreserve` 选项来保留调试执行程序所需的任何存储器。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做出任何假设。关于调试器使用哪些资源的完整信息，请查看 MPLAB X IDE **Start page**（起始页）中 **Learn and Discover**（学习和发现）下方的 Reserved Resources（保留资源）信息。

另请参见为什么某些对象被定位到保留的存储器中？。

3.4.4 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 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 32 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。您可以通过查看汇编器列表文件来确定某个变量是否是以原子方式访问的（更多信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN））。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。更多信息，请参见允许/禁止中断。

3.4.5 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（见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？）。

PIC32C/SAM 编译器尚未提供填充存储器功能。

3.4.6 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）。例如，您可以选择将 printf 输出送到 LCD、SPI 模块或 USART。

关于标准 printf 函数的更多信息，请参见 *Microchip Unified Standard Library Reference Guide*。

要使编译器静态分析传递给 printf 函数的格式字符串，可以使用-msmart-io 选项（特定于 PIC32C/SAM 器件的选项）。此外，也可使用-Wformat 选项指定当提供给函数的参数不具有对应于指定格式字符串的类型时产生警告（见用于控制警告和错误的选项）。

如果要创建自己的 printf 类型的函数，需要使用属性 format 和 format_arg，如函数属性所述。

3.4.7 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。

Microchip 建议不要在 PIC32/SAM 器件上使用软件延时，因为有许多变量可以影响时间，例如 L1 高速缓存、预取高速缓存和闪存等待状态的配置。在这些器件上，可以选择使用硬件定时器进行计时。

3.4.8 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 32 位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。

对于以下示例 C 代码：

```
unsigned rotate_left (unsigned a, unsigned s)
{
  return (a << s) | (a >> (32 - s));
}
```

编译器可能会生成类似于以下内容的汇编指令：

```
rotate_left:
  rsb r1, r1, #32
  rors r0, r0, r1
  bx lr
```

3.5 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？
- 编译器支持哪些器件？
- 如何知道编译器生成哪些代码？
- 如何确定某个函数有多大？
- 如何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？
- 如何在项目中使用库文件？
- 如何定制 C 运行时启动代码？
- 如何设置警告/错误消息？
- 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？
- 使用中断时，什么原因会导致变量损坏和代码失败？
- 如何编译库？
- MPLAB X 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？

3.5.1 免费和专业模式之间有何差别？

这些模式（或版本）主要差别在于编译时执行的优化（见优化）。在免费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。在免费或专业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低于专业模式生成的输出代码。

3.5.2 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。关于列表文件的信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）。

使用可能的最小数据类型，因为访问它们需要的代码较少（这也会减少 RAM 使用量）。例如，该编译器支持的 `short` 整型类型。关于所有数据类型和大小，请参见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。在可能的情况下，将浮点型变量替换为整型变量。对于许多应用程序，比例转换变量的值有可能可以消除浮点运算。

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。所以，可以将以上代码重写为：

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

确保使能您的编译器版本允许的所有优化。如果您具有专业版，则可以使用-Os 选项（见用于控制优化的选项）来优化大小。否则，请选择可用的最高优化。

了解编译器会执行哪些优化，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。

3.5.3 如何减少 RAM 使用量？

考虑使用 auto 变量，而不是 global 或 static 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堆栈中，如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。在传递较大结构体时，尤应如此。

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（见程序存储器中的变量）。这可以释放宝贵的 RAM，但会降低执行速度。

3.5.4 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。要实现该目的，请参见如何让代码更小？和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。

根据您的编译器版本，您可能能够使用-O3 选项（见用于控制优化的选项）来优化速度。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。

3.5.5 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段中，然后它们会被定位到代表器件存储器的容器中。要查看对象放置在哪些段中，可以使用选项-ai 查看汇编器列表文件中的该信息。

例外情况是绝对变量，它们会被放置在定义它们时的特定地址处，不放入段。对于设置绝对变量，请使用变量属性下指定的 address() 属性。

3.5.6 如何让中断程序更快？

对于所有中断代码，考虑采用如何让代码更小？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。

除了您在中断服务程序（Interrupt Service Routine，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。该代码是最优的，因为该代码只会保存在 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。

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。关于列表文件的信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）。

避免从 ISR 中调用其他函数。除了函数调用的额外开销之外，编译器还会保存所有通用寄存器，被调用函数可能使用也可能不使用这些寄存器。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 volatile 类型限定符），请参见如何在中断与主干代码之间共享数据？。

如果您的目标器件支持紧耦合存储器（见紧耦合存储器），则会有助于提高 ISR 的确定性和性能。

3.5.7 C 变量最大可以多大？

这个问题特别涉及到各个 C 对象（如数组或结构体）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；其他对象将被放入数据存储器。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，非自动变量分配和非自动变量长度限制分别讨论这些对象的大小限制。

3.5.8 代码将被应用哪些优化？

可用的代码优化取决于编译器的版本（见优化）。用于控制优化的选项给出了优化选项的说明。

3.5.9 编译器支持哪些器件？

通常每个编译器版本都会添加一些新器件。关于编译器版本支持的器件的完整列表，请查看自述文件。

通常，当适用于新器件的器件系列包（Device Family Pack，DFP）上线后，无需更新编译器即可添加对新器件的支持。DFP 可以从 MPLAB X IDE 中进行下载。另请参见 Dfp 选项。

3.5.10 如何知道编译器生成哪些代码？

可以使用汇编器列表文件选项设置汇编列表文件，使之包含关于代码的大量信息，例如几乎整个程序的汇编输出，包括链接到程序中的库程序、段信息和符号列表等。

列表文件可使用如下方式生成：

- 在命令行上，使用选项创建一个基本列表文件：
  `-Wa, -a=projectname.lst`。
- 对于 MPLAB X IDE，右键单击您的项目并选择“Properties”（属性）。在 Project Properties 窗口中，在“Categories”（类别）下单击“xc32-as”。从“Option categories”（选项类别）中，选择“Listing file options”（列表文件选项），并确保选中“List to file”（列表至文件）。

默认情况下，汇编列表文件将具有 `.lst` 扩展名。

关于列表文件的信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）。

3.5.11 如何确定某个函数有多大？

可以通过汇编列表文件确定某个函数的这种大小（对于该函数生成的汇编代码量）。关于创建汇编列表文件的更多信息，请参见如何知道编译器生成哪些代码？。

3.5.12 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。关于创建汇编列表文件的更多信息，请参见如何知道编译器生成哪些代码？。

要查看关于函数调用的信息，可以在 MPLAB X IDE 中查看调用图（*Window > Output > Call Graph*（窗口 > 输出 > 调用图））。只有处于调试模式下时，才能查看该图。右键单击某个函数并选择“Show Call Graph”（显示调用图）可查看哪些函数调用了该函数以及它调用了哪些函数。

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，请参见自动变量分配和访问。

3.5.13 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（由汇编器生成）或映射文件（由链接器生成）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）。

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。与某个变量关联的符号将被分配该变量最低字节的地址；对于函数，它是对于该函数生成的第一条指令的地址。

关于汇编列表文件和链接器映射文件的信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）。

3.5.14 为什么某些对象被定位到保留的存储器中？

大多数变量和函数都会被放到链接描述文件中定义的段中。关于链接描述文件的详细信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）。但是，一些变量和函数会被明确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，如 3.3.3.1 “如何将变量定位到指定的地址处？”和 3.3.3.2 “如何将函数定位到指定的地址处？”所述。

检查汇编列表文件，确定包含对象和代码的段的名称。检查映射文件中的链接器选项，确定是否已显式地链接这些段，或者是否在某个类中将它们链接到任意位置。关于其中每个文件的信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）。

3.5.15 如何知道还有多少存储器可用？

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

此外，还提供了 Memory Report by Module（按模块划分的存储器报告），以显示每个目标文件的存储器使用情况（text、data 和 bss 段）。该报告中显示最终 ELF 文件中每个输入目标的大小。此外，还有一个其他条目会显示无法确定输入目标文件或无法确定存储器类型的段。

要确定每个链接器类中究竟哪些存储器仍然可用，请查看链接器映射文件。如果存在存储器分页，则该文件还会指示该类中最大的连续块。关于映射文件的信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）。

3.5.16 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。

例如：

```
xc32-gcc -mprocessor=ATSAME70J19B main.c int.c mylib.a
```

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。

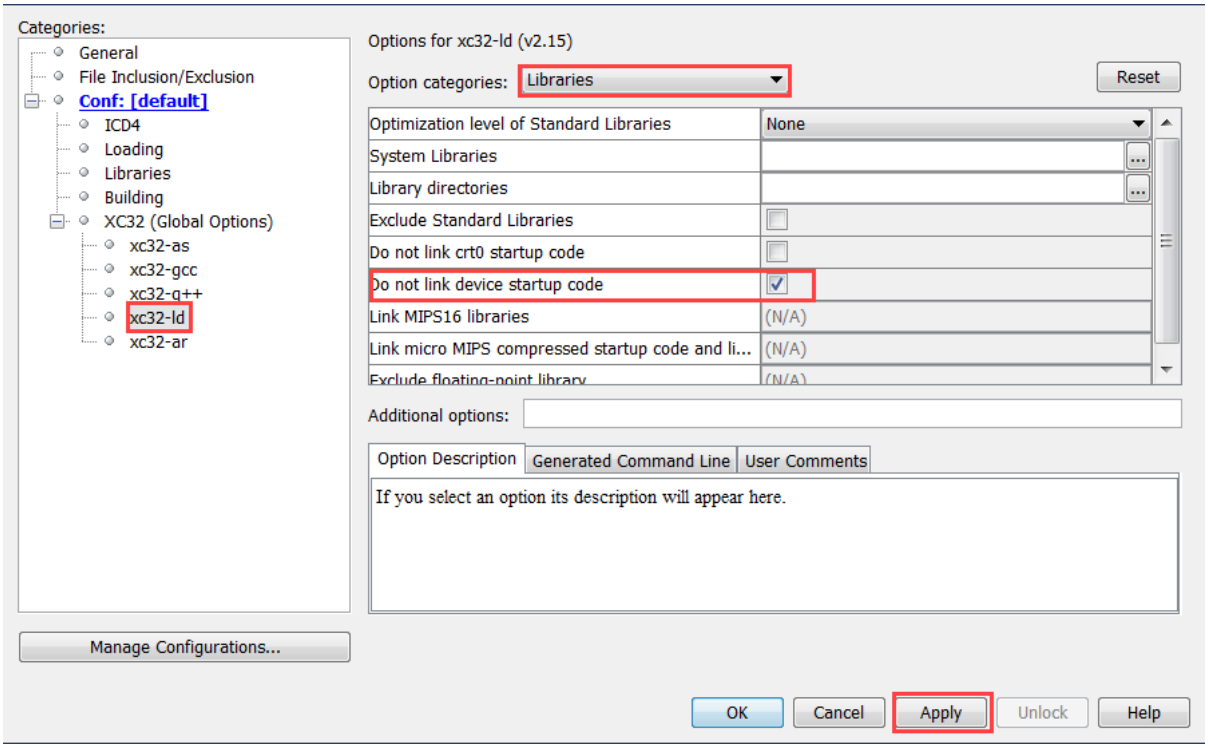
3.5.17 如何定制 C 运行时启动代码？

一些应用程序可能需要定制版的 C 运行时启动代码。例如，通过自举程序加载的应用程序可能就需要修改启动代码。

要针对应用程序定制启动代码，请按照以下步骤操作：

1. 首先从`<install-directory>/pic32c/lib/proc/<device>/startup_<device>.c` 路径下复制一份默认启动代码。该文件也可以从器件系列包（DFP）中获取。
   复制该 `.c` 文件后，对其进行重命名并添加到项目中。
2. 更改 MPLAB X 项目以排除默认启动代码，方法是在链接时将`-mno-device-startup-code` 选项传递给 xc32-gcc 驱动程序。该选项在 MPLAB X 项目属性的 Libraries（库）类别中的 xc32-ld 对应的 Options（选项）下以“Do not link device startup code”（不链接器件启动代码）的形式提供。编译项目时，MPLAB X 将编译新的定制版启动代码，而不是链接默认启动代码。

图 3-1. 启动代码属性设置



3.6 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。

- 如何设置警告/错误消息？
- 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？
- 使用中断时，什么原因会导致变量损坏和代码失败？
- 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？

3.6.1 如何设置警告/错误消息？

要控制消息输出，请参见用于控制警告和错误的选项。

3.6.2 如何在程序中查找导致编译器错误或警告的代码？

在大多数情况下，如果错误是与源代码相关的语法错误，则编译器产生的消息会指示引起问题的代码行。如果在 MPLAB 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，直到它扫描下一条语句之后才能确定。假设存在以下代码：

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

赋值语句缺少分号这一错误将会标记在包含 `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。

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。XC32 工具链和 MPLAB X IDE 中给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。关于链接器尝试将对象定位到何处的信息，请参见映射文件。关于列表和映射文件的信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）。

3.6.3 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。请总是检查您的代码，以确认它不是可能的错误源。在许多情况下，代码是有效的，警告是多余的。在这种情况下，您可以：

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

关于详细信息，请参见用于控制警告和错误的选项。

3.6.4 为什么甚至无法使 LED 闪烁？

即使您已经设置了端口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，如配置位访问所述。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。请参见配置位访问和您的器件数据手册。
- 要确保器件不会由于看门狗时间而复位，请在配置位中关闭定时器或在您的代码中清零定时器。您可以使用一些库函数来处理看门狗定时器，如“*32-bit Language Tool Libraries*”手册（DS51685）中所述。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。

3.6.5 使用中断时，什么原因会导致变量损坏和代码失败？

这通常是由于中断和主干代码同时使用变量而引起的。如果编译器优化了对某个变量的访问，或者访问被中断程序中断，则可能会发生损坏。更多信息，请参见如何在中断与主干代码之间共享数据？。

4. XC32 工具链和 MPLAB X IDE

32 位语言工具可以与 MPLAB X IDE 配合使用，为 PIC32 MCU 器件系列的应用程序代码提供 GUI 开发。这些工具包括：

- MPLAB XC32 C/C++编译器
- MPLAB XC32 汇编器
- MPLAB XC32 目标链接器
- MPLAB XC32 目标归档器/库管理器和其他 32 位实用程序

4.1 MPLAB X IDE 和工具安装

为了配合使用 32 位语言工具与 MPLAB X IDE，必须安装：

- MPLAB X IDE，它可从 Microchip 网站上免费获取。
- MPLAB XC32 C/C++编译器，它包含所有 32 位语言工具。该编译器可从 Microchip 网站上免费获取（免费版与评估版）或购买（专业版）。

注意：该 C 编译器版本需要与 MPLAB X IDE v5.50 或更高版本配合使用。

默认情况下，32 位语言工具将安装在以下目录中：

- Window 操作系统—— `C:\Program Files\Microchip\xc32\x.xx`
- Mac 操作系统—— `/Applications/microchip/xc32/x.xx`
- Linux 操作系统—— `/opt/microchip/xc32/x.xx`

其中，`x.xx` 是版本号。

每个工具的可执行文件将位于 `bin` 子目录中：

- C 编译器—— `xc32-gcc.exe`
- 汇编器—— `xc32-as.exe`
- 目标链接器—— `xc32-ld.exe`
- 目标归档器/库管理器—— `xc32-ar.exe`
- 其他实用程序—— `xc32-utility.exe`

所有器件包含（头）文件均位于`/pic32c/include/proc` 子目录中。`#include <xc.h>`头文件时，会自动包含这些文件。

代码示例位于 examples 目录中。

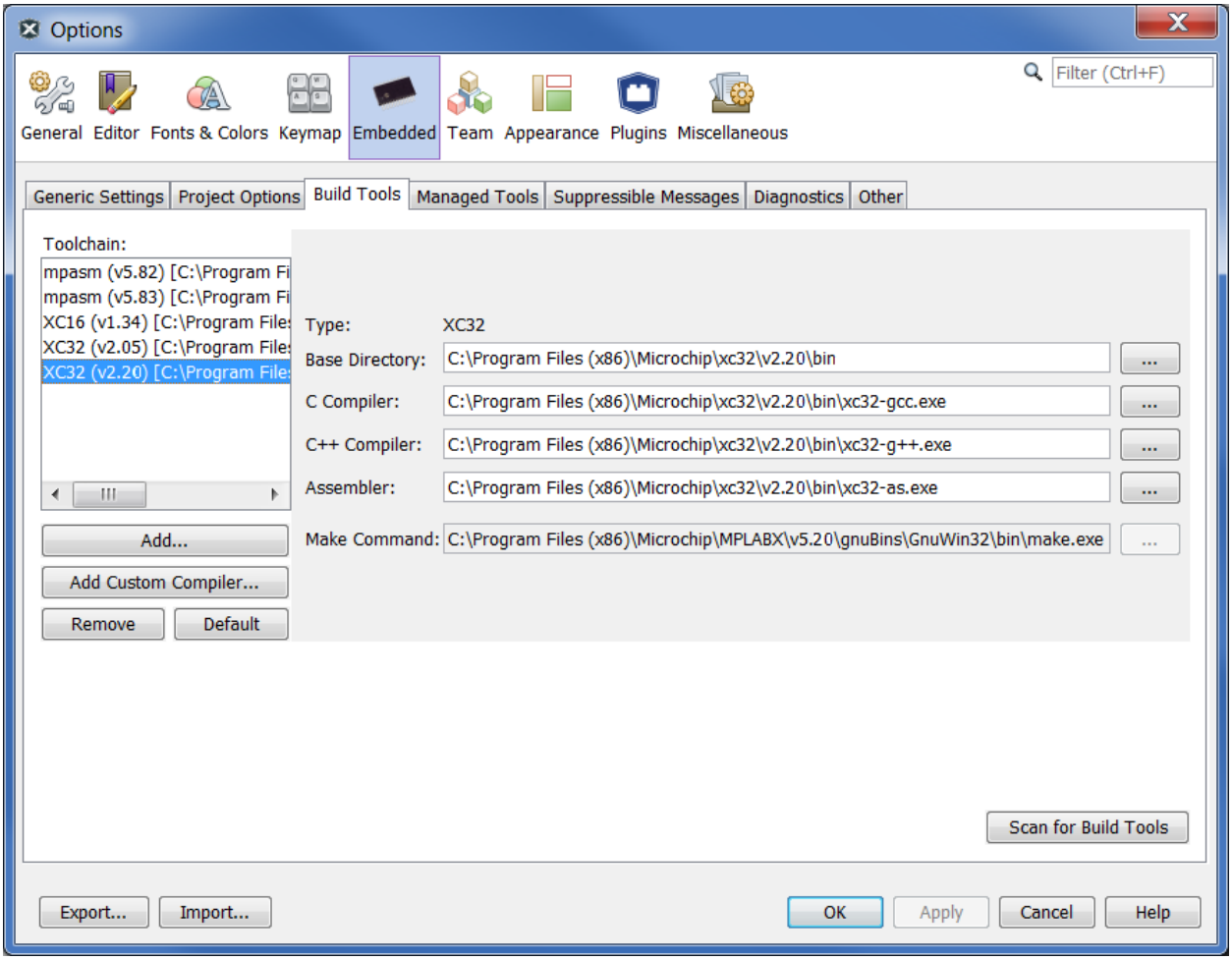
4.2 MPLAB X IDE 设置

在主机上安装 MPLAB X IDE 之后，启动该应用程序，并检查下面的设置，以确保正确识别 32 位语言工具。

1. 从 MPLAB X IDE 菜单栏中，选择 *Tools > Options*（工具 > 选项）打开 Options 对话框。单击 **Embedded** 按钮，并选择 **Build Tools**（编译工具）选项卡。
2. 确保安装的 MPLAB XC32 编译器版本已在 **Toolchain：**（工具链：）下列出。
3. 选择所需的 XC32 工具，确保安装的路径正确。

4. 单击 **OK**（确定）。

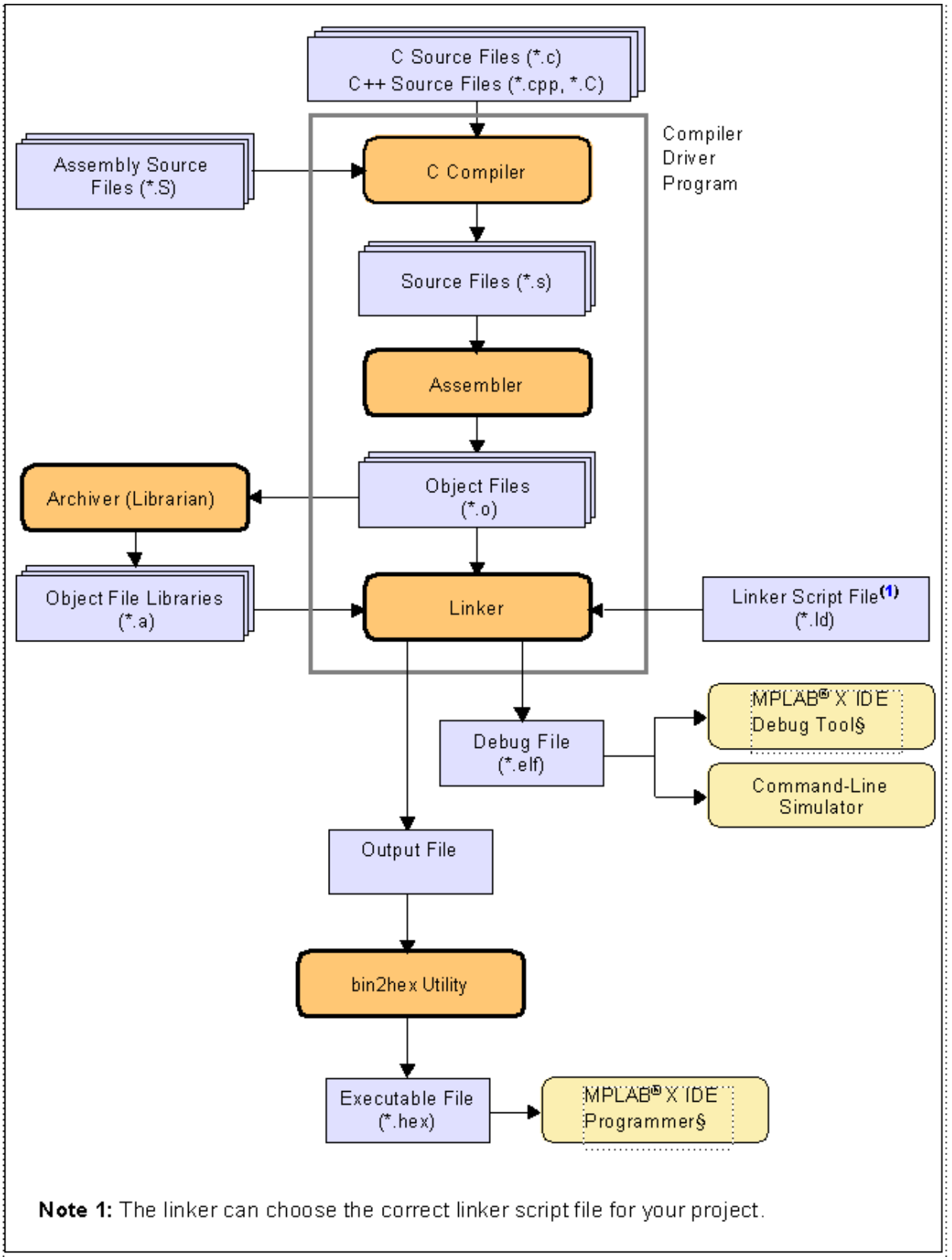
图 4-1. Windows®操作系统中的 XC32 工具套件位置



4.3 MPLAB X IDE 项目

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

图 4-2. 编译器项目关系



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

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

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

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

关于项目和相关工作区的更多信息，请参见 MPLAB X IDE 文档。

4.4 项目设置

要首次设置 MPLAB X IDE 项目，请使用内置的 Project Wizard（项目向导）（*File > New Project*（文件 > 新建项目））。在该向导中，可以选择使用 32 位语言工具的语言工具套件。关于该向导和 MPLAB X IDE 项目的更多信息，请参见 MPLAB X IDE 文档。

设置项目之后，可以在 MPLAB X IDE 中设置工具的属性。

1. 从 MPLAB X IDE 菜单栏中，选择 *File > Project Properties*（文件 > 项目属性）打开一个窗口来设置/检查项目编译选项。
2. 在“Conf:[*default*]”下，从工具集合中选择要设置的工具。

4.4.1 XC32 (Golbal Options)（XC32（全局选项））

设置所有 32 位语言工具的全局选项。另请参见“选项页面功能”。

表 4-1. XC32 (Global Options) (All Options)（所有选项）类别

选项	说明	命令行

Stack Smashing Protector（堆栈溢出保护器）	选择受堆栈溢出保护的函数。	`-fstack-protector`
		`-fstack-protector-strong`
		`-fstack-protector-all`

Override default device support（改写默认器件支持）	**Do not override**（不改写）选项用于让 MPLAB® X IDE 提供其自己的可选 DFP 列表。**Compiler location**（编译器位置）将使用编译器（而非 IDE）随附的 DFP。	`-mdfp`
Don't delete intermediate files（不要删除中间文件）	不要删除中间文件。将它们放在目标目录中，并基于源文件命名它们。	`-save-temps=obj`
Use Whole-Program and Link-Time Optimizations（使用全程序和链接时优化）	使能该功能时，编译会被限制为以下方式： - 每文件编译设置将被忽略 - 编译将不再为增量编译（仅完全编译）	`-fwhole-program` `-flto`
Common include dirs（公共包含目录）	在此处输入的目录路径会被追加到编译器现有包含路径后。 相对路径是相对于 MPLAB X IDE 项目目录。	`-I dir`
Data TCM size in bytes（数据 TCM 大小（字节））	使能指定大小的数据紧耦合存储器。	`-mdtcm=n`，其中 *n* 是用户指定的大小
Instruction TCM size in bytes（指令 TCM 大小（字节））	使能指定大小的指令紧耦合存储器。	`-mitcm=n`
Locate stack in data TCM（将堆栈定位到数据 TCM 中）	将堆栈定位到数据紧耦合存储器中。	`-mstack-in-tcm`

4.4.2 xc32-as（32 位汇编器）

可以在 MPLAB X IDE 中指定命令行选项的一个子集。选择某个类别，然后设置汇编器选项。关于其他选项，请参见 MPLAB XC32 汇编器文档。另请参见“选项页面功能”。

表 4-2. xc32-ld General（常规）类别

选项	说明	命令行
Have symbols in production build（获取生产编译中的符号）	生成用于在 MPLAB® X 中进行源代码级调试的调试信息。	`--gdwarf-2`
Keep local symbols（保留局部符号）	选中则保留局部符号，即以.L（仅大写）开头的标号。 取消选中则丢弃局部符号。	`--keep-locals` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Preprocessor macro definitions（预处理器宏定义）	通过编译器的 `-D` 选项传递的特定于项目预处理器宏定义。	`-Dmacro[=defn]`，请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Assembler symbols（汇编器符号）	将符号“sym”定义为给定值“value”。	`--defsym sym=value` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Preprocessor Include directories（预处理器包含目录）	相对路径是相对于 MPLAB X 项目目录。	`-I dir`，请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Assembler Include directories（汇编器包含目录）	相对路径是相对于 MPLAB X 项目目录。 将某个目录添加到汇编器搜索 `.include` 伪指令中指定文件的目录列表中。 您可以根据需要添加任意多个目录，以包含一系列路径。当前的工作目录是总是最先搜索，然后是 `-I` 目录，按此处指定它们的顺序（从左到右）进行搜索。	`-I dir` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。

表 4-3. xc32-as Other Options（其他选项）类别

选项	说明	命令行
Diagnostics level（诊断级别）	选择要在 Output（输出）窗口中显示的警告。选择“Generate warnings”（生成警告）可使编译器发出通常情况下的警告；选择“Suppress warnings”（禁止警告）可仅显示错误；选择“Fatal Warnings”（致命警告）可使汇编器将警告视为错误。	`--[no-]warn` `--fatal-warnings`，请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Include source code（包含源代码）	选中则产生高级语言列表。高级列表要求由编译器生成汇编源代码、向编译器传递诸如 `-g` 之类的调试选项，并请求产生汇编列表（`-al`）。 取消选中则产生普通列表。	`-ah` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Expand macros（展开宏）	选中则展开列表中的宏。 取消选中则折叠宏。	`-am` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Include false conditionals（包含虚假条件语句）	选中则在列表中包含虚假条件语句（`.if` 和 `.ifdef`）。 取消选中则省略虚假条件语句。	`-ac` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Omit forms processing（省略表单处理）	选中则关闭由列表伪指令 `.psize`、`.eject`、`.title` 和 `.sbttl` 执行的所有表单处理。 取消选中则由列表伪指令进行处理。	`-an` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。

.....（续）		
选项	**说明**	**命令行**
Include assembly（包含汇编）	选中则包含汇编列表。该-a 子选项可与其他子选项配合使用。 取消选中则排除汇编列表。	-al 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
List symbols（列表符号）	选中则包含符号表列表。 取消选中则从列表中排除符号表。	-as 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Omit debugging directives（省略调试伪指令）	选中则从列表中省略调试伪指令。这可以使列表更简洁。 取消选中则包含调试伪指令。	-ad 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
List to file	如果需要项目中任何汇编源文件的汇编列表，则使用该选项。 汇编列表的基本名称与源文件相同，扩展名为.lst。	-a=*file*.lst 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。

4.4.3 xc32-gcc（32 位 C 编译器）

虽然 MPLAB XC32 C/C++编译器与 MPLAB X IDE 配合工作，但它必须单独购买。可以购买完整版本，也可以免费下载学生（有限功能）版本。关于详细信息，请访问 Microchip 网站（www.microchip.com）。

可以在 MPLAB X IDE 中指定命令行选项的一个子集。选择某个类别，然后设置编译器选项。关于其他选项，请参见《MPLAB® X IDE 用户指南》（DS50002027E_CN），它也可从 Microchip 网站上获取。

另请参见选项页面功能。

表 4-4. xc32-gcc General 类别

选项	说明	命令行
Enable unaligned access（使能未对齐访问）	使能（或禁止）从非 16 位或 32 位对齐的地址读取和写入 16 位和 32 位值。默认情况下，所有 Armv6 之前和所有 Armv6-M 架构均禁止未对齐访问，所有其他架构均使能未对齐访问。如果未使能未对齐访问，则连续存放数据结构中的字将一次访问一个字节。	-m[no-]unaligned-access
Have symbols in production build	向生成的 ELF 文件中添加调试信息。	-g
Isolate each function in a section（将每个函数隔离在一个段中）	该选项通常与链接器的--gc-sections 选项配合使用，以删除未被引用的函数（请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》）。 选中则在输出文件中将每个函数放入它自己的段。函数的名称决定输出文件中段的名称。 **注：**指定该选项时，汇编器和链接器可能会创建更大的目标文件和可执行文件，执行速度会较慢。 取消选中则将多个函数放在一个段中。	-ffunction-sections
Place data into its own section（将数据放入它自己的段中）	该选项通常与链接器的--gc-section 选项配合使用，以删除未被引用的静态分配的变量（请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》）。 在输出文件中将每个数据项放入它自己的段中。 数据项的名称决定段的名称。指定该选项时，汇编器和链接器可能会创建更大的目标文件和可执行文件，速度也会较慢。	-fdata-sections
Enable toplevel reordering（使能顶层重新排序）	使能重新排序顶层函数、变量和 asm 语句，所以它们可能不会按照在源文件中出现的顺序输出。禁止此功能时，如果使用-fno-toplevel-reorder 选项，则未引用的静态变量不会被删除。将此选项与优化级别 1 或更高级别结合使用。	-f[no-]toplevel-reorder

.....（续）		
选项	**说明**	**命令行**
Use indirect calls（使用间接调用）	使能全范围的调用，在调用另一个存储器区域中的函数时通常需要此选项。	`-mlong-calls`

请注意，除 Optimization（优化）之外的项目属性类别中的字段指定的某些编译器选项可能会影响项目的大小和执行速度。当使用不同的编译器选项组合进行编译时，可以考虑使用 Compiler Advisor（可通过 MPLAB X IDE**Tools** > **Analysis** > **Compiler Advisor**（工具 > 分析 > Compiler Advisor）菜单项进行访问）来比较项目的大小。

表 4-5. xc32-gcc Optimization 类别

选项	说明	命令行
Optimization Level（优化级别）	选择优化级别。您的编译器版本可能仅支持某些优化。等效于 `-On` 选项，其中的 *n* 是以下选项之一： 0 ——不优化。编译器的目标是降低编译开销，以及使调试产生预期的结果。 1 ——优化。优化编译需要的时间稍长一些，对于大函数需要更多的主机存储器。编译器会尝试减小代码长度和执行时间。 2 ——更多优化。编译器执行几乎所有不涉及空间-速度权衡的受支持优化。 3 ——优化但更倾向于速度（O2 的超集）。 s ——针对长度优化。此级别会使能通常不会增加代码长度的所有 `-O2` 优化。它还会执行旨在减小代码长度的进一步优化。	`-O0 -O1 -O2 -O3 -Os`
Unroll loops（展开循环）	该选项通常可以提高执行速度，代价是代码长度较大。 选中则执行循环展开优化。这仅仅针对可以在编译时或运行时确定迭代次数的循环。 取消选中则不展开循环。	`-funroll-loops`
Omit frame pointer（省略帧指针）	选中时，对于不需要帧指针的函数，不会在寄存器中保留帧指针。 取消选中则保留帧指针。	`-fomit-frame-pointer`
Pre-optimization instruction scheduling（预优化指令调度）	尝试重新排序指令以消除指令停顿。选择“Default for optimization level”（优化级别的默认设置），以使此功能完全由 `-O` 级别选择控制。	`-f[no-]schedule-insns`
Post-optimization instruction scheduling（后优化指令调度）	尝试重新排序指令以消除指令停顿。选择“Default for optimization level”，以使此功能完全由 `-O` 级别选择控制。	`-f[no-]schedule-insns2`
Use Common Tentative Definition（使用公共暂定定义）	控制在无初始化情况下定义的全局变量的位置，在 C 标准中称为暂定定义。暂定定义不同于使用 `extern` 关键字进行的变量声明，后者不分配存储空间。 当使能该功能（使用 `-fcommon` 选项）时，将在公共存储器块中为未初始化的全局对象分配存储空间。这样，链接器便可跨翻译单元解析同一对象的暂定定义和非暂定定义。 当禁止该功能（使用 `-fno-common` 选项）时，编译器会将未初始化的全局对象放入相应目标文件的数据段中。这样做将阻止链接器跨翻译单元关联暂定定义，否则在遇到此类定义时会导致多重定义错误。	`-f[no-]common`

表 4-6. xc32-gcc Preprocessing and Messages（预处理和消息）类别

选项	说明	命令行
Preprocessor macros（预处理器宏）	通过编译器的 `-D` 选项传递的特定于项目预处理器宏定义。	*`-Dmacro[=defn]`*
Include directories（包含目录）	搜索这些目录来查找特定于项目的包含文件。	*`-Ipath`*

.....（续）		
选项	**说明**	**命令行**
Make warnings into errors（使警告变为错误）	选中则基于警告以及错误来暂停编译。 取消选中则仅基于错误来暂停编译。	`-Werror`
Additional warnings（额外的警告）	选中则使能所有警告。 取消选中则禁止警告。	`-Wall`
Enable address warning attribute（使能地址警告属性）	对 `address` 属性的所有使用发出警告。该选项仅供工程支持使用。	`-Waddress-attribute-use`
support-ansi	选中则发出严格 ANSI C 要求的所有警告。 取消选中则发出所有警告。	`-ansi`
strict-ansi	发出严格 ISO C 和 ISO C++要求的所有警告；拒绝所有使用已禁止扩展的程序，以及其他一些不遵循 ISO C 和 ISO C++的程序。	`-pedantic`

4.4.4 xc32-g++（32 位 C++编译器）

可以在 MPLAB X IDE 中指定命令行选项的一个子集。选择某个类别，然后设置链接器选项。关于其他选项，请参见用于 32 位器件 MPLAB 目标链接器的文档。另请参见选项页面功能。

表 4-7. xc32-g++ c++ specific（C++特定）类别

选项	说明	命令行
Generate run time type descriptor information（生成运行时类型描述符信息）	使能生成每个具有虚拟函数的类的信息，供 C++运行时类型识别功能（“dynamic_cast”和“typeid”）使用。如果不使用语言的这些部分，可以通过禁止该选项来节省一些空间。请注意，异常处理程序使用相同的信息，但它将根据需要生成它。 “dynamic_cast”操作符仍然可以用于不需要运行时类型信息的强制类型转换，即，强制类型转换为 void *或明确的基类。	`-f[no-]rtti`
Enable C++ exception handling（使能 C++异常处理）	使能异常处理。生成传播异常所需的额外代码。	`-f[no-]exceptions`
Check that the pointer returned by operator 'new' is non-null（检查操作符 new 返回的指针是否为非空）	在尝试修改所分配的存储空间之前，检查操作符 new 返回的指针是否为非空。	`-fcheck-new`
Generate code to check for violation of exception specification（生成用以检查是否违反异常规范的代码）	生成用于在运行时检查是否违反异常规范的代码。使用此选项可能会增加生产编译中的代码长度。	`-fenforce-eh-specs`

表 4-8. xc32-g++ General 类别

选项	说明	命令行
Enable unaligned access	使能（或禁止）从非 16 位或 32 位对齐的地址读取和写入 16 位和 32 位值。默认情况下，所有 Armv6 之前的所有架构和所有 Armv6-M 架构均禁止未对齐访问，所有其他架构均使能未对齐访问。如果未使能未对齐访问，则连续存放数据结构中的字将一次访问一个字节。	`-m[no-]unaligned-access`
Have symbols in production build	针对在生产编译映像中进行调试而编译。	`-g`
Isolate each function in a section	如果目标支持任意段，则在输出文件中将每个函数放入它自己的段。函数的名称或数据项的名称决定输出文件中段的名称。该选项可用于与链接器的`--gc-sections`选项配合使用，以删除未被引用的函数。	`-ffunction-sections`

.....（续）

选项	说明	命令行
Place data into its own section	如果目标支持任意段，则在输出文件中将每个数据项放入它自己的段。函数的名称或数据项的名称决定输出文件中段的名称。该选项可用于与链接器的--gc-sections 选项配合使用，以删除未被引用的变量。	-fdata-sections
Enable toplevel reordering	使能重新排序顶层函数、变量和 asm 语句，所以它们可能不会按照在源文件中出现的顺序输出。禁止此功能时，如果使用-fno-toplevel-reorder 选项，则未引用的静态变量不会被删除。将此选项与优化级别 1 或更高级别结合使用。	-f[no-]toplevel-reorder
Use indirect calls	使能全范围的调用，在调用另一个存储器区域中的函数时通常需要此选项。	-mlong-calls

表 4-9. xc32-g++ Optimization 类别

选项	说明	命令行
Optimization Level	选择优化级别。您的编译器版本可能仅支持某些优化。等效于-O*n* 选项，其中的 *n* 是以下选项之一： 0——不优化。编译器的目标是降低汇编开销，以及使调试产生预期的结果。 1——优化。优化编译需要的时间稍长一些，对于大函数需要更多的主机存储器。编译器会尝试减小代码长度和执行时间。 2——更多优化。编译器执行几乎所有不涉及空间-速度权衡的受支持优化。 3——优化但更倾向于速度（O2 的超集）。 S——优化但更倾向于长度（O2 的超集）。	-O0 -O1 -O2 -O3 -Os
Unroll loops	选中则执行循环展开优化。这仅仅针对可以在编译时或运行时确定迭代次数的循环。 取消选中则不展开循环。	-funroll-loops
Omit frame pointer	选中时，对于不需要帧指针的函数，不会在寄存器中保留帧指针。 取消选中则保留帧指针。	-fomit-frame-pointer
Pre-optimization instruction scheduling	尝试重新排序指令以消除指令停顿。选择“Default for optimization level”，以使此功能完全由-O 级别选择控制。	-f[no-]schedule-insns
Post-optimization instruction scheduling	尝试重新排序指令以消除指令停顿。选择“Default for optimization level”，以使此功能完全由-O 级别选择控制。	-f[no-]schedule-insns2

表 4-10. xc32-g++ Preprocessing and Messages（预处理和消息）类别

选项	说明	命令行
Preprocessor macros	通过编译器的 -D 选项传递的特定于项目预处理器宏定义。	-D*macro*[=*defn*]
Include directories	搜索这些目录来查找特定于项目的包含文件。	-I *dir*
Make warnings into errors	选中则基于警告以及错误来暂停编译。 取消选中则仅基于错误来暂停编译。	-Werror
Additional warnings	选中则使能所有警告。 取消选中则禁止警告。	-Wall
Enable Address-attribute warning	对 address() 属性的所有使用发出警告。该选项仅供工程支持使用。	-Waddress-attribute-use
strict-ansi	发出严格 ISO C 和 ISO C++要求的所有警告；拒绝所有使用已禁止扩展的程序，以及其他一些不遵循 ISO C 和 ISO C++的程序。	-pedantic

4.4.5 xc32-ld（32 位链接器）

可以在 MPLAB X IDE 中指定命令行选项的一个子集。选择某个类别，然后设置链接器选项。关于其他选项，请参见用于 32 位器件 MPLAB 目标链接器的文档。另请参见选项页面功能。

表 4-11. xc32-ld General 类别

选项	说明	命令行
Heap Size (bytes)（堆大小（字节））	指定堆的大小，以字节为单位。分配指定字节大小运行时堆，供 C 程序使用。堆从未用数据存储器中分配。如果没有足够的存储器可用，则会报告错误。	`--defsym=_min_heap_size=<size>` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Minimum stack size (bytes)（最小堆栈大小（字节））	指定堆栈的最小大小，以字节为单位。默认情况下，链接器会分配所有未用的数据存储器作为运行时堆栈。使用该选项可以确保至少有最小大小的堆栈可用。链接映射输出文件中会报告实际堆栈大小。如果没有最小大小可用，则会报告错误。	`--defsym=_min_stack_size=<size>` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Allow overlapped sections（允许重叠的段）	勾选以忽略段地址的重叠。 此功能仅用于诊断目的，应仅用于诊断与段分配问题相关的链接器错误。取消选中则检查是否有重叠。	`--[no-]check-sections` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Remove unused sections（删除未用的段）	选中以使能对未用输入段的垃圾回收（在某些目标器件上）。这通常与将数据/函数放入其自己的段的控制一起使用。 取消选中则禁止垃圾回收。	`--[no-]gc-sections` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Use response file to link（使用响应文件进行链接）	在一个文件中而不是在命令行上传递链接器选项。在 Windows®系统上，该选项使您可以正确地链接具有大量目标文件的项目，而通常情况下大量的目标文件会超出 Windows 操作系统的命令行长度限制。	
Write start linear address record（写入起始线性地址记录）	使用 ELF 文件的入口点字段将起始线性地址（Start Linear Address，SLA）记录（类型 0x05）写入十六进制文件。对于需要确定应用程序入口点的自举程序而言，十六进制记录可能很有用。可以使用 `ENTRY` 命令行链接描述文件设置该值。	`--write-sla` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Additional driver options（额外的驱动程序选项）	在此处输入 Project Properties 对话框中没有专用 GUI 小部件的任何额外驱动程序选项。此处输入的字符串将与其他驱动程序选项一起逐字发出。	

表 4-12. xc32-ld Libraries 类别

选项	说明	命令行
Optimization level of Standard Libraries（标准库的优化级别）	选择编译库时使用的优化级别。您的编译器版本可能仅支持某些库优化。等效于编译的链接阶段发出的`-On`选项，其中的 *n* 是以下选项之一： `0` ——不优化。编译器的目标是降低编译开销，以及使调试产生预期的结果。 `1` ——优化。优化编译需要的时间稍长一些，对于大函数需要更多的主机存储器。编译器会尝试减小代码长度和执行时间。 `2` ——更多优化。编译器执行几乎所有不涉及空间-速度权衡的受支持优化。 `3` ——优化但更倾向于速度（`-O2` 的超集）。 `s` ——针对长度优化。此级别会使能通常不会增加代码长度的所有`-O2` 优化。它还会执行旨在减小代码长度的进一步优化。	`-On` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
System Libraries（系统库）	添加要与项目文件链接的库。您可以添加多个。	`--library=name` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Library directories（库目录）	将某个库目录添加到库搜索路径中。您可以添加多个。	`--library-path="name"` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Exclude Standard Libraries（排除标准库）	选中则在链接时不使用标准系统启动文件或库。仅使用在命令行上指定的库目录。 取消选中则使用标准系统启动文件和库。	`-nostdlib` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。

.....（续）		
选项	**说明**	**命令行**
Do not link crt0 startup code（不链接 crt0 启动代码）	排除默认启动代码，因为项目提供了特定于应用程序的启动代码。	`-nostartfiles` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Do not link device startup code（不链接器件启动代码）	不链接默认的器件特定启动代码（例如，`startup_device.c`）	`-mno-device-startup-code` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。

表 4-13. xc32-ld Diagnostics（诊断）类别

选项	说明	命令行
Generate map file（生成映射文件）	创建映射文件。	`-Map="file"` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Display memory usage（显示存储器使用情况）	选中则打印存储器使用情况报告。 取消选中则不打印报告。	`--report-mem` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Generate cross-reference file（生成交叉引用文件）	选中则创建交叉引用表。 取消选中则不创建该表。	`--cref` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Warn on section realignment（产生关于段重新对齐的警告）	选中则在由于对齐而导致某个段的起始位置发生变化时打印警告。此功能仅用于诊断目的，应仅用于诊断段对齐问题。 取消选中则禁止警告。	`--warn-section-align` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Trace Symbols（跟踪符号）	添加/删除跟踪符号。	`-Y symbol` 或 `--trace-symbol symbol` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。

表 4-14. xc32-ld Symbols and Macros（符号和宏）类别

选项	说明	命令行
Linker symbols（链接器符号）	在输出文件中创建包含绝对地址的全局符号（*expr*）。您可以根据需要在命令行中任意多次地使用该选项来定义多个符号。对于该上下文中的 *expr*，支持一种有限形式的算术运算：您可以给出一个十六进制常数或现有符号的名称，或者使用+和-来对十六进制常数或符号进行加减运算。	`--defsym=sym` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Preprocessor macro definitions	定义用于链接描述文件的预处理器宏（当选项直接传递给 xc32-gcc 时）。	`-Dmacro` 请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。
Symbols（符号）	指定输出中的符号信息。	`-S` 或 `--strip-debug`; `-s` 或 `--strip-all`，请参见《*MPLAB® XC32 汇编器、链接器和实用程序用户指南*》。

4.4.6 分析

选择某个类别，然后设置分析选项。

另请参见选项页面功能。

表 4-15. 分析 General 选项类别

选项	说明	命令行
Code coverage instrumentation（代码覆盖插装）	使能代码覆盖功能	`-mcodecov`

.....（续）		
选项	**说明**	**命令行**
Stack guidance（堆栈指导）	选中以使能堆栈指导功能，该功能可估计堆栈的使用情况。	`-mchp-stack-usage`

4.4.7 选项页面功能

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

表 4-16. 页面功能选项

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

4.5 项目示例

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

4.5.1 运行项目向导

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

1. **选择项目：**对于类别选择“Microchip Embedded”（Microchip 已安装工具），对于项目选中“Standalone Project”（独立项目）。单击 **Next>**（下一步）继续。
2. **选择器件：**选择 ATSAME70N20B。单击 **Next>**继续。
3. **选择调试头：**该器件没有调试头，所以跳过该步骤。
4. **选择工具：**从列表中选择一种开发工具。选定器件的工具支持会在工具旁以彩色圆圈的形式显示。将鼠标光标悬停在圆圈上，可以文本形式查看支持。单击 **Next>**继续。
5. **选择编译器：**选择 XC32 工具链的一个版本。单击 **Next>**继续。
6. **选择项目名和文件夹：**输入项目名称，例如 `MyXC32Project`。然后选择项目文件夹的位置。单击 **Finish**（完成）来完成项目创建和设置。

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

4.5.2 设置编译选项

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

1. 在“Conf:[default]>XC32 (Global Options)”下，选择“xc32-gcc”。
2. 在“Conf:[default]>XC32 (Global Options)”下，选择“xc32-ld”。
3. 从“Option Category”中选择“Diagnostics”。然后，在“Generate map file”中输入文件名，例如，`example.map`。
4. 单击对话框底部的 **OK**，接受编译选项并关闭对话框。

4.5.3 编译项目

在项目树中右键单击项目名称“`MyXC32Project`”，然后从弹出菜单中选择“Build”（编译）。Output 窗口将显示编译结果。

如果编译未成功完成，请检查以下各项：

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

4.5.4 输出文件

通过在 MPLAB X IDE 中打开文件来查看项目输出文件。

1. 选择 *File>Open File*（文件 > 打开文件）。在 Open（打开）对话框中，查找项目目录。
2. 在“Files of type”（文件类型）下选择“All Files”（所有文件）来查看所有项目文件。
3. 选择 *File>Open File*。在 Open 对话框中，选择“`example.map`”。单击 **Open** 在 MPLAB X IDE 编辑器窗口中查看链接器映射文件。关于该文件的更多信息，请参见链接器文档。
4. 选择 *File>Open File*。在 Open 对话框中，返回到项目目录，然后转到 *dist>default>production*（分发 > 默认 > 生产）目录。请注意，其中只有一个 Hex 文件“`MyXC32Project.X.production.hex`”。它是主输出文件。单击 **Open**，在 MPLAB X IDE 编辑器窗口中查看 Hex 文件。关于该文件的更多信息，请参见实用程序文档。
   另外还有一个文件“`MyXC32Project.X.production.elf`”该文件包含调试信息，由调试工具使用来调试代码。关于选择调试文件类型的信息，请参见 XC32 (Golbal Options)（XC32（全局选项））。

4.5.5 进一步开发

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

开发完您的代码之后，您会希望通过编程将它写入器件。同样，有几个编程器可以与 MPLAB X IDE 配合工作，帮助您完成这项工作。要了解这些工具可以如何帮助您，请参见这些工具的文档。在编程时，您需要使用调试工具栏上的“Make and Program Device Project”（编译并编程器件项目）按钮。关于该控件的信息，请参见 MPLAB X IDE 文档。

5. 命令行驱动程序

可以调用 MPLAB XC32 C/C++编译器命令行驱动程序（xc32-gcc 或 xc32-g++）来执行编译的所有方面，包括 C/C++代码生成、汇编和链接步骤。建议使用该方法来调用编译器，因为这样可以规避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。即使使用 IDE 来协助进行编译，IDE 最终也会调用 xc32-gcc（对于 C 项目）或 xc32-g++（对于 C++项目）。MPLAB X IDE 使用各种启发式方法来确定项目语言。在某些情况下，它会添加-x 标志来选择正确的语言（C 或 C++），并使用两个驱动程序中的任意一个。

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。

5.1 调用编译器

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

5.1.1 驱动程序命令行格式

编译驱动程序（xc32-gcc）可以编译、汇编和链接 C 和汇编语言模块和库归档。当模块源文件以 C++编写时，必须使用 xc32-g++驱动程序。大多数编译器命令行选项对于 GCC 工具集的所有实现是通用的（MPLAB XC32 使用 GCC 工具集；XC8 不使用）。一些选项是特定于编译器的。

编译器命令行的基本形式为：

```
xc32-gcc [options] files
xc32-g++ [options] files
```

例如，要编译、汇编和链接 C 源文件 hello.c，从而创建可执行文件 hello.elf，可以执行以下命令：

```
xc32-gcc -o hello.elf hello.c
```

或者，要编译、汇编和链接 C++源文件 hello.cpp，从而创建可执行文件 hello.elf，可以执行：

```
xc32-g++ -o hello.elf hello.cpp
```

驱动程序选项说明介绍了可用的选项。约定是提供 *options*（通过文件名之前的前导*短划线*“-”表示），虽然这不是强制性的。

files 可以是 C/C++和汇编器源文件、可重定位目标文件（.o）或归档文件的任意混合。文件的顺序很重要。它可能会影响代码或数据在存储器中出现的顺序或符号的搜索顺序。归档文件通常在源文件之后指定。输入文件类型列出了文件类型。

注：命令行选项和文件名是区分大小写的。

除了标准 C 库外，*库*是用户定义的目标代码库文件的列表，链接器会搜索这些库。这些文件的顺序将决定对它们进行搜索的顺序。它们通常放置在源文件名之后，但这不是强制性的。

本手册假定编译器应用程序位于控制台的搜索路径中、已指定了相应的环境变量，或者在执行任何应用程序时指定了完整的路径。

5.1.2 环境变量

本节中的变量是可选的，但如果定义了它们，它们将由编译器使用。如果以下一些环境变量未设置，编译器驱动程序或其他子程序可以选择确定它们适当的值。驱动程序或其他子程序可以利用有关编译器安装的内部知识。只要安装结构保持不变，所有子目录和可执行文件保持在相同的相对位置中，驱动程序或子程序就能够确定一个可用的值。对于新项目，应使用“XC32”变量；但对于旧项目，可以使用“PIC32”变量。

表 5-1. 编译器相关的环境变量

选项	定义
XC32_C_INCLUDE_PATH PIC32_C_INCLUDE_PATH	该变量的值是以分号分隔的目录列表，非常类似于 PATH。在编译器搜索头文件时，它会尝试在使用-I 指定的目录之后、但在标准头文件目录之前搜索在该变量中列出的目录。 如果该环境变量未定义，则预处理器会基于标准安装选择一个适当的值。默认情况下，会在以下目录中搜索包含文件： <install-path>\xc32\include
XC32_COMPILER_PATH PIC32_COMPILER_PATH	XC32_COMPILER_PATH 的值是以分号分隔的目录列表，非常类似于 PATH。在搜索子程序时，如果使用 XC32_EXEC_PREFIX 无法找到子程序，编译器会尝试由此指定的目录。
XC32_EXEC_PREFIX PIC32_EXEC_PREFIX	如果设置了 XC32_EXEC_PREFIX，则它指定在由编译器执行的子程序的名称中使用的前缀。将该前缀与子程序名称组合时，不会添加目录分隔符，但如果需要，您可以指定一个以斜线结尾的前缀。如果编译器无法使用指定的前缀找到子程序，它将尝试在 PATH 环境变量中查找。 如果 XC32_EXEC_PREFIX 环境变量未设置或设置为空值，编译器驱动程序将基于标准安装选择一个适当的值。如果安装未进行任何修改，这将会使驱动程序能够找到所需的子程序。 使用-B 命令行选项指定的其他前缀将优先于用户或驱动程序定义的 XC32_EXEC_PREFIX 值。 在正常情况下，最好将该值保留未定义，让驱动程序自己查找子程序。
XC32_LIBRARY_PATH PIC32_LIBRARY_PATH	该变量的值是以分号分隔的目录列表，非常类似于 PATH。该变量指定要传递给链接器的目录列表。驱动程序对该变量的默认求值为： <install-path>\lib; <install-path>\xc32\lib.
TMPDIR	如果设置了 TMPDIR，则它指定要用于临时文件的目录。编译器使用临时文件来存放某个编译阶段的输出，用作下一个阶段的输入：例如，预处理器的输出，它是相应编译器的输入。

5.1.3 输入文件类型

编译驱动程序可以识别以下文件扩展名，它们区分大小写。

表 5-2. 文件名

扩展名	定义
file.c	必须进行预处理的 C 源文件。
file.cpp	必须进行预处理的 C++源文件。
file.h	头文件（不进行编译或链接）。
file.i	已经进行预处理的 C 源文件。
file.o	目标文件。
file.ii	已经进行预处理的 C++源文件。
file.s	汇编语言源文件。
file.S	必须进行预处理的汇编语言源文件。
其他	要传递给链接器的文件。

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

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

源文件是包含全部或部分程序的文件。它们可能包含 C/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。所有预处理器伪指令和命令（一些用于调试的命令可能除外）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。

5.2 C 编译序列

5.2.1 单步 C 编译

可以使用单个命令行指令来编译一个文件或多个文件。

编译单个 C 文件

本节演示如何编译和链接单个文件。为了本节讨论的目的，假定编译器的<install-dir>/bin 目录已添加到 PATH 变量中。以下是需要注意的其他目录：

- <install-dir>/pic32c/include ——标准 C 头文件的目录。
- <install-dir>/pic32c/include/proc ——特定于 PIC32 器件的头文件的目录。
- <install-dir>/pic32c/lib ——标准库和启动文件的目录结构。
- <install-dir>/pic32c/lib/proc ——可找到特定于器件的链接描述文件片段、寄存器定义文件和配置数据的目录。

下面是将两个数字相加的简单 C 程序。使用任意文本编辑器创建以下程序，并将其保存为 ex1.c。

```
/* ex1.c*/
// ATSAME70N20B Configuration Bit Settings

// Config Source code for XC32 compiler.
// GPNVMBITS
#pragma config SECURITY_BIT = CLEAR
#pragma config BOOT_MODE = CLEAR
#pragma config TCM_CONFIGURATION = 0x0 // Enter Hexadecimal value

// ONLY Set LOCKBITS are generated.
// LOCKBIT_WORD0

// LOCKBIT_WORD1

// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.

#include <xc.h>

unsigned int x, y, z;

unsigned int
add(unsigned int a, unsigned int b) {
    return (a + b);
}

int
main(void) {
    x = 2;
    y = 5;
    z = add(x, y);
    return 0;
}
```

该程序包含了头文件 xc.h，该文件提供了该器件上所有特殊功能寄存器（SFR）的定义。一些文档使用术语“外设寄存器”来描述这些器件寄存器。

通过在提示符下输入以下命令来编译该程序：

```
xc32-gcc -mprocessor=ATSAME70N20B -o ex1.elf ex1.c
```

命令行选项-o ex1.elf 用于命名输出可执行文件（如果未指定-o 选项，则输出文件命名为 a.out）。可使用 File-Import-Hex/Elf (prebuilt) File 将该可执行文件导入 MPLAB X IDE。

如果需要 hex 文件（例如，装入到器件编程器），则使用以下命令：

```
xc32-bin2hex ex1.elf
```

这将创建一个名为 ex1.hex 的 Intel® hex 文件。

编译多个 C 文件

本节演示如何在单个步骤中编译和链接多个文件。将 add() 函数移入一个名为 add.c 文件中，以演示如何在应用程序中使用多个文件。即：

文件 1

```
/* ex1.c*/
// ATSAME70N20B Configuration Bit Settings

// Config Source code for XC32 compiler.
// GPNVMBITS
#pragma config SECURITY_BIT = CLEAR
#pragma config BOOT_MODE = CLEAR
#pragma config TCM_CONFIGURATION = 0x0 // Enter Hexadecimal value
// ONLY Set LOCKBITS are generated.
// LOCKBIT_WORD0
// LOCKBIT_WORD1
// #pragma config statements should precede project file includes.
// Use project enums instead of #define for ON and OFF.

#include <xc.h>

int main(void);
unsigned int add(unsigned int a, unsigned int b);
unsigned int x, y, z

int main(void) {
    x = 2;
    y = 5;
    z = add(x, y);
    return 0;
}
```

文件 2

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

通过在提示符下输入以下命令来编译这两个文件：

```
xc32-gcc -mprocessor=ATSAME70N20B -o ex1.elf ex1.c add.c
```

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

5.2.2 多步 C 编译

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

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

如果使用 make 实用程序来调用编译器，则需要将 make 文件配置为使用中间文件（.o 文件）和用于生成中间文件的选项（-c，请参见用于控制输出类型的选项）。Make 实用程序通常会多次调用编译器：一次是为每个源文件生成一个中间文件，一次是执行编译的第二阶段。

例如，可以使用以下命令行单独编译单步 C 编译小节“编译多个 C 文件”列出的文件 ex1.c 和 add.c：

```
xc32-gcc -mprocessor=ATSAME70N20B -c ex1.c
xc32-gcc -mprocessor=ATSAME70N20B -c add.c
xc32-gcc -mprocessor=ATSAME70N20B -o ex1.elf ex1.o add.o
```

5.3 C++编译序列

5.3.1 单步 C++编译

可以使用单个命令行指令来编译一个文件或多个文件。

编译单个 C++文件

本节演示如何编译和链接单个文件。为了本节讨论的目的，假定编译器的<install-dir>/bin 目录已添加到 PATH 变量中。以下是需要注意的其他目录：

- <install-dir>/pic32c/include/c++——标准 C++头文件的目录。
- <install-dir>/pic32c/include/proc——特定于 PIC32 器件的头文件的目录。
- <install-dir>/pic32c/lib——标准库和启动文件的目录结构。
- <install-dir>/pic32c/lib/proc——特定于器件的链接描述文件片段、寄存器定义文件和配置数据的目录。

以下是一个简单的 C++程序。使用任意纯文本编辑器创建以下程序，并将其保存为 ex1.cpp。

文件 1

```
/* ex1.cpp */

// ATSAME70Q21B Configuration Bit Settings
#pragma config SECURITY_BIT = CLEAR
#pragma config BOOT_MODE = CLEAR
#pragma config TCM_CONFIGURATION = 0x0 // Enter Hexadecimal value

#include <xc.h>
#include <iostream>
using namespace std;

unsigned int add(unsigned int a, unsigned int b)
{
    return (a + b);
}
int main(void)
{
    int myvalue = 6;

    std::cout << "original value: " << myvalue << endl;
    myvalue = add(myvalue, 3);
    std::cout << "new value: " << myvalue << endl;
    while (1);
}
```

该程序包含头文件 xc.h，该文件为目标器件上的所有特殊功能寄存器（SFR）（有时也称为外设寄存器）提供定义。<iostream>头文件为外设库提供了必需的原型。为了补全代码，用户必须为文件 IO 提供函数 _mon_getc 和 _mon_putc 的实际实现。默认情况下，XC32 编译器为这些函数链接不执行任何操作的桩（stub）。

通过在命令提示符下输入以下命令来编译该程序。

```
xc32-g++ -mprocessor=ATSAME70Q21B -Wl,--defsym=_min_heap_size=0xF000 -o
ex1.elf ex1.cpp
```

选项 -o ex1.elf 用于命名输出可执行文件。该 elf 文件可以导入 MPLAB X IDE。

如果需要 hex 文件（例如，装入到器件编程器），则使用以下命令

```
xc32-bin2hex ex1.elf
```

这将创建一个名为 ex1.hex 的 Intel hex 文件。

5.3.2 编译多个 C++文件

本节演示如何在单个步骤中编译和链接多个 C 和 C++文件。

文件 1

```
/* ex1.cpp */

// ATSAME70Q21B Configuration Bit Settings
#pragma config SECURITY_BIT = CLEAR
#pragma config BOOT_MODE = CLEAR
#pragma config TCM_CONFIGURATION = 0x0 // Enter Hexadecimal value

#include <xc.h>
#include <iostream>
using namespace std;

extern unsigned int add(unsigned int a, unsigned int b);

int main(void)
{
    int myvalue = 6;

    std::cout << "original value: " << myvalue << endl;
    myvalue = add(myvalue, 3);
    std::cout << "new value: " << myvalue << endl;
    while (1);
}
```

文件 2

```
/* add.cpp */
unsigned int
add(unsigned int a, unsigned int b)
{
  return(a+b);
}
```

通过在提示符下输入以下命令来编译这两个文件：

```
xc32-g++ -mprocessor=ATSAME70Q21B -Wl,--defsym=_min_heap_size=0xF000 -o
ex1.elf ex1.cpp add.cpp
```

该命令将编译模块 ex1.cpp 和 add.cpp。编译的模块将与 C++的编译器库进行链接，提供堆，并创建可执行文件 ex1.elf。

注：为了将项目与项目中的 C++源文件必需的 C++支持库进行链接，请使用 xc32-g++驱动程序（而不是 xc32-gcc 驱动程序）。

5.4 运行时文件

除了在命令行上指定的 C/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/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

5.4.1 库文件

Microchip 统一标准 C 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。当您的项目包含纯 C 语言源代码并使用 xc32-gcc 驱动程序编译时，Microchip 统一标准库将与您的项目链接。当您使用 xc32-g++驱动程序编译包含 C++代码的项目时，将链接 Microchip 统一标准库和 libstdc++库。

目标库（称为 multilib）使用一组有序选项编译多次。当调用编译器驱动程序来编译和链接应用程序时，驱动程序选择适用于选定选项的 multilib 库。通常无需指定标准库的搜索路径，也无需手动将库文件包含到项目中。库中介绍了如何使用这些函数。

multilib 启动库位于编译器分发的 lib/gcc/pic32c/*gcc-version* 目录中，目标特定的库存储在 pic32c/lib 目录下的目录中。

对于所有器件，可通过在链接时将-mthumb 选项指定给 xc32-gcc 来选择 Thumb ISA 库版本。对于基于 Arm9、Cortex-A5 和 Cortex-A7 的器件，可通过在链接时指定-marm 选项额外选择 Arm ISA 库版本。

随编译器分发的目标库是针对以下命令行选项的组合编译的：

- 对齐（-mno-unaligned-access）
- 浮点应用程序二进制接口（-mfloat-abi softfp, hard）

随编译器分发的 C++目标库是使用以下命令行选项的组合编译的：

- 异常支持（-fno-exceptions 与-fno-rtti）

以下示例提供了关于选择哪个 multilib 子目录的详细信息。为了确保程序正确运行并实现最佳性能，在编译所有模块和链接时应一致地使用确定所选库的选项。

1.
```
xc32-gcc foo.c
xc32-g++ foo.cpp
```
对于该示例，未指定任何命令行选项（即，使用默认的命令行选项）。在这种情况下，使用上述提到的默认目录。

2.
```
xc32-gcc -mfloat-abi=softfp foo.c
xc32-g++ -mfloat-abi=softfp foo.cpp
```
对于此示例，使用 soft multilib 子目录。

3.
```
xc32-gcc -mfloat-abi=hard foo.c
xc32-g++ -mfloat-abi=hard foo.cpp
```
对于此示例，使用 hard multilib 子目录。

5.4.2 外设库函数

对于所有 PIC32 器件，请参见使用库程序。

5.4.3 启动和初始化

C/C++运行时启动代码是特定于器件的。xc32-gcc 和 xc32-g++编译驱动程序在链接时根据-mprocessor 选项选择适当的运行时启动代码。

启动模块的源代码用于初始化运行时环境，它是特定于平台的，可以在编译器分发目录中的 startup_*device*.c 目录内与 startup_*device*.S 或 pic32c/lib/proc/*DEVICE*/匹配的文件中找到。通常，默认的器件特定启动代码是用 C 语言（对于单片机（MCU））和汇编语言（对于微处理器（MPU））编写的，因为微处理器初始化代码需要一些标准 C 代码中通常没有的指令。

注：特定于器件的运行时启动源代码也打包在与您的目标器件相关的器件系列包（DFP）中。但是，一些框架（例如 MPLAB Harmony v3）提供了自己的应用程序特定启动代码，而不是 DFP 中提供的默认代码。

预编译的启动目标文件位于编译器分发目录的 lib/gcc/pic32c/<gcc-version>/目录下的架构特定目录中。

启动和运行时的目标文件名为：crti.o、crtn.o、crtbegin.o 和 crtend.o。也可以在其中找到其他相关库。为了支持器件系列之间的架构差别，提供了这些模块的 Multilib 版本。

关于这些启动模块中代码的实际功能的详细信息，请参见运行时启动代码。

5.5 编译器输出

编译器在编译过程中会创建许多文件。其中的大量文件是中间文件，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。

5.5.1 输出文件

编译驱动程序可以产生具有以下扩展名的输出文件，这些扩展名是区分大小写的。

表 5-3. 文件名

扩展名	定义
file.hex	Intel® HEX 可执行文件
file.elf (a.out)	ELF 调试文件
file.o	目标文件（中间文件）
file.s	汇编代码文件（中间文件）
file.i	预处理的 C 文件（中间文件）
file.ii	预处理的 C++文件（中间文件）
file.map	映射文件

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.o 的目标文件。

如果不指定-o 选项来重命名最终输出，则编译器会写入名为 a.out 的文件。这是一个 ELF 文件。通常会为 ELF 文件手动分配.elf 扩展名。

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。除非用户另外指定，则该项目的名称将用作项目范围输出文件的基本名称。请查看您使用的 IDE 的手册，了解更多详细信息。

注：在本手册中，*项目名称*一词将指代在 IDE 中创建的项目的名称。

编译器能够直接生成许多由 Microchip 开发工具使用的输出文件格式。

xc32-gcc 和 xc32-g++的默认行为是产生 ELF 输出。要更改文件的输出或文件名，请参见驱动程序选项说明。

5.5.2 诊断文件

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

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

在汇编器中创建列表文件的选项为-a（或-Wa,-a，如果传递给驱动程序）。该选项具有许多变化形式，《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）中进行了介绍。要从编译器中传递选项，请参见用于汇编的选项。

每次编译都会产生一个列表文件。每个翻译单元具有一个汇编器列表文件。它是一个预链接汇编器列表，所以它不会显示最终的地址。因此，如果需要为每个源文件产生一个列表文件，则必须单独编译这些文件，请参见多步 C 编译。如果使用 MPLAB X IDE 进行编译，就是如此。每个列表文件将被分配模块名称和扩展名.lst。

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

创建映射文件的链接器选项为`-Map file`（或 `-Wl,-Map=file`，如果传递给驱动程序），《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）进行了介绍。要从编译器驱动程序中传递选项，请参见用于链接的选项。

在编译项目时，如果执行了链接器并运行完成，则会生成一个映射文件。

5.6 编译器消息

有 3 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。

- **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。

关于控制编译器错误、警告或注释输出的选项的信息，请参见用于控制警告和错误的选项。

5.7 驱动程序选项说明

使用传递给命令行驱动程序 `xc32-gcc` 的选项可以对编译过程的大多数方面进行控制。

除了本文档中讨论的选项之外，MPLAB XC32 C/C++编译器所基于的 GCC 编译器还提供了许多选项。建议避免使用此处未给出的任何选项，尤其是用于控制代码生成或优化的选项。

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：`-Idir`。这些选项区分大小写，所以`-c` 是不同于`-C` 的选项。所有选项都以单或双前导短划线字符标识，例如：`-c` 或`--version`。

使用`--help` 选项可获取关于命令行可接受选项的简要说明。

如果在 MPLAB X IDE 中进行编译，则它会根据项目的 **Project Properties** 对话框中的选择向编译器发出显式的选项。默认项目选项可能与在命令行上运行时编译器使用的默认选项有所不同，因此应检查这些默认项目选项以确保它们是可接受的。

5.7.1 特定于 PIC32C/SAM 器件的选项

表 5-4. 特定于 PIC32C/SAM 器件的选项

选项 （链接至说明部分）	定义
`-f[no-]stack-protector-f[no-]stack-protector-strong-f[no-]stack-protector-all`	控制堆栈保护。
`-m[no-]allow-partial-config-words`	指定仅使用一部分定义的配置字即可编译程序。
`-mchp-stack-usage`	生成堆栈使用信息和警告。
`-mcodecov=options`	插装输出以提供代码覆盖信息。
`-mdfp=path`	指定要使用的器件系列包。
`-mdtcm=size`	指定数据紧耦合存储器的大小（字节）。
`-mfloat-abi=name`	指定要使用的浮点 ABI。
`-mitcm=size`	指定指令紧耦合存储器的大小（字节）。
`-m[no-]long-calls`	指定是否使用存储在寄存器中的地址来调用函数。
`-mprint-builtins`	打印已使能的内置函数列表。

.....（续）	
选项 （链接至说明部分）	定义
-mprocessor	选择编译所针对的器件。
-m[no-]pure-code	该选项可确保编译器不会将使用 const 限定的数据放入包含可执行代码的任何段中。
-msmart-io=[0|1|2]	控制链接的 IO 库的功能集。
-mtcm=*size*	指定单存储区的紧耦合存储器的大小（字节）。
-mthumb	使用 Thumb 指令集生成代码。
-m[no-]unaligned-access	控制对非 16 位或 32 位对齐地址的 16 位和 32 位值访问。
--nofallback	需要 MPLAB XC32 专业版许可证，不要回退到较低版本的许可证。

5.7.1.1 Stack-protector 选项

-fstack-protector 选项可为包含以下内容的脆弱函数使能堆栈保护：

- 大于 8 字节的字符数组
- 大于 8 字节的 8 位整型数组
- 对大小为变量或常量且大于 8 字节的 alloca() 的调用

该选项的-fstack-protector-strong 形式可为包含以下内容的脆弱函数使能堆栈保护：

- 任意大小和类型的数组
- 对 alloca() 的调用
- 已获取自身地址的自动局部变量

该选项的-fstack-protector-all 形式可为所有函数（无论是否敏感）添加堆栈保护。

该选项的-fno-stack-protector 形式可禁止堆栈保护，如果未指定该选项的形式，则默认使用这种形式。

注：如果指定了该选项的多种形式，则指定的最后一种形式将生效。

5.7.1.2 Allow-partial-config-words 选项

-mallow-partial-config-words 选项允许只使用通过 config pragma 伪指令指定的一部分配置字来编译程序。请谨慎使用该选项的这种形式，因为已指定的配置位与保留为默认状态的配置位的组合可能会导致器件无法正常工作。

该选项的-mno-allow-partial-config-words 形式将在已通过 config pragma 伪指令指定只使用一部分配置字时阻止编译程序。如果未指定该选项的形式，则默认使用这种形式。使用该选项时，可在未指定任何配置字的情况下编译程序，但如果指定了任何配置字，则必须全部在单个翻译单元内指定。

5.7.1.3 堆栈指导选项

-mchp-stack-usage 选项用于分析程序并报告针对程序所使用的任何堆栈估计的最大堆栈深度。只有具备 PRO 许可证才能使能该选项。

有关编译器生成的堆栈指导报告的更多信息，请参见堆栈指导。

5.7.1.4 Codecov 选项

-mcodecov=*suboptions* 选项可将诊断代码嵌入到程序的输出中，从而允许分析程序的源代码已执行到什么程度。有关更多信息，请参见代码覆盖。

必须指定一个子选项，此时惟一可用的子选项是 ram。

5.7.1.5 Dfp 选项

-mdfp=*path* 选项指示应从某个器件系列包（DFP）的内容中获取对目标器件（由-mprocessor 选项指示）的器件支持，其中 *path* 是该 DFP 的 XC32 子目录的路径。

如果未使用该选项，则 xc32-gcc 驱动程序将尽可能使用编译器发行版中提供的器件特定文件。

Microchip 开发环境自动使用该选项通知编译器要使用的器件特定信息。如果已为编译器获取了其他 DFP，则在命令行上使用该选项。

DFP 可能包含特定于器件的头文件、配置位数据和库等，从而不必更新编译器即可使用新器件上的功能。DFP 始终不包含可执行文件，也不提供针对任何现有工具或标准库函数的缺陷修复或改进。

5.7.1.6 Dtcm 选项

-mdtcm=*size* 选项指定在器件中与指令紧耦合存储器（Tightly Coupled Memory，TCM）分开实现的数据紧耦合存储器的大小（字节）。请参见紧耦合存储器。

5.7.1.7 Float-abi 选项

-mfloat-abi=*name* 选项指定要使用的浮点 ABI。允许使用的值包括：soft、softfp 和 hard。

指定 soft 作为参数可使 XC32 生成包含浮点运算库调用的输出。指定 softfp 作为参数可使用硬件浮点指令生成代码，但仍遵循软件浮点调用约定。指定 hard 作为参数可生成浮点指令并遵循特定于浮点单元（Floating Point Unit，FPU）的调用约定。

默认值取决于具体的目标配置。请注意，硬件浮点 ABI 与软件浮点 ABI 并非链接兼容；必须使用同一种 ABI 编译整个程序并与一组兼容的库链接。

5.7.1.8 Itcm 选项

-mitcm=*size* 选项指定在器件中与数据 TCM 分开实现的指令紧耦合存储器的大小（字节）。请参见紧耦合存储器。

5.7.1.9 Long-calls 选项

-mlong-calls 选项指示编译器在执行函数调用时，先将函数的地址装入寄存器，然后对该寄存器执行子程序调用。如果目标函数在基于偏移量的子程序调用指令的寻址范围之外，则需要进行上述转变。寻址范围因器件和 ISA 而异。通常，在调用另一个存储器区域中的函数时需要该选项。例如，当从闪存分配的函数调用 RAM 分配的函数时，目标地址可能超出短调用的范围。

即使使能该功能，也并非所有函数调用都会转变成长调用。据推断，static 函数、具有 short_call 属性的函数、在#pragma no_long_calls 伪指令范围内的函数，以及已在当前编译单元内编译定义的函数不会转变为长调用。但此规则存在例外，即具有 long_call 属性或 section 属性的函数以及在 #pragma long_calls 伪指令范围内的函数始终转变为长调用。该功能不会影响编译器如何生成通过指针间接调用函数的代码。

该选项的-mno-long-calls 形式将不会使用长调用。如果未指定该选项的形式，则默认使用这种形式。

5.7.1.10 Print-builtins 选项

-mprint-builtins 选项打印已使能的内置函数列表，然后停止编译。

5.7.1.11 Processor 选项

-mprocessor=*device* 选项选择编译针对的目标器件。关于支持的所有器件的列表，请参见编译器发行说明。请注意，器件名称必须大写，例如-mprocessor=ATSAME70N20B。

5.7.1.12 Pure-code 选项

-mpure-code 选项可确保编译器不会将使用 const 限定的数据放入包含可执行代码的任何段中。如果要将代码放入仅执行存储器（eXecute-Only Memory，XOM），则需使用该选项。此外，该选项还会在生成的 ELF 文件中将包含可执行代码的段标记为 SHF_ARM_PURECODE 标志，以指示这些段中仅包含代码而没有数据。将纯代码段放入 XOM 需单独执行（见更改默认的函数分配）。

该选项适用于基于 Arm Cortex-M23 和 Cortex-M0+的目标器件。请查看具体目标器件的数据手册来确定该目标器件是否支持 XOM 及其更多使用信息。不得将该选项与 MPLAB 代码覆盖功能一起使用，否则会触发编译器错误。

使用该选项可能会影响程序的代码长度和执行速度。使用 const 限定的数据通常放入程序存储器，并在需要时进行读取。这在该选项生效后将无法实现，编译器将需要生成更多/更长的指令来将常量作为立即数操作数进行装载。例如，要装载使用 int const 限定的常量（存储在存储器中），将需要 4 个字节的常量数据，并且通常需要 2 个字节的 ldr 指令将该常量装入寄存器。如果改为使用带立即数操作数的 movw + movt 指令序列装载该常量，则将需要两个 4 字节指令。对于无法使用此类指令序列的器件（如 Cortex M0+），只能逐字节装载常量，相关代码明显会更长。使能该选项时，请务必验证代码长度或执行时间延长后是否仍满足应用程序的时序要求。

MPLAB XC32 提供的预编译库（如标准 C 库）并非针对纯代码而编译。在应用程序链接描述文件中将这些函数映射到 XOM 区域可能会导致代码失败。

当同时使用-mpure-code 和-ffunction-sections 选项时，生成的输入段将根据-ffunction-sections 选项的规则命名，确保更加难以在定制链接描述文件中将代码映射到 XOM。在这种情况下，不能依赖 SHF_ARM_PURECODE 段标志，而必须按照名称来映射段。

将-mpure-code 选项与直接或间接影响函数位置的函数属性结合使用时同样如此。这类属性包括 tcm、ramfunc、address 和 always_inline 等。同理，在定制链接描述文件中仍按照名称（而非使用 SHF_ARM_PURECODE 段标志）来映射段，以确保代码受到完全保护。

使用-mpure-code 选项时，过程抽象优化（-mpa）可能会从被指定为纯代码的段中抽象出代码，所抽象代码随后需要在定制链接描述文件中适当地映射到 XOM。

该选项的-mno-pure-code 形式不会将可执行代码与使用 const 限定的数据区分开。如果未指定该选项的形式，则默认使用这种形式。

5.7.1.13 Smart-io 选项

-msmart-io=*level* 选项与在程序中检测到的 IO 格式字符串转换规范一起控制所链接库代码的功能集（以及长度），从而通过 printf 等函数执行格式化 IO。有关智能 IO 功能工作原理的更多信息，请参见智能 IO 程序。

可以指定工作级别数字，其含义如下表所示。

表 5-5. 智能 IO 实现级别

级别	智能 IO 功能；链接库
0	禁止；全功能库（最大代码长度）
1	使能；最小功能库（最小代码长度）
2	手动控制；仅整型库

禁止智能 IO 功能（-msmart-io=0）时，IO 函数的完整实现将链接到程序中。IO 库函数的所有功能都将可用，这些功能可能会占用目标器件上大量可用的程序和数据存储空间。

默认设置是使能智能 IO 并使用最小功能集。可以使用-msmart-io=1 或-msmart-io 选项将其设为显式。使能时，编译器将链接到复杂度最低的 IO 库形式中，该形式基于程序的 IO 函数格式字符串中检测到的转换规范实现程序需要的所有 IO 功能。这样可以大幅降低程序对存储器的需求，尤其是无需在程序中使用浮点功能时。

如果 IO 函数调用中的格式字符串并非字符串字面值，则编译器将无法检测到 IO 函数的确切使用，IO 库的全功能形式将链接到程序映像中（即使使能了智能 IO）。

所有程序模块应一致使用这些选项，以确保程序映像中包含的库程序为最佳选择。

5.7.1.14 Stack-in-tcm 选项

-mstack-in-tcm 选项将运行时堆栈放入数据紧耦合存储器（TCM）区域。运行时启动代码会在调用 main() 函数之前将堆栈从系统 SRAM 转移到 DTCM。无论是编译时还是链接时，都必须使用该选项。

5.7.1.15 Tcm 选项

-mtcm=*size* 选项指定在器件中作为单个存储器区域实现的紧耦合存储器的大小（字节）。请参见紧耦合存储器。

5.7.1.16 Thumb 选项

-mthumb 选项通知编译器使用 16 位 Arm Thumb 指令集生成代码。这样可能会生成更适合目标应用程序的代码。使用-marm 选项将明确表明不想使用 Arm Thumb 指令集生成代码。

5.7.1.17 Unaligned-access 选项

-munaligned-access 选项使能从非 16 位或 32 位对齐的地址读取和写入 16 位和 32 位值。该选项不影响取指。

该选项的-mno-unaligned-access 形式以每次一个字节的方式访问数据结构中的 16 位和 32 位值。

如果未指定该选项的形式，则 Arm v6 之前的所有架构、Arm v6-M 和 Arm v8-M 架构均禁止未对齐访问，所有其他架构均使能未对齐访问。

5.7.1.18 Nofallback 选项

--nofallback 选项可用于确保编译器不会意外执行优化级别低于-O 选项所指定的级别的优化。

例如，在无许可证的情况下请求编译器运行 s 级优化时，如果未使用该选项，则通常会恢复到较低的优化级别并继续。如果使用该选项，编译器将发出错误，同时编译将终止。因此，该选项可确保在具备适当许可证的情况下使用编译器进行编译。

5.7.2 用于控制输出类型的选项

以下选项控制编译器产生的输出的类型。

表 5-6. 输出类型控制选项

选项（链接至说明部分）	定义
-c	在链接步骤之前停止编译，生成中间文件。
-E	在预处理完成后停止编译，生成预处理文件。
-o *file*	将输出放入指定名称的文件。
-S	在汇编步骤之前停止编译，生成汇编文件输出。
-specs=*file*	改写标准规范文件。
-v	打印在每个编译阶段过程中执行的命令。
-x	指定源文件的语言，与文件扩展名无关。
--help	打印命令行选项的说明。

5.7.2.1 C：编译为中间文件

-c 选项用于为命令行上列出的每个源文件生成一个中间文件。

对于所有源文件，编译均将在执行汇编器后终止，并留下扩展名为.o 的可重定位目标文件。

该选项通常用于通过 make 实用程序实现多步编译。

5.7.2.2 E：仅预处理

-E 选项用于生成预处理的 C/C++源文件（也称为模块或翻译单元）。

预处理的输出将打印到 stdout，但是您可以使用-o 选项将其重定向到文件。

您可以检查预处理的源文件，以确保预处理器宏已扩展为您希望的内容。该选项还可用于创建不需要任何单独头文件的 C/C++源文件。需要将文件发送给同事，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）。

5.7.2.3 O：指定输出文件

-o 选项指定输出文件的基本名称和目录。

例如，选项-o main.elf 会将生成的输出置于名为 main.elf 的文件中。可用文件名指定现有目录的名称，例如-o build/main.elf，这样输出文件便会出现在该目录中。

不能使用该选项来更改输出文件的类型（格式）。

5.7.2.4 S：编译为汇编

-S 选项用于为命令行上列出的每个源文件生成一个汇编文件。

使用该选项时，编译序列将提前终止，并留下基本名称与相应源文件相同且扩展名为.s 的汇编文件。

例如，以下命令：

```
xc32-gcc -mprocessor=ATSAME70N20B -S test.c io.c
```

会生成名为 test.s 和 io.s 的两个汇编文件，其中包含基于相应的源文件生成的汇编代码。

该选项对于检查编译器输出的汇编代码可能很有用，因为不需要为汇编列表文件中的行号和操作码信息分心。此外，汇编文件也可用作编写汇编代码的基础。

5.7.2.5 Specs 选项

-specs=*file* 选项可用于改写标准规范文件。

规范文件是用于构造规范字符串的纯文本文件，规范字符串用于控制编译器应调用哪些程序以及需要传递哪些命令行选项。编译器有一个标准规范文件，该文件定义了编译器的内部应用程序将使用哪些选项来执行。

编译器在读入标准规范文件后会处理使用该选项指定的 *file* 文件，以改写 xc32-gcc 驱动程序在编译传递给 xc32-as 和 xc32-ld 等的命令行选项时使用的默认设置。可以在命令行上指定多个-specs 选项，它们将按顺序从左到右处理。

5.7.2.6 V：详细编译

-v 选项用于指定详细编译。

使用该选项时，将在执行内部编译器应用程序时显示其名称和路径，然后显示传递给每个应用程序的命令行参数。

使用该选项，可以确认您的驱动程序选项是否已按预期进行处理，或者查看哪个内部应用程序正在发出警告或错误。

5.7.2.7 X：指定源语言选项

-x *language* 选项指定命令行上的源文件的语言。

编译器通常使用输入文件的扩展名确定文件的内容。该选项用于明确声明文件的语言。该选项将一直保持有效，直到使用-x 选项指定其他语言或使用-x none 选项完全关闭后续文件的语言规范。下表列出了允许使用的语言。

表 5-7. 源文件语言

语言	文件语言
assembler	汇编源文件
assembler-with-cpp	具有 C 预处理器伪指令的汇编文件
c	C 源文件
c++	C++源文件
cpp-output	预处理 C 源文件
c-header	C 头文件

.....（续）	
语言	**文件语言**
none	完全基于文件的扩展名

-x assembler-with-cpp 语言选项可确保汇编源文件在汇编之前进行预处理，从而允许对汇编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，未使用.s 扩展名的汇编文件不进行预处理。

可以使用该选项创建预编译头文件，例如：

```
xc32-gcc -mprocessor=ATSAME70N20B -x c-header init.h
```

将创建名为 init.h.gch 的预编译头文件。

5.7.2.8 帮助

--help 选项显示有关 xc32-gcc 编译器选项的信息，随后驱动程序将终止。

例如：

```
xc32-gcc --help
Microchip Language Tool Shell Version 4.20 (Build date: Sep 16 2022).
Copyright (c) 2012-2017 Microchip Technology Inc. All rights reserved

  -omf=elf        Select elf object module format
Usage: pic32m-gcc [options] file...
Options:
  -pass-exit-codes         Exit with highest error code from a phase.
  --help                   Display this information.
  --target-help            Display target specific command line options.
  --help={common|optimizers|params|target|warnings|[^]{joined|separate|undocumented}}[,...].
                           Display specific types of command line options.
  (Use '-v --help' to display command line options of sub-processes).
  --version                Display compiler version information.
  -dumpspecs               Display all of the built in spec strings.
  -dumpversion             Display the version of the compiler.
  -dumpmachine             Display the compiler's target processor.
  -print-search-dirs       Display the directories in the compiler's search path.
  -print-libgcc-file-name  Display the name of the compiler's companion library.
```

5.7.3 用于控制 C 方言的选项

下表所列的选项用于定义编译器使用的 C 方言类型，这些选项将在后面的章节中详细讨论。

表 5-8. C 方言控制选项

选项 （链接至说明部分）	定义
-ansi	支持（且仅支持）所有 ANSI 标准 C 程序。
-aux-info *filename*	基于 C 模块生成函数原型，将输出放入指定名称的文件。
-f[no-]freestanding	断言项目将在独立式（而非托管式）环境中进行编译。
-f[no-]asm	限制对特定关键字的识别，从而释放这些关键字以用作标识符。
-fno-builtin -fno-builtin-*function*	不识别不以__builtin_作为前缀开头的内置函数。
-f[no-]unsigned-bitfields- f[no-]signed-bitfields	更改普通 int 位域的符号性。
-f[no-]signed-char- f[no-]unsigned-char	更改普通 char 类型的符号性。

5.7.3.1 Ansi 选项

-ansi 选项可确保 C 程序严格符合 C90 标准。

指定后，该选项将在编译 C 源代码时禁止特定的 GCC 语言扩展。此类扩展包括 C++样式注释以及关键字（如 asm 和 inline）。使用该选项时，将定义宏__STRICT_ANSI__。有关确保严格符合 ISO 标准的信息，另请参见-Wpedantic。

5.7.3.2 Aux-info 选项

-aux-info 选项基于 C 模块生成函数原型。

例如，-aux-info main.pro 选项用于打印到所编译模块中声明和/或定义的所有函数的 main.pro 原型声明中，包括头文件中的原型声明。使用该选项时，命令行上只能指定一个源文件，以防输出文件被改写。在除 C 之外的任何其他语言中，均会静默地忽略该选项。

输出文件还会使用每个声明的注释、源文件和行号来指示该声明为隐式声明、原型声明还是非原型声明。这通过在行号和冒号后的第一个字符中使用代码 I 或 N（新样式）和 O（旧样式）来指示，来自声明还是定义则通过在后面的字符中使用代码 C 或 F 来指示。对于函数定义，还会在声明之后的注释内提供 K&R 样式的参数列表及其声明。

例如，使用以下命令编译时：

```
xc32-gcc -mprocessor=ATSAME70N20B -aux-info test.pro test.c
```

可能生成包含以下声明的 test.pro，之后可以根据需要对其进行编辑：

```
/* test.c:2:NC */ extern int add (int, int);
/* test.c:7:NF */ extern int rv (int a); /* (a) int a; */
/* test.c:20:NF */ extern int main (void); /* () */
```

5.7.3.3 Enforce-eh-specs 选项

-fenforce-eh-specs 选项生成用于在运行时检查是否违反异常规范的代码。如果未指定该选项的形式，则默认使用这种形式。

该选项的-fno-enforce-eh-specs 形式不生成验证代码，这样做虽然违反 C++标准，但可能会减小经验证源代码的生产编译中的代码长度。即使在指定该选项时，编译器也仍会基于规范来优化异常代码，所以抛出意外的异常将导致未定义的行为。

5.7.3.4 Freestanding 选项

-ffreestanding 选项断言项目将在独立式（而非托管式）环境中进行编译。

在独立式环境中，标准库可能尚未完全实现，程序启动和终止由实现定义。

该选项与-fno-hosted 选项相同，隐含-fno-builtin 选项。

-fno-freestanding 选项与-fhosted 选项相同，指示项目将在托管式环境中进行编译。

5.7.3.5 Asm 选项

-fasm 选项将 asm、inline 和 typeof 保留用作关键字，以防止它们被定义为标识符。如果未指定该选项的形式，则默认使用这种形式。

该选项的-fno-asm 形式限制对这些关键字的识别。可以使用关键字__asm__、__inline__和__typeof__代替，其含义相同。

-ansi 选项隐含-fno-asm。

5.7.3.6 No-builtin 选项

-fbuiltin 选项让编译器生成专用代码，避免调用许多内置函数。生成的代码通常更小、更快，但由于对这些函数的调用不再出现在输出中，因此无法在这些调用上设置断点，也无法通过链接不同的库更改函数的行为。如果未指定该选项的形式，则默认使用这种形式。

-fno-builtin 选项将阻止编译器生成不带__builtin_前缀的内置函数的特殊代码。

该选项的-fno-builtin-*function*形式用于防止使用指定函数的内置版本。 在这种情况下，*function*不得以__builtin_开头。该选项没有-fbuiltin-*function*形式。

5.7.3.7 Signed-bitfields 选项

-fsigned-bitfield 选项用于控制普通 int 位域类型的符号性。

默认情况下，普通 int 类型用作位域的类型时与 signed int 等效。该选项指定编译器将针对普通 int 位域使用的类型。使用-fsigned-bitfield 或-fno-unsigned-bitfield 选项会强制普通 int 位域变为有符号类型。

可以考虑在定义位域时明确声明它们的符号性，而不是依赖分配给普通 int 位域类型的类型。

5.7.3.8 Signed-char 选项

-fsigned-char 选项会强制普通 char 对象变为有符号类型。

默认情况下，普通 char 类型等效于 unsigned char。-funsigned-char（或-fno-signed-char 选项）可显式设置该类型。

-fsigned-char（或-fno-unsigned-char 选项）可显式设置普通 char 类型被视为有符号整数。

可以考虑在定义 char 对象时明确声明它们的符号性，而不是依赖编译器为普通 char 对象分配类型。

> **注意：**根据 *Arm C 语言扩展*和 *Arm 过程调用标准*规范定义，普通 char 等效于 unsigned char。使用-fsigned-char 选项时，生成的代码将不符合这些规范。

5.7.3.9 Unsigned-bitfields 选项

-funsigned-bitfield 选项用于控制普通 int 位域类型的符号性。

默认情况下，普通 int 类型用作位域的类型时与 signed int 等效。该选项指定编译器将针对普通 int 位域使用的类型。使用-funsigned-bitfield 或-fno-signed-bitfield 选项会强制普通 int 变为无符号类型。

可以考虑在定义位域时明确声明它们的符号性，而不是依赖分配给普通 int 位域类型的类型。

5.7.3.10 Unsigned-char 选项

-funsigned-char 选项会强制普通 char 对象变为无符号类型。

默认情况下，普通 char 类型等效于 unsigned char。-funsigned-char（或-fno-signed-char 选项）可显式设置该类型。

可以考虑在定义 char 对象时明确声明它们的符号性，而不是依赖编译器为普通 char 对象分配类型。

5.7.4 用于控制 C++方言的选项

下表所列的选项用于定义编译器使用的 C++方言类型，这些选项将在后面的章节中详细讨论。请注意，关于 C 方言的相关选项，请参见用于控制 C 方言的选项一节。

表 5-9. C++方言控制选项

选项 （链接至说明部分）	定义
-ansi	支持（且仅支持）所有 ANSI 标准 C++程序。
-aux-info *filename*	基于 C 模块生成函数原型，将输出放入指定名称的文件。
-f[no-]check-new	检查操作符 new 返回的指针是否为非空。
-f[no-]enforce-eh-specs	指定生成用于在运行时检查是否违反异常规范的代码。

.....（续）	
选项 （链接至说明部分）	定义
-f[no-]freestanding	断言项目将在独立式（而非托管式）环境中进行编译。
-f[no-]asm	限制对特定关键字的识别，从而释放这些关键字以用作标识符。
-fno-builtin -fno-builtin-*function*	不识别不以__builtin_作为前缀开头的内置函数。
-f[no-]rtti	指定是否应生成运行时类型识别功能的代码。
-f[no-]signed-bitfields -f[no-]unsigned-bitfields	更改普通 int 位域的符号性。
-f[no-]signed-char- f[no-]unsigned-char	更改普通 char 类型的符号性。

5.7.4.1 Check-new 选项

-fcheck-new 选项在尝试修改所分配的存储空间之前检查 operator new 返回的指针是否为非 NULL。

通常情况下，不需要进行这项检查。使用该选项的-fno-check-new 形式可确保不进行检查。如果未指定该选项的形式，则默认使用这种形式。

5.7.4.2 Rtti 选项

-frtti 选项使能生成每个具有虚拟函数的类的信息，供 C++运行时类型识别（Run Time Type Identification，RTTI）功能使用，例如 dynamic_cast 和 typeid 操作符。

该选项的-fno-rtti 形式禁止生成该信息。如果不使用语言的 RTTI 功能，则使用该选项可能会减小代码长度。请注意，异常处理程序使用相同的信息，但它将根据需要生成该信息。dynamic_cast 操作符仍然可以用于不需要 RTTI 的强制类型转换，即，强制类型转换为 void *或明确的基类。

确保在编译所有模块时和链接时指定该选项的同一形式。

5.7.5 用于控制警告和错误的选项

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

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

以下各表列出的选项是更常用的警告选项，用于控制编译器发出的消息。后续（链接）章节将详细介绍通常会影响警告的选项。

表 5-10. 所有警告隐含的警告和错误选项

选项 （链接至说明部分）	定义
-fsyntax-only	检查代码的语法，但除此之外不执行任何操作。
-pedantic	发出严格 ANSI C 要求的所有警告。拒绝所有使用已禁止扩展的程序。
-pedantic-errors	类似于-pedantic，只是产生错误而不是警告。
-w	禁止所有警告消息。
-Wall	使能关于此类构造的所有警告：一些用户视为可疑且易于避免，即使是与宏配合使用。
-Waddress	产生关于存储器地址的可疑使用的警告。它们包括在条件表达式中使用函数的地址（如 void func(void)；和 if (func)）以及与字符串字面值的存储器地址进行比较，例如(x == "abc")。此类用法通常表明存在程序员错误：函数的地址总是求值为 true，所以在条件语句中使用它们通常表明程序员忘记了函数调用中的括号；而与字符串字面值进行比较则会产生未规定的行为，并且在 C 中是不可移植的，所以它们通常表明程序员希望使用 strcmp。

.....（续）

选项 （链接至说明部分）	定义
-Warray-bounds	当与-02 或更大值结合使用时，会针对许多实例超限数组索引和指针偏移量产生警告。该功能无法检测所有情况。通过-Wall 命令使能。
-Wchar-subscripts	数组索引具有 char 类型时产生警告。
-Wcomment	每当/*注释中出现注释开始序列/*，或每当//注释中出现反斜杠换行符时，产生警告。
-Wdiv-by-zero	产生关于编译时整数除以零的警告。要禁止警告信息，请使用-Wno-div-by-zero。不会产生关于浮点除以零的警告，因为它是获取无穷大和 NaN 的合法方式。 这是默认设置。
-Wformat	检查对于 printf 和 scanf 等的调用，以确保所提供参数具有对应于指定格式字符串的类型。
-Wimplicit	等效于同时指定-Wimplicit-int 和-Wimplicit-function-declaration。
-Wimplicit-function-declaration	每当函数在声明之前使用时产生警告。
-Wimplicit-int	声明未指定类型时产生警告。
-Wmain	main 的类型可疑时产生警告。 main 应为具有外部链接的函数，返回 int，接受 0 个、2 个或 3 个适当类型的参数。
-Wmissing-braces	聚合或联合的初始化中大括号不完整时产生警告。在以下示例中，a 的初始化中大括号不完整，但 b 的完整。 int a[2][2] = { 0, 1, 2, 3 }; int b[2][2] = { { 0, 1 }, { 2, 3 } };
-Wmultistatement-macros	针对不安全的宏展开作为语句（如 if, else、while、switch 或 for）主体的情况产生警告。通过-Wall 命令使能。

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

-Wparentheses	如果在某些上下文中省略了括号（例如，在期望得到真值的上下文中进行赋值，或者对操作符进行了嵌套，而其优先顺序不明确），则产生警告。
-Wreturn-type	每当函数定义为其返回类型默认设为 int 时产生警告。此外，对于返回类型不是 void 的函数，如果 return 语句没有返回值，也产生警告。
-Wsequence-point	对由于违反 C 标准中的序列点规则而可能具有未定义语义的代码产生警告。 C 标准以序列点的形式定义了对 C 程序中的表达式进行求值的顺序，序列点代表程序各个部分之间的执行偏序：在序列点之前执行的那些部分和在它之后执行的那些部分。它们出现在以下位置：在完整表达式（该表达式不属于更大表达式的一部分）的求值之后；在&&、||、？ :或，（逗号）操作符第一个操作数的求值之后；在调用函数之前（但在其参数和表示被调用函数的表达式的求值之后）；以及在某些其他位置。除了序列点规则所表述的之外，并未规定表达式的子表达式的求值顺序。所有这些规则仅描述偏序，而不是全序。举例来说，如果在一个表达式中调用了两个函数，它们之间不存在序列点，则调用这两个函数的顺序是未指定的。但是，标准委员会已经规定，函数调用不发生重叠。 并未规定对象值的修改在各序列点间何时生效。行为取决于这一点的程序具有未定义的行为。C 标准规定“在上一个和下一个序列点之间，对象的存储值最多只能被表达式求值修改一次。此外，只有在确定将要保存的值时才能读取前一个值”。如果程序违反了这些规则，则任何特定实现上的结果将是完全不可预测的。 具有未定义行为的代码的示例：a = a++;、 a[n] = b[n++] 和 a[i++] = i;。该选项不会诊断一些更复杂的情况，并且它偶尔可能会给出误报结果，但通常情况下，它对于检测程序中的此类问题是相当有效的。
-Wsizeof-pointer-div	针对指针大小与其指向的元素大小的可疑除法（看似通常用于计算数组大小的方法，但对于指针来说却无法正确计算）产生警告。

.....（续）	
选项（链接至说明部分）	**定义**
-Wswitch	每当 switch 语句具有枚举类型的索引，但该枚举的一个或多个指定代码缺少 case 语句时，产生警告（存在 default 标号会阻止该警告）。在使用该选项时，超出枚举范围的 case 标号也会产生警告。
-Wsystem-headers	打印关于在系统头文件中找到的构造的警告消息。通常假定来自系统头文件的警告并不表明真正的问题，只会使编译器输出更难以阅读，所以通常会禁止这些警告。使用该命令行选项将指示编译器发出来自系统头文件的警告，如同它们是出现在用户代码中。但请注意，将-Wall 与该选项一起使用并不会对系统头文件中的未知 pragma 伪指令产生警告。对于这种情况，还必须使用-Wunknown-pragmas。
-Wtrigraphs	遇到任何三字符组合（假设已使能它们）时产生警告。
-Wuninitialized	如果未先进行初始化而使用了自动变量，则产生警告。 只有在使能优化时才可能产生这些警告，因为它们需要只有在优化时才会计算的数据流信息。 只有对于寄存器分配的候选变量，才会出现这些警告。因此，对于声明为 volatile、已获取地址，或大小不为 1、2、4 或 8 字节的变量，不会出现这些警告。此外，对于结构体、联合体或数组（即使它们处于寄存器中时），也不会出现这些警告。 请注意，对于仅用于计算一个自身从未被使用的值的变量，可能不会出现任何警告，因为数据流分析可能会在打印警告之前已删除这种计算。
-Wunknown-pragmas	当遇到编译器无法理解的#pragma 伪指令时，产生警告。如果使用了该命令行选项，则甚至会对于系统头文件中的未知 pragma 伪指令发出警告。如果只是通过-Wall 命令行选项使能警告，则不会发生这种情况。
-Wunused	每当一个变量在声明之外未被使用过，每当函数声明为静态但从未定义，每当声明了标号但未使用过，以及每当某个语句计算一个明显未被使用的结果时，产生警告。 要获取关于未用函数参数的警告，必须同时指定-W 和-Wunused。 将表达式强制类型转换为 void 会禁止对表达式产生该警告。类似地，unused 属性会禁止对未用变量、参数和标号产生该警告。
-Wunused-function	每当声明了静态函数但未定义或非内联静态函数未被使用过时，产生警告。
-Wunused-label	每当声明了标号但未使用过时，产生警告。要禁止该警告，请使用 unused 属性。
-Wunused-parameter	每当函数参数除了声明之外未被使用过时，产生警告。要禁止该警告，请使用 unused 属性。
-Wunused-variable	每当局部变量或非常量静态变量除了声明之外未被使用过时，产生警告。要禁止该警告，请使用 unused 属性。
-Wunused-value	每当语句计算一个明显未被使用的结果时，产生警告。要禁止该警告，请将表达式强制类型转换为 void。

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

表 5-11. 所有警告不隐含的警告和错误选项

选项	定义
-Wextra （原名） -W	将不通过-Wall 使能的额外警告标志使能。
-Waggregate-return	如果定义或调用了返回结构体或联合体的函数，则产生警告。
-Wbad-function-cast	每当函数调用被强制类型转换为非匹配类型时，产生警告。例如，如果 int foof() 被强制类型转换为任何*类型，则产生警告。
-Wcast-align	每当对指针进行强制类型转换，从而增加了目标所需的对齐时，产生警告。例如，如果 char * 被强制类型转换为 int *，则产生警告。
-Wcast-qual	每当对指针进行强制类型转换，从而从目标类型中去除一个类型限定符时，产生警告。例如，如果 const char *被强制类型转换为普通 char *，则产生警告。

.....（续）

选项	定义
-Wconversion	如果原型导致的类型转换与不存在原型时同一个参数发生的类型转换不同，则产生警告。这包括定点到浮点的转换（以及反之）和更改定点参数宽度或符号的转换，但与默认提升相同时除外。此外，如果某个负整数常量表达式隐式转换为无符号类型，则产生警告。例如，如果 x 为无符号，则对赋值 x = -1 产生警告。但不对诸如(unsigned) -1 之类的显式强制类型转换产生警告。
-Werror	使所有警告变为错误。
-Winline	如果某个函数不能进行内联，但将它声明为内联或提供了-finline-functions 选项，则产生警告。
-Wlarger-than-len	每当定义的对象大于 len 个字节时产生警告。
-Wlong-long -Wno-long-long	如果使用了 long long 类型，则产生警告。这是默认设置。要禁止警告信息，请使用-Wno-long-long。只有在使用-pedantic 标志时，才会考虑标志-Wlong-long 和-Wno-long-long。
-Wmissing-declarations	如果定义了某个全局函数，但无事先声明，则产生警告。即使定义本身提供了原型也如此。
-Wmissing-format-attribute	如果使能了-Wformat，则也对于可能为 format 属性候选的函数产生警告。请注意，这些只是可能的候选函数，不是绝对的。除非使能了-Wformat，否则该选项不起作用。
-Wmissing-noreturn	对于可能为 noreturn 属性候选的函数产生警告。这些只是可能的候选函数，不是绝对的。应仔细地手动验证这些函数。事实上，请不要在添加 noreturn 属性之前返回，否则可能会引入难以琢磨的代码生成问题。
-Wmissing-prototypes	如果定义了某个全局函数，但无事先的原型声明，则产生警告。即使定义本身提供了原型，也会发出该警告。该选项可以用于检测未在头文件中声明的全局函数。
-Wnested-externs	如果在函数内遇到 extern 声明，则产生警告。
-Wno-deprecated-declarations	不产生关于使用通过 deprecated 属性标记为已弃用的函数、变量和类型的警告。
-Wpadded	如果某个结构体中包含填充（用于对齐结构体的某个元素或对齐整个结构体），则产生警告。
-Wpointer-arith	如果任何对象依赖于函数类型的大小或 void 的大小，则产生警告。编译器会将这些类型的大小赋值为 1，以方便计算 void *指针和指向函数的指针。
-Wredundant-decls	如果任何对象在同一作用域内声明多次（即使多个声明均有效，不改变任何设置），则产生警告。
-Wshadow	每当某个局部变量遮蔽另一个局部变量时，产生警告。
-Wsign-compare -Wno-sign-compare	如果在有符号值和无符号值之间进行比较，当有符号值被转换为无符号时，可能产生错误的结果，则产生警告。该警告也通过-W 使能。要获得-W 的其他警告而不产生该警告，请使用-W -Wno-sign-compare。
-Wstrict-prototypes	如果某个函数在声明或定义时未指定参数类型，则产生警告（如果前面有指定参数类型的声明，则允许使用旧式函数定义而不会发出警告）。
-Wtraditional	对某些在传统和 ANSI C 中具有不同行为的构造产生警告。 • 宏主体中的字符串常量内出现的宏参数。在传统 C 中，它们将替换参数，但在 ANSI C 中，它们是常量的一部分。 • 某个函数在一个块中声明为外部函数，然后在该块结束之后使用了该函数。 • 具有 long 类型操作数的 switch 语句。 • 非静态函数声明之后跟随一个静态函数声明。一些传统 C 编译器不接受该构造。
-Wundef	如果在#if 伪指令中对未定义的标识符进行求值，则产生警告。
-Wunreachable-code	如果编译器检测到代码永远不会执行，则产生警告。 即使在一些情况下，受影响代码行的一部分会被执行，该选项也可能会产生警告，因此在删除看上去不可达的代码时应小心。例如，对某个函数进行内联时，产生的警告可能意味着代码行仅在该函数的某个内联副本内不可达。
-Wwrite-strings	为字符串常量赋予 const char[length]类型，从而在将一个字符串常量的地址复制到非 const char *指针时获得警告。在编译时，这些警告帮助您查找可以尝试写入字符串常量的代码，但前提是您在声明和原型中使用 const 时很小心。否则，它只会令人烦扰，这也是-Wall 不请求产生这些警告的原因。

5.7.5.1 Syntax-only 选项

-fsyntax-only 选项用于检查 C 源代码的语法错误，然后终止编译。

5.7.5.2 严格程度选项

-pedantic 选项可确保程序不使用禁止的扩展并在程序未遵循 ISO C 标准时发出警告。

5.7.5.3 Pedantic-errors 选项

-pedantic-errors 选项的工作方式与-pedantic 选项相同，只是在程序不符合 ISO 标准时发出错误而非警告。

5.7.5.4 W：禁止所有警告选项

-w 选项用于禁止所有警告消息，因此应谨慎使用。

5.7.5.5 Wall 选项

-Wall 选项用于使能与一些用户认为有问题但容易避免的构造有关的所有警告（甚至与宏配合使用）。

请注意，-Wall 并未隐含一些警告标志。在这些警告中，有一些与用户通常不认为有问题但可能希望偶尔进行检查的构造相关。另一些与在某些情况下需要或难以避免的构造相关，无法通过修改代码来轻松禁止。其中一些警告可使用-Wextra 选项来使能，但很多警告必须单独使能。

5.7.5.6 Wextra 选项

-Wextra 选项在以下情况下生成额外警告。

- 非可变自动变量可能被对 longjmp 的调用更改。只有在优化编译时才可能出现这些警告。编译器只能检测到对 setjmp 的调用。它无法知道 longjmp 会在哪里被调用。事实上，信号处理程序可以在代码中的任意点调用它。因此，即使事实上不存在任何问题，因为事实上无法在会导致问题的位置调用 longjmp，也可能会产生警告。
- 函数有可能通过 return *value*;退出，也有可能通过 return;退出。不传递任何返回语句而结束函数主体的情况会被当做 return;处理。
- 表达式语句或逗号表达式的左侧不包含副作用。要禁止该警告，请将未用表达式强制类型转换为 void。例如，诸如 x[i,j]之类的表达式会产生警告，但 x[(void)i,j]不会。
- 无符号值通过<或<=与零进行比较。
- 出现诸如 x<=y<=z 之类的比较。这等效于(x<=y ? 1 : 0) <= z，这是一种不同于普通数学符号的解释。
- 诸如 static 之类的存储类说明符不是声明中的第一个符号。根据 C 标准，这种用法是过时的。
- 如果还指定了-Wall 或-Wunused，则会产生关于未用参数的警告。
- 在有符号值和无符号值之间进行比较，当有符号值被转换为无符号时，可能产生错误的结果（但如果也指定了-Wno-sign-compare，则不产生警告）。
- 聚合具有大括号不完整的初始化。例如，以下代码将会产生此类警告，因为 x.h 的初始化两边缺少大括号：

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

- 聚合具有不初始化所有成员的初始化。例如，以下代码将产生此类警告，因为 x.h 会隐式地初始化为零：

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

5.7.6 用于调试的选项

下表所列的选项用于控制编译器产生的调试输出，这些选项将在后面的章节中详细讨论。

表 5-12. 调试选项

选项 （链接至说明部分）	定义
-f[no-]eliminate-unused-debug-symbols	消除与程序中未使用的任何 C/C++符号相关的调试信息。
-g	产生调试信息。
-Q	打印每个轮次的函数名称和统计信息。
-save-temps[=*dir*]	不要删除中间文件。

5.7.6.1 Eliminate-unused-debug-symbols 选项

-feliminate-unused-debug-symbols 选项消除与程序中未使用的任何 C/C++符号相关的调试信息。如果未指定该选项的形式，则默认使用这种形式。消除该信息将会减小输出文件大小并缩短输出文件加载时间。

如果要为所有符号生成调试信息，则使用该选项的-fno-eliminate-unused-debug-symbols 形式。

5.7.6.2 G：产生调试信息选项

-g 选项指示编译器产生附加信息，硬件工具可以使用这些信息来调试程序。

该选项可与-O 配合使用，以便能够调试经过优化的代码。优化代码所采取的捷径有时可能会产生令人惊讶的结果：

- 一些已声明的变量可能根本不存在
- 控制流可能短暂地异常转移
- 某些语句可能因为计算常量结果或已得到其值而不会被执行
- 某些语句可能因为被移出循环而在不同的位置执行

尽管如此，证明还是可以对优化输出进行调试。这使得对可能存在问题的程序使用优化器是合理的。

5.7.6.3 Q：打印函数信息选项

-Q 选项指示编译器在编译每个函数时打印其名称，并在完成时打印关于每个轮次的统计信息。

5.7.6.4 Save-temps 选项

-save-temps 选项指示编译器在编译完成后保留临时文件。您可能会发现生成的.i 和.s 临时文件对于故障排除特别有用，当您输入支持工单时，Microchip 支持团队会经常使用这两个临时文件。

中间文件将放置在当前目录中，并具有一个基于相应源文件的名称。因此，使用-save-temps 编译 foo.c 将产生 foo.i、foo.s 和 foo.o 目标文件。

-save-temps=cwd 选项等效于-save-temps。

该选项的-save-temps=obj 形式类似于-save-temps，但是如果指定了-o 选项，则这些临时文件将与输出目标文件置于同一目录中。如果未指定-o 选项，则-save-temps=obj 开关的行为类似于 - save-temps。

以下示例将创建 dir/xbar.i 和 dir/xbar.s，因为使用了-o 选项。

```
xc32-gcc -save-temps=obj -c bar.c -o dir/xbar.o
```

5.7.7 用于控制优化的选项

下表所列的选项用于控制编译器优化，这些选项将在后面的章节中详细介绍。

对于在使用优化级别 2 或更低的优化级别时默认使能的选项，即使没有编译器许可证（免费版）也可以使用并产生作用。任何形式的禁止优化的选项都可以无条件使用。

表 5-13. 常规优化选项

选项 （链接至说明部分）	版本	控制
-O0	所有版本	不优化。
-O -O1	所有版本	优化级别 1。
-O2	所有版本	优化级别 2。
-O3	仅专业版	针对速度的优化。
-Og	所有版本	减少优化以便更好地进行调试。
-Os	仅专业版	针对大小的优化。
-falign-functions[=*n*] -f[no-]align-functions	所有版本	将函数起始位置对齐。
-falign-labels[=*n*] -f[no-]align-labels	所有版本	将所有转移目标对齐。
-falign-loops[=*n*] -f[no-]align-loops	所有版本	将循环对齐。
-f[no-]caller-saves	所有版本	分配到会被函数调用破坏的寄存器中。
-f[no-]cse-follow-jumps	所有版本	定制公共子表达式消除优化。
-f[no-]cse-skip-blocks	所有版本	定制公共子表达式消除优化。
-f[no-]data-sections	所有版本	将对象放入输出文件中的某个段。
-f[no-]defer-pop	所有版本	何时弹出函数调用的参数。
-f[no-]expensive-optimizations	所有版本	开销相对较大的次要优化。
-f[no-]function-cse	所有版本	函数地址的位置。
-f[no-]function-sections	所有版本	将函数放入输出文件中的某个段。
-f[no-]gcse	所有版本	全局公共子表达式消除轮次。
-f[no-]gcse-lm	所有版本	定制处理装入序列的公共子表达式消除。
-f[no-]gcse-sm	所有版本	定制处理存储序列的公共子表达式消除。
-finline	所有版本	控制对标有 inline 关键字的函数的处理。
-f[no-]inline-functions	所有版本	将所有简单函数合并到其调用方中。
-f[no-]inline-limit=n	所有版本	内联函数的大小限制。
-f[no-]keep-inline-functions	所有版本	输出内联函数的一个独立的运行时可调用版本。
-f[no-]keep-static-consts	所有版本	static const 对象的输出。
-f[no-]lto	仅专业版	标准链接时优化器。
-f[no-]omit-frame-pointer	所有版本	对于不需要帧指针的函数，在寄存器中保留帧指针。
-f[no-]optimize-sibling-calls	所有版本	优化同属和尾递归调用。
-m[no-]pa	专业版	程序抽象优化。
-f[no-]peephole -f[no-]peephole2	所有版本	特定于机器的窥孔优化。
-f[no-]rename-registers	所有版本	使用空闲寄存器以避免经过调度的代码中的假依赖性。

.....（续）

选项（链接至说明部分）	版本	控制
-f[no-]rerun-cse-after-loop	所有版本	何时应执行公共子表达式消除优化。
-f[no-]rerun-loop-opt	所有版本	旧版选项，将被忽略。
-f[no-]schedule-insns -f[no-]schedule-insns2	所有版本	重新排序指令以消除指令停顿。
-f[no-]strength-reduce	所有版本	旧版选项，将被忽略。
-f[no-]strict-aliasing	所有版本	采用适用于所编译语言的最严格别名规则。
-f[no-]thread-jumps	所有版本	跳转至转移优化。
-f[no-]toplevel-reorder	所有版本	重新排序顶层函数、变量和 asm 语句。
-f[no-]unroll-loops -f[no-]unroll-all-loops	所有版本	循环展开。

5.7.7.1 O0：0 级优化

-O0 选项仅执行基本优化。这是未指定-O 选项时的默认优化级别。

选择该优化级别后，编译器的目标是降低编译开销，以及使调试产生预期的结果。

使用该优化级别进行编译时，各语句之间是相互独立的。如果使用语句之间的断点停止程序，则可以为任意变量赋予新值或将程序指针更改为指向函数中的任何其他语句，并得到预期从源代码获得的结果。

编译器只会将声明为 register 的变量分配到寄存器中。

5.7.7.2 O1：1 级优化

-O1 或-O 选项用于请求 1 级优化。

使用-O1 时进行的优化旨在缩短代码长度和执行时间，但可调试性仍保持在合理水平。

无需编译器许可证即可使用该级别。

5.7.7.3 O2：2 级优化选项

-O2 选项用于请求 2 级优化。

在此级别下，编译器几乎执行了在不涉及空间与速度间权衡的情况下支持的所有优化。

无需编译器许可证即可使用该级别。

5.7.7.4 O3：3 级优化选项

-O3 选项用于请求 3 级优化。

该选项将请求所有支持的优化以缩短执行时间，但程序的大小可能会增大。

只有具备编译器许可证才能使用该级别。

5.7.7.5 Og：更好调试选项

-Og 选项用于禁止会严重干扰调试的优化，从而在保持快速编译和良好调试体验的同时提供合理的优化水平。

该选项为标准“编辑-编译-调试”周期选择最佳优化级别。它将使能不会干扰调试的所有 1 级优化标志，被禁止的优化无法通过其对应的选项重新使能。与 0 级优化相比，该优化级别更适合用于生成可调试的代码，因为 0 级优化会禁止一些收集调试信息的编译器轮次。

5.7.7.6 Os：s 级优化选项

-Os 选项用于请求针对空间的优化。

该选项将请求所有支持的通常不会增加代码长度的优化。

只有具备编译器许可证才能使用该级别。

5.7.7.7 Align-functions 选项

-falign-functions=*n* 选项将函数起始位置对齐到下一个大于 *n* 的 2 的幂，最多跳过 *n* 个字节。对于 PIC32C/SAM 器件，n 不得超过 64。

例如，-falign-functions=32 将函数对齐到下一个 32 字节边界处；但是，-falign-functions=24 仅在跳过的字节数不超过 23 字节时才能将函数对齐到下一个 32 字节边界处。

该选项在优化级别-O2 和-O3 下自动使能。

该选项的-fno-align-functions 形式等效于-falign-functions=1，暗示不对函数进行对齐。

该选项的-falign-functions 形式（无参数）仅执行常规对齐，即对于使用 Arm 指令集的代码对齐到 4 字节边界处，对于使用 Thumb 指令集的代码对齐到 2 字节边界处。

5.7.7.8 Align-labels 选项

-falign-labels=*n* 选项将所有转移目标对齐到下一个大于 *n* 的 2 的幂，最多跳过 *n* 个字节。例如，-falign-labels=8 将函数对齐到下一个 8 字节边界处；但是，-falign-functions=9 仅在跳过的字节数不超过 9 字节时才能将函数对齐到下一个 16 字节边界处。

该选项很容易会使代码速度变慢，因为它必须在正常代码流中到达转移目标时插入空操作。

如果使用选项-falign-loops 或-falign-jumps 并且二者中任一选项的参数大于 *n*，则改为使用其参数值来确定如何对齐标号。

该选项在优化级别-O2 和-O3 下自动使能。

该选项的-fno-align-labels 形式等效于-falign-labels=1，暗示不对函数进行对齐。该选项的-falign-labels 形式（无参数）也不执行对齐。

5.7.7.9 Align-loops 选项

-falign-loops=*n* 选项将循环对齐到下一个大于 *n* 的 2 的幂，最多跳过 *n* 个字节。例如，-falign-loops=32 将循环对齐到下一个 32 字节边界处；但是，-falign-loops=24 仅在跳过的字节数不超过 23 字节时才能将循环对齐到下一个 32 字节边界处。

该选项的-fno-align-loops 形式等效于-falign-loops=1，暗示不对函数进行对齐。该选项的-falign-loops 形式（无参数）也不执行对齐。

预期目的是循环会被执行许多次，以补偿所执行的所有空操作。

5.7.7.10 Caller-saves 选项

-fcaller-saves 选项允许编译器将值分配到会被函数调用破坏的寄存器中。如果将值分配到这些寄存器，则会在函数调用前后生成用于保存和恢复被破坏的寄存器的额外代码。仅当该代码的预期性能比其他方式产生的效果更好时，才会进行这种分配。

该选项在优化级别-O2、-O3 和-Os 下自动使能。

该选项的-fno-caller-saves 形式永远不会将值分配到会被函数破坏的寄存器中。

5.7.7.11 Cse-follow-jumps 选项

-fcse-follow-jumps 选项指示公共子表达式消除（Common Subexpression Elimination，CSE）优化在跳转目标除了跳转没有任何其他路径可达时逐条扫描跳转指令。例如，当 CSE 遇到一个带有 else 子句的 if() 语句时，CSE 会在测试的条件为假时跟随跳转。

该选项在优化级别-O2、-O3 和-Os 下自动使能。

该选项的-fno-cse-follow-jumps 形式不执行该扫描。

5.7.7.12 Cse-skip-blocks 选项

-fcse-skip-blocks 选项执行的任务与-fcse-follow-jumps 选项类似，但会指示公共子表达式消除（CSE）优化遵循条件性跳过块的跳转。当 CSE 遇到没有 else 子句的简单 if() 语句时，该选项将遵循 if() 语句主体前后的跳转。

该选项在优化级别-O2、-O3 和-Os 下自动使能。

该选项的-fno-cse-skip-blocks 形式不执行该扫描。

5.7.7.13 Data-sections 选项

-fdata-sections 选项用于将每个对象放入其自己的段中，以帮助垃圾回收并可能缩短代码长度。

使能该选项时会将每个对象放入其自己以对象名命名的段中。与链接器执行的垃圾回收结合使用时（使用-Wl,--gc-sections 驱动程序选项使能），最终输出可能较小。但是，这会对其他代码生成优化产生负面影响，因此需要确认该选项对于每个项目是否有益。

使用链接时优化（-flto）进行编译时，该选项不起作用。请对需要放入指定段的对象使用 section 属性。

该选项的-fno-data-sections 形式不强制将每个对象放入惟一的段中。如果未指定该选项的形式，则默认使用这种形式。

5.7.7.14 Defer-pop 选项

-fdefer-pop 选项指定编译器应允许几个函数调用的函数参数累积在堆栈中，然后在一个步骤中将所有参数弹出堆栈。

该选项在优化级别-O1、-O2、-O3 和-Os 下自动使能。

该选项的-fno-defer-pop 形式让编译器在函数返回时立即弹出每个函数调用的参数。

5.7.7.15 Expensive-optimizations 选项

-fexpensive-optimizations 选项让编译器执行许多开销相对较大但效果甚微的优化。

该选项在优化级别-O2、-O3 和-Os 下自动使能。

该选项的-fno-expensive-optimizations 形式不执行这些优化。

5.7.7.16 Function-cse 选项

该选项的-ffunction-cse 形式允许将被调用函数的地址保存在寄存器中。如果未指定该选项的形式，则默认使用这种形式。

该选项的-fno-function-cse 形式不会将函数地址放入寄存器中。调用常量函数的每条指令显式地包含函数的地址。该选项产生的代码效率较低，但不使用该选项时执行的优化可能会混淆一些改变汇编器输出的奇怪技巧。

5.7.7.17 Function-sections 选项

-ffunction-sections 选项用于将每个函数放入其自己的段中，以帮助垃圾回收并可能缩短代码长度。

使能该选项时会将每个函数放入其自己以函数名命名的段中。与链接器执行的垃圾回收结合使用时（使用-Wl,--gc-sections 驱动程序选项使能），最终输出可能较小。但是，-ffunction-sections 选项会妨碍其他代码生成优化，因此需要确认该选项对于每个项目是否有益。

另请注意，如果删除了函数，其调试信息将保留在 ELF 文件中，这可能会危及代码的可调试性。当显示源代码信息时，调试器会寻找地址与其保存的代码所属的函数之间的关联，可能会误认为某个地址属于已删除的函数。由于已删除的函数没有分配存储器，其“分配的”地址会保持 0 不变，因此调试器在解析更接近 0 的地址时更有可能显示对已删除函数的引用。

使用链接时优化（-flto）进行编译时，该选项不起作用。请对需要放入指定段的函数使用 section 属性。

该选项的-fno-function-sections 形式不会强制将每个函数放入惟一的段中。如果未指定该选项的形式，则默认使用这种形式。

5.7.7.18 Gcse 选项

-fgcse 选项执行全局公共子表达式消除轮次。该轮次也会执行全局常量和复制传播。

该选项在优化级别-O2、-O3 和-Os 下自动使能。

该选项的-fno-gcse 形式不执行该轮次。

5.7.7.19 Gcse-lm 选项

-fgcse-lm 选项尝试让全局公共子表达式消除优化移动只能被向其中存储破坏的装入序列。这允许将包含装入/存储序列的循环更改为循环外的装入，以及循环内的复制/存储。

该选项随-fgcse 一起使能。

该选项的-fno-gcse-lm 形式不移动这些序列。

5.7.7.20 Gcse-sm 选项

-fgcse-sm 选项在全局公共子表达式消除优化之后运行存储移动轮次。该轮次会尝试将存储移出循环。与-fgcse-lm 配合使用时，包含装入/存储序列的循环可以更改为循环之前的装入和循环之后的存储。

该选项不会在任何优化级别下自动使能。

该选项的-fno-gcse-sm 形式不移动这些序列。

5.7.7.21 Inline 选项

-finline 选项允许编译器使用 inline 关键字来扩展函数。如果未指定该选项的形式，则默认使用这种形式。

该选项的-fno-inline 形式永远不会内联函数，即使函数已被标记为 inline 也不例外。

5.7.7.22 Inline-functions 选项

-finline-functions 选项考虑内联所有函数，即使函数未声明为 inline 也是如此。

如果对某个给定函数的所有调用均被内联，并且该函数声明为 static，则通常该函数不作为独立汇编程序输出。

该选项在优化级别-O3 和-Os 下自动使能。

该选项的-fno-inline-functions 形式永远不会内联未标记为 inline 的内联函数。

5.7.7.23 Inline-limit 选项

-finline-limit=*n* 选项控制标记为 inline 的函数的内联大小限制。参数 *n* 是伪指令条数（不包括参数处理），是函数大小的抽象测量。该值不代表汇编指令条数，因而，其确切含义可能会因编译器版本而变化。*n* 的默认值为 10000。

增大内联限制会导致被内联的代码更多，代价是增加编译时间和存储器消耗。

5.7.7.24 Keep-inline-functions 选项

-fkeep-inline-functions 选项确保输出函数的单独运行时可调用汇编代码，即使对给定函数的所有调用都是内联的，并且函数声明为 static。该开关不会影响 extern inline 函数的输出。

该选项的-fno-keep-inline-functions 形式不会为对其的所有调用均已内联的 static 函数发出输出。如果未指定该选项的形式，则默认使用这种形式。

5.7.7.25 Keep-static-consts 选项

-fkeep-static-consts 选项在禁止优化器时发出 static const 对象，即使这些对象未被引用。如果未指定该选项的形式，则默认使用这种形式。

该选项的-fno-keep-static-consts 形式强制编译器只有在某个对象被引用时才将其发出，无论是否使能优化器。

5.7.7.26 Lto 选项

-flto 选项用于运行标准链接时优化器。

使用源代码调用时，编译器会将代码的内部字节码表示添加到目标文件中的特定段中。当目标文件链接到一起时，所有函数主体均从这些段中读取并按照属于同一个转换单元的情况进行实例化。链接时间优化不需要整个程序的存在来运行。

要使用链接时优化器，请在编译时和最终链接期间指定-flto。例如：

```
xc32-gcc -c -O3 -flto -mprocessor=ATSAME70N20B foo.c
xc32-gcc -c -O3 -flto -mprocessor=ATSAME70N20B bar.c
xc32-gcc -o myprog.elf -flto -O3 -mprocessor=ATSAME70N20B foo.o bar.o
```

使能链接时优化的另一种（更简单）的方法是：

```
xc32-gcc -o myprog.elf -flto -O3 -mprocessor=ATSAME70N20B foo.c bar.c
```

字节码文件具有不同版本，并且存在严格的版本检查，因此在一个版本的 XC32 编译器中生成的字节码文件可能不适用于不同的版本。

该选项的-fno-lto 形式不运行标准的链接时优化器。如果未指定该选项，则默认使用这种形式。

5.7.7.27 Omit-frame-pointer 选项

-fomit-frame-pointer 选项用于指示编译器直接使用堆栈指针来访问堆栈中的对象，同时尽可能省略用于保存、初始化和恢复帧寄存器的代码。该选项在所有非零优化级别下自动使能。

该选项的否定形式-fno-omit-frame-pointer 可帮助调试优化后的代码，但不能保证将始终使用帧指针。

5.7.7.28 Optimize-sibling-calls 选项

-foptimize-sibling-calls 选项优化了同属调用（函数的最后一个操作是调用另一个与被调用方的堆栈占用空间兼容的函数）和尾递归调用（函数的最后一个操作是调用其自身）。

该选项在优化级别-O2、-O3 和-Os 下自动使能。

该选项的-foptimize-sibling-calls 形式不会使能这些优化。

5.7.7.29 Pa 选项

-mpa 选项针对 Cortex-M 器件和 Thumb/2 指令集使能程序抽象优化。只有具备编译器许可证才能使用该优化。

程序抽象会查找汇编代码指令的公共块，并将它们分解成新的可调用过程。随后使用对此过程的新调用替换指令块的每个实例。这实际上是一个反向内联过程。

重要：
确保在编译所有模块时指定该选项的同一形式。此外，必须在链接时指定该选项。

此选项在与响应文件链接时可用，并与链接时优化（-flto）兼容。

重要：程序抽象是一种积极的代码长度优化。这样，会对已编译应用程序的可调试性和性能产生负面影响。

当该选项生效时，可以使用 nopa 属性在每个函数上禁止过程抽象。

-mno-pa 选项禁止程序抽象优化。如果未指定该选项的形式，则默认使用这种形式。

5.7.7.30 Peephole/peephole2 选项

-fpeephole 选项使能特定于机器的窥孔优化。窥孔优化发生在编译期间的不同时间点。如果未指定该选项的形式，则默认使用这种形式。

该选项的-fpeephole2 形式使能高级窥孔优化。该选项在优化级别-O2、-O3 和-Os 下使能。

这些选项的-fno-peephole 和-fno-peephole2 形式分别禁止标准和高级窥孔优化。应同时使用这两个选项来完全禁止窥孔优化。

5.7.7.31 Rename-registers 选项

-frename-registers 选项尝试通过使用在寄存器分配之后留下的寄存器来避免经过调度的代码中的假依赖性。这种优化最有利于具有许多寄存器的处理器，但它会降低代码的可调试性，因为变量可能不再分配给“归位寄存器”。

使用-funroll-loops 时，此选项自动使能。

该选项的-fno-rename-registers 形式不使用留下的寄存器。如果未指定任何选项，且未使用-funroll-loops 选项，则默认使用这种形式。

5.7.7.32 Rerun-cse-after-loop 选项

-frerun-cse-after-loop 选项在执行循环优化之后重新运行公共子表达式消除（CSE）优化。

该选项在优化级别-O2、-O3 和-Os 下自动使能。

该选项的-fno-rerun-cse-after-loop 形式不会重新运行 CSE 优化。

5.7.7.33 Rerun-loop-opt 选项

-frerun-loop-opt 选项不再具有任何效果，且会被忽略以与旧项目兼容。

5.7.7.34 Schedule-insns/schedule-insns2 选项

-fschedule-insns 选项尝试对指令重新排序，以消除由于所需数据不可用而产生的指令停顿。

该选项在优化级别-O2 和-O3 下自动使能。

该选项的-fschedule-insns2 形式类似于-fschedule-insns，但它要求在执行寄存器分配之后再执行一次指令调度。

该选项在优化级别-O2、-O3 和-Os 下自动使能。

这些选项的-fno-schedule-insns 和-fno-schedule-insns2 形式不会对指令进行重新排序。

5.7.7.35 Strength-reduce 选项

-fstrength-reduce 选项不再具有任何效果，且会被忽略以与旧项目兼容。

5.7.7.36 Strict-aliasing 选项

-fstrict-aliasing 选项允许编译器采用适用于所编译语言的最严格别名规则。对于 C，这会基于表达式的类型激活优化。特别是，假定一种类型的对象永远不会与不同类型的对象位于同一地址，除非类型几乎相同。

例如，unsigned int 类型可以为 int 类型的别名，但不能为 void *或 double 类型的别名。字符类型可以为任何其他类型的别名。

请特别注意类似以下形式的代码：

```
union a_union {
  int i;
  double d;
};
int f() {
  union a_union t;
  t.d = 3.0;
  return t.i;
}
```

不读取最近写入的联合成员，而读取其他联合成员的做法（称为“类型双关”）很常见。即使对于-fstrict-aliasing，也允许类型双关，前提是通过 union 类型来访问存储器。上面的代码会按预期方式运行。但是，以下代码可能不会：

```
int f() {
  a_union t;
  int * ip;
  t.d = 3.0;
  ip = &t.i;
  return *ip;
}
```

该选项在优化级别-O2、-O3 和-Os 下自动使能。

该选项的-fno-strict-aliasing 形式放宽了别名规则，从而会阻止执行某些优化。

5.7.7.37 Thread-jumps 选项

-fthread-jumps 选项会使能检查，以确定比较的转移目标是否为包含在第一个转移之内的进一步比较。如果是这样，则将第一个转移重定向到第二个转移的目标位置或紧随其后的位置，这取决于第二个转移条件是真还是假。

该选项在优化级别-O2、-O3 和-Os 下自动使能。

该选项的-fno-thread-jumps 的形式不执行这些检查。

5.7.7.38 Toplevel-reorder 选项

-ftoplevel-reorder 选项允许编译器重新排序顶层函数、变量和 asm 语句，在这种情况下，它们的输出顺序可能与它们在输入文件中出现的顺序不同。该选项还允许编译器删除未引用的 static 变量。

在-O0 级别下使能。使用-fno-toplevel-reorder 选项显式禁止时，它隐含着-fno-section-anchors，该选项在某些目标上会在-O0 级别下使能。

5.7.7.39 Unroll-loops/unroll-all-loops 选项

-funroll-loops 和-funroll-all-loops 选项用于控制针对速度的优化，此类优化会尝试移除循环中的转移延时。展开的循环通常会提高所生成代码的执行速度，但代价是增大代码长度。

-funroll-loops 选项用于展开在编译时或代码进入循环时可以确定迭代次数的循环。该选项通过-fprofile-use 使能。而-funroll-all-loops 选项能展开所有循环，甚至在迭代次数未知时也是如此。在提高执行速度方面，它通常不如-funroll-loops 选项有效。

这两个选项都隐含着-frerun-cse-after-loop、-fweb 和-frename-registers。

这些选项的-fno-unroll-loops 和-fno-unroll-all-loops 形式不会展开任何循环，并且如果未指定选项，则默认使用这些形式。

5.7.8 用于控制预处理器的选项

下表所列的选项用于控制预处理器，这些选项将在后面的章节中详细讨论。

表 5-14. 预处理器选项

选项 （链接至说明部分）	定义
-C	保留注释。
-d*letters*	对预处理宏进行调试转储。
-Dmacro[=defn]	定义宏。
-H	打印使用的每个头文件的名称。
-imacros file	仅包含文件宏定义。
-include file	在处理常规输入文件之前，将文件作为输入进行处理。
-M	生成 make 规则。
-MD	将相关性信息写入 file。
-MF file	指定相依性关系文件。
-MG	忽略缺少的头文件。
-MM	为加引号的头文件生成 make 规则。
-MMD	为用户头文件生成 make 规则。
-MP	为相依性关系添加虚假目标。
-MQ	使用引号更改规则目标。
-MT *target*	更改规则目标。
-nostdinc	从头文件搜索中省略系统目录。
-P	不要生成#line 伪指令。
-fno-show-column	省略诊断中的列编号。
-trigraphs	支持 ANSI C 三字符组合。
-Umacro	取消定义宏。
-undef	不要预定义非标准宏。

5.7.8.1 C：保留注释选项

-C 选项用于指示预处理器不要丢弃输出中的注释。将该选项与-E 选项结合使用可查看已注释但经过预处理的源代码。

5.7.8.2 d：预处理器调试转储选项

-d*letters* 选项可使预处理器在编译期间生成由 *letters* 指定的调试转储。该选项应与-E 选项结合使用。

-dM 选项为预处理器执行过程中定义的所有宏（包括预定义宏）生成#define 指令的列表。如果定义了替换字符串，则这些指令将指示替换字符串。例如，输出可能包括：

```
#define _CP0_BCS_TAGLO(c,s) _bcsc0(_CP0_TAGLO, _CP0_TAGLO_SELECT, c, s)
#define PORTE PORTE
#define _LATE_LATE1_LENGTH 0x00000001
#define _IPC22_w_POSITION 0x00000000
```

您可以使用该选项来显示那些由编译器或头文件预定义的宏。

下表列出了-d 可接受的字母参数。

表 5-15. 预处理器调试信息

字母	产生的结果
D	与-dM 的输出类似，但缺少预定义的宏。正常的预处理器输出也包含在此输出中。
M	完整的宏输出，如本表之前的正文所述。
N	与-dD 的输出类似，但每个定义仅输出宏名称。

5.7.8.3 D：定义宏

-D*macro* 选项允许定义一个预处理器宏，而该选项的-D*macro*=*text* 形式还允许随宏一起指定用户定义的替换字符串。该选项和宏名称之间可存在空格。

当宏名称后面没有替换文本时，-D*macro* 选项将定义一个名为 *macro* 的预处理器宏，并将其替换文本指定为 1。其用途相当于在每个正在编译的模块的顶部放置#define *macro* 1。

该选项的-D*macro*=*text* 形式用于定义一个名为 *macro* 的预处理器宏并指定替换文本。其用途相当于在每个正在编译的模块的顶部放置#define *macro* *text*。

该选项的任一种形式都会创建一个标识符（宏名称），其定义可以通过#ifdef 或#ifndef 伪指令检查。例如，当使用选项-DMY_MACRO（或-D MY_MACRO）并编译以下代码时：

```
#ifdef MY_MACRO
int input = MY_MACRO;
#endif
```

将编译 int 变量 input 的定义，并将变量赋值为 1。

如果上面的示例代码改为使用选项-DMY_MACRO=0x100 进行编译，则最终编译的变量定义将是： int input = 0x100;

有关如何使用替换文本的说明，请参见预处理器算术。

将宏定义为 C 字符串字面值时需要对字符串使用的引号字符（" "）进行转义。如果要包含引号并将其传递给编译器，则应使用反斜杠字符（\）对其进行转义。如果字符串包括空格字符，则还应在字符串两端加引号。

例如，要传递 C 字符串"hello world"（替换文本中包含引号字符和空格），应使用-DMY_STRING="\"hello world\""。另外，也可以在整个选项两端加引号："-DMY_STRING=\"hello world\""。这些格式可以在任何平台上使用。只有 macOS 和 Linux 系统允许对空格字符进行转义（如-DMY_STRING=\"hello\ world\"中），Windows 系统并不允许，因此建议在整个选项的两端加引号以确保可移植性。

在任何-U 选项之前处理命令行中的所有-D 实例。

5.7.8.4 H：打印头文件选项

除了一些常规操作之外，-H 选项还可将使用的每个头文件的名称打印到控制台。

5.7.8.5 Imacros 选项

-imacros *file* 选项用于按照-include 选项的方式处理指定文件，但文件扫描产生的任何输出都将被丢弃。文件定义的宏在处理期间保持已定义状态。由于从该文件生成的输出被丢弃，因此该选项的惟一作用是使文件中定义的宏在主输入中可用。

命令行上的任何-D 和-U 选项均始终在-imacros 选项前处理，而与这些选项的放置顺序无关。所有-include 和-imacros 选项按这些选项的写入顺序处理。

5.7.8.6 Include 选项

-include *file* 选项按照#include "file"出现在主源文件的第一行来处理 *file*。实际上，将先编译 *file* 的内容。命令行上的任何-D 和-U 选项均始终在-include 选项前处理，而与这些选项的写入顺序无关。所有-include 和-imacros 选项按这些选项的写入顺序处理。

5.7.8.7 M：生成 Make 规则

-M 选项指示预处理器输出适合 make 的规则，该规则描述每个目标文件的相依性关系。

对于每个源文件，预处理器都输出一个 make 规则，其目标是该源文件的目标文件名，其相依性关系是它包含的所有头文件。 此规则可以是单行，如果过长，可以用反斜杠换行序列延续。

规则列表打印在标准输出上，而不是预处理的 C 程序中。

-M 选项隐含-E。

5.7.8.8 MD：将相依性关系信息写入文件选项。

-MD 选项用于将相依性关系信息写入文件。

该选项与-M 类似，只是将相依性关系信息写入文件并继续编译。包含相依性关系信息的文件与扩展名为.d 的源文件同名。

5.7.8.9 MF：指定相依性关系文件选项

-MF *file* 文件选项用于指定一个文件，以便在其中写入-M 或-MM 选项的相依性关系。如果没有提供-MF 选项，则预处理器会将规则发送到预处理输出被发送到的位置。

与驱动程序选项-MD 或-MMD 一起使用时，-MF 会覆盖默认的相依性关系输出文件。

5.7.8.10 MG：忽略缺少的头文件选项

-MG 选项用于将缺少的头文件视为生成的文件并将其添加到相依性关系列表中，而且不会产生错误。

该选项假定缺少的文件与源文件位于相同的目录中。如果指定了-MG，则也必须指定-M 或-MM。-MD 或-MMD 不支持该选项。

5.7.8.11 MM：为引用的头文件生成 Make 规则选项

-MM 选项执行的任务与-M 基本相同，只是系统头文件不会包含在输出中。

5.7.8.12 MMD：为用户头文件生成 Make 规则选项

-MMD 选项执行的任务与-MD 基本相同，只是用户头文件会包含在输出中。

5.7.8.13 MP：为相依性关系添加虚假目标选项

-MP 选项用于指示预处理器为主文件以外的每种相依性关系添加虚假目标，使它们没有依赖的目标。如果移除头文件但不更新要匹配的 make 文件，这些 MP 规则可以解决 make 错误。

以下是典型的输出：

```
test.o: test.c test.h
test.h:
```

5.7.8.14 MQ：使用引号更改规则目标选项

-MQ 选项与-MT 类似，但将为被 make.视为特殊的任何字符加引号。

```
-MQ '$(objpfx)foo.o' gives $$(objpfx)foo.o: foo.c
```

默认目标像通过-MQ 指定一样自动加引号。

5.7.8.15 MT：更改规则目标选项

-MT 目标选项用于更改因产生相依性关系而发出的规则的目标。

默认情况下，预处理器将接受主输入文件的名称（包含任何路径），删除任何文件后缀（如.c）并附加平台的常规对象后缀。结果就是目标。-MT 选项会将目标设置为与您指定的字符串完全相同。如果需要多个目标，可以将它们指定为-MT 的单个参数，或使用多个-MT 选项。

例如：

```
-MT '$(objpfx)foo.o' might give $(objpfx)foo.o: foo.c
```

5.7.8.16 Nostdinc 选项

-nostdinc 选项用于阻止预处理器搜索头文件所在的标准系统目录。只搜索通过-I 选项指定的目录（如果适用，还搜索当前目录）。

使用-nostdinc 和-iquote 可以将包含文件搜索路径限定为显式指定的目录。

5.7.8.17 P：不要生成#line 伪指令选项

-P 选项用于指示预处理器不要在预处理输出中生成#line 伪指令，与-E 选项结合使用。

5.7.8.18 No-show-column 选项

-fno-show-column 选项用于控制诊断中是否将打印列编号。

如果由不了解列编号的程序（如 DejaGnu）扫描诊断，可能需要该选项。

5.7.8.19 Trigraphs 选项

-trigraphs 选项用于开启对 ANSI C 三字符组的支持。-ansi 选项也具有该作用。

5.7.8.20 U：取消定义宏

-U*macro* 选项用于取消定义宏 macro。

该选项将取消定义任何内置宏或使用-D 定义的宏。所有-U 选项在所有-D 选项之后、所有-include 和-imacros 选项之前评估。

5.7.8.21 Undef 选项

-undef 选项用于阻止预定义任何系统特定或 GCC 特定的宏（包含架构标志）。

5.7.9 用于汇编的选项

下表所列的选项用于控制汇编器操作，这些选项将在后面的章节中详细讨论。

表 5-16. 汇编选项

选项 （链接至说明部分）	定义
-Wa,*option*	将 option 传递给汇编器。

5.7.9.1 Wa：将 Option 传递给汇编器选项

-Wa,*option* 选项用于将其 *option* 参数直接传递给汇编器。如果 *option* 包含逗号，则表示以逗号分隔的多个选项。例如，-Wa,-a 将-a 选项传递给汇编器，以请求生成汇编列表文件。

5.7.10 用于链接的选项

下表所列的选项用于控制链接器操作，这些选项将在后面的章节中详细讨论。如果使用选项-c、-S 或-E 中的任何一个，将不会执行链接器。

表 5-17. 链接选项

选项 （链接至说明部分）	定义
--dinit-compress=*level*	将指定级别的优化应用于数据初始化模板，该模板会初始化 RAM 中的对象和 ramfunc 函数。
-l*library*	链接时搜索名为 *library* 的库。
-nodefaultlibs	链接时不要使用标准系统库。
-nostdlib	在链接时不使用标准系统启动文件或库。
-s	从输出中移除所有符号表和重定位信息。
-u symbol	添加将在链接阶段出现的未定义符号。
-Wl,*option*	将选项传递给链接器。
-Xlinker *option*	将系统特定选项传递给链接器。

5.7.10.1 Dinit-compress 选项

`-dinit-compress=level` 选项用于使能将优化应用于指定级别的数据初始化模板，该模板会初始化 RAM 中的对象和 `ramfunc` 属性函数。有关数据初始化模板和记录格式操作的更多信息，请参见初始化对象和 RAM 函数。

当参数 *level* 设置为 0 时，编译器将使用未经优化的传统 `.dinit` 格式，这种格式由 v4.10 和之前版本的编译器使用。

级别 1 会将那些已初始化的对象合并到同一个 `.dinit` 记录下的连续存储器中。

级别 2 执行 1 级优化，但会另外将相同（非零）初始值分组到格式#2 或格式#3 的记录中（如初始化对象和 RAM 函数中所述），同时分别指定将被多次复制的单个 16 位或 32 位初始值。当初始值序列中存在 16 位重复值（例如 0x13571357...）时，会选择格式 2 记录；当初始值序列中存在 32 位重复值（例如 0x6701238067012380...）时，会选择格式 3 记录。此选项会增加运行时启动代码的大小。

级别 3 可以执行较低级别中提供的任何优化，并且还可以对对象和 `ramfunc` 函数执行基于 PackBits 的运行长度编码压缩，并将其存储在记录格式 4 中。仅当 pack-bits 解压缩算法节省的空间量等于或大于需要额外链接的解压缩算法的大小时，才会使用该算法；但是，初始化的任何重复值（如上面所述）都可能会增加运行时启动程序的大小。这是未指定选项时的默认优化级别。

5.7.10.2 L：指定库文件选项

`-llibrary` 选项用于在链接时扫描已命名库文件中的未解析符号。

使用该选项时，链接器将搜索标准目录列表中名称为 *library*`.a` 的库。搜索的目录包括标准系统目录，以及用 `-L` 选项指定的目录。

链接器按照库文件和目标文件的指定顺序处理它们，因此将该选项置于命令中的不同位置将产生不同的影响。选项（按照相应顺序）`foo.o -llibz bar.o` 在文件 `foo.o` 之后、但在 `bar.o` 之前搜索库 `libz.a`。如果 `bar.o` 引用了 `libz.a` 中的函数，则可能不会装入这些函数。

通常，通过这种方式找到的文件是库文件（成员为目标文件的归档文件）。链接器处理归档文件的方式是扫描归档文件中用于定义已引用但尚未定义的符号的成员。但如果找到的文件是一个普通目标文件，则它以通常的方式链接。

使用 `-l` 选项（如 `-lmylib`）与指定文件名（如 `mylib.a`）之间的惟一差别在于编译器将在多个通过 `-L` 选项指定的目录中搜索使用 `-l` 指定的库。

默认情况下，会指示链接器搜索 `<install-path>/lib` 以查找使用 `-l` 选项指定的库。使用环境变量可以改写该行为。

另请参见 `INPUT` 和 `OPTIONAL` 链接描述文件伪指令。

5.7.10.3 Nodefaultlibs 选项

`-nodefaultlibs` 选项可阻止标准系统库链接到项目。仅将您指定的库传递给链接器。

编译器可能会生成对 `memcmp`、`memset` 和 `memcpy` 的调用，即使指定了此选项。 由于这些符号通常由标准编译器库中的入口解析，因此当指定此选项时，它们应通过其他某种机制提供。

5.7.10.4 Nostdlib 选项

`-nostdlib` 选项可阻止标准系统启动文件和库链接到项目。没有启动文件，仅将您指定的库传递给链接器。

编译器可能会生成对 `memcmp()`、`memset()` 和 `memcpy()` 的调用。这些入口通常由标准编译器库中的入口解析。指定此选项时，应通过其他某种机制提供这些入口点。

此外，您还必须提供您自己的 `__pic32c_data_initialization()` 和 `__libc_init_array()` 函数的实现，这两个函数由默认器件启动代码调用。

5.7.10.5 S：移除符号信息选项

`-s` 选项用于从输出中移除所有符号表和重定位信息。

5.7.10.6 U：添加未定义的符号选项

`-u` *`symbol`* 选项用于添加将在链接阶段出现的未定义符号。为了解析符号，链接器将搜索库模块中的符号定义，因此如果要强制链接库模块，该选项会十分有用。可以合法地通过不同符号多次使用该选项来强制装入额外的库模块。

5.7.10.7 Wl：将 Option 传递给链接器选项

`-Wl,`*`option`* 选项将 *`option`* 传递给链接器应用程序，在此它将被解释为链接器选项。如果 *`option`* 包含逗号，则表示以逗号分隔的多个选项。任何指定的链接器选项都将添加到驱动程序传递的默认链接器选项中，并且这些选项将在链接器的默认选项之前执行。

5.7.10.8 Xlinker 选项

`-Xlinker` *`option`* 选项将 *`option`* 传递给链接器，在此它将被解析为链接器选项。可以使用该选项来提供编译器不知道如何识别的特定于系统的链接器选项。

5.7.11 用于目录搜索的选项

下表所列的选项用于控制搜索目录操作，这些选项将在后面的章节中详细讨论。

表 5-18. 目录搜索选项

选项 （链接至说明部分）	定义
`-B`*`prefix`*	该选项指定在何处查找可执行文件、库、包含文件和编译器数据文件。
`-I` *`dir`*	添加用于搜索头文件的目录
`-Idirafter` *`dir`*	在所有其他路径下均未搜到所需头文件之后添加用于搜索头文件的目录
`-Iquote` *`dir`*	在处理-I 选项目录之前添加用于搜索“带引号”头文件的目录
`-L`*`dir`*	指定额外的库搜索目录。

5.7.11.1 B：指定编译器组件搜索路径选项

`-B`*`prefix`* 选项指定可找到可执行文件、库、包含文件和编译器数据文件的路径。

编译器驱动程序运行一个或多个内部应用程序 `xc32-cpp`、`xc32-as` 和 `xc32-ld`。它会尝试使用 *`prefix`* 作为它尝试运行的每个应用程序的前缀。

对于要运行的每个应用程序，编译器驱动程序会先尝试添加 *`prefix`*。如果找不到应用程序，则驱动程序将搜索由应用程序的 `PATH` 环境变量指定的搜索路径。

如果 *`prefix`* 指定目录名称，则该路径也适用于链接器使用的库，因为驱动程序会将其转换为链接器的`-L` 选项。这也适用于包含文件搜索路径，因为编译器会将这些选项转换为用于预处理器的`-isystem` 选项。在这种情况下，编译器会在前缀之后附加 `include`。

5.7.11.2 I：指定包含文件搜索路径选项

`-I`*`dir`* 选项将目录 *`dir`* 添加到用于搜索头文件的目录列表的开头。该选项和目录名称之间可能存在空格。

该选项可以指定绝对路径或相对路径，如果要搜索多个其他目录，则可以多次使用该选项，这种情况下将从左到右进行扫描。对该选项指定的目录完成扫描后，将搜索标准系统目录。

在 Windows 操作系统下，使用目录反斜杠字符可能会无意间形成转义序列。如果要指定的包含文件路径以目录分隔符结尾并加引号，请使用`-I "E:\\"`形式（而非`-I "E:\"`），以避免形成转义序列`\"`。请注意，MPLAB X IDE 将为您在项目属性中指定的任何包含文件路径加双引号，并且搜索路径是相对于输出目录（而不是项目目录）而言。

注：不要使用该选项指定任何 MPLAB XC32 系统包含路径。编译器驱动程序会自动为您选择默认语言库及其各自的包含文件目录。手动添加系统包含路径可能会破坏这种机制，导致将错误的文件编译到项目中，从而导致包含文件和库之间的冲突。请注意，向项目属性中添加系统包含路径从来不是建议的做法。

5.7.11.3 Idirafter 选项

`-idirafter` *`dir`* 选项在预处理期间将目录 *`dir`* 添加到用于搜索头文件的目录列表中。仅在所有其他搜索路径（包括标准搜索目录以及通过`-I` 和`-Iquote` 选项指定的路径）中均完成搜索后才搜索该目录。如果多次使用该选项，则将按照命令行上出现的顺序从左到右搜索它们指定的目录。

5.7.11.4 Iquote 选项

`-iquote` *`dir`* 选项在预处理期间将目录 *`dir`* 添加到用于搜索头文件的目录列表中。使用该选项指定的目录仅适用于伪指令的加引号形式（如`#include "file"`），并且仅在当前工作目录中完成搜索后才搜索该目录。如果多次使用该选项，则将按照命令行上出现的顺序从左到右搜索它们指定的目录。

5.7.11.5 L：指定库搜索路径选项

`-L`*`dir`* 选项允许您指定一个额外的目录，用于搜索已使用`-l` 选项指定的库文件。编译器将自动搜索标准库位置，因此仅在链接自己的库时才需要使用该选项。

5.7.12 用于代码生成约定的选项

下表所列的选项用于控制在生成代码时所用的独立于机器的约定，这些选项将在后面的章节中详细讨论。

表 5-19. 代码生成约定选项

选项 （链接至说明部分）	定义

`-fargument-alias`
`-fargument-noalias`
`-fargument-noalias-global`
	指定参数之间以及参数与全局数据之间可能存在的关系。

`-fcall-saved-`*`reg`*	将 *`reg`* 视为由函数保存的可分配寄存器。
`-fcall-used-`*`reg`*	将 *`reg`* 视为被函数调用破坏的可分配寄存器。
`-f[no-]common`	控制在无初始化情况下定义的全局变量的位置。
`-f[no-]exceptions`	生成用于传播异常的额外代码。
`-ffixed-`*`reg`*	将 *`reg`* 视为代码中不会使用的固定寄存器。
`-f[no-]ident`	忽略`#ident` 伪指令。
`-fpack-struct`	将所有结构体成员连续存放在存储器中，中间不留空隙。
`-f[no-]pcc-struct- return`	将短 `struct` 和 `union` 值返回到存储器中，而不是返回到寄存器中。
`-f[no-]short-enums`	指定 `enum` 类型的长度。
`-f[no-]verbose-asm`	在生成的汇编代码中放入额外的注释信息，使它更具可读性。

5.7.12.1 Argument-alias 选项

该组选项指定参数之间以及参数与全局数据之间可能存在的关系。

`-fargument-alias` 选项指定参数可以互为别名，并且可以为全局存储的别名。

`-fargument-noalias` 指定参数不互为别名，但可以为全局存储的别名。

`-fargument-noalias-global` 选项指定参数不互为别名，也不为全局存储的别名。

每种语言会自动使用语言标准所要求的适当选项，因此您不需要自己使用这些选项。

5.7.12.2 Call-saved-reg 选项

`-fcall-saved-`*`reg`* 选项会让编译器将名为 *`reg`* 的寄存器视为由函数保存的可分配寄存器。使用该选项编译的函数将在使用指定寄存器时对其进行保存和恢复。

该选项中指定的寄存器可分配给跨函数调用存活的对象或临时对象。

使用该选项指定帧指针或堆栈指针寄存器是错误的。使用该选项指定在机器的执行模型中具有固定重要作用的其他寄存器可能会导致代码失败。使用该选项指定用于返回函数值的寄存器也可能会导致代码失败。

该选项应对所有项目模块一致地使用。

5.7.12.3 Call-used-reg 选项

`-fcall-used-`*`reg`* 选项会让编译器将名为 *`reg`* 的寄存器视为被函数破坏的可分配寄存器。使用该选项编译的函数将在使用指定寄存器时不对其进行保存和恢复，这意味着寄存器的内容可能会丢失。

该选项中指定的寄存器可分配给不能跨函数调用存活的对象或临时对象。

使用该选项指定帧指针或堆栈指针寄存器是错误的。使用该选项指定在机器的执行模型中具有固定重要作用的其他寄存器可能会导致代码失败。使用该选项指定用于返回函数值的寄存器也可能会导致代码失败。

该选项应对所有项目模块一致地使用。

5.7.12.4 Common 选项

`-fcommon` 选项指示编译器允许链接器合并暂定定义。如果未指定该选项的形式，则默认使用这种形式。

在 C 标准中，没有存储类说明符或无初始化的文件范围对象的定义称为暂定定义。此类定义在指定`-fcommon` 选项后被视为外部引用，并将被放入公共存储器块。因此，如果另一个编译单元对同名对象有完整定义，则会合并定义并分配存储空间。如果无法找到完整定义，链接器将为暂定定义的对象分配惟一的存储空间。暂定定义不同于使用 `extern` 关键字进行的变量声明，后者永远不会分配存储空间。

在以下代码示例中：

extra.c
```
int a = 42;  /* full definition */
```

main.c
```
int a;       /* tentative definition */
int main(void) {
  ...
}
```

对象 `a` 在 `extra.c` 中定义，并在 `main.c` 中暂定定义。此类代码在使用`-fcommon` 选项时将会进行编译，因为链接器会在 `extra.c` 中将 `a` 的暂定定义解析为完整定义。

该选项的`-fno-common` 形式禁止链接器合并暂定定义，如果到达翻译单元的末尾仍未出现含初始化的定义，则将暂定定义视为完整定义。例如，编译上述代码将导致 `multiple definition of 'a'`错误，因为暂定定义与已初始化定义都会尝试为同一个对象分配存储空间。如果使用该选项的这种形式，则应将 `main.c` 中的 `a` 定义编写为 `extern int a;`以允许编译程序。

5.7.12.5 Exceptions 选项

`-fexceptions` 选项生成传播异常所需的额外代码。在编译需要与以 C++编写的异常处理程序进行正确互操作的 C 代码时，可能需要指定该选项。

该选项的`-fno-exceptions` 形式禁止异常处理。如果未指定该选项的形式，则默认使用这种形式。

确保在编译所有模块时和链接时指定该选项的同一形式。如果在链接时额外使用该选项，则链接器可选择在禁止 C++异常的情况下编译的库版本，从而缩短代码长度。

5.7.12.6 Fixed-reg 选项

`-ffixed-reg` 选项让编译器将名为 *`reg`* 的寄存器视为固定寄存器。编译器生成的代码永远不会使用该寄存器（除非该寄存器具有固定作用，例如用作堆栈指针或帧指针）。

寄存器名称可以是寄存器编号，例如`-ffixed-3` 指代 r3。

5.7.12.7 Ident 选项

`-fident` 选项强制编译器处理每一条`#ident` 伪指令，将伪指令的字符串常量参数放入目标文件的特殊段。如果未指定该选项的形式，则默认使用这种形式。

该选项的-fno-ident 形式强制编译器忽略每一条该伪指令。

5.7.12.8 Pack-struct 选项

-fpack-struct 选项将所有结构体成员连续存放在存储器中，而不在各结构体成员之间填充字节。通常不使用该选项，因为结构体成员可能会变得不对齐，并且访问它们所需的代码会变得不够优化。此外，此类结构体中各成员的偏移量与系统库中未连续存放（对齐）结构体的偏移量不一致。

代码必须特别小心地通过指针访问连续存放的结构体成员。一些 Cortex-M 器件不允许进行未对齐访问，尝试进行这种访问将导致硬故障异常。

-fno-pack-struct 选项选择一种将生成最优代码的结构体排列，可能会在各成员之间放置填充字节以确保它们与器件正确对齐。如果未指定该选项的形式，则默认使用这种形式。

5.7.12.9 Pcc-struct-return 选项

-fpcc-struct-return 选项强制编译器将较小的 struct 和 union 对象（其大小和对齐方式与整型类型一致）返回到存储器中，而不是返回到寄存器中。该约定的效率较低，但可以与使用其他编译器编译的文件兼容。

该选项的-fno-pcc-struct-return 形式将较小的 struct 和 union 对象返回到寄存器中。如果未指定该选项的形式，则默认使用这种形式。

5.7.12.10 Short-enums 选项

-fshort-enums 选项将最小可能整型类型（大小为 1、2 或 4 个字节）分配给 enum，以便可以容纳可能值的范围。

该选项的-fno-short-enums 形式强制每个 enum 类型的宽度为 4 字节（int 类型的大小）。 如果未指定该选项的形式，则默认使用这种形式。 使用该选项时生成的代码与不使用该选项时生成的代码无法二进制兼容。

5.7.12.11 Verbose-asm 选项

-fno-verbose-asm 选项在生成的汇编代码中放入额外的注释信息，使它更具可读性。

该选项的-fno-verbose-asm 形式会省略该额外信息，这在比较两个汇编文件时很有用。如果未指定该选项的形式，则默认使用这种形式。

6. C 标准问题

除非另外说明，否则该编译器是一种独立实现，符合针对编程语言的 ISO/IEC 9899:1990 标准（简称为 C90 标准）和 ISO/IEC 9899:1999 标准（简称为 C99 标准）。

6.1 不符合 C99 标准的方面

在 MPLAB XC32 C 编译器中实现的 C 语言存在不符合 C99 标准的方面，具体如以下各节所述。

6.1.1 不符合 C99 标准的库

例如，MPLAB XC32 不支持标准 pragma 伪指令`#pragma STDC FP_CONTRACT`。

此外，默认运行时启动代码在从 `main()` 返回后不会调用 `exit()`。这是为了减小代码长度；但可根据需要提供调用 `exit()` 的定制启动代码。

6.2 C99 标准的扩展

编译器允许使用 C 标准中未提供的某些源构造（详见以下章节）。

6.2.1 关键字扩展

编译器支持 C 标准中未提供的一些关键字和属性。这些关键字和属性属于基本 GCC 实现的一部分，所引用章节中的讨论基于标准 GCC 文档，并针对 GCC 的 32 位编译器移植版本的特定语法和语义进行讨论。

- 指定变量的属性—— 变量属性
- 指定函数的属性—— 函数属性
- 内联函数—— 内联函数
- 指定寄存器中的变量—— 变量属性
- 复数—— 复数数据类型
- 双字整数—— 整型数据类型
- 使用 `typeof` 引用类型—— 类型引用

6.2.2 语句和表达式扩展

编译器可编译使用 C 标准不支持的特性的表达式。这些特性属于基本 GCC 实现的一部分，所引用章节中的讨论基于标准 GCC 文档，并针对 GCC 的 32 位编译器移植版本的特定语法和语义进行讨论。

- 标号作为值—— 标号作为值
- 带省略操作数的条件语句—— 条件操作符的操作数
- case 范围—— case 范围
- 二进制常量——常量类型和格式

6.3 实现定义的行为

ANSI C 标准的某些特性具有实现定义的行为。这意味着一些 C 代码的确切行为会因编译器不同而不同。本文档将详细介绍 MPLAB XC32 C/C++编译器的确切行为，实现定义的行为也对其进行了全面总结。

7. 与器件相关的特性

MPLAB XC32 C/C++编译器支持许多特殊的 C/C++语言特性和扩展，这些特性和扩展旨在减轻生成基于 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。

7.1 器件支持

MPLAB XC32 C/C++编译器旨在支持所有 PIC32 器件。但是，这些系列中经常会发布新的器件。

7.2 器件头文件

有一个头文件建议在编写每个源文件时都包含。该文件为<xc.h>，它是一个通用文件，它会在您编译项目时包含其他特定于器件的头文件。

包含该文件可访问内核和外设模块的 SFR。更多信息，请参见从 C 代码中使用 SFR。

7.3 配置位访问

#pragma config 伪指令指定要编程到运行应用程序的器件中的配置字。该 pragma 伪指令可用于 C 或 C++程序。

所有 32 位目标器件都有多个配置字。这些配置字中的位控制基本的器件操作，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。

config 伪指令的一般形式如下：

```
#pragma config setting = value
```

其中，*setting* 是配置字位域名称（如 WDT_ENABLE），*value* 可以是所需状态的文本描述（如 CLEAR）或数值（取决于设置要求）。可以使用多个 pragma 伪指令来逐步指定完整的器件状态；但每个 pragma 伪指令可以指定多个以逗号分隔的设置值对。

以下示例显示了一些与 SAME54P20A 器件相关的 pragma 伪指令。关于设置描述符使用的配置字位域名称，可在 MPLAB X IDE 中的 **Configuration Bits**（配置位）视图进行查看。

```
#pragma config BOD33_DIS = SET
#pragma config BOD33USERLEVEL = 0x1c, BOD33_ACTION = RESET
#pragma config WDT_PER = CYC8192
```

应仅在单个翻译单元（或模块）中指定#pragma config 伪指令。应将这些伪指令放在函数定义之外，因为它们不定义可执行代码。

编译器会验证所指定的配置设置对于目标器件是否有效。如果配置字中的某个给定设置未在任何#pragma config 伪指令中指定，则与该设置关联的位默认设为未编程值。建议明确编程所有配置字以确保器件正常工作。

config pragma 伪指令可用于必须按字节寻址（而非按字（4 字节）寻址）配置单元的器件。

如果不同配置字中的多个配置设置使用相同的配置字位域名称，则可以在配置字位域名称中添加限定符。配置字位域（如 BOD33_DIS）可使用寄存器名称（如 USER_WORD_0）作为前缀，中间用下划线字符分隔，例如 USER_WORD_0_BOD33_DIS。如果需要进一步说明，可以在设置中继续添加寄存器组地址空间（如 fuses）作为前缀，中间仍用下划线字符分隔，例如 fuses_USER_WORD_0_BOD33_DIS。如果需要，可以使用更长的设置描述符形式重新编写上述示例中的 pragma 伪指令，如下所示（尽管使用该器件时并不需要此类名称）：

```
#pragma config USER_WORD_0_BOD33_DIS = SET
#pragma config fuses_USER_WORD_0_BOD33USERLEVEL = 0x1c, fuses_USER_WORD_0_BOD33_ACTION = RESET
#pragma config USER_WORD_1_WDT_PER = CYC8192
```

如果需要完全限定的名称但未使用，编译器将发出错误并提出建议。

7.4 从 C 代码中使用 SFR

特殊功能寄存器（SFR）是一些控制器件上 MCU 或外设模块各个方面的寄存器。这些寄存器进行存储器映射，这意味着，它们出现在器件存储器映射中的特定地址处。对于一些寄存器，寄存器内的位控制一些独立的功能。

可使用 C 语言接口读取或修改 SFR。可通过在源代码中包含<xc.h>头文件（见器件头文件）访问 SFR 接口定义。

C 接口中用于 SFR 和 SFR 位域/位的名称基于器件数据手册中指定的名称。每个外设组件的寄存器均通过固定基址（例如，定义为 WDT_BASE_ADDRESS）进行访问，该地址是包含组件所有存储器映射寄存器的结构体（如 WDT_CR）的地址。

可通过多种方法来使用每个 SFR 的位域或位。组件结构体中的寄存器位域既可以引用完整的寄存器值，也可以按名称引用寄存器的各个位或位域。此外，还提供了宏定义以允许使用位操作访问各个位域。 SFR 寄存器定义 提供了有关 SFR 接口的更多信息。

<xc.h>头文件将包含特定于所用器件的头文件。这些特定于器件的头文件位于编译器发行版的 pic32c/include/proc 目录下特定于架构的目录中；但是，不得将这些头文件显式地包含在代码中，否则会降低代码可移植性。

要确定所用器件的 SFR 接口，应查看特定于器件的头文件目录的 component 子目录中的头文件。这些头文件会自动包含在特定于器件的头文件中，并根据器件数据手册中的组件名称进行命名，例如 wdt.h。 请记住，不需要将任何特定于器件的头文件直接包含在源代码中——包含<xc.h>头文件将确保包含所有必要的头文件。

除了外设模块之外，通过同一接口还可以访问控制 MCU 各个方面的 SFR。特定于内核的 SFR 在所有特定于器件的头文件自动包含的头文件中定义，因此包含<xc.h>即以访问所有内核 SFR。 特定于内核的头文件位于 pic32c/include 目录中。

7.4.1 SFR 寄存器定义

本节介绍 SFR 接口的约定。下面以 WDT（看门狗定时器）组件为例。请始终查看器件数据手册和特定于器件的头文件，以确认器件的具体功能和名称。

每个组件内的 SFR 均通过基址指针进行访问，该指针在头文件中定义为宏，例如_WDT_REGS。它用作按名称指向包含组件所有 SFR 的_wdt_registers_t 类型结构体的指针，例如_WDT_REGS->WDT_CR。该结构体的位域类型反映每个寄存器的属性是只读、只写还是读写；但是，应始终查看器件数据手册以获取有关 SFR 读/写属性的详细信息。

每个 SFR 位域本身都是一种允许将 SFR 数据作为整体进行访问（如 WDT_CR.w）或单独访问各个位或位域（如 WDR_CR.KEY）的结构体。因此，SFR 中的任何保留位都不能单独访问。例如，

```
/* get entire WDT CR contents */
 uint32_t _wdt_cr = _WDT_REGS->WDT_CR.w;

 /* Set KEY field of CR (8 bits) */
 _WDT_REGS->_WDT_CR.KEY = 0x1F;

 /* Read LOCKMR bit of CR */
 uint32_t lock = _WDT_REGS->WDT_CR.LOCKMR;

 /* Set entire WDT_CR - including reserved bits */
 _WDT_REGS->WDT_CR.w = 0xDEADBEEFu;
```

请注意，本例及后续示例仅用于演示语言接口，并不演示特定 SFR 的任何使用。

对于 SFR 的每个位域，还提供了宏以便使用位操作访问该位域。例如，WDR_CR_KEY_Pos 定义为 KEY 位域的最低有效位位置，WDT_CR_KEY_Msk 定义为与 SFR 宽度相同的整数掩码（其中，该位域的所有位均置 1，所有其他位均为 0）。因此，可通过如下代码提取 KEY 位域：

```
/* Read KEY field by reading entire register and extracting
bits by mask/shift operations */
uint32_t wdt = _WDT_REGS->WDT_CR.w;
uint32_t key = (wdt & WDT_CR_KEY_Msk) >> WDT_CR_KEY_Pos;

/* Update KEY field by masking/inserting bits */
wdt = (wdt & ~WDT_CR_KEY_Msk) | (0x1F << WDT_CR_KEY_Pos);

/* Set WDRSTT bit by bit operations */
wdt = wdt | (1u << WDT_CR_WDRSTT_Pos);

/* Write back updated register contents */
_WDT_REGS->WDR_CR.w = wdt;
```

此外，类似 WDR_CR_KEY_Value 函数的宏定义为将值放入位域中的适当位位置，以便 WDR_CR_KEY_Value(0x3) 生成如下值：

```
WDT_CR_KEY_Msk & ((0x3) << WDT_CR_KEY_Pos)
```

使用掩码和位置宏的显式位操作与位域操作可以自由混用；但是，WDT_BASE_ADDRESS 指针只能使用定义的结构体进行访问，而不是强制转换为整型。请始终确保根据器件数据手册来确认外设模块的操作。

7.5 紧耦合存储器

编译器支持允许将代码和对象放入紧耦合存储器（TCM）的属性和选项。访问该存储器中的代码和数据可避免系统总线矩阵的复杂性并提供固定的延时，从而实现确定性的程序执行。

一些 Cortex-M7 器件在 SRAM 中实现了 TCM，其中包括用于保存代码的指令 TCM（或 ITCM）和用于保存数据的数据 TCM（或 DTCM）。这两个存储器可通过独立的器件寄存器进行使能和配置。TCM 区域的内部构成和大小在不同器件中差异很大，具体信息可参见所选器件的数据手册。

tcm 属性将对象或函数放入适当的 TCM，例如 uint32_t __attribute__((tcm)) var;会将对象 var 放入 DTCM。

在同时实现独立的 ITCM 和 DTCM 的器件系列中，可分别通过独立的驱动程序选项-mitcm=*size* 和-mdtcm=*size* 设置指令 TCM 和数据 TCM 这两个存储器区域的大小（字节）。指定的 *size* 必须在器件的允许范围内，并且既可以指定为十进制数，也可以指定为带 0x 前缀的十六进制数。当指定的大小无效时，可查看错误消息以了解该器件可接受的大小。使用这两个选项将确保特定于器件的运行时启动代码和链接描述文件在调用 main() 函数之前使能并初始化 TCM。使能 TCM 后，将定义预处理器宏 __XC32_ITCM_LENGTH 和/或__XC32_DTCM_LENGTH，二者分别等于相应 TCM 区域的大小。

默认情况下，会尽可能将向量表分配给 TCM。-mno-vectors-in-tcm 选项使向量表保持在 TCM 之外，非常适合在所选器件的存储器有限的情况下使用。

通过确保在编译时和链接时均发出-mstack-in-tcm 驱动程序选项，可将数据堆栈移动到 TCM。链接器会向 DTCM 分配堆栈，启动代码会在调用 main() 函数之前将堆栈从系统 SRAM 转移到 DTCM。

7.5.1 基于 Cortex-M 高速缓存控制器（CMCC）的紧耦合存储器

一些基于 Cortex-M4 的 SAM 单片机（如 SAME54 或 SAMD51）具有 Arm Cortex-M 高速缓存控制器（Cortex-M Cache Controller，CMCC）外设。借助 CMCC，可将关键代码加载到一个高速缓存路中并锁定，从而将此部分高速缓存用作紧耦合存储器（TCM）以实现确定性代码性能。

要将函数和对象放入基于 CMCC 的 TCM，应对这些需要重新定位的函数和对象使用 tcm 属性，然后使用-mtcm=*size* 选项设置 TCM 的大小（字节）并确保特定于器件的运行时启动代码和链接描述文件配置 TCM。指定的 *size* 必须在器件的允许范围内，并且既可以指定为十进制数，也可以指定为带 0x 前缀的十

六进制数。当指定的大小无效时，可查看错误消息以了解该器件可接受的大小。使能 TCM 后，将定义预处理器宏__XC32_TCM_LENGTH，它等于 TCM 区域的大小。

7.6 代码覆盖

购买分析工具套件许可证（SW006027-2）后，即可使用编译器的代码覆盖功能帮助分析项目的源代码已执行到什么程度。

使能后，该功能会在项目的程序映像中插装少量汇编序列。执行程序映像时，这些序列将记录它们在器件 RAM 的保留区中表示的代码的执行。记录存储在器件中，之后可用来分析，以确定项目源代码的哪些部分已执行。不插装编译器提供的库代码。

使能代码覆盖后，编译器将执行名为 xc-ccov 的外部工具，以确定插装项目的最高效方式。该工具会认为程序的基本块（可视为仅有一个入口点的一条或多条指令的序列）位于序列的起始位置，末端只有一个出口。并非所有这些块都需要插装，该工具确定允许对程序进行全面分析的最小块集合。

使用-mcodecov 选项可以在编译器中使能代码覆盖。使能该功能后，将定义预处理器宏__CODECOV。

使用代码覆盖时，用来编译项目的所有编译器选项都非常重要，因为它们将影响最终插装的程序映像。为确保分析能准确反映最终交付产品，编译选项应与最终发布版本将使用的选项相同。

如果使能代码覆盖，将为每个插装基本块分配 1 位 RAM，这将使项目的数据存储空间需求增加。

每个插装基本块中都插入了少量汇编指令序列，用以将相应的覆盖位置 1。

必须执行插装的项目代码才能生成代码覆盖数据，而由于汇编指令序列的增加，这一执行的速度将略有减缓。为运行中的程序提供应模拟程序代码的所有部分的输入和激励，这样便可以记录程序源代码所有部分的执行。

代码覆盖数据可以在 MPLAB X IDE 中分析。编译器生成的 ELF 文件中的信息允许插件定位并读取包含代码覆盖结果的器件存储器，并以可用格式将其显示出来。关于代码覆盖功能和其他分析工具的更多信息，请参见 Microchip 的分析工具套件许可网页。

8. 支持的数据类型和变量

MPLAB XC32 C/C++编译器支持一系列数据类型和属性。此处将介绍这些数据类型和变量。关于变量在存储器中的存储位置的信息，请参见存储器分配和访问。

8.1 标识符

C/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。

标识符区分大小写，所以 `main` 不同于 `Main`。

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

8.2 数据表示

编译器使用小尾数法格式来存储多字节值。也即，低字节存储在最低地址上。

例如，32 位值 `0x12345678` 将按以下形式存储在地址 `0x100` 处：

地址	0x100	0x101	0x102	0x103
数据	0x78	0x56	0x34	0x12

8.3 整型数据类型

编译器中的整型值使用二进制补码表示，其长度为 8 至 64 位。通过 imits.h，可在编译代码中使用这些值。

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

8.3.1 有符号和无符号字符类型

每个实现都必须定义普通 `char` 是有符号还是无符号。对于 PIC32C 和所有其他 Arm 平台，普通 `char` 默认为无符号。请注意，这一点与 PIC32M 不同。`-funsigned-char` 和`-fsigned-char` 选项始终可用于改变给定翻译单元的默认类型。

8.3.2 `limits.h`

`limits.h` 头文件定义了整型可以表示的值范围。

表 8-1. limits.h 头文件

宏名称	值	说明
`CHAR_BIT`	8	最小非位域对象的长度，以位为单位。
`SCHAR_MIN`	-128	`signed char` 类型的对象的最小可能值。
`SCHAR_MAX`	127	`signed char` 类型的对象的最大可能值。
`UCHAR_MAX`	255	`unsigned char` 类型的对象的最大可能值。
`CHAR_MIN`	-128（或 0，请参见有符号和无符号字符类型）	`char` 类型的对象的最小可能值。

.....（续）		
宏名称	**值**	**说明**
CHAR_MAX	127（或 255，请参见有符号和无符号字符类型）	char 类型的对象的最大可能值。
MB_LEN_MAX	16	任意区域设置中的多字节字符的最大长度。
SHRT_MIN	-32768	short int 类型的对象的最小可能值。
SHRT_MAX	32767	short int 类型的对象的最大可能值。
USHRT_MAX	65535	unsigned short int 类型的对象的最大可能值。
INT_MIN	-2^{31}	int 类型的对象的最小可能值。
INT_MAX	$2^{31}-1$	int 类型的对象的最大可能值。
UINT_MAX	$2^{32}-1$	unsigned int 类型的对象的最大可能值。
LONG_MIN	-2^{31}	long 类型的对象的最小可能值。
LONG_MAX	$2^{31}-1$	long 类型的对象的最大可能值。
ULONG_MAX	$2^{32}-1$	unsigned long 类型的对象的最大可能值。
LLONG_MIN	-2^{63}	long long 类型的对象的最小可能值。
LLONG_MAX	$2^{63}-1$	long long 类型的对象的最大可能值。
ULLONG_MAX	$2^{64}-1$	unsigned long long 类型的对象的最大可能值。

8.4 浮点型数据类型

编译器使用 IEEE-754 浮点格式的 32 位和 64 位形式来存储浮点值。适用于翻译单元的实现限制包含在 float.h 头文件中。

注意：一些目标 PIC32C/SAM 器件实现了浮点单元（FPU）。编译器实现了本指南中所述的某些特性来支持该硬件。

下表列出了数据类型及其对应的长度和算术类型。

类型	位数
float	32
double	64
long double	64

变量可以分别使用 float、double 和 long 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。所有浮点值都使用最低有效字节（Least Significant Byte，LSB）位于低地址的小尾数法格式表示。

下表描述了这种格式，其中：

- “符号”为指示数字是正数还是负数的符号位。
- 对于 32 位浮点值，指数为 8 位，它以加 127 的形式存储（即，指数 0 存储为 127）。
- 对于 64 位浮点值，指数为 11 位，它以加 1023 的形式存储（即，指数 0 存储为 1023）。
- “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。

对于 32 位浮点值，该数字的值为：

$(-1)^{符号} \times 2^{(指数-127)} \times 1.尾数$

对于 64 位值

$(-1)^{符号} \times 2^{(指数-1023)} \times 1.尾数$。

下表列出了 32 位和 64 位 IEEE 754 格式的示例。请注意，尾数列的最高位（即小数点左边的位）是隐含位，除非指数为零（这种情况下浮点数为零），否则假定它为 1。

表 8-2. 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	0x7DA6B69C	11111011b（251）	1.01001101011011010011100b（1.3024477959）	2.770000117e+37 —
64 位	0x47B4D6D37131A DE	10001111011b (1147）	1.0100110101101101001101110001001100011010011011011110b（1.3024477407110946）	2.77e+37 —

表中的示例可以如下手动计算。

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：

1×2.126764793256e+37×1.302447676659

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，64 位浮点格式允许范围较大的值，并且可以更精确地进行表示。

例如，如果使用 32 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（大约）为 95000.00781，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/C++代码可能无法产生预期的行为。例如：

```
volatile float myFloat;
myFloat = 95000.006;
if(myFloat == 95000.007) // value will be rounded
   LATA++;               // 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。

表 8-3 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` 代表 float 类型的最大浮点值。 `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 8-3. 浮点型值的范围

符号	含义	32 位值	64 位值
*XXX*_RADIX	指数表示形式的基数	2	2
*XXX*_ROUNDS	加法的舍入模式	1	
*XXX*_MIN_EXP	使 FLT_RADIX^{n-1} 为归一化浮点值的最小 *n*	-125	-1021
*XXX*_MIN_10_EXP	使 10^{n} 为归一化浮点值的最小 *n*	-37	-307
*XXX*_MAX_EXP	使 FLT_RADIX^{n-1} 为归一化浮点值的最大 *n*	128	1024
*XXX*_MAX_10_EXP	使 10^{n} 为归一化浮点值的最大 *n*	38	308
*XXX*_MANT_DIG	FLT_RADIX 尾数位数	24	53

.....（续）

符号	含义	32 位值	64 位值
*XXX*_EPSILON	与 1.0 相加不会得到 1.0 的最小数字	1.1920929e-07	2.2204460492503131e-16

8.5 结构体和联合体

MPLAB XC32 C/C++编译器支持 `struct` 和 `union` 类型。结构体和联合体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。完全支持位域。

结构体和联合体可作为函数参数和函数返回值自由地传递。完全支持指向结构体和联合体的指针。

8.5.1 结构体和联合体限定符

MPLAB XC32 C/C++编译器支持对于结构体使用类型限定符。对结构体应用限定符时，其所有成员都会继承该限定。在以下示例中，结构体使用 `const` 进行限定。

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

在此例中，整个结构体都会被放入程序存储器，并且每个成员都是只读的。请记住，如果结构体使用 `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。

如果结构体的成员单独使用 `const` 进行限定，但不对结构体进行这种限定，则结构体将被定位到 RAM，但每个成员都是只读的。将以下结构体与上面的结构体进行比较。

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

8.5.2 结构体中的位域

MPLAB XC32 C/C++编译器完全支持结构体中的位域。

位域总是分配到 8 位存储单元中，虽然它在定义中通常使用 `unsigned int` 类型。存储单元对齐到 32 位边界上，虽然这可以使用 `packed` 属性进行更改。

定义的第一个位将为存储它的字中的最低位。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，会在结构体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 个字节的结构体。

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`，以上结构体可以声明为：

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

带有位域的结构体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体可以进行初始化。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`。

MPLAB XC 编译器支持匿名联合体。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。这些联合体可以在放入结构体中之后使用。例如：

```
struct {
        union {
        int x;
        double y;
   };
} aaa;

int main(void)
{
    aaa.x = 99;
    // ...}
```

此处，联合体未命名，其成员可以作为结构体的一部分进行访问。匿名联合体不属于任何 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。

8.6 指针类型

MPLAB XC32 C/C++编译器支持两种基本指针类型：数据指针和函数指针。数据指针存放可以由程序间接读取，并可能可间接写入的变量的地址。函数指针存放可通过指针间接调用的可执行函数的地址。

8.6.1 组合类型限定符和指针

先回顾一下 ANSI C/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/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/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。

8.6.2 数据指针

编译器中的指针长度均为 32 位。它们可以存放能到达所有存储单元的地址。

8.6.3 函数指针

MPLAB XC 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/C++数组中的几个函数地址之一，该数组就如同一个查找表。

函数指针的长度总是为 32 位，存放要调用的函数的地址。

尝试使用包含 NULL 的函数指针调用函数将导致 ifetch 总线错误。

8.6.4 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或长度有关的任何信息。如果将整型地址赋予指针并进行解引用，则这种代码是不可移植的，并且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 PIC32C/SAM 器件。

将地址赋予一个指针时，请总是采用 C/C++对象的地址。如果在目标地址处未定义任何 C/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可以访问的存储单元的范围匹配。

例如，要生成从地址 0xA0001000 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：

```
int * cp;
```

```
cp = 0xA0001000; // what resides at 0xA0001000???
```

并随数据递增指针。更好的解决方案是：

```
int * cp;
int __attribute__((address(0xA0001000))) inputData [1000];
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。数组长度和类型会指示指针目标的长度。

进行指针比较（相减）时需要小心。例如：

```
if(cp1 == cp2)
```

```
; take appropriate action
```

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

将指针与整型常量进行比较的风险更高，例如：

```
if(cp1 == 0xA0000100)

; take appropriate action
```

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。

8.7 复数数据类型

MPLAB XC32 C/C++编译器目前未实现复数数据类型。

8.8 常量类型和格式

常量用于表示源代码中的数值，例如 123 是一个常量。与所有值一样，常量必须具有 C/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。整型常量的格式会指定其基数。MPLAB XC32 C/C++支持 ANSI 标准基数说明符，以及用于在 C 代码中指定二进制常量的说明符。

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

表 8-4. 基数格式

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

整型常量将具有 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 8-5 列出了在分配类型时考虑的后缀和类型的可能组合。例如，如果指定了后缀 l，并且值为一个十进制常量，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 8-5. 后缀和分配的类型

后缀	十进制	八进制或十六进制
u 或 U	unsigned int unsigned long int unsigned long long int	unsigned int unsigned long int unsigned long long int
l 或 L	long int long long int	long int unsigned long int long long int unsigned long long int
u 或 U，以及 l 或 L	unsigned long int unsigned long long int	unsigned long int unsigned long long int
ll 或 LL	long long int	long long int unsigned long long int
u 或 U，以及 ll 或 LL	unsigned long long int	unsigned long long int

以下给出了一个可能由于分配给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;

int main(void)
```

```
{
    shifter = 40;
    result = 1 << shifter;
    // code that uses result
}
```

常量 1 将被分配 int 类型，因而移位运算的结果将为 int 类型，无论对常量进行多少次移位，long 变量 result 的高位都永远不会被设置。在此例中，值 1 被左移 40 位将产生结果 0，而不是 0x10000000000。

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;
```

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 指定 long double 类型。

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，虽然在编译之后可能会将其优化为 char 类型。

编译器可接受多字节字符常量，但标准库不支持。

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。

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"
```

将字符串字面值赋予指向非 const char 类型的指针时，编译器将产生警告。该代码是合法的，但指针尝试写入字符串的行为将失败。例如：

```
char * cp= "one"; // "one" in ROM, produces warning
```

```
const char * ccp= "two"; // "two" in ROM, correct
```

定义并使用字符串初始化非 const 数组（即，不是指针定义）

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

是一种特殊情况，将生成一个位于数据空间中的数组，并在启动时使用字符串"two"初始化（从程序空间中复制）；而在其他上下文中使用的字符串常量则代表使用 const 限定的未命名数组，直接在程序空间中进行访问。

对于具有完全相同的字符序列的字符串，MPLAB XC32 C/C++编译器将使用相同的存储单元和标号。例如，在以下代码片段中

```
if(strncmp(scp, "hello world", 6) == 0)
   fred = 0;
if(strcmp(scp, "hello world") == 0)
   fred++;
```

两个相同的问候字符串将共用相同的存储单元。当字符串可能位于不同模块时，必须使能链接时优化，以允许这种优化。

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";
```

将为指针赋予字符串"hello world"的地址。

8.9 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 XC32 C/C++编译器支持标准 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 PIC32C/SAM 架构。

8.9.1 const 类型限定符

MPLAB XC32 C/C++编译器支持使用 ANSI 类型限定符 `const` 和 `volatile`。

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

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。

8.9.2 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：

```
extern volatile unsigned int WDTCON __attribute__((section("sfrs")));
```

`volatile` 限定符并不保证任何访问都是原子的，但编译器会尝试实现它。

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/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/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。

8.10 特定于编译器的限定符

MPLAB XC32 C/C++编译器目前没有实现任何非标准限定符。使用属性来控制变量和函数。

8.11 变量属性

编译器关键字`__attribute__`可以用于指定变量或结构体位域的特殊属性。该关键字后跟包含在双括号内的属性说明。

此外，还可通过在每个关键字的前后两端使用`__`（双下划线）来指定属性（例如，`__aligned__`代替`aligned`）。这样，在头文件中使用这些属性时便不必担心可能与宏同名。

要指定多个属性，请在双括号内通过逗号分隔它们，例如：

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

注：在整个项目中一致地使用变量属性非常重要。例如，如果某个变量在文件 A 中定义时带有某种属性，而在文件 B 中通过 `extern` 声明时未带有这种属性，则可能会产生链接错误。

address (*addr*)

为变量指定绝对虚拟地址。该属性可以与段属性配合使用。

该属性可以用在某个特定地址处开始一组变量：

```
int foo __attribute__((section("mysection"),address(0xA0001000)));

int bar __attribute__((section("mysection")));

int baz __attribute__((section("mysection")));
```

请记住，编译器不会对指定的地址执行错误检查。段将被定位到指定地址处，无论链接描述文件中列出的存储器区域范围或目标器件上的实际范围如何。该应用程序代码需要负责确保该地址对于目标器件和应用程序是有效的。

此外，为了有效利用绝对段和新的最佳适应分配器，不应在链接描述文件中映射标准程序存储器和数据存储器段。内置的链接描述文件不会映射大多数标准段，例如.text、.data、.bss 或.ramfunc 段。通过不在链接描述文件中映射这些段，可以允许使用最佳适应分配器而不是顺序分配器来分配这些段。未在链接描述文件中映射的段可以分布在绝对段周围，而链接描述文件映射的段则会被组合在一起并按顺序分配，这有可能导致与绝对段发生冲突。

注：在几乎所有情况下，都需要组合使用 address 属性与 space 属性，分别通过 space(prog) 或 space(data) 来指示代码或数据。此外，请参见 space 属性的说明。

aligned (*n*)

带属性的变量将对齐到下一个 *n* 字节边界处。

此外，aligned 属性也可以用于结构体成员。此类成员将在结构体内对齐到指定边界处。

如果省略了对齐值 *n*，则变量的对齐值设置为 8（基本数据类型的最大对齐值）。

请注意，aligned 属性用于增大变量的对齐值，而不是减小它。要减小变量的对齐值，请使用 packed 属性。

cleanup (*function*)

指示在带属性的自动函数作用域变量超出作用域时调用的函数。

所指示的函数应接受单个参数，即指向与带属性变量兼容的类型的指针，并具有 void 返回类型。

externally_visible

该属性与全局对象一起使用时可确保对象在当前编译单元之外保持可见。这可能会阻止对对象执行某些优化。

packed

带该属性的变量或结构体成员将具有最小的可能对齐值。也即，不会为声明分配任何对齐填充存储单元。packed 与 aligned 属性配合使用，可以用于为变量或结构体成员的类型设置大于或小于默认对齐值的任意对齐限制。

persistent

该属性可阻止启动代码的数据初始化代码对变量进行初始化。当需要在软复位过程中保留变量值时，该行为会很有用。*注意事项：*请注意，在具有 L1 高速缓存的单片机上，高速缓存重新初始化可能会导致变量的运行时值发生变化。对于具有 L1 高速缓存和存储器保护单元（Memory Protection Unit，MPU）的器件，可选择使用存储器保护单元来创建未高速缓存的 RAM 区域并将持久性变量放入该区域。

section ("*section-name*")

将变量放入指定的段。

例如，

```
unsigned int dan __attribute__ ((section (".quixote")))
```

变量 dan 将放入段.quixote。

除非同时指定了 unique_section，否则-fdata-sections 命令行选项对使用 section 属性定义的变量不起作用。

space(*memory-space*)

space 属性可以用于指示编译器在特定存储器空间中分配变量。对于程序存储器，有效的存储器空间为 prog，对于数据存储器为 data。data 空间是非 const 变量的默认空间。

prog 和 data 空间通常分别对应于 rom 和 ram 存储器区域，如默认特定于器件的链接描述文件中所述。

该属性还控制如何处理已初始化的数据。对于默认的 space(data)，链接器会在数据初始化模板中生成一个条目，但它不会对于 space(prog)生成一个条目，因为变量位于非易失性存储器中。通常，这意味着 space(data)适用于在运行时启动时初始化的变量；而 space(prog)适用于将由在线编程器或自举程序编程的变量。例如：

```
const unsigned int __attribute__((space(prog))) jack = 10;
signed int __attribute__((space(data))) oz = 5;
```

unique_section

将变量放入惟一命名的段中，就如同指定了-fdata-sections。如果该变量还具有 section 属性，将使用该段的名称作为前缀来生成惟一的段名称。

例如，

```
int tin __attribute__ ((section (".ofcatfood"), unique_section)
```

变量 tin 将放入段.ofcatfood。

unused

向编译器指示该变量可能不会被使用。如果该变量未被使用，编译器不会发出警告。

used

指示编译器，该对象总是会被使用，即使编译器无法检测到对该对象的引用，也必须为该对象分配存储空间。例如，只有行内汇编代码引用了某个对象时。

weak

weak 属性会使声明作为一个弱符号发出。弱符号指示如果有同一符号的全局版本可用，则改为使用该版本。

当向对外部符号的引用应用 weak 时，链接时可以不需要该符号。例如：

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

在以上程序中，如果其他模块未定义 s，程序将仍可以进行链接，但不会为 s 赋予地址。条件语句会验证 s 是否已定义（如果已定义，则返回其值）。否则返回 0。该功能有很多用途，主要是为了提供能与可选库链接的通用代码。

9. 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。以下几节分别介绍了这两组变量的存储器分配。

9.1 地址空间

不同于 8 位和 16 位 PIC 器件，PIC32 具有统一的编程模型。PIC32 器件为所有代码、数据、外设和配置位提供了单个 32 位宽的地址空间。

该单一地址空间中的存储器区域被指定为不同的用途；例如，作为用于指令代码的存储器或用于数据的存储器。在内部，这些器件使用独立的总线[(1)] 来访问这些区域中的指令和数据，从而支持并行访问。在 8 位和 16 位 PIC 器件上使用的程序存储器和数据存储器这两个术语仍然适用于 PIC32 器件，但存储空间较小的器件会在不同地址空间中实现它们。

器件内的 CPU 使用的所有地址都是虚拟地址。这些地址由系统控制处理器映射到物理地址。

9.2 数据存储器中的变量

大多数变量最终会被定位到数据存储器。例外情况是使用 `const` 限定的非 `auto` 变量（它们会被放入程序存储空间），请参见 const 类型限定符。

由于 `auto` 变量和非 `auto` 变量的存储器分配方式在根本上不同，以下将分别对它们进行讨论。使用 C/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的变量。但是，C/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 `static` 变量的作用域仅为该文件内部，所以它不可全局访问，但它可以由文件内的多个函数访问，所以它也不是任何一个函数的局部变量。从存储器分配的角度来说，变量基于是否为 `auto` 类型来分配空间，从而有下面各小节中的分组。

9.2.1 非自动变量分配

非 `auto` 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器。这是一个两阶段的过程：将每个变量放入相应的段中，稍后将该段链接到数据存储器中。

编译器会考虑 3 种类别的非 `auto` 变量，这 3 种类别都与程序开始时变量应包含的值有关。对于所描述的类别，将使用以下段。

- `.bss` 该段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` 该段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 映像。

请注意，用于存放已初始化变量的数据段是存放 RAM 变量本身的段。存在一个与之对应的段（名为`.dinit`），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。

9.2.2 静态变量

所有 `static` 变量都具有永久存储持续时间，即使是在函数内定义的那些“局部静态”变量。局部 `static` 变量的作用域仅为定义它们的函数或块内部，但与 `auto` 变量不同，在程序的整个持续时间中都会为它们保留存储器。因而，它们将像其他非 `auto` 变量一样分配存储器。静态变量可由其他函数通过指针访问，因为它们具有永久持续时间。

程序将确保 `static` 变量在函数调用之间保留其值，除非通过一个指针显式地进行修改。

---

1 根据该器件的内部总线安排，该器件可视为哈佛架构。

属于 static 类型且进行初始化的变量在程序执行过程中仅进行一次初始值赋值。因此，它们可能优于已初始化 auto 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 auto 已初始化对象相同，请参见外设库函数。静态变量与非 static 变量会被定位到相同的段中。

9.2.3 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）。结构体和联合体聚合类型也是，请参见结构体和联合体。不存在关于这些对象可以多长的理论限制。

9.2.4 更改默认的非自动变量分配

有几种方法可以将非 auto 变量定位到默认位置之外的位置。

变量可以通过使用限定符放入其他器件存储空间。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 const 类型限定符）。

如果要阻止所有变量使用一个或多个数据存储单元，以便这些存储单元可以用于其他目的，最好使用 address 属性定义一个变量（或数组），从而使它占用该存储空间，请参见变量属性。

如果只有几个非 auto 变量需要定位到数据存储空间中的特定地址处，则可以使用 address 属性定位变量。该属性在变量属性中描述。

9.2.5 数据存储器分配宏

sys/attribs.h 头文件提供了许多用于常用属性的宏，以提高代码的可读性。

__ramfunc__	将带属性的函数定位到 RAM 函数代码段中。
__longramfunc__	将带属性函数定位到 RAM 函数代码段中，并应用 long_call 属性。

9.3 自动变量分配和访问

本节介绍 auto 变量（那些具有自动存储持续时间的变量）分配。这也包括函数参数变量（其行为类似于 auto（*automatic* 的简称）变量）以及由编译器定义的临时变量。

auto 变量是局部变量的默认类型。除非显式地声明为 static 类型，否则局部变量将设为 auto 类型。如果需要，可以使用 auto 关键字。

顾名思义，auto 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。

在 PIC32C 器件上，寄存器 r4-r8、r10 和 r11 用于保存函数的自动变量的值。函数必须保存寄存器 r4-r8、r10、r11 和 r13/堆栈指针（Stack Pointer，SP）的内容。

通常，PIC32C 的软件堆栈用于存储 auto 变量。寄存器的值被压入堆栈。 函数是可重入的，函数的每个实例都在堆栈中具有自己的存储器区域，用于存放其自动变量和参数变量。

在 PIC32C 器件上，堆栈指针（SP）为寄存器 r13。内核使用满递减堆栈。满递减堆栈是指堆栈指针保存存储器中最后一个压入堆栈的对象的地址，并且堆栈向低地址增长。当内核将一个新对象压入堆栈时，它会递减堆栈指针，然后将对象写入新的存储单元。

标准限定符 const 和 volatile 可同时用于 auto 变量，它们不会影响这些变量在存储器中的定位方式。这意味着，使用 const 限定的局部对象仍是 auto 对象，因此将在数据存储器中的堆栈中分配存储器，而不是像非 auto const 对象一样在程序存储器中分配。

9.3.1 局部变量长度限制

自动变量不存在理论的最大长度。

9.4 程序存储器中的变量

只有不属于 auto 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 auto 变量都会与其他 auto 变量一起放入堆栈。

任何使用 const 限定的（auto 或非 auto）变量都永远是只读的，尝试在源代码中写入这些变量将导致编译器发出错误。

const 对象通常会定义初始值，因为程序无法在运行时写入这些对象。但是，这并不是一项要求。 未初始化的 const 对象与其他未初始化 RAM 变量一起在 bss 段中分配空间，但编译器仍然会将其视为只读。

```
const char IOtype = 'A'; // initialized const object

const char buffer[10]; // I just reserve memory in RAM
```

9.4.1 const 变量的长度限制

const 变量不存在理论的最大长度。

9.4.2 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，则可以选择在定制链接描述文件中调整存储器区域。

如果只有几个非 auto const 变量需要定位到程序空间存储器中的特定地址处，则这些变量应使用 address 属性来将它们定位到所需位置。该属性在变量属性中描述。

如果无法将使用 const 限定的对象放入包含可执行代码的段，则可使用-mpure-code 选项，如特定于 PIC32C/SAM 器件的选项所述。

9.5 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 XC32 C/C++编译器，变量可能会被作为代码优化的一部分而分配到寄存器中。对于优化级别 1 和更高级别，赋予变量的值可能会存入寄存器。在此期间，与变量关联的存储单元存放的可能不是实时值。

register 关键字可以用于指示您偏向将变量分配到寄存器中，但这只是建议，也可以不这样做。此外，还可以指示特定的寄存器，但建议不要这么做，因为您的寄存器选择可能会与编译器的需求相冲突。在代码中使用特定寄存器可能会导致编译器生成效率较低的代码。

如寄存器约定所示，可以通过寄存器将参数传递给函数。

与此相关的源代码位于以下位置的 pic32c-libs.zip 文件中： <install-directory>/pic32-libs/

解压该文件后，可在以下位置找到源代码： pic32m-libs/libpic32/stubs/pic32_init_tlb_ebi.S

例 9-1. 寄存器中的变量

```
volatile unsigned int special;
unsigned int example (void)
{
  register unsigned int my_reg __asm__("V1");
  my_reg += special;
  return my_reg;
}
```

9.6 动态存储器分配

运行时堆是数据存储器中的未初始化区域，用于使用标准 C 库动态存储器管理函数 calloc、malloc 和 realloc 以及 C++的 new 操作符进行动态存储器分配。大多数 C++应用程序都会需要堆。

如果不使用以上任何函数，则不需要分配堆。默认情况下，不会创建堆。

在 MPLAB X 中，可以在项目属性中为 xc32-ld 链接器指定堆大小。在编译项目时，MPLAB X 会自动将该选项传递给链接器。

如果希望直接（通过调用存储器分配函数之一）或间接（通过使用将使用这些函数之一的标准 C 库函数）使用动态存储器分配，则必须创建堆。堆通过在链接器命令行上使用`--defsym=_min_heap_size` 链接器命令行选项指定其大小来创建。使用命令行分配 512 字节堆的示例如下：

```
xc32-gcc foo.c -Wl,--defsym=_min_heap_size=512
```

使用 xc32-g++驱动程序分配 `0xF000` 字节的堆的示例如下：

```
xc32-g++ vector.cpp -Wl,--defsym=_min_heap_size=0xF000
```

链接器会紧邻在堆栈之前分配堆。

10. 芯片级安全性和 Arm® TrustZone®技术

SAM L11 MCU 集成了芯片级安全性和 Arm TrustZone 技术，有助于防御物理攻击和远程攻击。

XC32 v2.20 及更高版本支持使用-mcmse 选项进行 Arm v8-M CMSE 安全扩展。链接 TrustZone 应用程序（安全和非安全）时需要向链接器传递额外的参数以描述存储器区域大小。安全应用程序还必须传递链接器选项以创建安全网关模板。利用 TrustZone 编写的应用程序实际上是两个应用程序，分别是编译后的应用程序和链接后的应用程序。

10.1 Armv8-M 安全扩展

在支持 TrustZone 的器件上，存储器被划分为两个区域：安全区域和非安全区域。在非安全状态下对安全存储器的访问通过函数调用实现。此类调用源自非安全存储器并切换到安全存储器中执行。切换通过存储器另一特殊区域（非安全可调用（Non-Secure Callable，NSC））中的模板来实现。非安全可调用对模板的构造方式以及被链接器填充的方式有各种限制。它始终位于安全存储器区域的末尾。

-mcmse 选项使能 Armv8-M 安全扩展，如“*Armv8-M Security Extensions: Requirements on Development Tools - Engineering Specification*”（developer.arm.com/documentation/ecm0359818/latest）所述。

作为安全扩展的一部分，XC32 实现了两个新的函数属性：cmse_nonsecure_entry 和 cmse_nonsecure_call。此外，XC32 还实现了以下固有函数。此处使用 FPTR 来表示任何函数指针类型。

cmse_address_info_t cmse_TT (void *)	生成 TT 指令
cmse_address_info_t cmse_TT_fptr (FPTR)	生成 TT 指令。参数 FPTR 可以是任何函数指针类型。
cmse_address_info_t cmse_TTT (void *)	生成带有 T 标志的 TT 指令。
cmse_address_info_t cmse_TTT_fptr (FPTR)	生成带有 T 标志的 TT 指令。参数 FPTR 可以是任何函数指针类型。
cmse_address_info_t cmse_TTA (void *)	生成带有 A 标志的 TT 指令。
cmse_address_info_t cmse_TTA_fptr (FPTR)	生成带有 A 标志的 TT 指令。参数 FPTR 可以是任何函数指针类型。
cmse_address_info_t cmse_TTAT (void *)	生成带有 T 和 A 标志的 TT 指令。
cmse_address_info_t cmse_TTAT_fptr (FPTR)	生成带有 T 和 A 标志的 TT 指令。参数 FPTR 可以是任何函数指针类型。
void * cmse_check_address_range (void *, size_t, int)	对 C 对象执行权限检查。
typeof(p) cmse_nsfptr_create (FPTR)	返回 LSb 清零的 p 的值。参数 FPTR 可以是任何函数指针类型。
intptr_t cmse_is_nsfptr (FPTR)	如果 p 的 LSb 未置 1，则返回非零值，否则返回零。FPTR 参数可以是任何函数指针类型。
int cmse_nonsecure_caller (void)	如果从非安全状态调用入口函数，则返回非零值，否则返回零。

10.2 用于控制 TrustZone 的存储器区域的链接器宏

链接安全应用程序或非安全应用程序时需要链接器知道安全存储器区域、非安全存储器区域和非安全可调用存储器区域的地址和长度。下面给出了在链接 TrustZone 应用程序时用于设置和控制 TrustZone 的存储器区域的预处理器定义。它们以-Wl,-DNAME[=value]形式进行传递。

编译非安全应用程序时，以下预处理器定义会影响链接描述文件。

- BOOTPROT=*size*（可选）：定义引导保护大小（字节）。如果未提供，则默认值为 0。

- AS=*size*（建议）：定义闪存安全应用程序大小（字节）。如果未提供，则默认为 ROM 的 50%。
- RS=*size*（建议）：定义安全 RAM 的大小（字节）。如果未提供，则默认为 RAM 的 50%。

编译安全应用程序时，以下预处理器定义会影响链接描述文件。

- SECURE（必选）：使用针对安全应用程序的存储器布局。
- BOOTPROT=*size*（可选）：定义引导保护大小（字节）。如果未提供，则默认值为 0。
- AS=*size*（建议）：定义闪存应用程序安全存储器区域大小（字节）。如果未提供，则默认为 ROM 的 50%。
- ANSC=*size*（建议）：定义闪存应用程序非安全可调用存储器区域大小（字节）。如果未提供，则默认值为 0。
- RS=*size*（建议）：定义安全 RAM 的大小（字节）。如果未提供，则默认为 RAM 的 50%。

在所有情况下，如果未提供建议值，链接器预处理器将发出警告消息。

重要：存储器区域大小必须与 NVM 用户行（UROW）和 NVM 引导配置行（BOCOR）中的器件配置位编程的大小一致。这些可通过源代码中的#pragma config 设置为配置位。在这种情况下，配置位中的值必须与指示存储器区域的链接描述文件中的值一致，以确保正常工作。关于各位域的名称和值，请参见常规 config pragma 伪指令文档。

11. 浮点支持

一些 PIC32C/SAM 器件实现了符合 1985 IEEE-754 标准的浮点单元（FPU），支持单精度和双精度数据类型。

11.1 浮点调用约定

对于基于 Cortex-M 的器件（如 MEC17、CEC17 和 SAM 系列），XC32 默认使用硬件浮点单元（FPU）（如果可用）。对于需要遵循特定 FPU 调用约定的情况，可以指定以下选项。无论是编译时还是链接时，都必须发出这些选项。

-mfloat-abi=soft	指定“soft”可使 XC32 生成包含浮点运算库调用的输出。对于没有硬件 FPU 的器件，该设置为默认设置。
-mfloat-abi=softfp	指定“softfp”可使用硬件浮点指令生成代码，但仍遵循软件浮点调用约定。
-mfloat-abi=hard	指定“hard”可生成浮点指令并遵循特定于 FPU 的调用约定。对于具有硬件 FPU 的器件，该设置为默认设置。

12. 定点算术支持

MPLAB XC32 C/C++编译器支持定点算术。这符合 ISO/IEC TR 18037:2008 标准//ISO/IEC TR 18037 的 N1169 草案以及关于嵌入式 C 语言的 ISO C99 技术报告。该报告可通过以下链接获取：http://www.open-std.org/JTC1/SC22/WG14/www/projects#18037 或 standards.iso.org/ittf/PubliclyAvailableStandards/c051126_ISO_IEC_TR_18037_2008.zip

本章介绍了编译器根据该草案标准支持的类型和操作的特定于实现的详细信息。

由于 DSP 应用程序对性能敏感，因此应用程序开发人员历来倾向于用汇编语言编写函数。但是，通过使用 XC32 编译器，将很少甚至可能无需编写汇编代码。本章介绍了有助于对 DSP 应用程序实现最佳优化的编码风格和使用技巧。

12.1 使能定点算术支持

MPLAB XC32 C/C++编译器默认使能定点运算支持，允许使用内置定点类型、字面值和操作符。可包含 `<stdfix.h>`头文件以方便定义，如数据类型所述。

12.2 数据类型

支持 ISO/IEC TR 18037:2008 标准第 4.1 节“Overview and principles of the fixed-point data types”中所述的全部 12 种主要定点类型及其别名。定点数据值包含小数部分和可选的整数部分。下表说明了 XC32 中的定点数据格式。

在下表所示的格式中，s 是有符号类型的符号位（无符号类型没有符号位）。句点字符（.）是分隔整数部分与小数部分的说明符。数字表示整数部分或小数部分的位数。

表 12-1. 定点格式

类型	格式	说明
`short _Fract`	s0.7	1 个符号位，无整数位，7 个小数位
`unsigned short _Fract`	0.8	仅 8 个小数位
`_Fract`	s0.15	1 个符号位，15 个小数位
`unsigned _Fract`	0.16	仅 16 个小数位
`long _Fract`	s0.31	1 个符号位，无整数位，31 个小数位
`unsigned long _Fract`	0.32	仅 32 个小数位
`long long _Fract`	s0.63	1 个符号位，无整数位，63 个小数位
`unsigned long long _Fract`	0.64	仅 64 个小数位
`short _Accum`	s8.7	1 个符号位，8 个整数位，7 个小数位
`unsigned short _Accum`	8.8	8 个整数位，8 个小数位
`_Accum`	s16.15	1 个符号位，16 个整数位，15 个小数位
`unsigned _Accum`	16.16	16 个整数位，16 个小数位
`long _Accum`	s32.31	1 个符号位，32 个整数位，31 个小数位
`unsigned long _Accum`	32.32	32 个整数位，32 个小数位
`long long _Accum`	s32.31	1 个符号位，32 个整数位，31 个小数位
`unsigned long long _Accum`	32.32	32 个整数位，32 个小数位

`_Sat` 类型修饰符可与上表中的任何类型一起使用，以指示值已经饱和，如 ISO/IEC TR 18037:2008 标准中所述。例如，`_Sat short _Fract` 是 `short _Fract` 的饱和形式。有符号类型在相应格式可表示的最小负数和最大正数处饱和。无符号类型在零和格式可表示的最大正数处饱和。

MPLAB XC32 C 编译器提供了包含文件 `stdfix.h`，其中提供与定点支持相关的各种预处理器宏。下表列出了这些包含文件。

<stdfix.h>别名	类型
fract	_Fract
accum	_Accum
sat	_Sat

用例：

```
#include <stdfix.h>
void main () {
  int i;
  fract a[5] = {0.5,0.4,0.2,0.0,-0.1};
  fract b[5] = {0.1,0.8,0.6,0.5,-0.1};
  accum dp = 0;
  /* Calculate dot product of a[] and b[] */
  for (i=0; i<5; i++) {
      dp += a[i] * b[i];
  }
}
```

有符号或无符号类型溢出的默认行为是饱和。不支持 ISO/IEC TR 18037:2008 标准第 4.1.3 节“Rounding and Overflow”中所述的用于控制舍入和溢出行为的 pragma 伪指令。

下表列出了支持用于形成每种类型的定点字面值的定点字面值后缀。

表 12-2. 定点字面值后缀

类型	后缀
short _Fract	hr, HR
unsigned short _Fract	uhr, UHR
_Fract	r, R
unsigned _Fract	ur, UR
long _Fract	lr, LR
unsigned long _Fract	ulr, ULR
long long _Fract	llr, LLR
unsigned long long _Fract	ullr, ULLR
short _Accum	hk, HK
unsigned short _Accum	uhk, UHK
_Accum	k, K
unsigned _Accum	uk, UK
long _Accum	lk, LK
unsigned long _Accum	ulk, ULK
long long _Accum	llk, LLK
unsigned long long _Accum	ullk, ULLK

12.3 定点库函数

当前 MPLAB XC32 标准 C 库中不提供 ISO/IEC TR 18037:2008 标准第 4.1.7 节中所述的定点函数（舍入和转换函数等）。

12.4 定点变量操作

对定点类型的支持包括：

- 前缀和后缀递增和递减操作符（++和--）
- 一元算术操作符（+、-和!）
- 二元算术操作符（+、-、*和/）
- 二元移位操作符（<<和>>）

- 关系操作符（<、<=、>=和>）
- 赋值操作符（+=、-=、*=、/=、<<=和>>=）
- 与整型类型、浮点类型或定点类型之间的转换

12.5 不支持的特性

当前 MPLAB XC32 标准 C 库不支持 ISO/IEC TR 18037:2008 标准第 4.1.9 节“Formatted I/O functions for fixed-point arguments”中所述的用于格式化 I/O 的定点转换说明符。在格式化的 I/O 程序中使用定点参数时，必须适当进行与浮点表示之间的转换。例如：

```
#include <stdio.h>
#include <stdfix.h>

int main(void)
{
  fract a = 0.5;
  accum b;
  double d;

  scanf ("%lf", &d);           /* read into floating-point type */
  b = (accum) d;               /* convert to fixed-point type */
  printf ("%1.4f", (float) a); /* cast to floating-point type for output */
  return 0;
}
```

当前 MPLAB XC32 标准 C 库中不提供 ISO/IEC TR 18037:2008 标准第 4.1.7 节中所述的定点函数。

13. 操作符和语句

MPLAB XC32 C/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。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。

13.1 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。MPLAB XC32 C/C++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 signed 或 unsigned 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。

假设以下示例：

```
unsigned char count=0, 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 unsigned 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=0, 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 0xFFFFFFAA，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 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，MPLAB XC32 C/C++编译器不会执行整型提升，以提高代码效率。假设以下示例：

```
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 32 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。

13.2 类型引用

引用表达式的类型的另一种方式是使用 typeof 关键字。它是语言的一个非标准扩展。使用该功能会降低代码的可移植性。

使用该关键字的语法类似于 sizeof，但构造在语义上类似于使用 typedef 定义的类型名称。

编写 typeof 的参数有两种方式： 使用一个表达式或一个类型。下面是一个使用表达式的示例：

```
typeof (x[0](1))
```

它假设 x 是函数的数组；所描述的类型是函数的值的类型。

以下是一个使用 typename 作为参数的示例：

```
typeof (int *)
```

此处描述的类型是一个指向 int 类型的指针。

如果要编写的头文件必须在包含到 ANSI C 程序中时可工作，则编写__typeof__而不是 typeof。

可以在可使用 typedef 名称的所有位置使用 typeof 构造。例如，您可以在声明中、在强制类型转换中，或在 sizeof 或 typeof 内使用它。

- 以下语句使用 x 所指向的类型来声明 y： typeof (*x) y;
- 以下语句将 y 声明为此类值的数组： typeof (*x) y[4];
- 以下语句将 y 声明为指向字符的指针的数组： typeof (typeof (char *)[4]) y;
  它等效于以下传统的 C 声明：
  char *y[4];

为了说明使用 typeof 的声明的意义，以及它为什么是一种有用的编写方式，使用以下宏重新编写它：

```
#define pointer(T) typeof(T *)

#define array(T, N) typeof(T [N])
```

现在，可以使用以下方式重写声明：

```
array (pointer (char), 4) y;
```

因而，array (pointer (char), 4)的类型为由 4 个指向 char 类型的指针组成的数组。

13.3 标号作为值

您可以使用一元操作符“&&”，获取在当前函数（或包含函数）中定义的标号的地址。它是语言的一个非标准扩展。使用该功能会降低代码的可移植性。

返回的值具有类型 void *。该值是一个常量，可以在该类型常量有效的任意位置使用。例如：

```
void *ptr;
...
ptr = &&foo;
```

要使用这些值，需要能够跳转到其中一个值。这通过计算 goto 语句 goto *exp;来实现。例如：

```
goto *ptr;
```

允许 void *类型的任意表达式。

使用这些常量的一种方法是初始化一个用作跳转表的静态数组：

```
static void *array[] = { &&foo, &&bar, &&hack };
```

然后，可以使用索引选择一个标号，如下：

```
goto *array[i];
```

注：它不会检查索引是否处于界限内（C 中的数组索引从不这样做）。

这种标号值的数组的用途非常类似于 `switch` 语句。`switch` 语句更为简洁，因而优于数组。

标号值的另一个用途是在线程代码的解释器中。解释器函数内的标号可以存储在线程代码中，用于实现快速分派。

这种机制可能会被误用，将代码跳转到一个不同的函数中。编译器无法防止这种情况发生，所以必须小心确保目标地址对于当前函数是有效的。

13.4 条件操作符的操作数

条件表达式中的中间操作数可以省略。那么，如果第一个操作数为非零值，其值为条件表达式的值。它是语言的一个非标准扩展。使用该功能会降低代码的可移植性。

因此，表达式：

```
x ? : y
```

如果 x 为非零值，则具有 `x` 的值；否则为 `y` 的值。

该示例等效于：

```
x ? x : y
```

在该简单示例中，省略中间操作数的功能并不是特别有用。当第一个操作数（如果它是一个宏参数）包含或可能包含副作用时，它会变得很有用。此时，重复中间的操作数会将副作用执行两次。省略中间操作数时，将使用已计算的值，没有重新计算它的副作用。

13.5 case 范围

您可以在单个 case 标号中指定一个连续值的范围，如下：

```
case low ... high:
```

这与使用相应数量的独立 case 标号（每个标号对应于从 *low* 到 *high* 的一个整数值）具有相同的作用。它是语言的一个非标准扩展。使用该功能会降低代码的可移植性。

该功能对于表示 ASCII 字符编码的范围特别有用：

```
case 'A' ... 'Z':
```

请注意：请在...两边写空格，否则在对整数值使用它时，可能无法正确地解析它。例如，编写以下代码：

```
case 1 ... 5:
```

而不是以下代码：

```
case 1...5:
```

14. 寄存器使用

本章介绍编译器用于基于 C/C++源代码生成汇编代码的寄存器。

14.1 寄存器使用

当基于 C/C++源代码生成汇编代码时，编译器假定外部函数（根据调用约定）和行内汇编语句不会修改寄存器内容。可使用扩展行内汇编语法指示行内汇编语句使用和/或修改的硬件寄存器，以便编译器可以在有这些语句的情况下生成正确的代码。

14.2 寄存器约定

下表列出了 PIC32C 器件的 Arm Cortex-Mx 内核中的 16 个通用寄存器。某些寄存器被指定为编译器专用，或者具有同义词以表明它们在过程调用标准中的使用。在适用的情况下，会标明在汇编代码中使用以及用作专用寄存器时的特殊名称。

表 14-1. 寄存器约定

寄存器编号	特殊名称	使用
R0-R3		通用寄存器/函数参数和返回值寄存器。
R4-R8		通用寄存器/变量寄存器。
R9		通用寄存器/平台寄存器。
R10		通用寄存器/变量寄存器。
R11	FP	帧指针。
R12	IP	内部过程调用中间结果暂存寄存器。
R13	SP	堆栈指针。
R14	LR	链接寄存器。
R15	PC	程序计数器。

注：如果特殊寄存器 R7 和 R11 无需用作专用寄存器，则可作为通用寄存器使用。另请注意，所有寄存器名称（包括同义词和特殊名称）在汇编语言中都不区分大小写。

15. 堆栈

本章介绍 PIC32C/SAM 器件使用的软件堆栈。

15.1 软件堆栈

PIC32C/SAM 器件使用本用户指南中所称的“软件堆栈”作为堆栈。它是大多数计算机采用的典型堆栈安排，它是通过入栈和出栈类型的指令以及一个堆栈指针寄存器进行访问的普通数据存储器。“硬件堆栈”一词用于描述 Microchip 8 位器件采用的堆栈，它仅用于存储函数返回地址。

PIC32C/SAM 器件使用专用的堆栈指针寄存器 sp（寄存器 r13）作为软件堆栈指针。所有处理器堆栈操作（包括函数调用、中断和异常）都使用软件堆栈。它指向堆栈中的下一个可用单元。堆栈在运行时朝着存储器低地址向下增长。

默认情况下，堆栈的大小为 1024 字节。可通过使用--defsym 链接器选项将_min_stack_size 符号定义为所需大小（字节）来指定堆栈的大小。使用命令行分配 2048 字节堆栈的示例如下：

```
xc32-gcc -mprocessor=ATSAME70N20B foo.c -Wl,--defsym=__min_stack_size=2048
```

使用两个工作寄存器来管理堆栈：

- 寄存器 r13（sp）——它是堆栈指针。它指向堆栈中的下一个可用单元。
- 寄存器 r11（fp）——它是帧指针。它指向当前函数的帧。

未提供堆栈溢出检测功能。

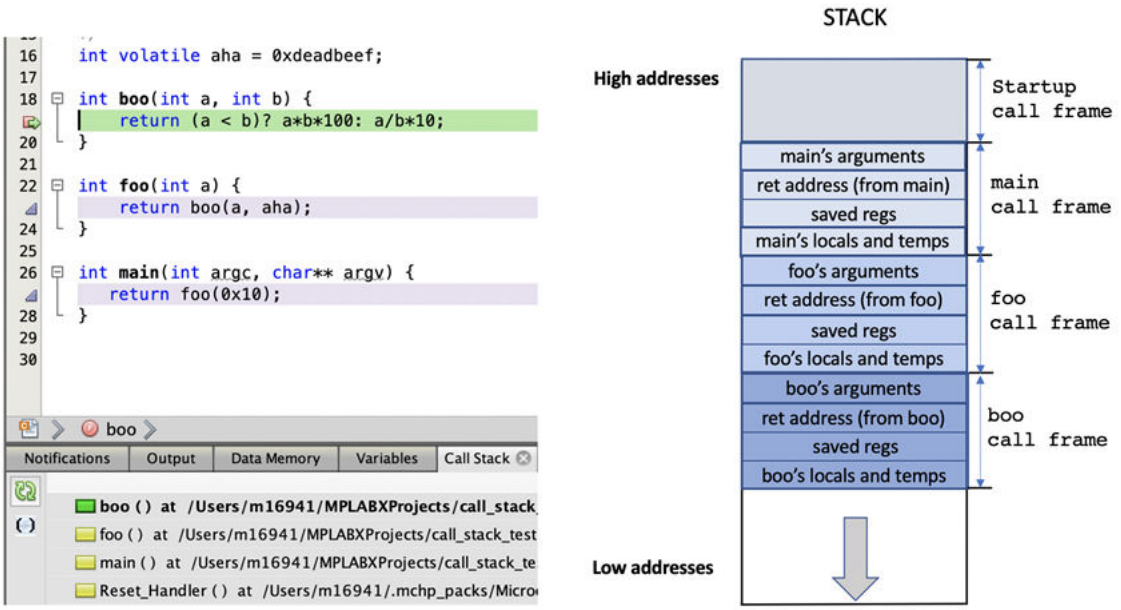
C/C++运行时启动模块会在启动和初始化序列期间初始化堆栈指针，请参见初始化堆栈指针和堆。

15.2 调用帧

堆栈是一个在运行时动态增长的存储器区域。它从高地址开始，向低地址增长（堆栈向下增长）。

堆栈具有多个调用帧，每个调用帧包含与活动函数调用相对应的数据。

图 15-1. 调用帧图



调用帧包含：

- 返回地址
- 被调用方保存的寄存器
- 通过堆栈传递的参数
- 局部变量
- 临时变量（由编译器插入）
- 中断现场

15.3 针对搭载 Arm Cortex 内核的目标器件的堆栈溢出保护器（Stack Smashing Protector，SSP）

背景知识

“Smashing the stack”是 Elias Levy（笔名 Aleph One）在 1996 年于 *Phrack Magazine* 发表的一篇标题为“Smashing the Stack for Fun and Profit”文章中提出的一个术语。这篇文章仍然可以从网上的各种来源找到，包括 duckduckgo.com/?q=Smashing+the+Stack+for+Fun+and+Profit+by+aleph+one&ia=web。

在用 C 语言编写的许多应用程序中，攻击者可使用自动存储类使编写的应用程序超出本地缓冲区阵列的末尾，从而破坏执行堆栈。攻击会改写或“破坏”堆栈中的关键数据，进而利用安全漏洞实施恶意操作。关于堆栈溢出攻击的细节不在本文档的讨论范围之内。

堆栈溢出保护器（SSP）编译器功能通过在运行时检测位于堆栈中的机密信息值（称为堆栈金丝雀）的变化，以帮助检测堆栈缓冲区溢出情况。该功能可帮助应用程序开发人员识别此类应用程序漏洞和缺陷，并有助于缓解堆栈破坏安全攻击。为了帮助检测这类问题，编译器会插装函数的序言（prologue）代码，将机密信息值（堆栈金丝雀）先于其他自动变量存储到堆栈中。之后，在从函数返回之前，函数的结语（epilogue）代码会检查存储的堆栈金丝雀值。如果函数的结语代码确定该值已被修改，则发生堆栈缓冲区溢出并调用失败回调（`__stack_chk_fail()`），相关详细信息如下文所述。

重要：堆栈溢出保护器（SSP）功能仅有助于检测堆栈缓冲区溢出情况，但并不能阻止溢出情况发生。此外，通过在准备输入时使堆栈金丝雀被正确的机密信息值改写是可以破解检测机制的，因此无法提供完美保护。因此，第一道防线必须是修复应用程序代码以消除缓冲区溢出的可能性。**不要**依靠该功能来捕捉所有缓冲区溢出情况。

由于该功能在函数序言和结语处都添加了代码，因此会导致每个受影响函数的代码长度增大、执行时间延长。下面列出的编译器选项可用于指定要插装的函数类型。

15.3.1 堆栈保护器函数属性

`__attribute__((stack_protect))`

如果设置了`-fstack-protector`、`-fstack-protector-strong` 或`-fstack-protector-explicit` 标志，则该属性会为函数添加堆栈保护代码。

优化可能会影响堆栈保护：

- 函数内联可能会影响函数是否受保护。
- 删除未使用的变量可阻止函数受保护。

例 15-1. 使用（test_stack_protect.c）

```
#include <stdint.h>
int32_t __attribute__((stack_protect)) func1()
{
    char ana[]="Test canary use";
    int32_t i;
```

```
    if (ana[1] == ana[14])
      return 1;

    return 0;
}

int main()
{
    return func1();
}
```

通过以下命令行进行编译：

```
xc32-gcc -fstack-protector -O2 test_stack_protect.c -o test_p.elf
-mprocessor=ATSAME70J19
```

在本例中，`func1()`函数将会插装堆栈保护代码。堆栈保护器功能在`func1()`的调用堆栈中添加一个堆栈金丝雀变量，并在函数的末尾检查变量值。如果值被修改，即表示堆栈已被破坏，则生成的代码将调用`__stack_chk_fail()`函数。

15.3.2 堆栈回调函数

当检查代码检测到堆栈中的保护变量已被修改时，会通过调用以下函数通知运行时环境： `void __stack_chk_fail(void);`

必须根据具体的应用程序来适当地提供该函数的实现。 正常情况下，此类函数会终止程序（可能在报告故障之后）。

XC32 默认提供`__stack_chk_fail()`的弱实现，几乎不占用空间。 `__stack_chk_fail()` 在检测到溢出时被调用，因为金丝雀已被改写。该函数不应该返回任何值。如果溢出是由攻击导致，最好通过暂停执行来进行阻止。

```
void __attribute__((noreturn)) __stack_chk_fail (void)
{
    // XC32 default weak implementation
    while(1)
    {
        // Replace this code with application-specific code
        __builtin_software_breakpoint();
    }
}
```

15.3.3 自定义堆栈金丝雀值

堆栈溢出保护器的实现依赖于受保护函数的调用堆栈中的*机密信息*值。该值称为金丝雀，在编译时即确定。应用程序代码可以（并且应该）定义特定于应用程序的值。该值必须对任何潜在攻击者保密。

```
extern unsigned long __stack_chk_guard;
void __attribute__((constructor, optimize("-fno-stack-protector")))
my_stack_guard_setup(void)
{
    __stack_chk_guard = 0x12345678; // the new secret value for the canary
}
```

- 定制 setter 函数必须是构造函数，以确保在执行应用程序之前进行调用。
- `__optimize__ ("-fno-stack-protector")))`属性禁止该函数的堆栈保护器选项，以避免在使用 `-fstack-protector-all` 时出现运行时错误。

15.3.4 工作中的堆栈溢出保护器

下面通过示例来展示堆栈溢出保护器的实用性：

```
extern unsigned long __stack_chk_guard;
void __attribute__((constructor, optimize("-fno-stack-protector")))
```

```
my_stack_guard_setup(void)
{
    __stack_chk_guard = 0xDEADBEEF; // the new value for the canary
}

void __attribute__((noreturn)) __stack_chk_fail(void)
{
    // handle the failure in a way that is appropriate for your application
    while(1)
    {
      // A software breakpoint is not likely to be an appropriate response to an attack!
      __builtin_software_breakpoint();
    }
}

void foo (char* ptr)
{
  int retval;
  char buffer[10];       // buffer located on the stack
  char *dest = buffer;

  // A buffer overrun here will cause a stack smash
  // The SSP feature helps you to identify & catch vulnerabilities in your application code.
  // It does not prevent them.
  while (*ptr != 0)
  {
      *dest++ = *ptr++;
  }

  return;
}

int main (void)
{
    // In a real application, this data may come from an external source,
    // making the application vulnerable to an attack.
    char string[] = "hello world!";

    foo(string);

    while(1);
}
```

本例通过 `-fstack-protector` 成功构建并在目标器件上完成调试，代码执行应在 `__stack_chk_fail()` 函数中的软件断点处停止。

15.4 堆栈指导

PRO 编译器许可证提供编译器的堆栈指导功能，可用于估算程序所用的任何堆栈的最大深度。

运行时堆栈溢出会导致程序失败，且难以继续跟踪，特别是当程序复杂并使用中断时。编译器的堆栈指导功能会构造程序的调用图并对其加以分析，以确定每个函数的堆栈使用情况，并生成一份报告，用以推断程序使用的堆栈的深度。在程序开发过程中就对堆栈的使用情况进行监视，将减少发生堆栈溢出的情况。

该功能通过`-mchp-stack-usage` 命令行选项来使能。

使能后，堆栈指导功能即会完全自动运行。对于编译器的命令行执行，编译成功后会直接在控制台上显示一个报告。在 MPLAB X IDE 中编译时，该报告将显示在 **Output**（输出）窗口的编译视图中。

如果使用`-Wl,-Map=`*`mapfile`* 命令行选项或 MPLAB X IDE 项目属性中的等效控件发出请求，映射文件中将提供有关堆栈使用的更详细信息及其永久记录。

15.4.1 什么是堆栈溢出？

如果堆栈指针超出为调用堆栈保留的 RAM 地址范围，即可能发生堆栈溢出。当应用程序占用的空间超过为堆栈保留的空间时，可能会改写和破坏其他数据，例如静态分配的变量和堆。此外，当访问其他变量时，也可能会破坏堆栈中的值。这种数据损坏可能难以调试，并导致应用程序的运行时行为不可预测。

15.4.2 估计堆栈使用量

在裸机嵌入式应用程序中，应用程序开发人员必须确定适合为堆栈保留的最小数据 RAM 量，并将该值传递给 XC32 链接器。链接器随后使用该值来确保为堆栈保留足够的 RAM。

但是，应用程序的确切堆栈要求只能在运行时确定，具体原因在下文说明。在链接时，XC32 可使用静态分析来*估计*应用程序所需的最大堆栈大小，并通过可读性高的报告提供指导。实现方法为先计算每个函数的堆栈使用量，然后使用应用程序的调用图来找到最大的堆栈使用量。

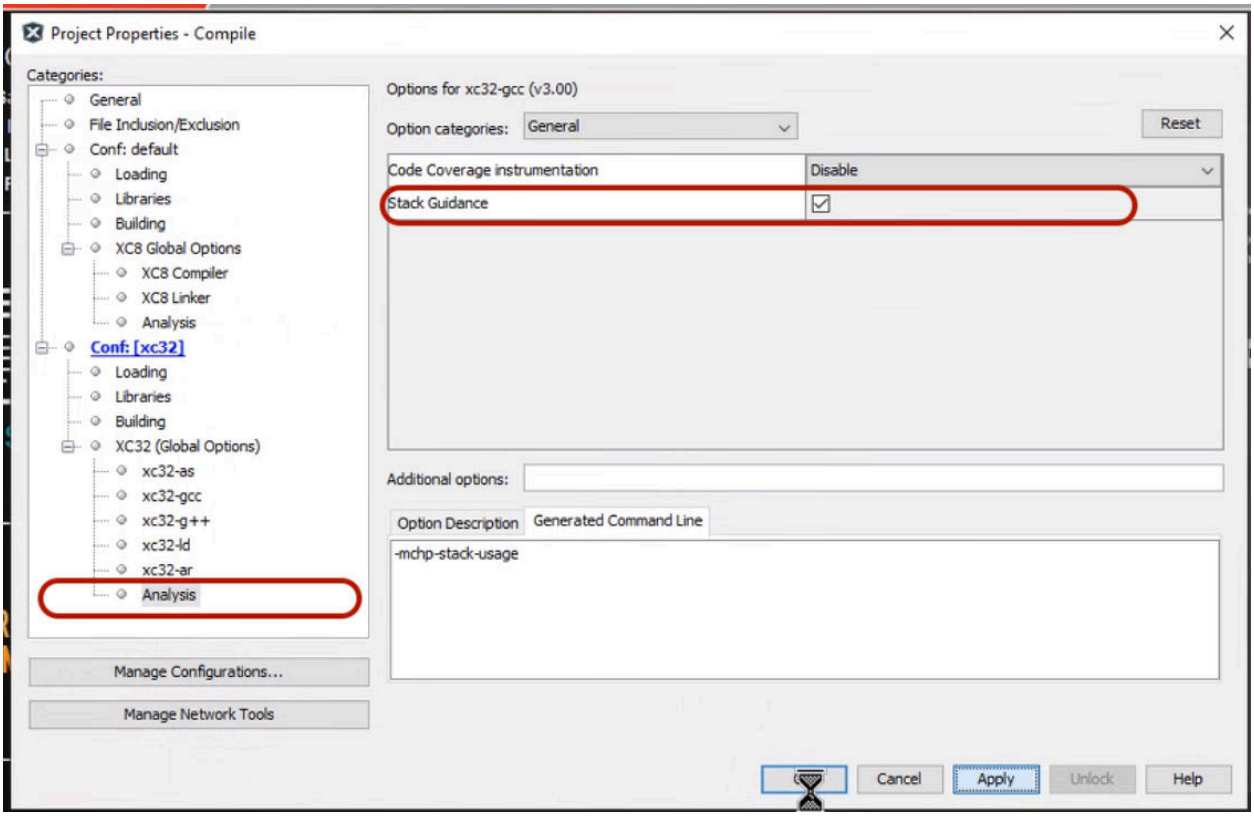
一般来说，该报告无法提供确切的值，但会提供可用于确定适合为堆栈保留的大小信息。*只有用户精准了解其应用程序的系统级运行时行为。*

15.4.3 使能堆栈使用量报告

仅当具备 PRO XC32 许可证时才能使用该功能。堆栈使用量报告通过`-mchp-stack-usage` 命令行选项使能，当在 MPLAB X Project Properties 中使能 Stack Guidance 复选框时，该报告将传递给 XC32。

要从 MPLAB X 使能 Stack Usage Analysis（堆栈使用量分析），应在 *Project Properties > XC32 > Analysis*（项目属性 > XC32 > 分析）中设置 Stack Guidance。

图 15-2. Project Properties 中的 Stack Guidance



或者通过命令行传递`-mchp-stack-usage`。

```
xc32/<version>/bin/xc32-gcc -mprocessor=ATSAME70N20B -mchp-stack-usage -O0 test.c
```

15.4.4 解读报告

堆栈使用量报告被写入以下位置：

- 直接写入标准误差（在命令行执行终端上或在 MPLAB X 输出窗口中）。

- 写入映射文件。

这两个位置上的信息相同，但映射文件可作为记录稍后查看。

报告内容如下：

- 静态分析可确定的最大堆栈使用量。
- 无法确定最大堆栈使用量的原因列表，例如递归或堆栈的变量调整。
- 并查集和/或中断处理函数的列表，这些函数不会被程序的主流程直接调用。

要根据具体应用程序确定合适的堆栈大小：

首先，从复位处理程序调用图中的初始值开始。在本例中，报告显示复位处理程序调用图需要 72 个字节。

堆栈使用量报告

```
============= STACK USAGE GUIDANCE =============
In the call graph beginning at Reset_Handler,
72 bytes of stack are required.
```

其次，根据具体应用程序的系统级运行时行为添加列出的堆栈限额。

堆栈使用量报告

```
However, the following cautions exists:
1. The following functions cannot be connected to the main call graph.
This is usually caused by some indirection:
frame_dummy uses 8 bytes
Dummy_Handler uses 0 bytes
__libc_init_array uses 40 bytes
You must add stack allowances for those functions.
================================================
```

第三，按照下一节所述为中断处理程序（中断服务程序）添加堆栈限额。

15.4.4.1 Cortex-M 中断

堆栈使用量报告打印应用程序中定义的中断处理程序列表及其堆栈消耗，以便可以估计中断产生的堆栈开销。

在该现场中，使用 `interrupt` 函数属性允许编译器将函数作为中断处理程序进行识别和分析，并将其报告为中断处理程序。

未使用__attribute__((interrupt))	使用__attribute__((interrupt))
1. The following functions cannot be connected to the main call graph. This is usually caused by some indirection: FstHandler uses 408 bytes SndHandler uses 408 bytes	1. The following functions are interrupt functions: FstHandler uses 408 bytes SndHandler uses 408 bytes

重要：
当 Cortex-M 目标器件上发生中断时，应对堆栈估计进行调整，即添加适当的空间进行中断现场保护，如下所示：

- 具有软件浮点支持——32 字节（4 字节 * 7 个调用方保存的寄存器（R0-R3、R12、LR 和 PC）以及退出中断处理程序时要执行的下一条指令的地址）。
- 使用硬件浮点单元（FPU）——164 字节（超过 32 字节 + 4 字节 * 33 个浮点寄存器）——假定禁止惰性现场保护。

在估计堆栈使用量时，务必添加适当的值以进行中断现场保护。中断现场由内核（而非编译器）保护。

15.4.4.2 Cortex-A 中断

当 Cortex-A 目标器件上发生中断时，最坏情况下应在堆栈指导的基础上调整 56 字节（R0-R12 和 PC x 4 字节）。

- 强烈建议对中断处理程序使用 `interrupt` 函数属性，因为可以重新定位向量表，但很难根据地址找到 ISR。此外，还可以在运行时设置/修改向量表。
- 如果允许嵌套中断，则应考虑嵌套成本。

15.4.5 堆栈使用限制

如果应用程序包含属于以下至少一个类别的函数，则对最大堆栈使用量的静态估计将不准确：

- 无法获得堆栈使用量信息的函数（未使用`-mchp-stack-usage` 进行编译/汇编）。
- 包含对其他函数的间接调用的函数（无法识别被调用方）。
- 包含变量堆栈调整、行内汇编或汇编程序生成的堆栈使用量信息（可能不准确）的函数。
- 递归函数（直接或间接）——仅当循环中至少有一个函数的堆栈消耗不为零时。

对于每种情况涉及的函数都提供了相关列表。

请注意，如果汇编代码会调整堆栈，则汇编文件的堆栈使用量估计可能不准确。在估计堆栈使用量时，务必考虑这些调整。

15.4.6 堆栈示例

举个简单示例：

test.c

```
int max(int a, int b){
        if (a < b)
                return b;
        return a;
}

int main() {
        int a[10], i, j;

        for (i =0; i < 10; i++)
                a[i] = i;
        for (i = 0; i < 10; i++)
                a[i] = max(a[i], a[10-i/2]);
        return a[0];
}
```

当使用

```
xc32-gcc -mprocessor=ATSAMD51N20A -mchp-stack-usage -O0 test.c
```

进行编译时，堆栈使用量报告显示如下

堆栈使用量报告

```
============= STACK USAGE GUIDANCE =============
In the call graph beginning at Reset_Handler,
    72 bytes of stack are required.

However, the following cautions exists:

1. Indeterminate stack adjustment has been detected:
    _init uses 24 bytes
    _fini uses 24 bytes
No stack usage predictions can be made.

2. The following functions cannot be connected to the main call graph.
This is usually caused by some indirection:
    _init uses 24 bytes
    _fini uses 24 bytes
    __do_global_dtors_aux uses 8 bytes
    frame_dummy uses 8 bytes
    Dummy_Handler uses 0 bytes
You must add stack allowances for those functions.
================================================
```

16. 函数

以下几节介绍如何编写函数的定义，特别是如何定制它们来满足您的应用程序。此外还介绍了用于参数和返回值的约定，以及汇编调用序列。

16.1 编写函数

函数可以根据 C/C++语言按照通常的方式编写。

惟一对函数具有作用的说明符为 `static`。中断函数通过使用 `interrupt` 属性定义，请参见函数属性和说明符。

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。`static` 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` 类型，汇编代码中用于表示函数的等效符号可能会发生变化，请参见静态变量。该说明符不会改变对函数进行编码的方式。

16.2 函数属性和说明符

16.2.1 函数属性

编译器关键字`__attribute__`用于指定函数的特殊属性。该关键字后跟包含在双括号内的属性说明。

此外，还可通过在每个关键字的前后两端使用`__`（双下划线）来指定属性（例如，`__noreturn__`代替 `noreturn`）。这样，在头文件中使用这些属性时便不必担心可能与宏同名。

要指定多个属性，请在双括号内通过逗号分隔它们，例如：

```
__attribute__ ((address(0x100), keep))
```

属性说明符可以放在函数的返回类型之前或之后。

注：在整个项目中一致地使用函数属性非常重要。例如，如果某个函数在文件 A 中定义时带有某种属性，而在文件 B 中通过 `extern` 声明时未带有这种属性，则可能会产生链接错误。

address(*addr*)

address 属性指定带该属性的程序在存储器中放置的绝对物理地址。该属性可与 C 和 C++函数一起使用。

编译器不会对地址值执行错误检查，因此应用程序必须确保该值对于目标器件是有效的。包含函数的段将被定位到指定地址处，无论链接描述文件中指定的存储器区域或目标器件上的实际存储器范围如何。应用程序代码必须确保地址对于目标器件是有效的。

为了有效利用绝对段和新的最佳适应分配器，不应在链接描述文件中映射标准程序存储器和数据存储器段。内置的链接描述文件不会映射大多数标准段，例如`.text`、`.data`、`.bss` 或`.ramfunc` 段。通过不在链接描述文件中映射这些段，可以允许使用最佳适应分配器而不是顺序分配器来分配这些段。未在链接描述文件中映射的段可以分布在绝对段周围，而链接描述文件映射的段则会被组合在一起并按顺序分配，这有可能导致与绝对段发生冲突。

alias ("*symbol*")

指示该函数是另一个符号的别名。例如：

```
void foo (void) { /* stuff */ }
__attribute__ ((alias("foo"))) void bar (void);
```

在上例中，符号 `bar` 会被视为符号 `foo` 的别名。

always_inline

指示编译器始终内联声明为 `inline` 的函数，即使未指定任何优化级别也不例外。

const

如果某个纯函数的结果完全通过其参数来确定（即，不取决于任何全局变量的值），则该函数可使用 const 属性进行声明，从而允许更为积极的优化。请注意，如果对指针参数进行解引用的函数取决于引用的值，则该函数无法声明为 const，因为引用的存储空间不被视为函数的参数。

deprecated

deprecated (msg)

当使用指定为 deprecated 的函数时，编译器将产生警告。可选的 *msg* 参数（必须为字符串）将打印在警告中（如存在）。deprecated 属性还可用于变量和类型。

externally_visible

该属性与函数一起使用时可确保函数在当前编译单元之外保持可见。这可能会阻止对函数执行某些优化。

format (*type*, *format_index*, *first_to_check*)

format 属性指示函数接受参数列表中 index 位置处的 printf、scanf、strftime 或 strfmon 样式的格式字符串和参数，并指示编译器根据格式字符串中的转换说明符对从 *first_to_check* 处开始的参数进行类型检查，就如对标准库函数执行的一样。

type 参数为 printf、scanf、strftime 或 strfmon 之一（可以具有前后的双下划线，例如，__printf__），并决定如何解释格式字符串。

format_index 参数指定格式字符串在函数参数中的位置。函数参数从左边开始从索引 1 开始编号。

first_to_check 参数指定要根据格式字符串检查的第一个参数的位置。由 *first_to_check* 指定的参数之后的所有参数都将接受检查。如果 *first_to_check* 为零，则不执行类型检查，编译器只检查格式字符串的一致性。

format_arg(*index*)

format_arg 属性指定函数处理 printf 样式的格式字符串，编译器应检查格式字符串的一致性。*index* 参数给出格式字符串在函数参数列表中的位置（从左边开始从索引 1 开始编号）。

interrupt

interrupt(*type*)

功能等效于 isr 属性。

isr

isr(*type*)

指示编译器为作为中断处理函数的函数生成序言和结语代码，如中断所述。对于 Cortex-A 器件，*type* 参数指定中断的类型，具体可以是标识符 irq、fiq、abort、undef 或 swi 之一（不区分大小写）。针对 Cortex-M 器件进行编译时，将忽略任何指定的参数。

keep

keep 属性阻止链接器在--gc-sections 生效时删除未使用的函数。

long_call

始终使用间接调用指令来调用函数。

malloc

malloc 属性断言来自函数的任何非空指针返回值都不会是在函数返回点存活的任何其他指针的别名。这使编译器可以执行更积极的优化。

naked

指示编译器不应为函数生成序言或结语代码。

noinline

无论优化级别如何，都永远不考虑对使用 noinline 属性声明的函数进行内联。

nonnull(*index*, ...)

指示编译器，函数的一个或多个指针参数必须为非空。当-Wnonnull 选项生效时，如果编译器可以确定在调用函数时有一个空指针被提供给任何 nonnull 参数，则会发出警告诊断。*index* 参数指示函数参数列表中需为非空的指针参数的位置（从左边开始从索引 1 开始编号）。如果未提供任何参数，则函数的所有指针参数均会被标记为非空。

nopa

指示编译器在执行过程抽象时不应考虑为该函数生成的汇编代码。

noreturn

告知编译器该函数永远不会将控制返回给其调用方。在某些情况下，这会使编译器可以在调用函数中生成更高效的代码，因为可以不考虑函数是否返回的行为而执行优化。声明为 noreturn 的函数的返回类型应始终为 void。

optimize(*optimization*)

optimize 属性允许使用与通过命令行指定的优化不同的优化来编译函数，并且该优化将应用于程序的其余部分。*optimization* 参数可以是数字或字符串。字符串参数表示用于控制优化的命令行选项，例如，要使能窥孔优化（-fpeephole），应使用 optimize("peephole")。-f 选项前缀不需要与参数一起指定。如果要指定多个优化，应将各个参数用逗号分隔，但不要使用空格字符。以 O 开头的参数会被假定为优化级别选项，例如 optimize("O1,unroll-loops")会开启 1 级优化和展开循环优化（由-funroll-loops 命令行选项控制）。此外，数字参数也会被假定为优化级别，例如 optimize(3)会开启 3 级优化，相当于下例中属性的完整使用。

```
int __attribute__((optimize("O3"))) pandora (void) {
  if (maya > axton)
    return 1;
  return 0;
}
```

例如，通过该功能，可以对频繁执行的函数使用更积极的优化选项进行编译，产生更快和更长的代码，而其他函数则可以使用较不积极的选项进行调用。但是，该功能通常不用于生产编译。

pure

向编译器指示该函数是纯函数，允许在调用该函数时进行更积极的优化。pure 函数除了其返回值之外没有任何副作用，并且返回值仅依赖于参数和/或（非可变）全局变量。

ramfunc

ramfunc 属性在通常从闪存执行代码的器件上将带该属性的程序定位到 RAM（而非闪存）中。该属性可用于 C 函数和 C++类方法。编译器的默认运行时启动代码在程序启动时使用数据初始化模板将与使用该属性的函数相关的代码从闪存复制到 RAM。

例如，下面定义了一个放置在数据存储器中的 C 函数：

```
__attribute__((ramfunc))
unsigned int myramfunct(void)
{ /* code */ }
```

<sys/attribs.h>头文件提供了用于此目的的__ramfunc__宏。例如：

```
#include <sys/attribs.h>
__ramfunc__ unsigned int myramfunct(void)
{ /* code */ }
```

在 C++代码中，它可与类方法一起使用，如下例所示。

```
class printmyname {
    // Access specifier
public:
    // Data  Members
    string myname;
    int  dummy ;
    // Member Functions()
    printmyname (){
        myname = "microchip";
    }

    void  __attribute__((ramfunc, long_call)) set_name(string newname){
        myname = newname;
        dummy = 9;
    }
    void printname() { cout << "name is:" << myname; }
};
```

转移/调用指令 BL 的范围有限（Arm 指令为 32 MB，Thumb2 指令为 16 MB，Thumb 指令为 4 MB）。因此，RAM 可能不在闪存中代码的调用范围内。要允许此类调用，需对 RAM 中函数的定义额外使用 long_call 属性，以便能够从闪存中的函数调用。例如：

```
__attribute__((ramfunc, long_call))
unsigned int myramfunct(void)
{ /* code */ }
```

<sys/attribs.h>头文件提供了__longramfunc__宏，该宏将同时指定 ramfunc 和 long_call 属性。例如：

```
#include <sys/attribs.h>
__longramfunc__ unsigned int myramfunct(void)
{ /* code */ }
```

如果闪存中的函数被 RAM 中的函数调用，则基于闪存的函数定义必须包含 long_call 属性。

```
__attribute__ ((long_call))
unsigned int myflashfunct(void)
{ /* code */ }

__attribute__((ramfunc))
unsigned int myramfunct(void)
{
        return myflashfunct();
}
```

将函数放置在 RAM 中而不是闪存中会增大代码长度并延长启动时间，因为在启动时必须将带属性的函数从闪存复制到 RAM。此外，还可能对运行时性能产生影响，具体取决于目标器件。请验证应用程序是否满足启动时序约束。此外，还需验证提升的运行时性能是否满足应用程序的时序要求，以及应用程序与受该设计选择负面影响的时序之间是否没有隐藏依赖性。

section("*name*")

将函数放入指定的段。例如：

```
void __attribute__ ((section (".wilma"))) baz () {return;}
```

在上例中，baz 函数将放入.wilma 段。-ffunction-sections 命令行选项对使用 section 属性声明的函数没有影响。

short_call

即使指定了-mlong-calls 命令行选项，也总是使用绝对调用指令调用函数。

space(*id*)

将函数放入由 id 参数标识的存储空间。id 参数可能是 prog（将函数放入程序空间（即 ROM）中）或 data（将函数放入数据段（即 RAM）中）。与 section 属性不同的是，不明确指定实际的段。

stack_protect

如果设置了以下选项之一，则该属性会向函数中添加堆栈保护代码：`-fstack-protector`、`-fstack-protector-strong` 或`-fstack-protector-explicit`。

优化可能会影响堆栈保护：

- 函数内联可能会影响函数是否受保护。
- 删除未使用的变量可阻止函数受保护。

tcm

尝试将函数放入紧耦合存储器（TCM），以提供高度一致的访问时间。实际的段将由编译器确定，可能基于其他属性（如 `space`）。例如：

```
void __attribute__((tcm)) foo (void) {return;}
```

在上例中，编译器会尝试将 `foo` 放入紧耦合程序存储器。请注意，目标器件上程序存储器或数据存储器中可用的 TCM 空间可能会有所不同，因此编译器无法确保带 `tcm` 属性的所有函数都能放入 TCM。

另请参见紧耦合存储器

unique_section

将函数放入惟一命名的段中，就如同`-ffunction-sections` 生效一样。如果该函数还具有 `section` 属性，将使用给定段的名称作为生成的惟一段名称的前缀。例如：

```
void __attribute__ ((section (".fred"), unique_section) foo (void) {return;}
```

在上例中，`foo` 函数将放入`.fred.foo` 段。

unsupported

指示编译器，该函数不受支持，类似于 `deprecated` 属性。如果调用该函数，将会发出警告。

unused

向编译器指示该函数可能不会被使用。如果该函数未被使用，编译器不会发出警告。

used

指示编译器，该函数总是会被使用，即使编译器无法确定该函数是否被调用（即，如果仅从行内汇编语句中调用状态函数），也必须为该函数生成代码。

warn_unused_result

如果调用方未使用指定函数的返回值，则发出警告。

weak

弱符号指示如果有同一符号的另一个版本可用，则改为使用该版本。例如，如果某个库函数已实现，可以通过该属性使用用户编写的函数覆盖它。

16.3 函数代码的分配

与 C/C++函数相关联的代码通常放置在目标器件的程序闪存中。或者，也可以通过使用`__ramfunc__`或`__longramfunc__`宏以及这两个宏代表的 `ram_func` 和 `long_call` 属性，将函数定位到 RAM 中而不是闪存中，并从其中执行。更多信息，请参见函数属性。

闪存的另一种替代方案是将函数放入这些器件的紧耦合存储器（如果可用）。有关该存储器的信息，请参见紧耦合存储器。

16.4 更改默认的函数分配

与 C/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` 属性并指定函数的虚拟地址来实现，请参见变量属性。

此外，还可以通过将函数放入用户定义的段，然后将该段链接到相应地址处而放置在特定位置，请参见变量属性。

要将可执行代码放入仅执行存储器（XOM）区域，应使用定制链接描述文件将文本输入段映射到新的输出段。接下来，将该输出段映射到所需的 XOM 区域。这必须结合-mpure-code 选项（见特定于 PIC32C/SAM 器件的选项）来执行。该选项可确保文本段不包含只读数据（如果 XOM 中存在此类数据，则无法访问），并在文本段上放置一个特殊标志以指示其中仅包含代码而不包含数据。使用 XOM 可帮助保护固件免遭第三方窃取或逆向工程。例如，将以下代码添加到链接描述文件中。

```
SECTIONS
{
 .purecode :
 {
 INPUT_SECTION_FLAGS (SHF_ARM_PURECODE) *(.text .text.*)
 } > purecode_memory
}
```

此更改将确保名称与.text 或.text.*匹配且标有 SHF_ARM_PURCODE 段标志（由-mpure-code 选项设置）的输入段将被放入.purecode 输出段。该.purecode 输出段将映射到名为 purecode_memory 的存储器区域，该存储器区域必须已事先使用 MEMORY 命令进行定义以与目标器件上所需的 XOM 单元相关联。关于链接描述文件的更多信息，请参见《MPLAB XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）。

16.5 函数长度限制

不存在关于函数可以多长的理论限制。

16.6 函数参数

MPLAB XC 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。

堆栈指针总是对齐到 4 字节边界上。

- 长度小于 32 位整型的所有整型都会先被转换为 32 位值。参数的前 4 个 32 位通过寄存器 R0-R3 传递（关于每种数据类型需要多少寄存器，请参见下表）。
- 在调用函数时：
  寄存器 R0-R3 用于向函数传递参数。这些寄存器中的值不会跨函数调用保存。
  寄存器 R0-R3、R12、R14 和 R15 是调用方保存的寄存器。调用函数必须将这些值压入堆栈，以便保存寄存器的值。
  寄存器 R4-R7 是被调用方保存的寄存器。被调用函数必须保存这些寄存器中它需要修改的任何寄存器。
  如果优化器取消将 R11 寄存器用作帧指针，则该寄存器是一个被保存的寄存器。否则，该寄存器是一个保留的寄存器。
  寄存器 R14 包含函数调用的返回地址。

表 16-1. 需要的寄存器

数据类型	需要的寄存器数
char	1
short	1
int	1
long	1
long long	2
float	1
double	2

.....（续）

数据类型	需要的寄存器数
long double	2
structure	最高为 4，取决于结构体的长度。

16.7 函数返回值

函数返回值在寄存器中返回。

整型或指针值均放入寄存器 R0，最多需要延伸至 R3。对于浮点值也是如此。

如果函数需要返回的聚合值太大而无法放入返回寄存器，则将被放入由调用方分配的堆栈空间。

16.8 调用函数

默认情况下，使用直接形式的调用（bl）指令来调用函数。该操作可以通过使用应用于函数的属性或命令行选项进行更改，从而进行较长但不受限制的调用。长调用通常用于两个存储器区域之间的调用，例如从 Flash 中的函数调用 RAM 中的函数或反之。

-mlong-calls 选项（见特定于 PIC32C/SAM 器件的选项）会强制在默认情况下采用寄存器形式的调用。生成的代码会较长，但从目标地址的角度来说，调用不受限制。

可以对某个函数定义使用 long_call 属性，从而对于该特定函数总是强制使用较长的调用序列。可以对某个函数使用 short_call 属性，使得对它的调用使用较短的直接调用，即使-mlong-calls 选项有效。

16.9 内联函数

通过将函数声明为 inline，可以指示编译器将该函数的代码合并到其调用方的代码中。这可以消除函数调用开销，通常可以使执行速度变快。此外，如果任何实际参数值为常量，其已知值可能允许在编译时进行简化，从而不需要包含内联函数的所有代码。对代码长度的影响较难预测。使用内联函数时的机器代码可能变长也可能变短，这取决于具体情况。

注：只有在函数的定义可见时（不仅仅是原型）才会发生函数内联。要将某个函数内联到多个源文件中，可以将函数定义放入每个源文件包含的头文件中。

要将某个函数声明为 inline，请在其声明中使用 inline 关键字，如下：

```
inline int
inc (int *a)
{
  (*a)++;
}
```

（如果使用了-traditional 选项或-ansi 选项，则写为__inline__而不是 inline）。此外，还可以通过命令行选项-finline-functions 将所有“足够简单”的函数设为内联。编译器会基于函数长度的估计值，启发式地决定哪些函数足够简单，值得以这种方式合并。

注：只有使用-finline 选项或使能优化时，才会识别 inline 关键字。

函数定义中的某些用法可能会使它不适合于内联替换。这些用法包括：使用 varargs、使用 alloca、使用长度可变的数据、使用计算 goto，以及使用非局部 goto。使用命令行选项-Winline 将在无法替换标记为 inline 的函数时产生警告，并给出失败的原因。

在编译器语法中，inline 关键字不会影响函数的链接。

当某个函数同时为 inline 和 static 时，如果对该函数的所有调用都被合并到调用方中，并且从不使用该函数的地址，则永远不会引用该函数自己的汇编代码。这种情况下，编译器不会实际输出该函数的汇编代码，除非您指定了命令行选项-fkeep-inline-functions。一些调用会由于各种原因而无法进行合并（特别是，无法合并函数的定义之前的调用，也无法合并定义中的递归调用）。如果存在无法合并的调用，则该函数将像正常情况下一样编译为汇编代码。如果程序引用了该函数的地址，则该函数也必须像正

常情况下一样进行编译，因为无法对这种情况进行内联。只有函数声明为 static，并且函数定义在所有函数使用之前，编译器才会消除 inline 函数。

当某个 inline 函数不为 static 时，则编译器必须假定可能存在来自其他源文件的调用。由于全局符号只能在所有程序中定义一次，所以绝对不能在其他源文件中定义该函数，导致无法合并其中的调用。因此，非 static 内联函数总是以通常的方式独立地编译。

如果在函数定义中同时指定了 inline 和 extern，则该定义仅用于内联。在任何情况下不会独立编译该函数，即使显式地引用了它的地址。此类地址会变为外部引用，如同仅声明了函数但未定义它。

inline 和 extern 的这种组合具有类似于宏的效果。将某个函数定义与这些关键字一起放入头文件，并将该定义（缺少 inline 和 extern）的另一个副本放入库文件。头文件中的定义将导致对函数的大多数调用被内联。如果仍然还有对该函数的任何调用，它们将引用库中的单个副本。

17. 中断

中断处理对于大多数单片机应用来说都是很重要的一个方面。中断用来使软件操作与实时发生的事件同步。当发生中断时，软件的正常执行流程被打断，调用特殊的函数来处理事件。当中断处理结束时，恢复先前的现场信息并继续正常执行流程。

PIC32 器件支持多个内部和外部中断源。这些器件允许高优先级中断抢占正在进行的任何较低优先级中断。

编译器为 C/C++或行内汇编代码中的中断处理提供了完全支持。本章对中断处理进行概述。

17.1 中断操作

编译器包含了允许完全从 C/C++代码中处理中断的特性。*中断代码*指的是由于发生中断而执行的任何代码。中断代码在执行相应的中断返回指令时执行完毕。这与*主子代码*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。

每个中断通常在特殊功能寄存器（SFR）中具有一个可以禁止该中断源的控制位。大多数中断还可以配置优先级。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和相应的技术参考手册。

17.2 编写中断服务程序

中断服务程序不接受任何参数也不返回任何结果，也就是说，其参数列表和返回类型都是 `void`。该模式并不强制，但执行不遵循该模式的 ISR 将导致不可预测的行为。

在 Cortex-M 内核上，硬件负责保护和恢复现场。当执行 ISR 时，硬件还会将寄存器 r0、r1、r2、r3、r12 和 r14 的内容保存在堆栈中，并在函数退出时将其恢复。因此，任何符合平台过程调用标准且没有返回值和参数的函数均可用作处理函数。无论如何，仍然应使用 `interrupt` 属性将函数标记为中断处理函数，以便编译器知道该函数已被使用。当发现 `interrupt` 属性时，编译器将生成代码以保证在进入函数时堆栈按 8 字节（双字）对齐，并在退出函数时将堆栈指针恢复为其原始值。这是一种安全措施，但如果器件配置为在进入异常时保证堆栈按 8 字节对齐，则不需要额外的指令。

以 Cortex-A MCU 为目标器件时，编译器必须生成应用程序代码来保存和恢复器件现场。因此，充当中断处理程序的函数必须使用 `interrupt` 属性，以便编译器可以确保生成正确的现场切换代码。

有关该器件的异常进入和退出行为的完整详细信息，请参见相应的架构参考手册。

17.2.1 interrupt 属性

`interrupt` 属性告知编译器该函数为中断处理程序。在针对 Cortex-M 器件编译中断处理程序时，可以选择使用该属性，但建议确保编译器生成代码以在进入函数时将堆栈指针对齐到 8 字节边界上。在针对 Cortex-A 器件编写中断处理程序时，强制使用该属性，因为编译器必须为此类器件上的中断处理程序生成现场切换代码。例如：

```
void __attribute__((interrupt)) hoppy (void)
{
  // interrupt code goes here
}
```

使用 C++编程时，必须将中断处理程序符号分配给 C 命名空间，这可以通过对定义使用 `extern "C"`来实现，例如：

```
extern "C"
void __attribute__((interrupt)) hoppy (void)
{
  // interrupt code goes here
}
```

注： `interrupt` 属性可以接受参数（如 GCC 手册中所述），但在目标器件搭载 Cortex-M 内核时会被忽略。

17.3 将处理函数与异常关联

每个异常处理程序（无论是内部还是外部）都与一个函数相关联。该函数处理的异常取决于其名称。处理函数的名称与其处理的异常的名称相对应，后缀为_Handler。例如，SysTick_Handler 是 SysTick 中断的异常处理程序。要定义定制处理程序，应编写一个名称与相应异常的默认处理程序名称一致的函数，并将其链接到应用程序中。除了标准的内部处理程序（见下节）之外，处理函数的名称均特定于器件。要查看处理函数名称的完整列表，请查看 pic32c/include/proc 中相应的器件头文件。

以下示例展示了如何创建定制 SysTick_Handler。

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

const static uint32_t LOWEST_IRQ_PRIORITY =
  (1UL << __NVIC_PRIO_BITS) - 1UL;

static uint32_t tick_counter;

__attribute__((interrupt)) void SysTick_Handler(void) {
    tick_counter += 1;
}

int main(void) {
    // Get the reload value for 10ms.
    uint32_t ticks = SysTick->CALIB & SysTick_CALIB_TENMS_Msk;

    // Set the IRQ priority, the SysTick reload value, the counter
    // value, then enable the interrupt. The same can be achieved
    // using the function SysTick_Config from the CMSIS API.
    NVIC_SetPriority(SysTick_IRQn, LOWEST_IRQ_PRIORITY);
    SysTick->LOAD = (uint32_t) ticks - 1UL;
    SysTick->VAL = 0UL;
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk
      | SysTick_CTRL_TICKINT_Msk
      | SysTick_CTRL_ENABLE_Msk;

    // Ensure the changes are written before continuing.
    __DSB();
    while (1) { __builtin_nop(); }
    return 0;
}
```

一些 PIC32C/SAM 器件中存在必须用软件使能的故障。以下示例将处理程序与 UsageFault 相关联并使能被零除错误。

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

__attribute__((interrupt)) void UsageFault_Handler(void);

void UsageFault_Handler(void) {
    __builtin_software_breakpoint();
}
uint32_t DemoFunction(uint32_t dividend, uint32_t divisor) {
    return dividend / divisor;
}

int main(void) {
    // Enable UsageFault and divide by zero errors
    SCB->SHCSR |= SCB_SHCSR_USGFAULTENA_Msk;
    SCB->CCR |= SCB_CCR_DIV_0_TRP_Msk;
    // Ensure the changes are written before continuing.
    __DSB();

    DemoFunction(2, 0);

    return 0;
}
```

下面是一个更长的示例，它使用两个中断来安全地访问共享数据，而无需禁止中断。tick_counter 的空间仅由 SysTick 和 PendSV 处理程序访问。两者以相同的优先级运行，这意味着它们不会中断对方。因此，对 tick_counter 的访问已正确序列化。

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

const static uint32_t LIMIT = 50;
const static uint32_t LOWEST_IRQ_PRIORITY =
    (1UL << __NVIC_PRIO_BITS) - 1UL;

static uint32_t tick_counter = 0;

__attribute__((interrupt)) void SysTick_Handler(void) {
    tick_counter += 1;
    __conditional_software_breakpoint(tick_counter <= LIMIT);
}

static uint32_t tick_limit = 0;
static uint32_t result = 0;

__attribute__((interrupt)) void PendSV_Handler(void) {
    if (tick_counter == tick_limit) {
        tick_counter = 0;
        result = 1;
    } else {
        result = 0;
    }
}

static inline void TriggerPendSV(void) {
    SCB->ICSR = SCB_ICSR_PENDSVSET_Msk;
    __DSB();
    __ISB();
}

uint32_t TickCounterReached(uint32_t limit) {
    tick_limit = limit;
    TriggerPendSV();
    return result;
}

int main(void) {
    const uint32_t ticks =
        (SysTick->CALIB & SysTick_CALIB_TENMS_Msk);

    SysTick_Config(ticks);
    NVIC_SetPriority(SysTick_IRQn, LOWEST_IRQ_PRIORITY);
    NVIC_SetPriority(PendSV_IRQn, LOWEST_IRQ_PRIORITY);
    __DSB();

    while (1) {
        if (TickCounterReached(LIMIT)) {
            __builtin_software_breakpoint();
        }
    }

    return 0;
}
```

为了在运行时设置中断处理程序，最好将向量表复制到 RAM 中。向量表通过*系统控制模块（System Control Block，SCB）*中的*向量表偏移寄存器（Vector Table Offset Register，VTOR）*定位。它通过 DeviceVectors 结构体描述，并位于 exception_table 变量中。这些定义位于通过 xc.h 包含的特定于器件的头文件中。要动态设置中断处理程序，应创建 DeviceVectors 结构体并将 exception_table 复制到其中，接着将 VTOR 指向它，然后根据需要更改处理程序。实际结构体定义位于默认器件启动代码中。它被定义为弱并被放入.vectors.default 段中，该段通过默认的链接描述文件映射到闪存区域。在 C 代码中，定义如下，其中省略了实际的函数指针。

```
__attribute__((section(".vectors.default"), weak, externally_visible))
const DeviceVectors exception_table = { ...
```

最好使用 CMSIS API（通过 xc.h 提供）的中断和异常部分来设置处理程序。每个中断在 IRQn_Type 枚举中都有一个对应的 IRQ 编号。IRQ 编号的名称与处理函数的名称类似，只是带有 _IRQn 后缀。要设置处理程序，应使用函数 NVIC_SetHandler() 为其提供 IRQ 编号和函数地址。

向量表对齐很重要，但特定于器件（有关详细信息，请参见相应的架构手册）。对于 Cortex-M7 器件（如 SAME70），向量表必须按大于或等于异常数量四倍的 2 的幂（最小 128 字节）对齐。目前，如果自行创建向量表，必须手动确定该数字。

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

// For a SAME70 device. Supports 90 exceptions (16 + 74).
// 90 * 4 = 360. Next highest power of 2 is 512.
#define TBL_ALIGN 512

DeviceVectors exn_table __attribute__((aligned(TBL_ALIGN)));
extern DeviceVectors exception_table;

static uint32_t counter;
__attribute__((interrupt)) void CustomSysTick_Handler(void) {
    counter += 1;
}

int main(void)
{
    memcpy(&exn_table, &exception_table, sizeof(DeviceVectors));

    // Disable interrupts, set the VTOR, ensure all data
    // operations are complete, then enable interrupts.
    __disable_irq();
    SCB->VTOR = (uint32_t) exn_table;
    __DSB();
    __enable_irq();

    NVIC_SetVector(SysTick_IRQn, (uint32_t) CustomSysTick_Handler

    // Code continues here.
}
```

17.4 异常处理程序

Cortex-M 内核支持用于内部事件的异常处理程序，但具体的内部事件集取决于架构。除了复位处理程序之外的所有内外部异常处理程序都有一个默认弱实现。该默认实现会在调试编译中执行 __builtin_software_breakpoint()，并在生产编译中进入无限循环。默认实现是名为 Dummy_Handler;的弱函数；如果希望定义自己的默认处理程序，可以改写默认实现。

为了方便起见，下面简要介绍一组常用于内部事件的异常处理程序。有关所有内部事件的列表以及内部事件的触发方式，请参见相应的架构参考手册。

17.4.1 复位

复位异常由函数 Reset_Handler 处理，其 IRQ 编号为 Reset_IRQn。该中断始终处于允许状态，其优先级在所有中断中最高。处理程序是 C 运行时启动代码的一部分，不得更改。可以通过定义某些函数来增强复位处理程序。有关详细信息，请参见 main、运行时启动和复位。

17.4.2 不可屏蔽中断（Non-Maskable Interrupt，NMI）

NMI 异常由函数 NonMaskableInt_Handler 处理，其 IRQ 编号为 NonMaskableInt_IRQn。该中断始终处于允许状态，其优先级在所有中断中仅次于复位中断。通常由硬件触发 NMI，但软件也可以使用以下代码触发 NMI。

```
SCB->ICSR = SCB_ICSR_NMIPENDSET_Msk;
```

17.4.3 硬故障

硬故障异常为通用故障机制，在某个故障没有其他适用的异常机制时使用。该异常由函数 `HardFault_Handler` 处理，其 IRQ 编号为 `HardFault_IRQn`。与 NMI 一样，该中断也始终处于允许状态，其优先级在所有中断中仅次于复位中断和 NMI。

17.4.4 SVCall

SVCall 异常由监控器调用（`SVC`）指令触发。其处理程序为 `SVCall_Handler`，IRQ 编号为 `SVCall_IRQn`，始终处于允许状态，并且优先级可配置。

17.4.5 PendSV

PendSV 为内部中断，通常用于强制通过软件进行现场切换。其处理程序为 `PendSV_Handler`，IRQ 编号为 `PendSV_IRQn`。该中断始终处于允许状态，并且优先级可配置（通常配置为最低优先级）。

可以使用以下代码触发 PendSV 中断。

```
SCB->ICSR = SCB_ICSR_PENDSVSET_Msk;
```

17.4.6 SysTick

SysTick 为内部中断，用于处理系统定时器引发的中断。其处理程序为 `SysTick_Handler`，IRQ 编号为 `SysTick_IRQn`，始终处于允许状态，并且优先级可配置。此外，也可以使用以下代码通过软件触发该处理程序。

```
SCB->ICSR = SCB_ICSR_PENDSTSET_Msk;
```

17.5 中断服务程序现场切换

使用 Cortex-M 器件时，硬件会在发生中断时保存参数寄存器（r0 至 r3）、IP 寄存器（r12）、链接寄存器（r14）、返回地址和程序状态寄存器的内容，并在中断返回时恢复这些内容。其他寄存器的内容必须由中断处理函数保存，这一点与任何其他函数一样。对于 C 函数，编译器将自动执行该操作。如果用汇编语言自行编写处理程序，请务必遵循过程调用标准。

现场切换期间可能会发生其他异常。这些异常由硬件处理。相关详细信息请参见相应的架构参考手册，但本质上不会发生两次现场切换。相反，在运行高优先级中断时会将其他中断设为等待处理。

以 Cortex-A MCU 为目标器件时，编译器必须生成相应的应用程序代码以保护和恢复器件现场；但中断处理程序永远不会保存浮点单元（FPU）寄存器的内容。如果这些器件需要从中断现场使用 FPU，则相应的应用程序需要定制中断处理程序。可以考虑用汇编代码编写中断处理程序，让其将所有需要保存的寄存器内容压入堆栈，调用处理程序的函数，恢复寄存器内容，然后返回。该汇编包装函数需要应用于每个中断处理程序（或者至少是使用 FPU 寄存器的中断处理程序）。在这种情况下，C 函数不应该设置 `interrupt` 属性。如果在处理程序中使用了寄存器 r0-r12、r14、d0-d7、d16-d32 和状态寄存器，则必须保存这些寄存器的内容。需要保存的寄存器数量可能会比较大，需要占用大量的堆栈空间，并且需要花费大量的时间来执行保存代码。

17.6 延时

从产生中断到执行 ISR 的第一条指令之间的时间称为*中断延时*。该延时受两个因素影响。

- **处理器中断处理**——这是处理器识别中断并转移到相关 ISR 所需的时间量。
- **保存 ISR 代码现场**——在进入 ISR 之前将寄存器内容保存到堆栈中所需的时间。

在大多数情况下，这些完全取决于 Cortex-M 器件上的硬件。具体来说，硬件将现场保存到堆栈中，从而无需编译器生成此类代码。因此，如果编写的 ISR 只用于通过硬件保存现场，不需要保存任何寄存器的内容，则 ISR 的中断延时完全取决于硬件。

在 Cortex-A 器件上，现场保护所需的时间取决于进入 ISR 时保存的寄存器数量，这取决于组成函数的代码。如果需要，该时间必须根据具体情况确定。

要确定中断延时的值，请参见具体器件的数据手册和相应的技术参考手册。

17.7 允许/禁止中断

CMSIS API 中的以下函数用于控制 CPU 的中断状态：

```
__enable_irq()
```

```
__disable_irq()
```

使用 Cortex-M 器件时，可使用 CMSIS API 来操作嵌套向量中断控制器（Nested Vector Interrupt Controller，NVIC），NVIC 控制几乎所有内外部中断的各个方面。更多信息，请参见 API 的*中断和异常（NVIC）*参考部分。NVIC API 函数在包含<xc.h>头文件后可用。

对于 Cortex-A 器件，可使用 CMSIS API 控制中断的各个方面。请参见*中断和异常*参考部分。这些函数在包含<xc.h>头文件后可用。

17.8 ISR 注意事项

编写中断服务程序时需要注意几个事项。

与所有编译器一样，限制中断函数或由中断函数调用的任何函数所使用的寄存器数量时，编译器生成和执行的现场切换代码会减少。使中断函数保持小而简单将有助于实现这一目标。此外，避免从中断现场使用硬件浮点单元（FPU），否则需要保存额外的 FPU 寄存器，进而增加中断延时。

需要考虑中断执行速度问题时，请避免从 ISR 中调用其他函数。您可能可以使用由应用程序的主控制循环处理的 `volatile` 标志来替换函数调用。

如果使用链接时优化（`-flto` 选项）进行编译，可能需要采取特殊步骤以确保与中断相关的代码不会被删除。这些优化大部分作用于整个程序。当分析整个程序时，会发现中断函数没有被任何其他函数调用，因此编译器认为可以将其删除。为了防止这种情况发生，必须使用 `used` 属性标记中断函数来告知编译器中断函数并非多余。对于中断函数所使用的数据对象，应使用相同的属性。或者，也可以考虑在禁止链接时优化的情况下编译包含中断函数的源文件。

18. main、运行时启动和复位

创建以 PIC32C/SAM/CEC MCU 为目标器件的 C/C++应用程序时，需在复位后、调用应用程序 `main()` 函数之前执行特定步骤来初始化器件、内核寄存器和 C/C++运行时环境。XC32 编译器为支持的每款器件提供启动代码，以便在启动过程的各个点执行用户定义的函数。本章将介绍这些功能以及在复位后、调用 `main()` 函数之前所执行的一般步骤。

18.1 main 函数

标识符 `main()` 保留作为应用程序代码的入口点。器件的启动代码将在执行完所有其他初始化步骤之后调用 `main()`。从 `main()` 函数返回后，控制权将返回给启动函数并将进入无限循环。调用标准 `exit` 或 `abort` 函数也会导致执行进入无限循环。启动函数不使用 `main()` 的返回值。

18.2 运行时启动代码

C/C++程序要求先初始化一些特定的对象，并且处理器处于某种特定状态，之后才会开始执行其 `main()` 函数。在执行 `main()` 之前执行这些任务是运行时启动代码的工作。

如果需要执行任何额外的初始化任务，通常不需要修改编译器提供的默认运行时启动代码，但需要更改启动代码所执行的任何现有操作时除外。 特殊的复位时和引导时钩子允许在启动序列中的不同位置定制启动代码。

PIC32 启动代码调用一个名为 `Reset_Handler()` 的函数以执行以下任务，这些任务将在后续章节中进行介绍。

1. 对于 Cortex-M 器件，确保堆栈指针寄存器初始化为指向系统堆栈的顶部；对于 Cortex-A 器件，确保堆栈指针寄存器初始化为指向相应堆栈的顶部。
2. 调用复位时程序（`_on_reset`）。
3. 使能浮点单元（FPU）器件（如果存在且已通过编译器选项使能）。
4. 使能指令和数据高速缓存（如果存在）。
5. 配置指令和/或数据紧耦合存储器（TCM）（如果存在且已使能）。另请参见紧耦合存储器。
6. 使用链接器生成的数据初始化模板初始化 data、bss 和 ramfunc 段。
7. 根据需要将堆栈重新定位到数据 TCM。
8. 将 `VTOR`（复位时向量表）寄存器初始化为中断向量表的地址（仅限 Cortex-M 器件）。
9. 执行标准 C 库的初始化。
10. 调用引导时程序（`_on_bootstrap`）。
11. 调用应用程序 `main()` 函数。
12. 从 `main()` 返回时，进入无限循环。

下面几节将进一步详细介绍每个步骤。

18.2.1 初始化堆栈指针和堆

只有某些器件明确执行该步骤。初始堆栈指针值使用符号 `_stack`（由链接器定义）定义，位于数据（RAM）存储器中。通过将符号 `_min_stack_size` 定义为正值（例如，通过使用链接器 `--defsym` 选项）来保留最少量的堆栈空间。请注意，尽管某些实现可能将堆栈基址定位在最高 RAM 地址，但 XC32 链接器可能将其放置在 RAM 中的其他位置。如果器件支持使用 `-mstack-in-dtcm` 选项将堆栈置于 TCM 中，则将遵循 TCM 初始化步骤将堆栈指针更新至 TCM 中的新位置。请参见将堆栈重新定位到 TCM 和紧耦合存储器。

尽管堆栈指针由 Cortex-M 器件上的硬件初始化，但在应用程序由其他代码自举的情况下，也可以从运行时启动代码初始化堆栈指针。请注意，以下示例中的代码由预处理器宏__REINIT_STACK_POINTER 保护。

```
#if defined (__REINIT_STACK_POINTER)
  /* Initialize SP from linker-defined _stack symbol. */
  __asm__ volatile ("ldr sp, =_stack" : : : "sp");
#ifdef SCB_VTOR_TBLOFF_Msk
  /* Buy stack for locals */
  __asm__ volatile ("sub sp, sp, #8" : : : "sp");
 #endif
  __asm__ volatile ("add r7, sp, #0" : : : "r7");
 #endif
```

对于 Cortex-A 器件，运行时启动代码为以下模式设置堆栈：FIQ、IRQ、中止、未定义、系统和监控器。完成后，系统处于监控器模式。

18.2.2 复位时程序

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。例如，可能需要在初始化程序的变量之前初始化 DDR RAM 控制器。为了实现这一点，提供了一个运行时钩子，这样便无需定制整个启动代码序列。

随启动代码提供了复位时程序（_on_reset）的空白弱实现。它在 C/C++语言现场完成最小程度的初始化之后由运行时启动代码调用。

如果确实需要自己定制该程序，可修改以下存根（注意函数名称中的前导下划线字符）。

```
void _on_reset(void)
{
  // Add code to be executed soon after reset here
}
```

该程序可放入项目中的任何源文件，并且无需调整任何编译器选项即可执行。

使用 C 或 C++语言编写该程序时需要注意一些特殊事项。代码不应假定已完全建立运行时环境。有关建立环境的顺序，请参见运行时启动代码。最重要的是，既不会初始化静态分配的变量（需要使用指定的初始化，或像对待未初始化变量那样赋 0 值），也不会针对 C++应用程序调用任何静态构造函数。在 C/C++应用程序中引用非自动变量可能会产生意外或不可预测的结果，但堆栈指针将已完成初始化。

18.2.3 使能 FPU 器件

在具有 FPU 的器件上，对于可使用浮点单元生成代码的应用程序，将使能该单元。仅当-mfloat-abi=hard|softfp 选项在链接器步骤生效时才执行该步骤，这是具有 FPU 的器件的默认行为。

18.2.4 使能高速缓存

在支持指令或数据可高速缓存存储器的器件上，将根据通过-mprocessor 选项控制的器件特定文件中的定义初始化高速缓存。

18.2.5 初始化对象和 RAM 函数

从 RAM 访问的对象和从 RAM 执行的函数必须对其分配的 RAM 进行初始化，之后才能开始执行 main() 函数，这是运行时启动代码执行的一项重要任务。

开始执行 main() 之前，必须将未初始化的非 auto 对象清零（赋 0 值）。此类对象在其定义中未赋值，例如以下示例中的 output：

```
int output;
int main(void) { ...
```

PIC32C/SAM 器件只有一个.bss 段，其中包含所有未初始化的对象。

运行时启动代码的另一个任务就是确保所有已初始化对象在程序开始执行之前包含其初始值。已初始化对象是那些不属于 auto 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 = 0x88;
int main(void) { ...
```

此类已初始化对象具有两个组成部分：其初始值（上例中的 0x0088），它存储在程序存储器中（即，放置在 HEX 文件中）；以及对象在程序执行期间（运行时）将驻留的并被访问的 RAM 中保留的空间。

只有一个.data 段，其中包含所有已初始化的对象。

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，从而使这些对象在执行 main() 之前包含正确的值。

由于 auto 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。auto 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 auto 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 auto 对象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 static 对象。

在复位时需要保存内容的对象应使用 persistent 属性进行限定，请参见标准类型限定符。这种对象链接到存储器的不同区域，运行时启动代码不会以任何方式更改它。

在执行使用 ramfunc 属性的任何函数之前，会将其从程序存储器复制到 RAM。这同样由运行时启动代码执行，并且与在访问已初始化对象之前复制其初始值的方式非常相似。

18.2.5.1 数据初始化模板

为了清零或初始化所有数据和 RAM 函数段，链接器会创建数据初始化模板，该模板装入名为.dinit 的输出段并在程序存储器中分配空间。libpic32c.a 中包含的 C/C++启动模块中的代码解释该模板，以指示必须如何初始化相应的段。

该模板初始化的段包括存放已初始化对象的段（如.data 段）以及包含 ramfunc 属性函数的段，所有这些段都必须将值从程序存储器中的模板复制到数据存储器，运行时将从数据存储器访问对象和函数。在调用 main() 函数之前，模板会将保存未初始化对象的其他数据段（如.bss 段）清零。运行时启动代码不考虑持久数据段（.pbss）。当应用程序的主程序获得控制权时，数据存储器中的所有对象和 RAM 函数都将已初始化。

对于需要初始化的每个输出段，数据初始化模板中均包含一个记录。每个记录都采用多种格式（在记录内由格式代码表示）之一。记录格式指定如何在记录中存储数据值，以及应如何使用数据值来初始化相应的段。--dinit-compress 选项（见 Dinit-compress 选项）控制模板可以使用哪些记录。数字格式代码及其代表的初始化类型如下：

#0	用零填充由相应输出段定义的 RAM。该记录中不存储任何数据字节。由 bss 段使用。
#1	将记录数组中的每个数据字节复制到与输出段关联的 RAM 中。由数据段以及与 ramfunc 函数关联的段使用。
#2	将同一 16 位值多次复制到与输出段关联的 RAM。由初始值为重复序列的数据段使用。
#3	将同一 32 位值多次复制到与输出段关联的 RAM。由初始值为重复序列的数据段使用。
#4	复制简化版 PackBits 编码的 data_record 并解压。 由数据段以及与 ramfunc 函数关联且包含大量连续零字节的段使用。

每种记录类型包含的数据都可以用如下所示的等效 C 结构来表示。记录的第一个元素是一个指针，指向数据存储器中的段。第二个元素是段长度或重复次数。第三个元素是格式代码，用于指示（上面列出的）记录的类型，以及应如何初始化相应的段。第四个元素用于对齐填充或初始值。第五个元素（如果存在）是数据字节数组。该模板通过两个空指令字终止。

```
/* For format values of 0 */
struct data_record_bss {
    uint32_t *dst;     /* destination address */
```

```
    uint32_t len;      /* length in bytes */
    uint16_t format;   /* format code */
    uint16_t padding;  /* padding for alignment */
};

/* For format values of 1 and 3 (also identical to format value 4) */
struct data_record_standard {
    uint32_t *dst;     /* destination address */
    uint32_t len;      /* length in bytes */
    uint16_t format;   /* format code */
    uint16_t padding;  /* padding for alignment or a 16-bit initialization value */
    uint32_t dat[0];   /* object-length data - holding initialization data */
};

/* For format values of 2 - objects are initialised with the same 16-bit value */
struct data_record_short_standard {
    uint32_t *dst;     /* destination address */
    uint32_t count;    /* count in bytes */
    uint16_t format;   /* format code */
    uint16_t dat       /* 16-bit repeated value data */
};

/* For format values of 4 - A simplified PackBits data compression is applied, where
each run of zeros is replaced by two 8-bit characters in the compressed array:
zero followed by the number of zeros in the original run. */
struct data_record_compressed {
    uint32_t *dst;               /* destination address */
    uint32_t count;              /* count in bytes */
    uint16_t format;             /* format code */
    uint16_t padding;            /* 16-bit repeated value data */
    uint32_t compressed_data[0]; /* compressed intialized data */
};
```

18.2.6 配置紧耦合存储器

在支持一个或多个 TCM 的器件上，当使能 TCM 后，将调用特定于器件的代码来执行满足请求的 TCM 配置所需的任何配置或初始化。另请参见紧耦合存储器。

18.2.7 将堆栈重新定位到 TCM

使用-mstack-in-itcm 或类似选项时，可将运行时堆栈放入 TCM。完成该步骤后，运行时堆栈将位于所请求的存储器区域。

18.2.8 设置 VTOR 寄存器

将 VTOR（即，向量表偏移寄存器）设置为反映中断向量表（Interrupt Vector Table，IVT）的起始地址（仅限 Cortex-M 器件）。该值由特殊符号__svectors（由 XC32 链接器定义）确定。Cortex-A 器件对异常处理程序没有特殊的初始化要求。

18.2.9 C 库初始化

调用函数__libc_init_array()以执行标准 C 库所需的全部初始化。在该步骤之前，标准 C 库程序可能会产生意外的结果。

18.2.10 调用_on_bootstrap()函数

当存储器、CPU 和库完成初始化之后，如果在调用 main()函数之前需要执行特殊的初始化步骤，则应在应用程序中定义_on_bootstrap()。该函数可以用 C 语言实现且没有任何警告，这一点与_on_reset()有所不同。

18.2.11 调用 main 函数

调用 main()函数，不使用任何返回值。从 main()返回后，控制权将返回给 Reset_Handler()并将执行进入无限循环。在支持 Thumb-2 指令集的器件上，可定义预处理器宏__DEBUG 以在从 main()返回后立即插入软件断点指令（BKPT）。

18.2.12 异常处理程序

异常表由编译器定义。对于基于 Arm Cortex-M 内核的器件，该表将包含初始堆栈指针和程序起始地址（如 `Reset_Handler()` 函数地址），以及中断服务程序（ISR）向量。使用 Cortex-A 器件时也会创建类似的表，只是未定义堆栈指针。该表位于 `.vectors` 段中。

特定于器件的启动代码定义了默认向量表 `exception_table`，以及名为 `Dummy_Handler()` 的默认 ISR。使用 Cortex-M 器件时，除了 `Reset_Handler()` 之外，`exception_table` 中的所有指针均初始化为指向 `Dummy_Handler()`，即进入无限循环。使用 Cortex-A 器件时，并非所有处理程序均默认为 `Dummy_Handler()`，FIQ 和 IRQ 中断处理程序可能具有特定于器件的实现。对于支持 Thumb-2 指令集的器件，当定义 `__DEBUG` 时，将在无限循环之前插入软件断点指令。

可通过用户代码重新定义符号 `exeception_table`、`Reset_Handler()` 和 `Dummy_Handler()` 以提供定制实现。

19. 库

MPLAB XC32 C/C++编译器提供经过 C90 和 C99 标准验证的函数库、宏、类型和对象，可辅助代码开发。

有关标准 C 库的描述，另请参见 *Microchip Unified Standard Library Reference Guide*，其内容适用于所有 MPLAB XC C 编译器。

该库包括字符串操作、动态存储器分配、数据转换、计时等函数以及数学函数（三角函数、指数函数和双曲函数）。用于文件处理的标准 I/O 函数亦包括在内，并且在分发时，它们支持使用命令行模拟器对主机文件系统进行完全访问。

除了编译器附带的库，还可以从编写的源代码中自行创建库。

19.1 智能 IO 程序

对于与 IO 函数的打印和扫描系列相关联的库代码，可以通过编译器在每次编译时基于编译器选项和在项目中使用这些函数的方式进行定制。这样做可以减少链接到程序映像中的冗余库代码量，从而可以减少程序所使用的程序存储器和数据存储器。

智能输出（打印系列）函数包括：

printf	fprintf	snprintf	sprintf
vfprintf	vprintf	vsnprintf	vsprintf

智能输入（扫描系列）函数包括：

scanf	fscanf	sscanf
vfscanf	vscanf	vsscanf

使能该功能后，编译器在每次编译时都会分析项目的 C 源代码，搜索任何智能 IO 函数调用。格式字符串所包含的转换规范用于所有调用，因为应用了这些规范，所以在程序映像输出中包含了库程序以及关联的功能。

例如，如果程序仅包含以下调用：

```
printf("input is: %d\n", input);
```

使能智能 IO 后，编译器将注意到程序中的 `printf` 函数只使用了`%d` 占位符，因而定义 `printf` 的链接库程序将包含至少能处理十进制整数打印的基本功能。如果在程序中添加了以下调用：

```
printf("input is: %f\n", ratio);
```

编译器将检测到 `printf` 使用了`%d` 和`%f` 占位符。这样，链接库程序将获得额外功能，确保能满足程序的所有要求。

下面一节对该编译器的智能 IO 功能如何运行的具体细节进行了详细介绍。有关<stdio.h>头文件中包含的所有 IO 函数的语法和使用，请参见 *Microchip Unified Standard Library Reference Guide*。

19.1.1 PIC32C/SAM 器件的智能 IO

使用 MPLAB XC32 C/C++编译器时，有多种 IO 库形式（表示越来越复杂的 IO 功能子集）可供使用并基于`-msmart-io` 选项和项目源代码中使用智能 IO 函数的方式链接到程序中。

禁止智能 IO 功能（`-msmart-io=0`）时，IO 函数的完整实现将链接到程序中。IO 库函数的所有功能都将可用，这些功能可能会占用目标器件上大量可用的程序和数据存储空间。

使能智能 IO 功能（`-msmart-io=1` 或`-msmart-io`）时，编译器将链接到复杂度最低的 IO 库形式中，该形式基于程序的 IO 函数格式字符串中检测到的转换规范实现程序需要的所有 IO 功能。这样可以大幅降低程序对存储器的需求，尤其是无需在程序中使用浮点功能来调用智能 IO 函数时。这是默认的设置。

编译器将单独分析每个 IO 函数的使用，因此当特定程序的代码可能需要 printf 函数为全功能函数时，可能只需要 snprintf 函数的基本实现。

如果 IO 函数调用中的格式字符串并非字符串字面值，则编译器将无法检测到 IO 函数的确切使用，IO 库的全功能形式将链接到程序映像中（即使使能了智能 IO）。在这种情况下，可使用该选项的-msmart-io=2 形式。这使得编译器假定格式化的 IO 函数未使用浮点数，可以安全地链接到仅整型格式的 IO 库。必须确保程序仅使用指定的转换规范，否则 IO 函数无法按预期工作。

以下面四个对智能 IO 函数的调用为例。

```
vscanf("%d:%li", va_list1);
vprintf("%-s%d", va_list2);
vprintf(fmt1, va_list3); // ambiguous usage
vscanf(fmt2, va_list4);  // ambiguous usage
```

在处理最后两个调用时，编译器无法从任一格式字符串中推断出任何使用信息。如果已知 fmt1 和 fmt2 指向的格式字符串只共同使用%d、%i 和%s 转换说明符，则可使用该选项的-msmart-io=2 形式。

所有程序模块应一致使用这些选项，以确保程序映像中包含的库程序为最佳选择。

19.2 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。

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/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。

19.3 使用库程序

在源代码中引用库函数或程序之后，库函数或程序（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。

注：不要在项目属性中指定 MPLAB XC32 系统包含目录（如/pic32c/include/）。xc32-gcc 编译驱动程序会自动为您选择 XC libc 及其相应的包含文件目录。xc32-g++编译驱动程序会自动为您选择 C++库及其相应的包含文件目录。手动添加系统包含文件路径可能会破坏这种机制，导致将错误的 libc 包含文件编译到项目中，从而导致包含文件和库之间的冲突。请注意，向项目属性中添加系统包含路径从来不是建议的做法。

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）。

```
#include <math.h> // declare function prototype for sqrt

int main(void)
{
  double i;

  // sqrt referenced; sqrt will be linked in from library file
  i = sqrt(23.5);
}
```

MPLAB Harmony 包含一组外设库、驱动程序和系统服务，可轻松用于应用程序开发。如需访问 plib.h（外设头文件），请访问 Harmony 网站（www.microchip.com/mplab/mplab-harmony）下载 MPLAB Harmony。

20. 混合使用 C/C++和汇编语言

通过使用两种不同的技术，可以将汇编语言代码与 C/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/C++模块中。本节介绍如何一起使用汇编语言和 C/C++模块。它给出了在汇编代码中使用 C/C++变量和函数的示例，以及在 C/C++中使用汇编语言变量和函数的示例。

项目中包含的汇编代码越多，维护它的难度就越高，所需时间也越多。随着项目的开发，编译器可能会改变工作方式，因为一些优化会着眼于整个程序。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，汇编代码很可能会发生失败。这些因素不会影响以 C/C++编写的代码。

注：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。

20.1 混合使用汇编语言与 C 变量及函数

以下准则说明了如何将独立的汇编语言模块与 C 模块进行接口。

- 遵循寄存器约定中所述的寄存器约定。特别是，使用寄存器 r0-r3 来进行参数传递。汇编语言函数将在这些寄存器中接收参数，并应在这些寄存器中向被调用函数传递参数。
- 寄存器约定中的表格介绍了必须跨非中断函数调用保存哪些寄存器。
- 中断函数必须保存所有寄存器。不同于普通的函数调用，中断可能在程序执行过程中的任意点发生。返回到正常程序时，所有寄存器都必须与发生中断之前相同。
- 在独立的汇编文件中声明且将由任意 C 源文件引用的变量或函数应使用汇编器伪指令 `.global` 声明为全局。在汇编文件中使用的未声明符号将被视为外部定义。

以下示例说明了如何在汇编语言和 C 中使用变量和函数，无论它们最初在何处定义。

文件 `ex1.c` 定义了要在汇编语言文件中使用的 `cFunction` 和 `cVariable`。C 文件还说明了如何调用汇编函数 `asmFunction` 和如何访问汇编代码定义的变量 `asmVariable`。

例 20-1. 混合使用 C 语言和汇编语言

```
        .syntax unified
        .cpu cortex-m7
        .thumb

        .global asmVariable
        .type asmVariable,%object
        .data
        .align 2
asmVariable:
        .space 4
        @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        @ char *asmFunction (char *s)
        @ {
        @    asmVariable = 0;
        @    if (s) {
        @       char *d = s, c;
        @       while ((c = *d)) {
        @         if (cFunction (c)) {
        @           *d = c & cVariable;
        @           ++asmVariable;
        @         }
        @         ++d;
        @       }
        @    }
        @    return s;
        @ }
        @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
        .global asmFunction
        .type asmFunction,%function

        .text
```

```
        .align 1

        .thumb_func
asmFunction:
        @ if the input string is not NULL
        cbnz r0, .L_not_NULL

        @ set 'asmVariable' to zero and return
        ldr    r1, =asmVariable
        str    r0, [r1]
        bx     lr

.L_not_NULL:
        @ r4-r7 are callee-saved registers
        @ LR contains the return address
        @ r0 is the first argument and also the return value of the
function
        push {r0, r4-r7, lr}

        @ d = s;
        mov r4, r0

        @ r6 - the value of the C variable (8-bit AND mask)
        @ r7 - counter of changed chars
        ldr     r6, =cVariable
        movs    r7, #0
        ldrb    r6, [r6]

.L_while:
        @ while ((c = *d))
        ldrb r5, [r4]
        cbz r5, .L_end_while
        @ if (cFunction (c))
        mov   r0, r5
        bl    cFunction
        cbz r0, .L_next_char

        @ *d = c & cVariable;
        ands r5, r5, r6
        strb r5, [r4]

        @ ++ the number of changed chars
        adds r7, r7, #1

.L_next_char:
        @ ++d;
        adds r4, r4, #1
        b .L_while

.L_end_while:
        @ write the no. of changes to 'asmVariable'
        ldr    r0, =asmVariable
        str    r7, [r0]

        @ return s;
        pop {r0, r4-r7, pc}

        .pool
        .size asmFunction, .-asmFunction
```

文件 ex1.S 定义了需要在链接的应用程序中使用的 asmFunction 和 asmVariable。汇编文件还说明了如何调用 C 函数 cFunction 和如何访问 C 代码定义的变量 cVariable。

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

extern int asmVariable;
extern char *asmFunction (char *s);

char cVariable = 0xDF;

char cFunction (char c)
{
   return c >= 'a' && c <= 'z';
```

```
}

int main()
{
   char s[] = "heLLo, wOrlD!";
   printf ("%s\n", s);

   char *d = asmFunction (s);
   printf ("%s\nchanges: %d", d, asmVariable);

   return 0;
}
```

在 C 文件 ex2.c 中，尽管对于函数声明这不是必需的，但请注意 asmFunction 是 char *函数并相应地进行声明。

在汇编文件 ex1.S 中，符号 asmFunction 和 asmVariable 通过使用.global 汇编伪指令而设为全局可见，可以由任何其他源文件访问。

20.2 使用行内汇编语言

在 C/C++函数内，可以使用 asm 语句将一行汇编语言代码插入编译器将生成的汇编语言中。行内汇编代码具有两种形式：简单和扩展。

在简单形式中，汇编器指令使用以下语法编写：

```
asm ("instruction");
```

其中，*instruction* 是有效的汇编语言构造。如果在 ANSI C 程序中编写行内汇编代码，则编写__asm__而不是 asm。

注：只能向简单形式的行内汇编代码传递单个字符串。

在使用 asm 的**扩展**汇编器指令中，指令的操作数使用 C/C++表达式来指定。扩展语法为：

```
asm("template" [ : [ "constraint"(output-operand) [ , ... ] ]
          [ : [ "constraint"(input-operand) [ , ... ] ]
             [ "clobber" [ , ... ] ]
          ]
        ]);
```

您必须指定一个汇编器指令 *template*，再加上每个操作数的操作数 *constraint* 字符串。*template* 指定指令助记符，以及（可选）操作数的占位符。*constraint* 字符串指定操作数约束，例如，操作数必须处于寄存器内（通常情况），或操作数必须为立即数。

以下各表列出了编译器支持的约束字母和修饰符。

表 20-1. 编译器支持的寄存器约束字母

字母	约束
r	任何通用寄存器
l	在 Thumb 状态下，内核寄存器 r0-r7。在 Arm 状态下，这是“r”约束的别名。
h	在 Thumb 状态下，内核寄存器 r8-r15。
t	在 Arm/Thumb-2 状态下，VFP 浮点寄存器 s0-s31。
w	在 Arm/Thumb-2 状态下，VFP 浮点寄存器 d0-d15 或 d0-d31（对于 VFPv3）。
x	在 Arm/Thumb-2 状态下，VFP 浮点寄存器 d0-d7。
Ts	如果指定-mrestrict-it（对于 Arm-v8），内核寄存器 r0-r7。否则，GENERAL_REGS（r0-r12 和 r14）。

表 20-2. 编译器支持的整型约束字母

字母	约束
G	在 Arm/Thumb-2 状态下，浮点型常量 0。

.....（续）	
字母	**约束**
I	在 Arm/Thumb-2 状态下，可用作数据处理指令中的立即数的常量（即，0 至 255 范围内按 2 的倍数循环移位的整数）。 在 Thumb-1 状态下，0 至 255 范围内的常量。
j	在 Arm/Thumb-2 状态下，适用于 MOVW 指令的常量。
J	在 Arm/Thumb-2 状态下，-4095 至 4095 范围内的常量。 在 Thumb-1 状态下，-255 至-1 范围内的常量。
K	在 Arm/Thumb-2 状态下，取反（二进制反码）后满足“I”约束的常量。 在 Thumb-1 状态下，乘以 2 的任意幂后满足“I”约束的常量。
L	在 Arm/Thumb-2 状态下，求补（二进制补码）后满足“I”约束的常量。 在 Thumb-1 状态下，-7 至 7 范围内的常量。
M	在 Thumb-1 状态下，0 至 1020 范围内为 4 的倍数的常量。
N	在 Thumb-1 状态下，0 至 31 范围内的常量。
O	在 Thumb-1 状态下，-508 至 508 范围内为 4 的倍数的常量。
Pf	除放宽、消耗或释放这三种存储器模型之外的存储器模型。

表 20-3. 编译器支持的约束修饰符

字母	**约束**
=	表示该操作数对于该指令是只写的：先前的值会被丢弃，替换为输出数据。
+	表示指令既会读取也会写入该操作数
&	表示该操作数是一个 earlyclobber 操作数，即在使用输入操作数完成指令之前就会被修改。因此，该操作数不能位于用作输入操作数的寄存器中，也不能作为任何存储器地址的一部分

20.2.1 示例：

- 插入位域
- 多条汇编器指令

插入位域

该示例演示如何使用 BFI 指令向一个 32 位宽的变量中插入位域。这个类似于函数的宏使用行内汇编代码来发出 BFI 指令，C/C++代码通常不会生成该指令。

```
/* Thumb2 insert bits */
#define _ins(tgt,val,pos,sz) __extension__({                        \
    unsigned int __t = (tgt), __v = (val); \
    __asm__  ("bfi\t%0,%1,%2,%3"                        /* template  */ \
             : "+r" (__t)                               /* output    */ \
             : "r" (__v), "M" (pos), "M" (sz));         /* input     */ \
    __t;                                                            \
})
```

此处的__v、pos 和 sz 是输入操作数。__v 操作数被约束为类型“r”（寄存器）。pos 和 sz 操作数被约束为类型“M”（0-32 范围内的常量或 2 的任意幂）。

__t 输出操作数被约束为类型“r”（寄存器）。“+”修饰符意味着指令既会读取也会写入该操作数，所以该操作数同时作为输入和输出。

以下示例说明了该宏的使用。

```
unsigned int result;
void example (void)
{
    unsigned int insertval = 0x12;
    result = 0xAAAAAAAAu;
    result = _ins(result, insertval, 4, 8);
```

```
    /* result is now 0xAAAAA12A */
}
```

对于该示例，编译器可能会生成类似于以下内容的汇编代码。

```
    movs    r2, #18             @ 0x12
    mov     r3, #-1431655766    @ 0xaaaaaaaa

    bfi     r3,r2,#4,#8         @ inline assembly

    ldr     r2, .L2             @ load result address
    str     r3, [r2]            @ assign the result
    bx      lr                  @ return
    ...
 .L2:
    .word result
```

多条汇编器指令

该示例演示如何使用几条 REV 指令来进行 64 位字节交换。REV 指令交换（反转顺序）32 位字中的字节。这个类似于函数的宏使用行内汇编代码来创建“字节交换双字”，使用了 C/C++代码通常不会生成的指令。但是，使用 GCC 内置函数__builtin_bswap64()也可以获得相同的功能。一般来说，应尽可能优先采用内置函数而不是行内汇编。

以下给出了类似于函数的宏_bswapdw 的定义。

```
/* Thumb2 byte-swap double word */
#define _bswapdw(val) __extension__({            \
  union { uint32_t i[2]; uint64_t l; } __i, __o; \
  __i.l = (val);                                 \
  __asm__ ("rev\t%0, %3\n\t"                     \
           "rev\t%1, %2" /* template */          \
           : "=&r" (__o.i[0]), "=r" (__o.i[1])   \
           : "r" (__i.i[0]), "r" (__i.i[1]));    \
  __o.l; \
})
```

使用联合体引用 64 位整数的两半部分（各 32 位）。例如，输入操作数的 C 表达式分别为'__i.i[0]'和'__i.i[1]'，输出操作数的 C 表达式分别为'__o.i[0]'和'__o.i[1]'。

所有操作数都使用约束“r”（32 位寄存器）。请注意操作数 0 的“&”修饰符，该修饰符指示它是“早期破坏者”（在所有输入操作数被消耗之前写入，这意味着编译器将分配一个与输入寄存器不同的寄存器）。这是必要的，因为两半部分（各 32 位）本身需要进行交换。

以下示例显示了这个类似于函数的宏，它将 value 的内容进行交换之后赋值给 result。

```
uint64_t result;
int example (void)
{
   uint64_t value = 0x0123456789ABCDEFull;
   result = _bswapdw (value);
   /* result == 0xEFCDAB8967452301 */
}
```

对于该示例，编译器可能会生成类似于以下内容的汇编代码：

```
 ldr r2, .L6 @ r2 = 0x01234567
 ldr r3, .L6+4 @ r3 = 0x89ABCDEF

 rev r1, r2 @ from inline asm
 rev r3, r3 @ from inline asm

 ldr r2, .L6+8 @ r2 = address of 'result'
 stm r2, {r1, r3} @ store value to 'result'
 bx lr @ return
 ...
 .align 2
.L6:
```

```
  .word 19088743 @ 0x01234567
  .word -1985229329 @ 0x89ABCDEF
  .word result
```

20.2.2 等效汇编符号

C/C++符号可以不经修改直接在扩展汇编代码中访问。

20.3 预定义的宏

包含宏<xc.h>之后，会有一个预定义的宏可供使用，即_nop()。该宏会插入 nop 指令。

21. 优化

是否激活编译器许可证决定着可以使用哪些代码优化功能。

如果不激活许可证，编译器将在免费模式下工作，此时只能访问基本优化功能。用户可以随时购买 PRO 编译器许可证。激活 PRO 编译器许可证后将解锁针对速度和针对空间的全部可用优化功能。购买许可证之前，用户可根据需要获取并激活免费的 60 天 PRO 许可证进行评估试用，在此期间同样可以对源代码使用全部的优化功能，充分体验 PRO 许可证所带来的好处。

关于 C 和 C++许可证的更多信息，请访问 www.microchip.com/mplab/compilers。

MPLAB XC32 C/C++编译器许可证类型包括免费、EVAL 和 PRO。初次下载该编译器时会提供有效期为 60 天的评估（EVAL）许可证供用户进行评估试用，在此期间等同于激活了专业（PRO）许可证，可以使用最多的优化功能。免费许可证可用的优化功能最少。

不同的 MPLAB XC32 C/C++编译器版本支持不同的优化级别。一些版本可免费下载，而其他一些则必须购买。关于 C 和 C++许可证的更多信息，请访问 www.microchip.com/mplab/compilers。

编译器版本包括：

版本	成本	说明
专业（PRO）	有	实现了最高的优化和性能级别。
免费	无	实现具有最多代码优化限制。
评估（EVAL）	无	允许使用专业版 60 天，之后恢复为免费版。

21.1 优化功能汇总

激活编译器许可证后，可以解锁在免费模式下无法使用的优化功能。下表列出了 XC32 编译器在激活许可证之前（免费）和激活许可证之后（PRO 许可证）可用的优化功能。优化功能的名称取自通常用于使能或禁止它们的选项，例如“Defer pop”优化功能可使用`-fdefer-pop` 选项（如果尚未在所选优化级别使能）手动使能或使用`-fno-defer-pop` 手动禁止。关于用于设置优化的编译器选项的详细信息，请参见用于控制优化的选项。

表 21-1. 许可证优化功能

免费	PRO 许可证
• Align functions • Align labels • Align loops • Caller saves • Cse follow jumps • Cse skip blocks • Data sections • Defer pop • Expensive optimizations • Function cse • Function sections • Gcse • Gsce lm • Gsce sm • Inline • Inline functions • Inline limit • Keep inline functions • Keep static consts • Omit frame pointer • Optimize sibling calls • Peephole/2 • Rename registers • Rerun cse after loop • Rerun loop opt • Schedule insns/2 • Strength reduce • Strict aliasing • Thread jumps • Toplevel reorder • Unroll/all loops	在免费版的全部可用优化功能之外再加上： • Lto • Pa

22. 预处理

在编译之前，需要对所有 C/C++源文件进行预处理。使用.S 扩展名（大写）的汇编源文件也需要进行预处理。有许多选项可以控制预处理器和预处理代码的操作，请参见用于控制预处理器的选项。

22.1 预处理器伪指令

除了标准伪指令之外，XC32 还可接受几条专用的预处理器伪指令。这些伪指令均已在下表中列出。

表 22-1. 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
#define	定义预处理器宏。	#define SIZE (5) #define FLAG #define add(a,b) ((a)+(b))
#elif	#else #if 的简写。	请参见#ifdef
#else	根据条件包含源代码行。	请参见#if
#endif	终止条件包含源代码。	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* filename	（其中 *nn* 为编号，*filename* 为源文件名）其后的内容源自指定的文件和行号。	#20 init.c
#pragma	特定于编译器的选项。	请参见本指南中的“Pragma 伪指令”部分。
#undef	取消定义预处理器符号。	#undef FLAG
#warning	产生一条警告消息。	#warning Length not set

带参数的宏展开可以使用#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自身需要替换的情况，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展开的表达式。请记住，对某个宏标识符进行展开之后，如果它在连接之后出现，将不会再次对它进行展开。

22.1.1 预处理器算术

预处理器宏替换表达式是文本，不使用类型。除非宏包含在包含伪指令（下文讨论）的控制表达式中，否则预处理器不会对宏求值。当宏进行了文本展开且预处理完成后，展开即形成了一个 C 表达式，代码生成器以及其他 C 代码将对其求值。展开 C 表达式中的标记继承了一种类型，其值会进行整型提升和通常方式的类型转换。

如果宏包含在条件包含伪指令（`#if` 或`#elif`）的控制表达式中，则预处理必须对宏求值。求值结果通常不同于相同序列的 C 域的结果。预处理器将长度赋给控制表达式中的字面值（等于编译器可接受的最大整型长度），如`<stdint.h>`中定义的 `intmax_t` 的长度所指定。

对于 MPLAB XC32 C/C++编译器，此长度为 64 位。

22.2 C/C++语言注释

C/C++注释会被编译器忽略，可用于向阅读源代码的人员提供信息。应自由地使用它们。

可以通过将所需字符包含在`/*`和`*/`中来添加注释。注释可以跨越多行，但注释不能进行嵌套。注释可以放置在 C/C++代码中的任意位置，甚至可以放置在表达式中间，但不能放置在字符常量或字符串字面值中。

由于注释不能进行嵌套，可能需要使用`#if` 预处理器伪指令来注释掉已经包含注释的代码，例如：

```
#if 0
result = read(); /* TODO: Jim, check this function is right */
#endif
```

此外，也可以指定 C++样式的单行注释。`//`之后到行尾的所有字符都会被视为注释，将被编译器忽略，如下所示：

```
result = read(); // TODO: Jim, check this function is right
```

22.3 pragma 伪指令

`#pragma` 伪指令可用于修改编译器的行为。pragma 伪指令的一般格式为：

```
#pragma [GCC] keyword options
```

其中，`keyword` 是一组受支持的关键字之一，其中一些后面会跟随几个 `options`。

某些关键字的前面必须有 `GCC`，以指示该关键字为 GCC 扩展。编译器将忽略所有无法理解的关键字。 下面给出了 PIC32C/SAM 器件支持的关键字。

22.3.1 用于控制函数属性的 pragma 伪指令

#pragma long_calls

将 pragma 伪指令后面的所有函数设置为具有 `long_call` 函数属性。

#pragma no_long_calls

将 pragma 伪指令后面的所有函数设置为具有 `short_call` 属性。

#pragma long_calls_off

禁止任何前面出现的 `long_calls` 或 `long_calls_off` pragma 伪指令的作用，从而使其后函数不会隐式地设置任何 `long_call` 或 `short_call` 属性。

22.3.2 用于控制选项/优化的指令

#pragma GCC target ("string" ...)

该 pragma 伪指令可用于为所有后续函数定义设置特定于目标的选项。允许使用的参数为带`-m` 前缀的所有选项，因此`-m` 将被添加到每个给定字符串的前面以形成目标选项，即`#pragma` GCC 目标

("arch=armv7e-m")。该 pragma 伪指令后面的所有函数定义的行为将与对定义应用 ((target("string"))属性时相同。括号为可选项。

#pragma GCC optimize ("string" ...)

#pragma GCC optimize ("*string*"...) pragma 伪指令针对未指定这些属性的函数声明和定义设置默认优化属性。该 pragma 伪指令后面的所有函数都将进行相应的优化。括号为可选项。允许使用的参数可能是：

- 数字 *n*，将被解释为优化级别，即-On 选项
- 以 O 开头的字符串，将被解释为优化选项，即 -O*string*
- 否则，*string* 应为可使用-f 前缀的选项。

#pragma GCC reset_options pragma 伪指令会清除默认优化，因此后续函数的优化不受 optimize pragma 伪指令控制。

#pragma GCC push_options

#pragma GCC pop_options

这两个 pragma 伪指令可用于维护 target 和 optimize 选项的堆栈。push_options pragma 伪指令会将当前选项压入堆栈，如果 target 或 optimize pragma 伪指令均未生效，则将为命令行选项。pop_options 将使最后压入堆栈的选项恢复生效。

#pragma GCC reset_options

针对所有后续函数定义清除通过 target 或 optimize pragma 伪指令设置的任何当前选项。

22.3.3 MPLAB XC32 pragma 伪指令

以下 pragma 伪指令特定于 MPLAB XC32 编译器。

#pragma config identifier = value

config pragma 伪指令可用于为应用程序设置特定于器件的配置位。关于 config 选项的语法说明，请参见配置位访问。

#pragma default_function_attributes

#pragma default_function_attributes = [@ "*section*"] pragma 伪指令针对未指定该属性的函数声明和定义设置默认的段放置属性。该 pragma 伪指令后面的所有函数都将被放入其名称在@标记之后加双引号的段。使用该 pragma 伪指令的#pragma default_function_attributes =形式将取消段指定，以使后续函数不会被放入任何指定段。

#pragma default_variable_attributes

#pragma default_variable_attributes = [@ "*section*"] pragma 伪指令针对未指定该属性的变量声明和定义设置默认的段放置属性。该 pragma 伪指令后面的所有变量都将被放入其名称在@标记之后加双引号的段。使用该 pragma 伪指令的#pragma default_variable_attributes =形式将取消段指定，以使后续变量不会被放入任何指定段。

22.4 预定义的宏

编译器提供了许多宏定义，用于描述各种特定于目标的选项以及编译器和主机环境的其他方面。下表中列出了其中的几个宏定义。

关于其他可以用于确定选定器件上的可用功能的宏，另请参见特定于器件的包含文件（pic32c/include/proc/*family*/*device*.h）。您可以在头文件末尾附近找到这些宏。

此外，还可以通过-E 和-dM 选项运行编译器，让编译器打印出针对特定的项目和一组编译器选项定义的宏。

表 22-2. 预定义宏定义

宏	含义
__PIC32C __PIC32C__	使用-mprocessor 选项指定 PIC32CX 器件时定义。以 PIC32C/SAM MCU 为目标器件时总是定义。
__PIC32CZ	使用-mprocessor 选项指定 PIC32CZ 器件时定义。
__LANGUAGE_ASSEMBLY __LANGUAGE_ASSEMBLY__ _LANGUAGE_ASSEMBLY	编译预处理汇编文件（.S 文件）时定义。
LANGUAGE_ASSEMBLY	编译预处理汇编文件（.S 文件）且未指定-ansi 时定义。
__LANGUAGE_C __LANGUAGE_C__ _LANGUAGE_C	编译 C 文件时定义。
LANGUAGE_C	编译 C 文件且未指定-ansi 时定义。
__LANGUAGE_C_PLUS_PLUS __cplusplus _LANGUAGE_C_PLUS_PLUS__	编译 C++文件时定义。
__EXCEPTIONS	使能 C++异常时定义。
__GXX_RTTI	使能运行时类型信息时定义。
__*PROCESSOR*__	其中 *PROCESSOR* 是传递给-mprocessor 选项的大写参数。例如，使用-mprocessor=32CX0525SG12144 时，编译器将定义 __32CX05255SG12144__。
__XC	总是定义，用以指示它是 Microchip XC 编译器。
__XC32	总是定义，用以指示它是 XC32 编译器。
__VERSION__	__VERSION__宏会扩展为描述所使用的编译器的字符串常量。不要依赖它的内容会具有任何特定形式，但它至少应包含版本号。使用__XC32_VERSION 宏来获取数字版本号。
__XC32_VERSION 或 __C32_VERSION__	C 编译器会定义常量__XC32_VERSION，为版本标识符提供一个数值。该宏可以用于构造利用新的编译器功能的应用程序，同时仍然与较早版本保持向后兼容。该值基于当前发行版的主要和次要版本号。例如，发行版 1.03 的 __XC32_VERSION 定义为 1030。该宏可以与标准预处理器比较语句配合使用，根据条件来包含/排除各种代码构造。
__arm__	针对 Arm 架构进行编译时定义，无论是生成 Thumb 代码还是 Arm 代码。
__thumb__	需要指示编译器生成 Thumb 代码时定义。该定义需要使用-mthumb 和-marm 选项。
__thumb2__	针对支持 Thumb-2 指令集的目标处理器生成 Thumb 代码时定义。
__SOFTFP__	针对软件浮点（即，当-mfloat-abi=soft 生效时）进行编译时定义。
__ARM_FP	定义为描述当前目标处理器浮点能力的整数掩码。当软件浮点生效时，该值为 0。否则，通过将掩码的 bit 1、2 和 3 置 1 来分别指示支持 16、32 和 64 位硬件浮点。
__XC32_DTCM_LENGTH	定义为由-mdtcm 选项指定的数据紧耦合存储器的大小（字节）。
__XC32_ITCM_LENGTH	定义为由-mitcm 选项指定的指令紧耦合存储器的大小（字节）。
__XC32_TCM_LENGTH	定义为由-mtcm 选项指定的组合紧耦合存储器的大小（字节）。

23. 链接程序

关于链接描述文件的更多详细信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）。

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。

链接描述文件用于指定可用的存储器区域，以及段应定位在这些区域中的什么位置。

链接器会创建一个映射文件，详细说明分配给段的存储器。映射文件是查找存储器信息的最佳位置。

23.1 替换库符号

不同于 Microchip MPLAB XC8 编译器，使用 MPLAB XC32 C/C++编译器时，并不是所有库函数都可以替换为用户定义的程序。只有弱版本的库函数（见变量属性）可以通过这种方式替换。对于那些弱版本的函数，您在代码中编写的任意函数会替换库文件中的同名函数。

23.2 链接器定义的符号

32 位链接器定义了几个可以在 C 代码开发中使用的符号。更多信息，请参见《MPLAB® XC32 汇编器、链接器和实用程序用户指南》（DS50002186A_CN）。

链接器还定义了符号`_stack`，由运行时启动代码用于初始化堆栈指针。该符号代表软件堆栈的起始地址。

大多数程序很少需要上述所有符号，但如果您要编写自己的运行时启动代码，则可能会对您有所帮助。

24. 实现定义的行为

24.1 概述

ISO C 要求符合标准的实现用文档说明对于标准中定义为“实现定义的行为”所做的选择。以下几节列出了所有这些方面、对于编译器所做的选择，以及 ISO/IEC 9899:1999 标准中的相应章节编号。

24.2 翻译

ISO 标准：	“如何标识一条诊断消息（3.10 和 5.1.1.3）。”
实现：	`stderr` 的所有输出都是诊断消息。
ISO 标准：	“在翻译阶段 3 中，空白字符（换行符除外）组成的每个非空序列是保留还是替换为一个空格字符（5.1.1.2）。”
实现：	每个空白字符序列替换为一个字符。

24.3 环境

ISO 标准：	“在独立式环境中，程序启动时调用的函数的名称和类型（5.1.2.1）。”
实现：	`int main (void);`
ISO 标准：	“在独立式环境中，终止程序的影响（5.1.2.1）。”
实现：	执行一条无限循环（转移到自身）指令。
ISO 标准：	“main 函数的备用定义方式（5.1.2.2.1）。”
实现：	`int main (void);`
ISO 标准：	“赋给 main 的 `argv` 参数所指向的字符串的值（5.1.2.2.1）。”
实现：	不向 `main` 传递任何参数。对 `argc` 或 `argv` 的引用是未定义的。
ISO 标准：	“交互式设备的要素（5.1.2.3）。”
实现：	由应用程序定义。
ISO 标准：	“在程序启动时执行 `signal(sig, SIG_IGN);` 的等效函数所针对的信号（7.14.1.1）。”
实现：	信号由应用程序定义。
ISO 标准：	“当抛出 `SIGABRT` 信号但未被捕捉时，返回给主机环境以指示终止未成功的状态的形式（7.20.4.1）。”
实现：	主机环境由应用程序定义。
ISO 标准：	“`exit` 函数返回给主机环境，报告终止成功和不成功的状态的形式（7.20.4.3）。”
实现：	主机环境由应用程序定义。
ISO 标准：	“`exit` 函数的参数的值不为 0、`EXIT_SUCCESS` 或 `EXIT_FAILURE` 时，它返回给主机环境的状态（7.20.4.3）。”
实现：	主机环境由应用程序定义。
ISO 标准：	“环境名称集合以及用于更改 `getenv` 函数所使用的环境列表的方法（7.20.4.4）。”
实现：	主机环境由应用程序定义。
ISO 标准：	“system 函数执行字符串的方式（7.20.4.5）。”
实现：	主机环境由应用程序定义。

24.4 标识符

ISO 标准：	“可能出现在标识符中的附加多字节字符以及它们与通用字符名的对应关系（6.4.2）。”
实现：	无。
ISO 标准：	“标识符中有效初始字符的数量（5.2.4.1 和 6.4.2）。”
实现：	所有字符均有效。

24.5 字符

ISO 标准：	“字节中的位数（C90 3.4 和 C99 3.6）。”
实现：	8。
ISO 标准：	“执行字符集成员的值（C90 和 C99 5.2.1）。”
ISO 标准：	“为每个标准字母转义序列生成的执行字符集成员的惟一值（C90 和 C99 5.2.2）。”
实现：	执行字符集为 ASCII。
ISO 标准：	“存储了除基本执行字符集成员之外的任意其他字符的 char 对象的值（C90 6.1.2.5 和 C99 6.2.5)。”
实现：	char 对象的值是字符在源字符集中的 8 位二进制表示。即，不进行任何转换。
ISO 标准：	“signed char 和 unsigned char 中，哪一个与“普通”char 具有相同的范围、表示和行为（C90 6.1.2.5、C90 6.2.1.1、C99 6.2.5 和 C99 6.3.1.1）。”
实现：	在 PIC32C 上，默认情况下，unsigned char 在功能上等效于普通 char。
ISO 标准：	“源字符集成员（字符常量和字符串字面值）到执行字符集成员的映射（C90 6.1.3.4、C99 6.4.4.4、C90 和 C99 5.1.1.2）。”
实现：	源字符集的二进制表示形式保存到执行字符集。
ISO 标准：	“包含多个字符，或包含未映射到单字节执行字符的字符或转义序列的整型字符常量的值（C90 6.1.3.4 和 C99 6.4.4.4）。”
实现：	编译器以每次一个字符的方式确定多字符的字符常量的值。前一个值被左移 8 位，然后下一个字符的位组合以掩码方式移入。最终的结果为 `int` 类型。如果结果大于 `int` 的表示范围，那么会发出一个警告诊断消息，并且值被截断为 `int` 的长度。
ISO 标准：	“包含多个多字节字符，或包含不以扩展执行字符集表示的多字节字符或转义序列的宽字符常量的值（C90 6.1.3.4 和 C99 6.4.4.4）。”
实现：	参见上文。
ISO 标准：	“用于将包含单个多字节字符（该字符映射到扩展执行字符集的一个成员）的宽字符常量转换为相应的宽字符代码的当前语言环境（C90 6.1.3.4 和 C99 6.4.4.4）。”
实现：	LC_ALL
ISO 标准：	“用于将宽字符串字面值转换为相应的宽字符代码的当前语言环境（C90 6.1.4 和 C99 6.4.5）。”
实现：	LC_ALL
ISO 标准：	“包含不以执行字符集表示的多字节字符或转义序列的字符串字面值的值（C90 6.1.4 和 C99 6.4.5）。”
实现：	从源字符集保存字符的二进制表示。

24.6 整型

ISO 标准：	“实现中存在的所有扩展整型（C99 6.2.5）。”
实现：	不存在任何扩展整型。
ISO 标准：	“有符号整型是使用符号加数值、二进制补码还是二进制反码表示，以及异常值是陷阱表示还是普通值（C99 6.2.6.2）。”
实现：	所有整型都以二进制补码表示，所有位组合都是普通值。
ISO 标准：	“任意扩展整型相对于另一个具有相同精度的扩展整型的级别（C99 6.3.1.1）。”
实现：	不支持任何扩展整型。
ISO 标准：	“将某个整型转换为有符号整型，当前者的值无法由属于后者类型的目标表示时，所产生的结果或所抛出的信号（C90 6.2.1.2 和 C99 6.3.1.3）。”
实现：	在将值 X 转换为宽度为 N 的类型时，结果的值是 X 的二进制补码的低 N 位。也即，X 截断为 N 位。不抛出任何信号。
ISO 标准：	“对有符号整型的一些位操作的结果（C90 6.3 和 C99 6.5）。”
实现：	对有符号值进行位操作时，会对该值的二进制补码表示（包括符号位）进行操作。有符号值右移表达式的结果进行符号扩展。 C99 允许不定义有符号数“<<”操作的一些方面。该编译器不这样做。

24.7 浮点型

ISO 标准：	“浮点运算的精度，以及<math.h>和<complex.h>中返回浮点型结果的库函数的精度（C90 和 C99 5.2.4.2.2）。”
实现：	精度未知。
ISO 标准：	“<stdio.h>、<stdlib.h>和<wchar.h>中的库函数执行的浮点型内部表示和字符串表示之间的转换的精度（C90 和 C99 5.2.4.2.2）。”
实现：	精度未知。
ISO 标准：	“FLT_ROUNDS 的非标准值所表示的舍入行为（C90 和 C99 5.2.4.2.2）。”
实现：	不使用此类值。
ISO 标准：	“FLT_EVAL_METHOD 的非标准负值所表示的求值方法（C90 和 C99 5.2.4.2.2）。”
实现：	不使用此类值。
ISO 标准：	“当整数转换为无法精确表示原始值的浮点数时的舍入方向（C90 6.2.1.3 和 C99 6.3.1.4）。”
实现：	遵循 C99 附件 F。
ISO 标准：	“当浮点数转换为长度较短的浮点数时的舍入方向（C90 6.2.1.4 和 6.3.1.5）。”
实现：	遵循 C99 附件 F。
ISO 标准：	“如何为某些浮点常量选择最接近的可表示值，或者与最接近的可表示值接近的较大或较小可表示值（C90 6.1.3.1 和 C99 6.4.4.2）。”
实现：	遵循 C99 附件 F。
ISO 标准：	“当 FP_CONTRACT pragma 伪指令未禁止浮点表达式时，是否以及如何化简浮点表达式（C99 6.5）。”
实现：	未实现该 pragma 伪指令。
ISO 标准：	“FENV_ACCESS pragma 伪指令的默认状态（C99 7.6.1）。”
实现：	未实现该 pragma 伪指令。
ISO 标准：	“其他浮点异常、舍入模式、环境、分类及其宏名称（C99 7.6 和 7.12）。”
实现：	均不支持。
ISO 标准：	“FP_CONTRACT pragma 伪指令的默认状态（C99 7.12.2）。”
实现：	未实现该 pragma 伪指令。
ISO 标准：	“当实际舍入的结果与符合 IEC 60559 要求的实现中的数学结果相等时，是否抛出‘不精确’浮点异常（C99 F.9）。”
实现：	未知
ISO 标准：	“当结果很小，但在符合 IEC 60559 要求的实现中足够精确时，是否抛出‘下溢’（及‘不精确’）浮点异常（C99 F.9）。”
实现：	未知

24.8 数组和指针

ISO 标准：	“将指针转换为整型（或反之）的结果（C90 6.3.4 和 C99 6.3.2.3）。”
实现：	将整型强制转换为指针（或反之）的结果使用源类型的二进制表示，并按适合于目标类型的方式重新解释。 如果源类型的长度大于目标类型，则丢弃高位。当从指针强制转换为整型时，如果源类型的长度小于目标类型，那么将对结果进行符号扩展。当从整型强制转换为指针时，如果源类型的长度小于目标类型，那么结果将根据源类型是否有符号来进行扩展。
ISO 标准：	“将两个指向同一数组的两个元素的指针进行相减的结果的长度（C90 6.3.6 和 C99 6.5.6）。”
实现：	32 位有符号整型。

24.9 提示

ISO 标准：	“使用寄存器存储类说明符的建议的有效程度（C90 6.5.1 和 C99 6.7.1）。”
实现：	寄存器存储类说明符通常没有任何作用。
ISO 标准：	“使用内联函数说明符的建议的有效程度（C99 6.7.4）。”

实现：	如果指定了-fno-inline 或-O0，则即使使用 inlined 说明符进行指定，也不会对函数进行内联。否则，可能会也可能不会内联该函数，这取决于编译器的优化推断。

24.10 结构、联合、枚举和位域

ISO 标准：	“联合对象的成员使用不同类型的成员访问（C90 6.3.2.3）。”
实现：	联合对象的相应字节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。
ISO 标准：	“‘普通’ int 位域是作为 signed int 位域还是作为 unsigned int 位域处理（C90 6.5.2、C90 6.5.2.1、C99 6.7.2 和 C99 6.7.2.1）。”
实现：	在 PIC32C 上，默认情况下，普通 int 位域作为无符号整型处理。请注意，这一点与 PIC32M 不同。可通过-funsigned-bitfields 和-fsigned-bitfields 编译器标志显式设置默认行为。
ISO 标准：	“除_Bool、signed int 和 unsigned int 之外允许的位域类型（C99 6.7.2.1）”。
实现：	不支持任何其他类型。
ISO 标准：	“位域是否可以跨越存储单元边界（C90 6.5.2.1 和 C99 6.7.2.1）。”
实现：	否。
ISO 标准：	“位域在单元内的分配顺序（C90 6.5.2.1 和 C99 6.7.2.1）。”
实现：	位域从左到右分配。
ISO 标准：	“结构体的非位域成员的对齐（C90 6.5.2.1 和 C99 6.7.2.1）。”
实现：	每个成员根据成员类型的对齐限制，定位到允许的距当前位置偏移量最小的位置。
ISO 标准：	“与每种枚举类型兼容的整型（C90 6.5.2.2 和 C99 6.7.2.2）。”
实现：	如果枚举值全部非负，则类型为 unsigned int，否则为 int。-fshort-enums 命令行选项可以更改该行为。

24.11 限定符

ISO 标准：	“构成访问具有 volatile 限定类型的对象的要素（C90 6.5.3 和 C99 6.7.3）。”

实现：	使用 volatile 对象的值或向 volatile 对象存储值的任意表达式均视为对该对象进行访问。不能保证此类访问是原子的。
	如果一个表达式中含有对 volatile 对象的引用，但既不使用也不存储对象的值，那么表达式是否视为对 volatile 对象的访问取决于对象的类型。如果对象属于标量类型、带有单个标量类型成员的聚合类型，或者含有（仅）标量类型成员的联合，那么将表达式视为对 volatile 对象的访问。否则，将对该表达式进行求值来产生它的副作用，但不将其视为对 volatile 对象的访问。
	例如：
	`volatile int a;`
	`a; /* access to ‘a’ since ‘a’ is scalar */`

24.12 声明符

ISO 标准：	“可修改算术、结构体或联合体类型的声明符的最大数量（C90 6.5.4）。”
实现：	无限制。

24.13 语句

ISO 标准：	“switch 语句中的 case 值的最大数量（C90 6.6.4.2）。”
实现：	无限制。

24.14 预处理伪指令

ISO 标准：	“如何将两种头文件名称形式的序列映射到头文件或外部源文件名称（C90 6.1.7 和 C99 6.4.7）。”
实现：	定界符之间的字符序列视为一个字符串，作为主机环境的文件名。
ISO 标准：	“控制条件包含的常量表达式中的字符常量的值是否与执行字符集中的相同字符常量的值匹配（C90 6.8.1 和 C99 6.10.1）。”
实现：	是。

ISO 标准：	“控制条件包含的常量表达式中的单字符 character 常量的值是否可以具有负值（C90 6.8.1 和 C99 6.10.1）。”
实现：	是。
ISO 标准：	“所包含的以< >定界的头文件的搜索位置，以及如何指定这些位置或如何标识头文件（C90 6.8.2 和 C99 6.10.2）。”
实现：	<install directory>/lib/gcc/pic32c/6.2.1/include
	<install directory>/pic32c/include
ISO 标准：	“如何在所包含的以“”定界的头文件中搜索指定的源文件（C90 6.8.2 和 C99 6.10.2）。”
实现：	编译器首先在含有包含文件的目录中、由-iquote 命令行选项指定的目录（如果有）中搜索指定文件，然后在以< >定界的头文件的搜索目录中进行搜索。
ISO 标准：	“将预处理标记合并到头文件名称中的方法（C90 6.8.2 和 C99 6.10.2）。”
实现：	所有标记（包括空格）都视为头文件名称的一部分。对于定界符内部的标记，将不执行宏展开。
ISO 标准：	“#include 处理的嵌套限制（C90 6.8.2 和 C99 6.10.2）。”
实现：	无限制。
ISO 标准：	“每个识别到的非 STDC #pragma directive 伪指令的行为（C90 6.8.6 和 C99 6.10.6）。”
实现：	请参见变量属性。
ISO 标准：	“当转换的日期和时间分别不可用时，__DATE__和__TIME__的定义（C90 6.8.8 和 C99 6.10.8）。”
实现：	转换的日期和时间总是可用。

24.15 库函数

ISO 标准：	“宏 NULL 展开成的空指针常量（C90 7.1.6 和 C99 7.17）。”
实现：	(void *)0
ISO 标准：	“独立式程序可用的任何库工具，条款 4 所要求的最小工具集除外（5.1.2.1）。”
实现：	请参见 *32-Bit Language Tools Libraries*（DS51685）。
ISO 标准：	“assert 宏打印的诊断消息的格式（7.2.1.1）。”
实现：	“Failed assertion ‘*message*’ at line *line* of ‘*filename*’.\n”
ISO 标准：	“FENV_ACCESS pragma 伪指令的默认状态（7.6.1）。”
实现：	未实现。
ISO 标准：	“由 fegetexceptflag 函数存储的浮点异常标志的表示（7.6.2.2）。”
实现：	未实现。
ISO 标准：	“除了上溢或下溢异常之外，feraiseexcept 函数是否抛出不精确异常（7.6.2.3）。”
实现：	未实现。
ISO 标准：	“除 FE_DFL_ENV 之外，可以用作 fesetenv 或 feupdateenv 函数参数的浮点环境宏（7.6.4.3 和 7.6.4.4）。”
实现：	未实现。
ISO 标准：	“""C""和""""除外，可作为第二个参数传递给 setlocale 函数的字符串（7.11.1.1）。”
实现：	无。
ISO 标准：	“当 FLT_EVAL_METHOD 宏的值小于 0 或大于 2 时，为 float_t 和 double_t 定义的类型（7.12）。”
实现：	未实现。
ISO 标准：	“INFINITY 宏展开到的最大极限（如果有）（7.12）。”
实现：	未实现。
ISO 标准：	“NAN 宏展开到的静默 NaN（如果定义）（7.12）。”
实现：	未实现。
ISO 标准：	“除了本国际标准所要求的之外，数学函数的域错误（7.12.1）。”
实现：	无。
ISO 标准：	“发生域错误时数学函数返回的值，以及是否将 errno 设置为宏 EDOM 的值（7.12.1）。”

实现：	发生域错误时，errno 设置为 EDOM。
ISO 标准：	“发生上溢和/或下溢范围错误时，数学函数是否将 errno 设置为宏 ERANGE 的值（7.12.1）。”
实现：	是。
ISO 标准：	“FP_CONTRACT pragma 伪指令的默认状态（7.12.2）。”
实现：	未实现。
ISO 标准：	“当 fmod 函数的第二个参数为 0 时，是发生域错误还是返回 0（7.12.10.1）。”
实现：	返回 NaN。
ISO 标准：	“在减小商时，remquo 函数使用的模数以 2 为底的对数（7.12.10.3）。”
实现：	未实现。
ISO 标准：	“信号的集合，它们的语义，以及它们的默认处理（7.14）。”
实现：	信号的默认处理是总是返回失败。实际的信号处理由应用程序定义。
ISO 标准：	“在调用信号处理程序之前未执行 signal(sig, SIG_DFL);的等效函数的情况下，所执行的信号阻塞（7.14.1.1）。”
实现：	由应用程序定义。
ISO 标准：	“在调用信号 SIGILL 的信号处理程序之前，是否执行 signal(sig, SIG_DFL)的等效函数（7.14.1.1）。”
实现：	由应用程序定义。
ISO 标准：	“除 SIGFPE、SIGILL 和 SIGSEGV 之外，对应于计算异常的信号值（7.14.1.1）。”
实现：	由应用程序定义。
ISO 标准：	“文本流的最后一行是否需要一个终止换行符（7.19.2）。”
实现：	是。
ISO 标准：	“写到文本流中的紧接在换行符前面的空格字符在读入时是否出现（7.19.2）。”
实现：	是。
ISO 标准：	“可附加至写入二进制流的数据的空字符数（7.19.2）。”
实现：	不向二进制流附加任何空字符。
ISO 标准：	“附加模式流的文件位置说明符最初位于文件开始还是末尾（7.19.3）。”
实现：	由应用程序定义。系统级函数 open 使用 O_APPEND 标志进行调用。
ISO 标准：	“对文本流的写操作是否导致关联的文件在该点之后被截断（7.19.3）。”
实现：	由应用程序定义。
ISO 标准：	“文件缓冲的特征（7.19.3）。”
ISO 标准：	“零长度文件是否确实存在（7.19.3）。”
实现：	由应用程序定义。
ISO 标准：	“构造有效文件名的规则（7.19.3）。”
实现：	由应用程序定义。
ISO 标准：	“同一文件是否可以同时打开多次（7.19.3）。”
实现：	由应用程序定义。
ISO 标准：	“文件中多字节字符所用编码的性质和选择（7.19.3）。”
实现：	对于每个文件，编码是相同的。
ISO 标准：	“remove 函数对于已打开文件的作用（7.19.4.1）。”
实现：	由应用程序定义。系统函数 unlink 会被调用。
ISO 标准：	“如果在调用 rename 函数之前，已存在具有新名称的文件，会有什么影响（7.19.4.2）。”
实现：	由应用程序定义。系统函数 link 将会被调用以创建新文件名，然后 unlink 被调用以删除旧文件名。通常，如果新文件名已存在，link 将失败。
ISO 标准：	“在程序异常终止时，是否删除打开的临时文件（7.19.4.3）。”
实现：	否。
ISO 标准：	“当 tmpnam 函数的调用次数超过 TMP_MAX 次时，会发生什么情况（7.19.4.4）。”

实现：	将重新使用临时名称。
ISO 标准：	“允许哪些模式更改（如果有），以及在哪些情况下允许更改（7.19.5.4）。”
实现：	文件通过系统级 close 函数关闭，然后使用 open 函数以新模式重新打开。除了应用程序定义的那些限制之外，对于 open 和 close 函数没有任何其他限制。
ISO 标准：	“用于打印无穷大或 NaN 的样式，以及采用 *n-char-sequence* 打印 NaN 时，该样式的含义（7.19.6.1 和 7.24.2.1）。”
实现：	不打印任何字符序列。 NaN 打印为“NaN”。 无穷大打印为“[-/+]Inf”。
ISO 标准：	“fprintf 或 fwprintf 函数中%p 转换的输出（7.19.6.1 和 7.24.2.1）。”
实现：	功能上等效于%x。
ISO 标准：	“在 fscanf 或 fwscanf 函数中%[转换的扫描列表中，若-字符既不是第一个字符也不是最后一个字符，也不是^字符作为第一个字符时的第二个字符，此时对-字符的解释（7.24.2.1 和 7.19.6.2）。”
实现：	未知
ISO 标准：	“fscanf 或 fwscanf 函数中%p 转换所匹配的序列的集合（7.19.6.2 和 7.24.2.2）。”
实现：	与%x 所匹配的序列集合相同。
ISO 标准：	“fscanf 或 fwscanf 函数中与%p 转换对应的输入项的解释（7.19.6.2 和 7.24.2.2）。”
实现：	如果结果是无效指针，那么行为是未定义的。
ISO 标准：	“fgetpos、fsetpos 或 ftell 函数在失败时对宏 errno 设置的值（7.19.9.1、7.19.9.3 和 7.19.9.4）。”
实现：	如果结果超过 LONG_MAX，那么 errno 设置为 ERANGE。 其他错误由应用程序定义，依据是其对于 lseek 函数的定义。
ISO 标准：	“在由 strtod、strtof、strtold、wcstod、wcstof 或 wcstold 函数转换的字符串中，*n-char-sequence* 的含义（7.20.1.3 和 7.24.4.1.1）。”
实现：	该序列未附加任何含义。
ISO 标准：	“发生下溢时，strtod、strtof、strtold、wcstod、wcstof 或 wcstold 函数是否将 errno 设置为 ERANGE（7.20.1.3 和 7.24.4.1.1）。”
实现：	是。
ISO 标准：	“当所请求的长度为 0 时，calloc、malloc 和 realloc 函数是返回空指针还是返回指向已分配对象的指针（7.20.3）。”
实现：	返回指向静态分配的对象的指针。
ISO 标准：	“调用 abort 函数时，是否清空打开的输出流、关闭打开的流或删除临时文件（7.20.4.1）。”
实现：	否。
ISO 标准：	“abort 函数返回给主机环境的终止状态（7.20.4.1）。”
实现：	默认情况下，不存在主机环境。
ISO 标准：	“system 函数在其参数不是空指针时返回的值（7.20.4.5）。”
实现：	由应用程序定义。
ISO 标准：	“本地时区和夏令时（7.23.1）。”
实现：	由应用程序定义。
ISO 标准：	“clock 函数的年代（7.23.2.1）。”
实现：	由应用程序定义。
ISO 标准：	“在标准化的 tmx 结构中，tm_isdst 的正值（7.23.2.6）。”
实现：	1。
ISO 标准：	“在“C”语言环境中，strftime、strfxtime、wcsftime 和 wcsfxtime 函数的%Z 说明符的替换字符串（7.23.3.5、7.23.3.6、7.24.5.1 和 7.24.5.2）。”
实现：	未实现。
ISO 标准：	“在符合 IEC 60559 要求的实现中，三角函数、双曲函数、以 *e* 为底的指数函数、以 *e* 为底的对数函数、错误函数和对数 gamma 函数是否抛出不精确异常，以及何时抛出（F.9）。”

实现：	否。
ISO 标准：	“当实际舍入的结果与符合 IEC 60559 要求的实现中的数学结果相等时，是否抛出不精确异常（F.9）。”
实现：	否。
ISO 标准：	“当结果很小，但在符合 IEC 60559 要求的实现中足够精确时，是否抛出下溢（及不精确）异常（F.9）。”
实现：	否。
ISO 标准：	“函数是否遵从舍入方向模式（F.9）。”
实现：	并不强制要求遵从舍入模式。

24.16 架构

ISO 标准：	“为在头文件<float.h>、<limits.h>和<stdint.h>中指定的宏分配的值或表达式（C90、C99 5.2.4.2、C99 7.18.2 和 7.18.3）。”
实现：	请参见 limits.h。
ISO 标准：	“任意对象中的字节的数量、顺序和编码（当未在标准中明确指定时）（C99 6.2.6.1）。”
实现：	小尾数法形式，从最低字节开始填充。请参见数据表示。
ISO 标准：	“size of 操作符的结果值（C90 6.3.3.4 和 C99 6.5.3.4）。”
实现：	请参见数据表示。

25. C++实现定义的行为

ISO C++标准要求符合标准的实现用文档说明对于标准中定义为“实现定义的行为”所做的选择。下表列出了所有这些方面、对于编译器所做的选择，以及 ISO/IEC 14882:2014 C++标准中的相应章节编号。

表 25-1. 实现定义的行为

ISO 标准	实现
编译器的输出消息中哪些是诊断消息（1.3.6）	
	`stderr` 的所有输出都是诊断消息。
独立实现所需的库（1.4）	
	请参见*独立实现的头文件集（17.6.1.3）*。
1 个字节包含的位数（1.7）	
	8 位
交互式设备的要素（1.9）	
	如果 `isatty()` 为 true，则暗示支持交互式流。
独立实现下程序的线程数（1.10）	
	不直接支持线程。
物理源文件字符如何映射到基本源字符集（2.2）	
	语言环境设置确定默认源字符集，如果语言环境设置无法确定则为 UTF-8。基本源字符集为 UTF-8。
物理源文件字符（2.2）	
	默认为设置语言环境，如果不起作用则默认为 UTF-8。可以是系统的 `iconv()` 库程序支持的任何编码。
针对未映射到等效执行字符的源字符选择的执行字符成员。（2.2）	
	由 ABI 定义
是否必须提供翻译单元的源以定位模板定义（2.2）	
	必须提供源。
执行字符集和执行宽字符集（2.3）	
	执行字符集包含`[a-zA-Z0-9_{}[]#()<>%:;.?*+-/^&|~!=,\"']`、空格，以及代表水平制表符、垂直制表符、换页符、换行符、警告符、退格符、回车符和空字符的控制字符。执行宽字符集与执行字符集基本相同，只是空字符替换为空宽字符。编码可以是系统的 `iconv()` 库程序支持的任何编码，默认为 UTF-8（对于宽字符集，默认为 UTF-32）。
头文件或外部源文件的映射头文件名称（2.9）	
	与头文件同名（采用 UTF-8 编码）。
不在执行字符集中的通用字符名称的编码（2.14.3）	
	`\U`*NNNNNNNN* 或`\u`*NNNN*（其中 *N* 为四位十六进制数），分别对应于 ISO/IEC 10646 中的字符短名称 *NNNNNNNN* 或 `0000`*NNNN*。
非标准转义序列的语义（2.14.3）	
	作为扩展，GCC 支持`'\e'`序列作为 ASCII `ESC` 字符。
超出相应类型的范围的字符字面值（2.14.3）	
	构成字面值的一系列字节的最后一个字节。
包含单个不在执行宽字符集中的 *c-char* 的宽字符字面值（2.14.3）	
	构成字面值的一系列字节的最后四个字节。
当浮点字面值不在其类型可表示的值范围内（较大或较小）时如何选择其值（2.14.4）	
	遵循 C99 附件 F。
除标准完全指定的字符串字面值之外的字符串字面值的连接（2.14.5）	
	不支持额外的连接。
是否所有字符串字面值均不同（即，存储在不重叠的对象中）（2.14.5）	
	字符串字面值并非一定不同。

.....（续）

ISO 标准	实现
main() 的链接（3.6.1）	
	外部链接。
main 函数的参数（3.6.1）	

	`int main(void)`
	或
	`int main(int argc, char** argv)`

独立环境中的启动和终止的要素（3.6.1）	
	由特定于器件的启动代码确定。DFP 中随附的默认代码在 main() 之前针对具有静态存储持续时间的命名空间作用域的对象调用构造函数，但从 main() 返回之后不调用解构函数。
是否需要在独立环境中定义 main（3.6.1）	
	需要——可以定制特定于器件的启动代码以取消调用 main()，但有些功能（如堆栈使用量报告）需要调用 main()。
具有静态或线程存储持续时间的非局部变量的动态初始化是否在线程初始函数的第一条语句之前完成（3.6.2）	
	不直接支持线程。
具有静态存储持续时间的非局部变量的动态初始化是否在 main() 的第一条语句之前完成（3.6.2）	
	由特定于器件的启动代码确定，但 DFP 中提供的默认启动代码确实会调用构造函数。
实现具有放宽还是严格的指针安全性（3.7.4.3）	
	放宽的指针安全性。
扩展有符号整型（3.9.1）	
	无
char 的表示（3.9.1）	
	1 个字节。
普通 char 的符号（3.9.1）	
	普通 char 无符号。
浮点型的值表示（3.9.1）	
	IEEE-754
指针型的值表示（3.9.2）	
	32 位。
出于对齐目的可在两个连续地址之间为给定对象分配的字节数（3.11）	
	有效对齐为 2 的幂，最高为 2^{28}（含）。
是否支持任何扩展对象对齐以及支持它们的上下文（3.11）	
	最高支持 2^{28} 的扩展对齐，具体取决于器件存储器布局。
无符号至有符号转换结果的值（4.7）	
	用于转换至宽度为 *N* 的类型，值以 2^N 为模减小，从而确保处于该类型的范围内。
不精确浮点型转换的结果（4.8）	
	软件仿真使用舍入到最近值模式，但使用硬件 FPU 时可以更改模式。
不精确整型至浮点型转换结果的值（4.9）	
	遵循 C99 附件 F。
扩展有符号整型的级别（4.13）	
	不支持扩展有符号整型。
通过省略号传递类类型的参数（5.2.2）	
	支持
公开派生自可通过 typeid 返回的 std::type_info 的类（5.2.8）	
	未定义额外的类。结果为 std::type_info。
指针与整型之间的转换（5.2.10）	

.....（续）

ISO 标准	实现
	将整型强制转换为指针（或反之）使用源类型的二进制表示，并按适合于目标类型的方式重新解释。 如果源类型的长度大于目标类型，则丢弃高位。当从指针强制转换为整型时，如果源类型的长度小于目标类型，那么将对结果进行符号扩展。当从整型强制转换为指针时，如果源类型的长度小于目标类型，那么结果将根据源类型是否有符号来进行扩展。
将函数指针转换为对象指针类型（或反之）的含义（5.2.10）	
	支持该转换。保留二进制表示。
将 `sizeof` 应用于除 `char`、`signed char` 和 `unsigned char` 之外的基本类型时的结果（5.3.3）	
	请参见整型数据类型，注意 `sizeof()` 返回的大小以字节为单位。
是否支持过度对齐类型（5.3.4、20.7.9.1 和 20.7.11）	
	支持。
`ptrdiff_t` 的类型（5.7 和 18.2）	
	`int`
负值右移的结果（5.8）	
	按位操作符作用于值的表示，包括符号位和值位，其中符号位被视为紧挨着最高值位的上一位。
`attribute` 声明的含义（7）	
	请参见变量属性和函数属性。
枚举的底层类型（7.2）	
	如果枚举值全部非负，则类型为 `unsigned int`，否则为 `int`。
`asm` 声明的含义（7.4）	
	请参见使用行内汇编语言。
链接说明符的语义（7.5）	
	仅支持“C”和“C++”链接。
非标准属性的行为（7.6.1）	
	请参见变量属性和函数属性。
属性范围标记的行为（7.6.1）	
	属性在“gnu”命名空间中可用，相当于`__attribute__`。
由`__func__`产生的字符串（8.4.1）	
	被视为常量表达式，可用于 *constexpr* 上下文中。该字符串仅包含函数名称。
类对象内的位域分配（9.6）	
	位域从左到右分配。
类对象内的位域对齐（9.6）	
	每个成员根据成员类型的对齐限制，定位到允许的距当前位置偏移量最小的位置。
模板上的链接规范的语义（14）	
	仅支持`"C"`和`"C++"`链接。
在调用 `std::terminate()` 之前堆栈是否展开（15.3 和 15.5.1）	
	堆栈未展开。
违反 `std::terminate()` 规范时，在调用 `noexcept` 之前堆栈是否展开（15.5.1）	
	堆栈未展开。
`#if` 伪指令中的字符字面值的数值（16.1）	
	将其视为在目标上进行解释。
单字符字面值是否可为负值（16.1）	
	否
搜索包含的源文件的方式（16.2）	
	根据定义的搜索路径列表按顺序搜索。在出现第一个匹配项时即结束搜索。
`#include` 伪指令的嵌套限制（16.2）	
	XC32 要求不得超过 200 级嵌套，以避免失控递归。

.....（续）

ISO 标准	实现
使用""包含的头文件的搜索位置（16.2）	
	#include "*file*"伪指令用于您的项目定义的头文件。XC32 首先在包含当前源文件的目录中搜索该文件，然后在带引号的目录中搜索，最后在系统目录中搜索。可使用命令行选项在前面添加目录。
使用<>包含的头文件的搜索位置（16.2）	
	#include <*file*>伪指令用于通过 XC32 打包的系统头文件。可使用命令行选项在前面添加搜索目录。按以下顺序搜索各个目录： pic32c/include/CMSIS/Core/Include pic32c/include_mcc pic32c/include/c++/<gcc-ver> pic32c/include/c++/<gcc-ver>/pic32c/<isa-arch-ver>/nofp pic32c/include/c++/<gcc-ver>/backward pic32c/include pic32c/include/musl lib/gcc/pic32c/<gcc-ver>/include
搜索头文件的位置顺序（16.2）	
	带引号形式先搜索当前文件的目录，然后在系统目录中搜索。尖括号形式仅搜索系统目录。
#pragma 伪指令的行为（16.6）	
	请参见 pragma 伪指令。
转换的日期不可用时等同于__DATE__的文本（16.8）	
	扩展为 "??? ?? ????"
转换的时间不可用时等同于__TIME__的文本（16.8）	
	扩展为 "??? ?? ????"
__STDC_VERSION__是否为预定义的宏以及它的含义（16.8）	
	该宏为长整型常量，其格式为 *yyyymm*L，其中 *yyyy* 为编译器兼容的 C 标准版本的年份，*mm* 为相应的月份。这显示了编译器遵循的 C 标准版本。
__STDC__是否为预定义的宏以及它的含义（16.8）	
	在正常操作中，该宏扩展为常量 1，以表示该编译器符合 ISO 标准 C。
独立实现的头文件集（17.6.1.3）	
	标准要求的所有头文件（<atomic>除外），具体如下： <ciso646> <cstddef> <cfloat> <limits> <climits> <cstdint> <cstdlib> <new> <typeinfo> <exception> <initializer_list> <cstdalign> <cstdarg> <cstdbool> <type_traits>
使用外部链接声明的 C 标准库中的名称是否具有 extern "C"或 extern "C++"链接（17.6.2.3）	
	extern "C++"
没有异常规范的标准库函数抛出的异常（17.6.5.12）	
	不存在实现定义的异常类。
源自操作系统之外的错误的 error_category（17.6.5.14）	
	无。
size_t 的类型（18.2）	
	unsigned int

.....（续）

ISO 标准	实现
返回至主机环境的退出状态的形式（18.5）	
	不支持主机环境。
bad_alloc::what 的返回值（18.6.2.1）	
	类的名称。
type_info::name()的返回值（18.7.1）	
	损坏的类型名称。
bad_cast::what 的返回值（18.7.2）	
	类的名称。
bad_typeid::what 的返回值（18.7.3）	
	类的名称。
exception::what 的结果（18.8.1）	
	类的名称。
bad_exception::what 的返回值（18.8.2）	
	类的名称。
使用非 POF（非“普通旧函数”）作为信号处理程序（18.10）	
	不支持这些信号。
shared_ptr 构造函数失败时的异常类型（20.8.2.2.1）	
	std::bad_alloc
占位符类型是否为 CopyAssignable（20.9.9.1.4）	
	是。
是否支持扩展对齐（20.10.7.6）	
	支持。
在 time_t 值与 time_point 对象之间转换时，是将值舍入还是截断为所需的精度（20.12.7.1）	
	值被截断。
streamoff 的类型（21.2.3.1）	
	long long
streampos 的类型（21.2.3.1）	
	与 fpos<mbstate_t>同义
char_traits<char16_t>::eof 的返回值（21.2.3.2）	
	int_type(-1)
u16streampos 的类型（21.2.3.2）	
	与 fpos<mbstate_t>同义
char_traits<char32_t>::eof 的返回值（21.2.3.3）	
	int_type(-1)
u32streampos 的类型（21.2.3.3）	
	与 fpos<mbstate_t>同义
wstreampos 的类型（21.2.3.4）	
	与 fpos<mbstate_t>同义
语言环境对象为全局还是按线程（22.3.1）	
	语言环境对象为全局。
对调用 locale::global 的 C 语言环境的作用（22.3.1.5）	
	如果语言环境有名称，则与 setlocale()相同。
ctype<char>::table_size 的值（22.4.1.3）	
	SCHAR_MAX + 1
time_get::do_get_date 接受的额外格式（22.4.5.1.2）	
	无

.....（续）

ISO 标准	实现
time_get::do_get_year 是否接受两位数的年份（22.4.5.1.2）	
	接受两位数的年份。
time_put::do_put 在 C 语言环境中生成的格式化字符序列（22.4.5.3.2）	
	与 strftime() 相同
array::const_iterator 的类型（23.3.2.1）	
	const T*
array::iterator 的类型（23.3.2.1）	
	T*
unordered_map 的默认桶数（未在构造函数中指定时时）（23.5.4.2）	
	0
unordered_multimap 的默认桶数（未在构造函数中指定时时）（23.5.5.2）	
	0
unordered_set 的默认桶数（未在构造函数中指定时时）（23.5.6.2）	
	0
unordered_multiset 的默认桶数（未在构造函数中指定时时）（23.5.7.2）	
	0
random_shuffle 的随机数的根本来源（25.3.12）	
	使用 rand() 作为随机数的来源。
在进行任何输入或输出操作之后调用 ios_base::sync_with_stdio 对标准流的作用（27.5.3.4）	
	只涉及标准 C 实用程序与标准 C++对象之间的同步。用户声明的流不受影响。
用于构造由 basic_ios 标志函数抛出的 basic_ios::failure 的 fail 对象的参数值（27.5.5.4）	
	const char&
basic_stringbuf 传送构造函数是否复制序列指针（27.8.2.1）	
	复制序列指针。
调用含非零参数的 basic_streambuf::setbuf 的作用（27.8.2.4）	
	如果尚未创建缓冲区且两个参数均非零，则第一个参数用作缓冲区。
basic_filebuf 传送构造函数是否复制序列指针（27.9.1.2）	
	复制序列指针。
调用含非零参数的 basic_filebuf::setbuf 的作用（27.9.1.5）	
	第一个参数用作缓冲区。
存在获取区域时调用 basic_filebuf::sync 的作用（27.9.1.5）	
	无作用。
regex_constants::error_type 的类型（28.5.3）	
	无范围的枚举类型。
各种 ATOMIC_...LOCK_FREE 宏的值（29.4）	
	不支持。
如果实现具有放宽的指针安全性，则 get_pointer_safety 返回 pointer_safety::relaxed 还是 pointer_safety::preferred（29.4）	
	放宽的指针安全性。
是否存在 native_handle_type 和 native_handle 以及它们的含义（30.2.3）	
	不支持本机句柄。
ios_base::streamoff 的类型（D.6）	
	long long int
ios_base::streampos 的类型（D.6）	
	Typedef fpos<mbstate_t>

26. 内置函数

本附录列出了特定于 MPLAB XC32 C/C++编译器的内置函数。

内置函数使 C 语言程序员可以访问目前只能通过行内汇编访问但却十分有用、适用于一系列广泛应用程序的汇编器操作符或机器指令。内置函数在 C 源文件中编写，语法类似于函数调用，但它们被编译为直接实现函数的汇编代码，不涉及函数调用或库程序。

有一些原因使得提供内置函数优于要求程序员使用行内汇编。这些原因包括：

1. 提供用于特定目的的内置函数可以简化编码。
2. 使用行内汇编时，某些优化会被禁止。内置函数则不会如此。
3. 对于使用专用寄存器的机器指令，在编写行内汇编代码时，需要特别细心处理，避免寄存器分配错误。内置函数让这个过程变得简单，因为您不需要关心每条机器指令的特殊寄存器要求。

26.1 内置函数说明

本节介绍编译器内置函数的程序员接口。由于这些函数是“内置”的，所以不存在与之关联的头文件。类似地，不存在与内置函数关联的命令行开关——它们总是可用。所选择的内置函数名称使它们属于编译器的命名空间（均具有前缀 `__builtin_`），所以它们不会与程序员命名空间中的函数或变量名称发生冲突。

26.1.1 void __builtin_nop(void)

发出无操作指令。

原型

```
void __builtin_nop(void);
```

参数

无。

返回值

无。

汇编器操作符/机器指令

```
nop
```

错误消息

无。

26.1.2 __builtin_software_breakpoint 内置函数

插入软件断点。

请注意，在<assert.h>中定义的`__conditional_software_breakpoint()`宏提供了一个轻量版`assert(exp)`，在断言失败时只会插入软件断点而不会打印消息。在包含<assert.h>时，如果已定义名为 NDEBUG 的宏或未定义名为__DEBUG 的宏，则禁止该宏。例如：

```
__conditional_software_breakpoint(myPtr!=NULL);
```

原型

```
void __builtin_software_breakpoint(void)
```

参数

无。

返回值

无。

汇编器操作符/机器指令

`bkpt`

错误消息

无。

27. ASCII 字符集

表 27-1. ASCII 字符集

	高位字符								
低位字符	十六进制	0	1	2	3	4	5	6	7
	0	NUL	DLE	空格	0	@	P	'	p
	1	SOH	DC1	!	1	A	Q	a	q
	2	STX	DC2	"	2	B	R	b	r
	3	ETX	DC3	#	3	C	S	c	s
	4	EOT	DC4	$	4	D	T	d	t
	5	ENQ	NAK	%	5	E	U	e	u
	6	ACK	SYN	&	6	F	V	f	v
	7	Bell	ETB	'	7	G	W	g	w
	8	BS	CAN	(	8	H	X	h	x
	9	HT	EM	)	9	I	Y	i	y
	A	LF	SUB	*	:	J	Z	j	z
	B	VT	ESC	+	;	K	[	k	{
	C	FF	FS	,	<	L	\	l	|
	D	CR	GS	-	=	M	]	m	}
	E	SO	RS	.	>	N	^	n	~
	F	SI	US	/	?	O	_	o	DEL

28. 文档版本历史

28.1 文档版本历史

版本 A（2019 年 6 月）

- 本文档的初始版本。

版本 B（2021 年 10 月）

- 增加了代码覆盖信息
- 增加了堆栈指导
- 重新调整了格式和编号

版本 C（2023 年 4 月）

- 更新了库信息以反映最近的更改，包括 Microchip 统一标准库
- 更正了关于使用__pack 说明符的示例
- 删除了关于复位原因的“操作指南”，因为不适用于 PIC32C/SAM 器件
- 更新了浮点类型的大小以反映最近的编译器更改
- 更新了关于如何确保函数不被删除的信息
- 更新了关于使用调试器的信息
- 更新了关于 MPLAB X IDE 中提供的编译器选项控件的信息
- 更新了关于运行时启动代码的信息，包括关于数据初始化模板的信息
- 除了汇总表之外，还提供了专门的章节来介绍所有编译器选项
- 新增了关于-mdfp、-mpure-code、-feliminate-unused-debug-symbols、-Og 和--dinit-compress 选项的信息
- 删除了一些与 PIC32/SAM 器件无关的选项和说明
- 增加了“不符合 C99 语言标准的方面”
- 扩充了配置位访问信息
- 改进了关于紧耦合存储器的信息
- 增加了 used 变量属性
- 增加了 externally_visible 和 nopa 函数属性
- 增加了关于将代码放入仅执行存储器（XOM）的信息
- 更新了关于中断现场切换的信息
- 改进了关于智能 IO 功能的信息
- 进行了大量一般更正和改进

版本 D（2023 年 9 月）

- 更新了对 C++标准的引用，以反映现在支持 ISO/IEC 14882:2014 C++编程语言
- 新增章节定义了符合 C++14 标准的编译器实现定义的行为
- 增加了关于中断操作的附加信息
- 删除了 CCI 一章及其引用，因为 C/SAM 器件不支持 CCI，该功能在 MPLAB X IDE 中为禁止状态。

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。我们的网站提供以下内容：

- **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 设计伙伴计划成员名单
- **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

产品变更通知服务

Microchip 的产品变更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，请访问 www.microchip.com/pcn，然后按照注册说明进行操作。

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（ESE）
- 技术支持

客户应联系其代理商、代表或 ESE 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 www.microchip.com/support 获得网上技术支持。

Microchip 器件代码保护功能

请注意以下有关 Microchip 产品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规范。
- Microchip 确信：在正常使用且符合工作规范的情况下，Microchip 系列产品非常安全。
- Microchip 注重并积极保护其知识产权。严禁任何试图破坏 Microchip 产品代码保护功能的行为，这种行为可能会违反《数字千年版权法案》（Digital Millennium Copyright Act）。
- 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。

法律声明

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。

本出版物及其提供的信息仅适用于 Microchip 产品，包括设计、测试以及将 Microchip 产品集成到您的应用中。以其他任何方式使用这些信息都将被视为违反条款。本出版物中的器件应用信息仅为您提供便利，

将来可能会发生更新。如需额外的支持，请联系当地的 Microchip 销售办事处，或访问 www.microchip.com/en-us/support/design-help/client-support-services。

Microchip“按原样”提供这些信息。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，包括但不限于针对非侵权性、适销性和特定用途的适用性的暗示担保，或针对其使用情况、质量或性能的担保。

在任何情况下，对于因这些信息或使用这些信息而产生的任何间接的、特殊的、惩罚性的、偶然的或间接的损失、损害或任何类型的开销，Microchip 概不承担任何责任，即使 Microchip 已被告知可能发生损害或损害可以预见。在法律允许的最大范围内，对于因这些信息或使用这些信息而产生的所有索赔，Microchip 在任何情况下所承担的全部责任均不超出您为获得这些信息向 Microchip 直接支付的金额（如有）。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损害、索赔、诉讼或费用时，会维护和保障 Microchip 免于承担法律责任。除非另外声明，在 Microchip 知识产权保护下，不得暗中或以其他方式转让任何许可证。

商标

Microchip 的名称和徽标组合、Microchip 徽标、Adaptec、AVR、AVR 徽标、AVR Freaks、BesTime、BitCloud、CryptoMemory、CryptoRF、dsPIC、flexPWR、HELDO、IGLOO、JukeBlox、KeeLoq、Kleer、LANCheck、LinkMD、maXStylus、maXTouch、MediaLB、megaAVR、Microsemi、Microsemi 徽标、MOST、MOST 徽标、MPLAB、OptoLyzer、PIC、picoPower、PICSTART、PIC32 徽标、PolarFire、Prochip Designer、QTouch、SAM-BA、SenGenuity、SpyNIC、SST、SST 徽标、SuperFlash、Symmetricom、SyncServer、Tachyon、TimeSource、tinyAVR、UNI/O、Vectron 及 XMEGA 均为 Microchip Technology Incorporated 在美国和其他国家或地区的注册商标。

AgileSwitch、ClockWorks、The Embedded Control Solutions Company、EtherSynch、Flashtec、Hyper Speed Control、HyperLight Load、Libero、motorBench、mTouch、Powermite 3、Precision Edge、ProASIC、ProASIC Plus、ProASIC Plus 徽标、Quiet-Wire、SmartFusion、SyncWorld、TimeCesium、TimeHub、TimePictra、TimeProvider 和 ZL 均为 Microchip Technology Incorporated 在美国的注册商标。

Adjacent Key Suppression、AKS、Analog-for-the-Digital Age、Any Capacitor、AnyIn、AnyOut、Augmented Switching、BlueSky、BodyCom、Clockstudio、CodeGuard、CryptoAuthentication、CryptoAutomotive、CryptoCompanion、CryptoController、dsPICDEM、dsPICDEM.net、Dynamic Average Matching、DAM、ECAN、Espresso T1S、EtherGREEN、EyeOpen、GridTime、IdealBridge、IGaT、In-Circuit Serial Programming、ICSP、INICnet、Intelligent Paralleling、IntelliMOS、Inter-Chip Connectivity、JitterBlocker、Knob-on-Display、MarginLink、maxCrypto、maxView、memBrain、Mindi、MiWi、MPASM、MPF、MPLAB Certified 徽标、MPLIB、MPLINK、mSiC、MultiTRAK、NetDetach、Omniscient Code Generation、PICDEM、PICDEM.net、PICkit、PICtail、Power MOS IV、Power MOS 7、PowerSmart、PureSilicon、QMatrix、REAL ICE、Ripple Blocker、RTAX、RTG4、SAM-ICE、Serial Quad I/O、simpleMAP、SimpliPHY、SmartBuffer、SmartHLS、SMART-I.S.、storClad、SQI、SuperSwitcher、SuperSwitcher II、Switchtec、SynchroPHY、Total Endurance、Trusted Time、TSHARC、Turing、USBCheck、VariSense、VectorBlox、VeriPHY、ViewSpan、WiperLock、XpressConnect 和 ZENA 均为 Microchip Technology Incorporated 在美国和其他国家或地区的商标。

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

Adaptec 徽标、Frequency on Demand、Silicon Storage Technology 和 Symmcom 均为 Microchip Technology Inc.在除美国外的国家或地区的注册商标。

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

在此提及的所有其他商标均为各持有公司所有。

ISBN： 978-1-6683-4625-9

质量管理体系

有关 Microchip 质量管理体系的信息，请访问 www.microchip.com/quality。

全球销售及服务网点

美洲	亚太地区	亚太地区	欧洲
公司总部 2355 West Chandler Blvd. Chandler, AZ 85224-6199 电话：480-792-7200 传真：480-792-7277 技术支持： www.microchip.com/support 网址： www.microchip.com	**澳大利亚 - 悉尼** 电话：61-2-9868-6733	**印度 - 班加罗尔** 电话：91-80-3090-4444	**奥地利 - 韦尔斯** 电话：43-7242-2244-39 传真：43-7242-2244-393
亚特兰大 德卢斯，佐治亚州 电话：678-957-9614 传真：678-957-1455	**中国 - 北京** 电话：86-10-8569-7000	**印度 - 新德里** 电话：91-11-4160-8631	**丹麦 - 哥本哈根** 电话：45-4485-5910 传真：45-4485-2829
奥斯汀，德克萨斯州 电话：512-257-3370	**中国 - 成都** 电话：86-28-8665-5511	**印度 - 浦那** 电话：91-20-4121-0141	**芬兰 - 埃斯波** 电话：358-9-4520-820
波士顿 韦斯特伯鲁，马萨诸塞州 电话：774-760-0087 传真：774-760-0088	**中国 - 重庆** 电话：86-23-8980-9588	**日本 - 大阪** 电话：81-6-6152-7160	**法国 - 巴黎** 电话：33-1-69-53-63-20 传真：33-1-69-30-90-79
芝加哥 艾塔斯卡，伊利诺伊州 电话：630-285-0071 传真：630-285-0075	**中国 - 东莞** 电话：86-769-8702-9880	**日本 - 东京** 电话：81-3-6880-3770	**德国 - 加兴** 电话：49-8931-9700
达拉斯 阿迪森，德克萨斯州 电话：972-818-7423 传真：972-818-2924	**中国 - 广州** 电话：86-20-8755-8029	**韩国 - 大邱** 电话：82-53-744-4301	**德国 - 哈恩** 电话：49-2129-3766400
底特律 诺维，密歇根州 电话：248-848-4000	**中国 - 杭州** 电话：86-571-8792-8115	**韩国 - 首尔** 电话：82-2-554-7200	**德国 - 海尔布隆** 电话：49-7131-72400
休斯顿，德克萨斯州 电话：281-894-5983	**中国 - 香港特别行政区** 电话：852-2943-5100	**马来西亚 - 吉隆坡** 电话：60-3-7651-7906	**德国 - 卡尔斯鲁厄** 电话：49-721-625370
印第安纳波利斯 诺布尔斯维尔，印第安纳州 电话：317-773-8323 传真：317-773-5453 电话：317-536-2380	**中国 - 南京** 电话：86-25-8473-2460	**马来西亚 - 槟榔屿** 电话：60-4-227-8870	**德国 - 慕尼黑** 电话：49-89-627-144-0 传真：49-89-627-144-44
洛杉矶 米慎维荷，加利福尼亚州 电话：949-462-9523 传真：949-462-9608 电话：951-273-7800	**中国 - 青岛** 电话：86-532-8502-7355	**菲律宾 - 马尼拉** 电话：63-2-634-9065	**德国 - 罗森海姆** 电话：49-8031-354-560
罗利，北卡罗来纳州 电话：919-844-7510	**中国 - 上海** 电话：86-21-3326-8000	**新加坡** 电话：65-6334-8870	**以色列 - 霍德夏沙隆** 电话：972-9-775-5100
纽约，纽约州 电话：631-435-6000	**中国 - 沈阳** 电话：86-24-2334-2829	**台湾地区 - 新竹** 电话：886-3-577-8366	**意大利 - 米兰** 电话：39-0331-742611 传真：39-0331-466781
圣何塞，加利福尼亚州 电话：408-735-9110 电话：408-436-4270	**中国 - 深圳** 电话：86-755-8864-2200	**台湾地区 - 高雄** 电话：886-7-213-7830	**意大利 - 帕多瓦** 电话：39-049-7625286
加拿大 - 多伦多 电话：905-695-1980 传真：905-695-2078	**中国 - 苏州** 电话：86-186-6233-1526	**台湾地区 - 台北** 电话：886-2-2508-8600	**荷兰 - 德卢内市** 电话：31-416-690399 传真：31-416-690340
	中国 - 武汉 电话：86-27-5980-5300	**泰国 - 曼谷** 电话：66-2-694-1351	**挪威 - 特隆赫姆** 电话：47-72884388
	中国 - 西安 电话：86-29-8833-7252	**越南 - 胡志明市** 电话：84-28-5448-2100	**波兰 - 华沙** 电话：48-22-3325737
	中国 - 厦门 电话：86-592-2388138		**罗马尼亚 - 布加勒斯特** 电话：40-21-407-87-50
	中国 - 珠海 电话：86-756-3210040		**西班牙 - 马德里** 电话：34-91-708-08-90 传真：34-91-708-08-91
			瑞典 - 哥德堡 电话：46-31-704-60-40
			瑞典 - 斯德哥尔摩 电话：46-8-5090-4654
			英国 - 沃金厄姆 电话：44-118-921-5800 传真：44-118-921-5820