适用于 AVR® MCU 的 MPLAB® XC8 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>，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

开发工具客户须知 1

1. 前言 4
   - 1.1. 本指南使用的约定 4
   - 1.2. 推荐读物 5

2. 编译器概述 6
   - 2.1. 器件说明 6
   - 2.2. C 标准 6
   - 2.3. 主机和许可证 7
   - 2.4. 约定 7
   - 2.5. 兼容的开发工具 7

3. 命令行驱动程序 8
   - 3.1. 调用编译器 8
   - 3.2. 编译序列 9
   - 3.3. 运行时文件 12
   - 3.4. 编译器输出 13
   - 3.5. 编译器消息 14
   - 3.6. 选项说明 15
   - 3.7. MPLAB X IDE 集成 37
   - 3.8. Microchip Studio 集成 45

4. C 语言特性 56
   - 4.1. 符合 C 标准 56
   - 4.2. 与器件相关的特性 56
   - 4.3. 支持的数据类型和变量 60
   - 4.4. 存储器分配和访问 71
   - 4.5. 操作符和语句 75
   - 4.6. 寄存器使用 76
   - 4.7. 函数 76
   - 4.8. 中断 79
   - 4.9. main、运行时启动和复位 82
   - 4.10. 库 84
   - 4.11. 混合使用 C 代码和汇编代码 86
   - 4.12. 优化 93
   - 4.13. 预处理 93
   - 4.14. 链接程序 96

5. 实用程序 99
   - 5.1. 归档器/库管理器 99
   - 5.2. Objdump 100

6. 实现定义的行为 101
   - 6.1. 概述 101

6.2. 转换 101
6.3. 环境 101
6.4. 标识符 102
6.5. 字符 102
6.6. 整型 103
6.7. 浮点型 103
6.8. 数组和指针 104
6.9. 提示 105
6.10. 结构、联合、枚举和位域 105
6.11. 限定符 105
6.12. 预处理伪指令 105
6.13. 库函数 106
6.14. 架构 109

7. 库函数 110
7.1. 库示例代码 110
7.2. <boot.h>自举程序函数 112
7.3. <cpufunc.h> CPU 相关函数 119
7.4. <delay.h>延时函数 120
7.5. <pgmspace.h> 121
7.6. <sfr_defs.h> 128
7.7. <sleep.h> 130
7.8. 内置函数 133

8. 文档版本历史 138

Microchip 网站 140

产品变更通知服务 140

客户支持 140

Microchip 器件代码保护功能 140

法律声明 140

商标 141

质量管理体系 141

全球销售及服务网点 142

1. 前言

1.1 本指南使用的约定

本指南采用以下文档约定：

表 1-1. 文档约定

说明	表示	示例
Arial 字体：		
斜体字	参考书目	*MPLAB® IDE User's Guide*
	需强调的文字	...*仅有*的编译器...
首字母大写	窗口	Output 窗口
	对话框	Settings 对话框
	菜单选择	选择 File，然后单击 Save。
引用	窗口或对话框中的字段名	“Save project before build”
带右尖括号有下划线的斜体文字	菜单路径	*File>Save*
粗体字	对话框按钮	单击 **OK**
	选项卡	单击 **Power** 选项卡
N'Rnnnn	verilog 格式的数字，其中 N 为总位数，R 为基数，n 为其中一位。	4'b0010，2'hF1
尖括号< >括起的文字	键盘上的按键	按下<Enter>，<F1>
Courier New 字体：		
常规 Courier New	源代码示例	`#define START`
	文件名	`autoexec.bat`
	文件路径	`C:\Users\User1\Projects`
	关键字	`static, auto, extern`
	命令行选项	`-Opa+，-Opa-`
	二进制位值	`0 和 1`
	常量	`0xFF，‘A’`
斜体 Courier New	可变参数	*`file.o`*，其中 *`file`* 可以是任一有效文件名
方括号[ ]	可选参数	`xc8 [options] files`
花括号和竖线：{ | }	选择互斥参数；“或”选择	`errorlevel {0|1}`
省略号...	代替重复文字	`var_name [, var_name...]`

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

1.2 推荐读物

本用户指南介绍针对 AVR 目标器件进行编译以及使用针对编程语言的 ISO/IEC 9899:1999 标准（C99）时 MPLAB XC8 C 编译器的使用和特性。以下 Microchip 文档均已提供，并建议读者作为补充参考资料。

《适用于 PIC® MCU 的 MPLAB® XC8 C 编译器用户指南》
该版本编译器用户指南面向以 8 位 PIC MCU 为目标器件的项目。

MPLAB® XC8 C Compiler Release Notes for AVR® MCU
有关使用 MPLAB XC8 C 编译器的最新信息，请阅读编译器安装目录的 `docs` 子目录下的 MPLAB® XC8 C Compiler Release Notes（HTML 文件）。发行说明包含本用户指南中可能未提供的最新信息和已知问题。

Microchip Unified Standard Library Reference Guide
所有 MPLAB XC C 编译器均随附该指南，其中包含与 C 标准库定义的类型、宏和函数相关的信息和示例。

开发工具发行说明
有关使用其他开发工具的最新信息，请阅读相应工具的自述文件，自述文件位于 MPLAB X IDE 安装目录的 docs 子目录下。

2. 编译器概述

MPLAB XC8 C 编译器是一款经优化的独立 ISO C99 交叉编译器，适用于 C 编程语言。

它支持所有 8 位 PIC®和 AVR®单片机；但本文档介绍 `xc8-cc` 驱动程序的使用并假设程序针对 Microchip 8 位 AVR 器件进行编译。有关以 Microchip PIC MCU 为目标器件时如何使用该编译器的信息，请参见《适用于 PIC® MCU 的 MPLAB® XC8 C 编译器用户指南》（DS50002737F_CN）。

注：本文档所述的 MPLAB XC8 特性假设用户使用 Microchip AVR 器件。如果选择改为针对 Microchip PIC 器件进行编译，这些特性可能有所不同。

2.1 器件说明

本编译器指南介绍 MPLAB XC8 编译器对所有 8 位 Microchip AVR 器件（包括 tinyAVR®和 AVR XMEGA®器件）的支持。

只要可能，编译器都会利用目标器件的指令集、寻址模式、存储器和寄存器。下面汇总了各个器件系列。其中包含特殊功能寄存器的偏移量以及程序存储器映射到相关数据空间时的偏移量。

关于编译器支持的器件的完整列表的信息，请参见 3.6.2.4. Print-devices。下载更新后的器件系列包后，可能支持新的器件。

表 2-1. 支持的器件系列汇总

系列	ArchID	SFR 偏移量	映射的闪存地址
avr1	1	0x20	n/a
avr2	2	0x20	n/a
avr25	25	0x20	n/a
avr3	3	0x20	n/a
avr31	31	0x20	n/a
avr35	35	0x20	n/a
avr4	4	0x20	n/a
avr5	5	0x20	n/a
avr51	51	0x20	n/a
avr6	6	0x20	n/a
avrtiny	100	0x0	0x4000
avrxmega2	102	0x0	n/a
avrxmega3	103	0x0	0x8000
avrxmega4	104	0x0	n/a
avrxmega5	105	0x0	n/a
avrxmega6	106	0x0	n/a
avrxmega7	107	0x0	n/a

2.2 C 标准

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）。此外，还包含为 8 位 AVR 嵌入式控制应用定制的语言扩展。

2.3 主机和许可证

MPLAB XC8 C 编译器可用于多种常见的操作系统。有关适用于所用编译器版本的操作系统，请参见编译器发行说明。

无论是否具备许可证都可以运行编译器。许可证可以随时购买和应用，以便解锁更高级别的优化。但是，是否具备编译器许可证对于编译器的基本操作、支持的器件和可用存储器没有任何影响。

2.4 约定

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

类似地，“编译器”也常用于指代命令行驱动程序；虽然具体地说，MPLAB XC8 C 编译器软件包的驱动程序名为 `xc8-cc`。3.6. 选项说明介绍了驱动程序及其选项。相应地，“编译器选项”通常指代命令行驱动程序选项。

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

2.5 兼容的开发工具

编译器与许多其他 Microchip 工具配合使用，包括：

- 适用于所有 8 位 PIC 和 AVR 器件的 MPLAB X IDE（www.microchip.com/mplab/mplab-x-ide）
- 适用于 AVR 和 SAM 器件的 Microchip Studio（www.microchip.com/mplab/microchip-studio），适合所有 8 位 AVR 器件
- MPLAB X 软件模拟器
- 命令行 MDB 软件模拟器——请参见《Microchip 调试器（MDB）用户指南》（DS50002102H_CN）
- 所有 Microchip 调试工具和编程器（www.microchip.com/mplab/development-boards-and-tools）
- 支持 8 位 AVR 器件的演示板和入门工具包

可以查看工具的文档来确认其是否支持您计划使用的器件。

3. 命令行驱动程序

可以调用 MPLAB XC8 C 编译器命令行驱动程序 xc8-cc 来执行编译的所有方面，包括 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 最终还是会调用 xc8-cc。

如果编译旧版项目或想要使用旧版命令行驱动程序，则可以改为运行 avr-gcc 驱动程序应用程序，并为该驱动程序使用相应的命令行选项。这些选项可能不同于本指南所述的选项。

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。

3.1 调用编译器

本节介绍如何通过命令行调用 xc8-cc，并讨论可传递给编译器的输入文件。

3.1.1 驱动程序命令行格式

xc8-cc 驱动程序可用来编译和汇编 C 和汇编源文件，并能链接目标文件和库归档文件，以形成最终程序映像。

该驱动程序具有以下基本命令格式：

```
xc8-cc [options] files
```

例如，如果要编译并链接 C 源文件 hello.c，可以使用下面的命令：

```
xc8-cc -mcpu=atmega3290p -O2 -o hello.elf hello.c
```

在本手册中，假定编译器应用程序位于控制台的搜索路径中。有关指定搜索位置的环境变量的信息，请参见 3.1.2. 驱动程序环境变量。执行编译器时，也可以使用完整目录路径以及驱动程序名称。

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

命令行选项区分大小写，3.6. 选项说明中列出了其格式和说明。xc8-cc 所接受的很多命令行选项都是所有 MPLAB XC 编译器的通用选项，从而提高了器件与编译器之间的可移植性。

files 可以是 C 和汇编器源文件以及可重定位目标文件和归档文件的任意混合。这些文件的列出顺序并不对程序运行有直接影响，但会影响代码或数据的分配。请注意，归档文件的顺序即为搜索它们的顺序，在某些情况下，对哪些模块链接到程序会有影响。

3.1.1.1 长命令行

可以向 xc8-cc 驱动程序传递一个命令行文件，其中包含的驱动程序选项和参数可以规避操作系统对命令行长度施加的任何限制。

命令文件通过@符号指定，它后面应紧随（即，没有中间空格字符）包含参数的文件的名称。编译器驱动程序所调用的大多数内部应用程序均可使用与此相同的参数传递方法。

在该文件中，各参数之间必须由一个或多个空格分隔，使用反斜线返回序列时，可以扩展到多个行。文件中可以包含空白行，这些行将被忽略。

例如，以下是命令文件 xyz.xc8 的内容，该文件使用文本编辑器构建，包含编译项目所需的选项和文件名。

```
-mcpu=atmega3290p -Wl,-Map=proj.map -Wa,-a \
-O2 main.c isr.c
```

保存该文件之后，就可以使用以下命令调用编译器：

```
xc8-cc @xyz.xc8
```

命令文件可以用作 make 文件和实用程序的简单替代，可以方便地存储编译器选项和源文件名称。MPLAB X IDE 也可以使用此类文件。文件名在 Project Properties（项目属性）的 **XC8 Linker > Additional options > Use response file to link** > > （XC8 链接器 > 附加选项 > 使用响应文件进行链接）字段中指定。

3.1.2 驱动程序环境变量

编译器未定义环境变量，执行时也不需要环境变量。

调整 PATH 环境变量即可运行编译器驱动程序，无需指定完整的编译器路径。

在相应对话框中选中 **Add xc8 to the path environment variable**（将 xc8 添加到 path 环境变量中）复选框，安装编译器时该变量即会自动得到更新。

请注意，由 PATH 变量指定的目录仅用于查找编译器驱动程序。驱动程序一旦运行，即会管理内部编译器应用程序（如汇编器和链接器等）的访问。

通常，所用 IDE 允许在项目的属性中选择编译器，无需定义 PATH 变量。

3.1.3 输入文件类型

xc8-cc 驱动程序完全根据文件类型或扩展名来区分源文件、中间文件和库文件。可识别下面的表 3-1 中列出的扩展名（区分大小写）。

表 3-1. 输入文件类型

扩展名	文件格式
.c	C 源文件
.i	预处理 C 源文件
.s	汇编源文件
.S	需要预处理的汇编源文件
.o	可重定位目标代码文件
.a	归档（库）文件
其他	要传递给链接器的文件

编译器对源文件的基本名称不存在任何限制，但需要注意主机操作系统对于大小写形式、名称长度和其他方面的限制。

避免对汇编源文件和 C 源文件使用相同的基本名称，即使它们位于不同的目录中也要避免。例如，如果项目包含名为 init.c 的 C 源文件，则不要再为项目添加名为 init.s 的汇编源文件。还要避免源文件名称与包含这些文件的 IDE 项目的名称相同。

*源文件*和*模块*这两个词通常可以互换，但它们指代编译序列中位于不同点的源代码。

源文件是包含程序的全部或部分程序的文件。它可能包含 C 代码，以及预处理器伪指令和命令。源文件最初由编译器驱动程序传递给预处理器。

模块是给定源文件的预处理器输出，是在包含了通过#include 预处理器伪指令指定的所有头文件之后，以及在处理并接着删除其他预处理器伪指令（一些用于调试的命令可能除外）之后的输出。因此，模块通常是一个源文件和多个头文件的合并结果，也是传递给其余编译器应用程序的输出。模块也常被称为*翻译单元*。

这些术语同样适用于汇编源文件，可以对其进行预处理并包含其他（.inc）文件，以生成汇编模块。

3.2 编译序列

编译项目时，驱动程序会调用很多内部应用程序以执行相应的任务。本节将介绍这些内部应用程序，并说明它们如何与编译过程发生联系，特别是项目包含多个源文件的情况。如果您使用 make 系统来编译项目，应特别注意本节内容。

3.2.1 编译器应用程序

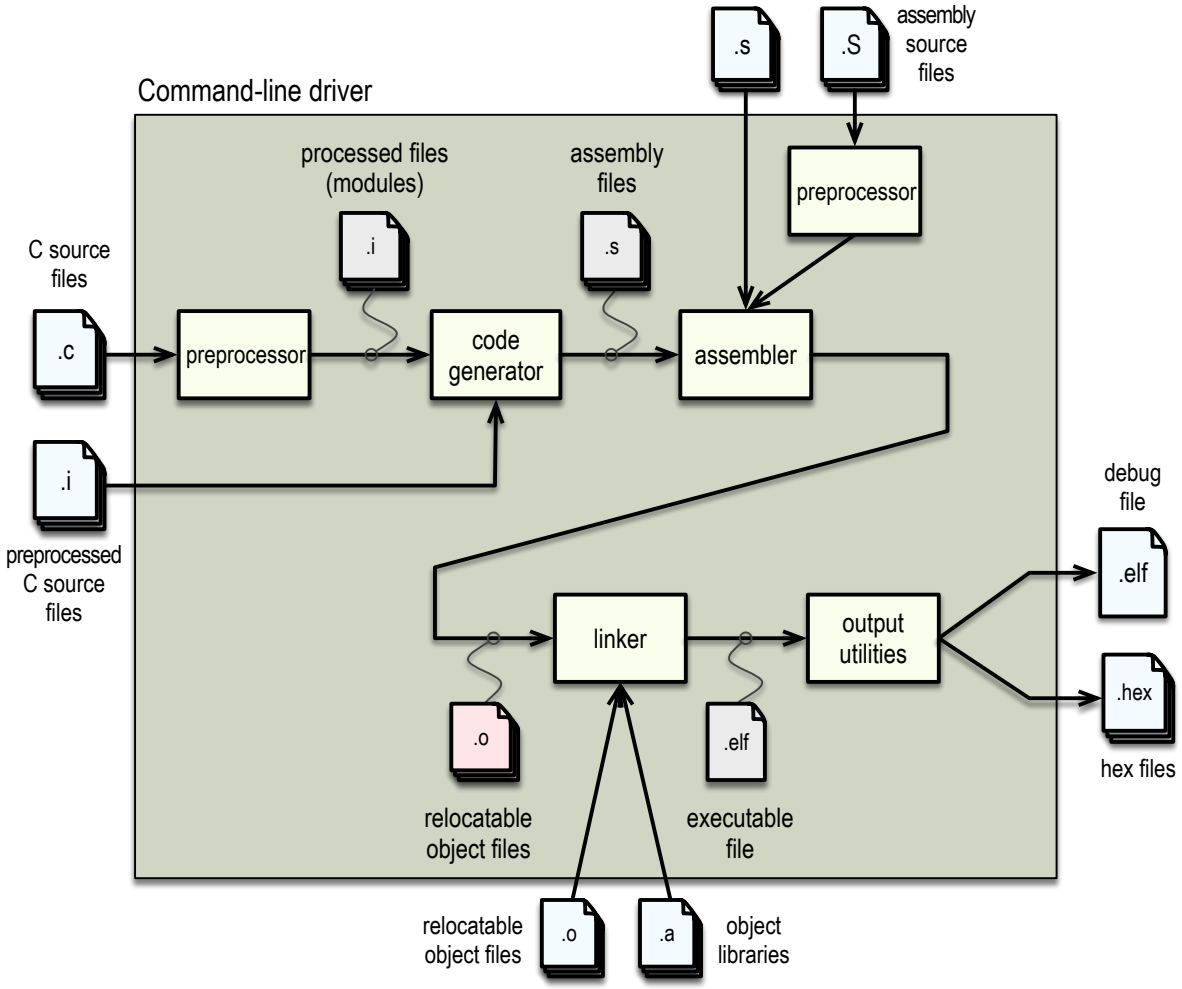
下面的图中显示了主要的内部编译器应用程序和文件。

大的阴影方框代表编译器，它通过命令行驱动程序 xc8-cc 进行控制。只知道 C 源文件（显示在最左侧）被传递给编译器，并生成最终输出文件（此处显示为最右侧的 HEX 和 ELF 调试文件），您可能就会满意；但是，在内部会生成许多应用程序和临时文件。了解编译器的内部操作虽然并非必需，但有助于工具的使用。

需要时，驱动程序会调用所需的编译器应用程序。这些应用程序位于编译器的 bin 目录中，在图中显示为驱动程序内的较小方框。

图中还显示了各个应用程序所生成的临时文件，并在编译序列中标记出生成这些文件的点。C 源文件的中间文件加上了红色阴影。编译结束后，其中一部分临时文件会保留下来。此外，也有一些驱动程序选项可用于要求编译序列在执行特定应用程序之后暂停，这样，该应用程序的输出将保留在文件中，可以进行检查。

图 3-1. 编译器应用程序和文件



建议仅直接执行归档器（xc8-ar）内部应用程序。5. 实用程序对归档器的命令行选项进行了说明。如果在编译项目后需要运行此 HEX 文件操作实用程序，请参见单独提供的 Hexmate 用户指南。

3.2.2 单步 C 编译

使用 xc8-cc 驱动程序可以仅通过一条命令执行一个或多个 C 源文件的完整编译（包括链接步骤）。

3.2.2.1 编译单个 C 文件

下面是将两个数字相加的简单 C 程序。为了说明如何编译和链接包含单个 C 源文件的程序，可将代码复制到任意文本编辑器中，并保存为 ex1.c 纯文本文件。

```
#include <xc.h>

unsigned int
```

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

int
main(void)
{
  unsigned int x, y, z;
  x = 2;
  y = 5;
  z = add(x, y);

  return 0;
}
```

为清楚起见，该代码不指定器件配置位，也没有任何有用的用途。

在所用终端的提示符后键入以下命令来编译程序。鉴于本讨论的目的，假定在所用终端上已切换到包含刚刚创建的源文件的目录，编译器安装在标准目录位置且位于主机的搜索路径中。

```
xc8-cc -mcpu=atmega3290p  -o ex1.elf ex1.c
```

该命令为 atmega3290p 器件编译 `ex1.c` 源文件，并将输出写入到 `ex1.elf` 中，供开发环境使用。

举例来说，如果需要将 HEX 文件载入到器件编程器中，则使用以下命令：

```
avr-objcopy -O ihex a.out ex1.hex
```

这将创建一个名为 `ex1.hex` 的 Intel HEX 文件。

驱动程序将编译此源文件，无论它自上次编译以来是否发生了更改。要实现增量编译，必须使用开发环境和 make 实用程序（见 3.2.3. 多步 C 编译）。

除非另外指定，否则将生成 ELF 文件（称为 `a.out`，不使用`-o` 选项）作为最终的输出。

有些中间文件在编译完成之后会保留，但大多数其他临时文件会被删除，除非使用`-save-temps` 选项（见 3.6.5.5. Save-temps 选项），该选项会保留除运行时启动文件之外的所有生成文件。请注意，如果使用 IDE 进行编译，则某些生成的文件可能与项目源文件位于不同的目录中（另见 3.6.2.3. O：指定输出文件）。

3.2.2.2 编译多个 C 文件

本节演示如何一步编译及链接包含多个 C 源文件的项目。

将所示的示例代码复制到名为 `add.c` 的文本文件中。

```
/* add.c */
#include <xc.h>

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

将以下代码放入另外一个文件 `ext.c` 中。

```
/* ex1.c */
#include <xc.h>

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

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

```
  return 0;
}
```

为清楚起见，该代码不指定器件配置位，也没有任何有用的用途。

在提示符后键入以下命令来编译这两个文件：

```
xc8-cc -mcpu=atmega3290p  -o ex1.elf ex1.c add.c
```

该命令将通过一个步骤编译模块 ex1.c 和 add.c。编译完的模块将与相关编译器库链接，并将创建可执行文件 ex1.elf。

3.2.3 多步 C 编译

多步编译方法可用于编译包含一个或多个 C 源文件的项目。Make 实用程序可以通过此功能注意到自上次编译以来哪些源文件发生了更改，从而加快编译速度。增量编译也可以由集成开发环境执行。

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，一次是执行第二阶段编译，在该阶段将链接中间文件以生成最终输出。如果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。

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

```
xc8-cc -mcpu=atmega3290p -c ex1.c
xc8-cc -mcpu=atmega3290p -c add.c
```

```
xc8-cc -mcpu=atmega3290p  -o ex1.elf ex1.o add.o
```

用于前两个命令的-c 选项可将指定文件编译为中间文件格式，但不执行链接步骤。生成的中间文件在最后一步链接，以创建最终输出 ex1.elf。执行第二阶段编译时，构成项目的所有文件都必须存在。

上例使用命令行驱动程序 xc8-cc 执行最终链接步骤。可以显式调用链接器应用程序 avr-ld，但不建议这样做，这是因为命令具有复杂性，如果直接驱动链接器应用程序，必须指定链接器选项而不是驱动程序选项，如上文所示。

您可能还希望生成中间文件来构造您自己的库归档文件。

3.2.4 汇编源文件编译

构成 C 项目的汇编源文件的编译方式与 C 源文件相似。编译器驱动程序知道这些文件应传递给不同的内部编译器应用程序集合，一个编译命令可包含 C 源文件和汇编源文件的混合，如下例所示。

```
xc8-cc -mcpu=atmega3290p proj.c spi.s
```

如果汇编源文件包含必须在传递给编译器之前进行预处理的 C 预处理器伪指令，则要确保源文件使用.S 扩展名，例如 spi.S。

通过-S 选项，编译器可用于基于 C 源代码生成汇编文件。然后可将汇编输出用作您自己的汇编程序的基础，随后使用命令行驱动程序进行编译。

3.3 运行时文件

除了 C 源文件和汇编源文件以及在命令行指定的用户定义库之外，编译器还可以链接项目中的编译器生成的源文件和预编译的库文件，这些文件的内容归属以下类别：

- C 标准库程序
- 隐式调用的算术库程序
- 运行时启动代码

3.3.1 库文件

C 标准库包含一组标准化的函数，例如字符串、数学和输入/输出程序。7.1. 库示例代码介绍了这些函数的使用和操作。有关创建和使用自己的库的更多信息，请参见 4.10. 库。

这些库通过一组与使用顺序无关的选项多次进行编译。当调用编译器驱动程序来编译和链接应用程序时，驱动程序选择已使用相同选项编译的相应目标库。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。

3.3.1.1 位置和命名约定

`libc.a` 等标准库位于 `avr/avr/lib` 目录中。不受硬件原生支持的操作的仿真程序是 `avr/lib/gcc/avr/`下的 `libgcc.a` 的一部分。`libm.a` 数学库也会自动链接，包含用于看门狗定时器、电源管理、EEPROM 访问等的器件特定程序的 `lib`*`devicename`*`.a`（如 `libatxmega128b1.a`）也是如此（更多信息，请参见 7.1. 库示例代码）。

3.3.2 启动和初始化

运行时启动代码执行一些初始化任务，这些任务必须先于 C 程序中的 `main()` 函数执行。有关该代码执行的任务的信息，请参见 4.9. main、运行时启动和复位。

编译器将基于所选目标器件和其他编译器选项来选择相应的运行时启动代码。

3.3.2.1 运行时启动代码

编译器提供包含运行时启动代码的预编译目标文件。这些文件构成器件系列包的一部分，位于项目目标器件相关目录下的`<dfp>/avr/lib` 中。

如果需要在复位后立即执行任何特殊的初始化，则应写入后面的 4.9.3. 上电程序中介绍的上电初始化程序。

3.4 编译器输出

编译器在编译过程中会创建许多文件。其中的大量文件是临时文件或中间文件，编译完成后将被删除，但有些文件会保留下来用于对器件进行编程或调试，在编译期间暂停编译的选项将保留中间文件，可对其进行检查。

3.4.1 输出文件

表 3-2 列出了通用输出文件类型和扩展名（区分大小写）。

表 3-2. 通用输出文件

扩展名	文件类型	创建方式
`.hex`	Intel HEX	`avr-objcopy` 应用程序
`.elf`	ELF（可执行且可链接格式），包含 Dwarf 调试信息	`-o` 选项
`.s`	汇编文件	`-S` 选项
`.i`	预处理的 C 文件	`-E` 和`-o` 选项

`xc8-cc` 的默认行为是生成名为 `a.out` 的 ELF 文件（除非使用`-o` 选项改写了该名称）。

调试器可利用 ELF/DWARF 文件获取关于项目的调试信息，从而获得比其他格式（如 COFF）更精准的调试结果。使用 COFF 时，调试器可能无法获知项目运行情况的某些方面。开发环境通常会要求编译器生成 ELF 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，如果使用`-c` 选项，源文件 `input.c` 将创建一个名为 `input.o` 的文件。

3.4.2 诊断文件

可由编译器生成的两个重要文件分别是汇编列表文件（由汇编器生成）和映射文件（由链接器生成）。这些均通过表 3-3 所示的选项生成。

表 3-3. 诊断文件

扩展名	文件类型	创建方式
file.lst	汇编列表文件	-Wa,-a=file.lst 驱动程序选项
file.map	映射文件	-Wl,-Map=file.map 驱动程序选项

汇编列表文件包含原始源代码与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存储区域时，它是必需的。

用于创建列表文件的汇编器选项是-a，可使用驱动程序选项（例如-Wa,-a=*file*.lst）从驱动程序传递给汇编器。

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。

用于在链接器应用程序中创建映射文件的链接器选项是-Map *file*，该选项可使用驱动程序选项（例如-Wl,-Map=*file*.map）从驱动程序传递给链接器。

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。

3.5 编译器消息

所有编译器应用程序在编译过程中都使用文本消息来报告反馈。

下面列出了几种不同类型的消息，以及遇到每种类型的消息时编译器的行为。

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（但初始错误的原因可能会引发更多错误），并会对项目中的其他模块进行编译，但不会链接项目。

致命消息 指示编译不能继续进行、需强制立即停止编译过程的情况。

3.5.1 改变消息的行为

对于编译器生成的消息，可以更改一些属性，有时甚至可以完全禁止消息。此外，还可以控制生成的警告消息数量来协助调试。

3.5.1.1 禁止消息

编译器将发出警告，提醒您源代码中潜在的问题。

可以使用-w 选项禁止所有警告消息。

可以使用-Wno-*message* 选项关闭显式警告，其中 *message* 与警告类型相关，例如，-Wno-return-type 选项将阻止与返回类型默认为 int 的函数相关的警告。当编译器产生警告时，用于控制该警告的相关警告选项将括上方括号。例如，如果编译器发出以下警告：

```
avr.c:13:1: warning: 'keep' attribute directive ignored [-Wattributes]
```

可以使用选项-Wno-attributes 禁止该警告。

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

可以使用-Wall 使能一组更完整的关于有问题构造的警告消息。-Wextra 选项用于开启额外消息。此外，也可以使用-W*message* 选项使能单个消息，例如，-Wunused-function 将确保为从未调用的函数产生警告。

3.5.1.2 更改消息类型

还可以更改某些消息的类型（以及行为）。

可以使用-Werror 选项将警告转变为错误。可以使用-Wfatal-errors 选项将错误转变为致命错误。

3.6 选项说明

使用传递给命令行驱动程序 xc8-cc 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 XC8 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。

所有选项均区分大小写，并通过一个前导短划线或两个前导短划线字符进行标识，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。

如果在开发环境中进行编译，则它会根据项目的属性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。

3.6.1 特定于 AVR 器件的选项

下表所示的选项在使用 MPLAB XC8 编译器为 8 位 Microchip AVR 器件进行编译时非常有用，这些选项将在后面的章节中详细讨论。

表 3-4. AVR 器件特定的选项

选项 （链接至说明部分）	控制
-m[no-]accumulate-args	如何在函数之间传递参数。
-m[no-]call-prologues	函数如何保存和恢复寄存器。
-m[no-]const-data-in-config-mapped-progmem	编译器是否会将 const 限定数据放入随后将映射到数据存储器中的 32k 闪存段。
-m[no-]const-data-in-progmem	const 限定对象放入程序存储器还是数据存储器中。
-mcpu=*device*	要进行编译的目标器件或架构。
-mdfp=*path*	器件特定信息源。
-mno-interrupts	如何更改堆栈指针。
-fno-jump-tables	switch()语句中是否使用跳转表。
-mrelax	优化调用/跳转指令。
-mreserve=*space@start:end*	存储空间中要保留的地址范围。
-mshort-calls	如何对函数调用进行编码。
-msmart-io=*level*	链接的 IO 库的功能集。
-msmart-io-format="*format*"	手动包含 IO 库功能。
-mstrict-X	使用 X 寄存器。
-mtiny-stack	堆栈指针的宽度。

3.6.1.1 Accumulate-args 选项

-maccumulate-args 选项用于防止函数参数入栈和出栈，而非生成在调用函数开始时调整一次堆栈指针的代码。调用不使用参数堆栈的函数时，该选项不起作用，但对于其他函数，如果多次调用这些函数，该选项会缩短代码长度。

3.6.1.2 Call-isr-prologues 选项

-mcall-isr-prologues 选项用于更改中断函数如何在进入中断函数时保存寄存器以及如何在中断程序终止时恢复这些寄存器。该选项的工作方式与-mcall-prologues 选项类似，但仅影响中断函数（ISR）。

如果未指定该选项，需要由 ISR 保留的寄存器将通过这些函数内的代码保存和恢复。当 ISR 调用另一个函数时，将有大量寄存器需要保留。如果使用-mcall-isr-prologues 选项，该保留代码将作为在 ISR 中的适当点处调用的子程序提取。

如果有多个中断函数包含对另一个函数的调用，使用该选项可以缩短代码长度，但会延长代码的执行时间。

3.6.1.3 Call-prologues 选项

`-mcall-prologues` 选项用于更改如何在进入函数时保存寄存器以及如何在退出函数时恢复这些寄存器。

如果未指定该选项或使用`-mno-call-prologues` 选项，则需要由每个函数保留的寄存器将通过这些函数内的代码保存和恢复。如果使用`-mcall-prologues` 选项，该保留代码将作为在函数中的适当点处调用的子程序提取。

使用该选项可以缩短代码长度，但会延长代码的执行时间。

3.6.1.4 Const-data-in-config-mapped-progmem 选项

`-mconst-data-in-config-mapped-progmem` 选项用于将 `const` 限定对象存储到程序存储器中的某个区域，编译器将确保该区域映射到一些器件的数据存储空间中。

某些器件（如 AVR DA 和 AVR DB 系列）具有 SFR（如 FLMAP），用于指定程序存储器的哪个 32 KB 段将映射到数据存储空间中。`-mconst-data-in-config-mapped-progmem` 选项可用于使链接器将所有 `const` 限定数据放入其中一个 32 KB 段中并自动初始化相关 SFR 寄存器以确保这些对象映射到数据存储器中，就程序大小而言，这些对象在数据存储器中的访问将更为高效。该选项必须与`-mconst-data-in-progmem` 选项结合使用；但该选项默认使能。

对于可支持该映射功能的器件，`-mconst-data-in-config-mapped-progmem` 选项自动使能。在这种情况下，可以使用`-mno-const-data-in-config-mapped-progmem` 选项禁止该功能，强制从程序存储器访问 `const` 限定对象。

由于`-mconst-data-in-config-mapped-progmem` 选项不仅影响代码的生成方式，还影响待链接库的选择，因此必须同时遵循编译过程的编译步骤和链接步骤来使用该选项（或其否定形式 `no-`）。

如果定义了超过 32 KB 的 `const` 限定数据或者该段无法放入映射的存储段中，则链接器将发出错误。

如果使能了该选项且 `const` 限定对象位于映射的存储段中，则定义宏 `__AVR_CONST_DATA_IN_CONFIG_MAPPED_PROGMEM__`。

3.6.1.5 Const-data-in-progmem 选项

`-mconst-data-in-progmem` 选项用于更改仅使用 `const` 类型限定符定义的对象的存储位置。

默认情况下，仅使用 `const` 类型限定符定义的所有对象均位于程序存储器中且从程序存储器进行访问，该操作可使用`-mconst-data-in-progmem` 选项设为显式。指向 `const` 限定对象的任何指针都将能够读取数据和程序存储器并在运行时确定要访问哪个空间。另请参见 4.4.3. 程序空间中的对象。

`-mno-const-data-in-progmem` 选项用于强制将仅使用 `const` 类型限定符定义的所有对象复制到数据存储器中，如有需要，可以使用不同指令读取数据存储器中的这些对象。对于闪存映射到数据存储器的器件（如 avrxmega3 和 avrtiny 架构），禁止该功能不会影响针对其编译的代码。指向仅使用 `const` 限定的对象的指针将始终读取数据存储器，必须使用通常由`<string.h>`提供的字符串函数的替代函数来访问位于程序存储器中的字符串。

如果使能了该功能且 `const` 限定对象位于程序存储器中并从程序存储器进行访问，则定义宏 `__AVR_CONST_DATA_IN_PROGMEM__`。

将 `const` 限定对象放入程序存储器中将释放更多宝贵的 RAM 资源，并且无需使用非标准关键字，方便指针访问此类对象。就程序大小而言，访问数据存储器中的 `const` 限定对象更为高效。

3.6.1.6 Cpu 选项

`-mcpu=`*`device`* 选项用于指定目标器件或至少指定目标架构系列。

例如：

```
xc8-cc -mcpu=atmega161 main.c
```

要查看可以使用该选项的支持器件的列表，请使用`-mprint-devices` 选项（见 3.6.2.4. Print-devices）。下表列出了可用的架构系列。

表 3-5. 可选架构系列

架构	架构特性
avr1	简单内核，无数据 RAM，仅支持汇编。
avr2	经典内核，最高 8 KB 的程序存储器。
avr25	具有 movw 和 lpm Rx,Z[+]指令的 avr2。
avr3	经典内核，最高 64 KB 的扩展程序存储器。
avr31	经典内核，128 KB 的程序存储器。
avr35	具有 movw 和 lpm Rx,Z[+]指令的 avr3。
avr4	增强型内核，最高 8 KB 的程序存储器。
avr5	增强型内核，最高 64 KB 的程序存储器。
avr51	增强型内核，128 KB 的程序存储器。
avr6	增强型内核，256 KB 的程序存储器。
avrxmega2	XMEGA 内核，最高 64 KB 的程序存储器，最高 64 KB 的数据地址空间。
avrxmega3	XMEGA 内核，程序存储器映射到数据地址空间。
avrxmega4	XMEGA 内核，最高 128 KB 的程序存储器，最高 64 KB 的数据地址空间。
avrxmega5	XMEGA 内核，最高 128 KB 的程序存储器，大于 64 KB 的数据地址空间。
avrxmega6	XMEGA 内核，大于 128 KB 的程序存储器，最高 64 KB 的数据地址空间。
avrxmega7	XMEGA 内核，大于 128 KB 的程序存储器，大于 64 KB 的数据地址空间。
avrtiny	AVR Tiny 内核，16 个寄存器。

3.6.1.7 Dfp 选项

-mdfp=*path* 选项指示应从某个器件系列包（Device Family Pack，DFP）的内容中获取对目标器件（由-mcpu 选项指示）的器件支持，其中 *path* 是该 DFP 的 xc8 子目录的路径。

如果未使用该选项，则 xc8-cc 驱动程序将尽可能在*<installation dir>*/dfp 中提供与编译器一起安装的相关 DFP 的路径。

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 可能包含器件特定的头文件、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。

在搜索标准搜索目录之前，预处理器将在*<DFP>*/xc8/avr/include/目录中搜索包含文件。对于库、启动文件和规范文件，编译器将在搜索标准搜索路径之前搜索*<DFP>*/xc8/avr/和*<DFP>*/xc8/avr/lib/。

3.6.1.8 No-interrupts 选项

-mno-interrupts 选项用于控制更改堆栈指针时是否应禁止中断。

对于大多数器件，状态寄存器 SREG 的状态保存在临时寄存器中，并且在调整堆栈指针前禁止中断。更改堆栈指针后，状态寄存器随之恢复。

如果程序不使用中断，无需通过这种方式保护堆栈调整。使用该选项时将省略用于禁止并可能重新允许中断的代码（在用于调整堆栈指针的代码前后），从而缩短代码长度和执行时间。

由于 AVR XMEGA 器件和具有 8 位堆栈指针的器件可以用原子方式更改由堆栈指针保存的值，因此不需要该选项，并且在针对其中一种器件进行编译时，该选项不起作用。

指定该选项会将预处理器宏__NO_INTERRUPTS__定义为值 1。

3.6.1.9 No-jump-tables 选项

`-fno-jump-tables` 选项用于控制 `switch()` 语句的编码方式。关于该选项的完整详细信息，请参见 4.5.3. switch 语句。

3.6.1.10 Relax 选项

`-mrelax` 选项用于控制长调用和跳转指令的优化，链接时这些调用和跳转指令始终由代码生成器输出为较短和/或较快的相对调用和跳转。只有确定指令的相对形式处于其目标地址范围内时，才能进行这些更改（有关更多信息，请参见 4.3.6.3. 函数指针）。

3.6.1.11 Reserve 选项

`-mreserve=`*`ranges`* 选项用于保留程序通常使用的存储器。该选项的一般形式为：

```
-mreserve=space@start:end
```

其中，*space* 可以是 `ram` 或 `rom`，分别表示数据和程序存储空间；*start* 和 *end* 是地址，表示要排除的范围。例如，`-mreserve=ram@0x100:0x101` 将保留数据存储器中从地址 100h 开始的两个字节。

3.6.1.12 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 4.10.1. 智能 IO 程序。

可以指定工作级别数字，其含义如下表所示。

表 3-6. 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）

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）。在这种情况下，`-msmart-io-format` 选项可用于强制编译器改为链接到复杂度较低的库形式中，该形式的确切功能集使用`-msmart-io-format` 选项（见 3.6.1.13. Smart-io-format 选项）控制。

3.6.1.13 Smart-io-format 选项

`-msmart-io-format="`*`fmt`*`"`选项（其中，*fmt* 是一个包含格式化的、printf 样式转换规范的字符串）用于通知编译器所列出的规范是供智能 IO 函数使用的，并且链接的 IO 库代码必须能够处理这些规范。关于该功能的更多信息，请参见 4.10.1. 智能 IO 程序。

为了缩短代码长度，编译器会根据打印和扫描系列智能 IO 函数的所有调用中所使用格式字符串中包含的转换规范来定制与这些函数相关的库代码。该功能由`-msmart-io` 选项控制。

在某些情况下，编译器无法从格式化的 IO 函数调用中确定使用信息。在这种情况下，可以使用`-msmart-io-format="`*`fmt`*`"`选项来指示链接的 IO 函数所需的转换规范，例如`-msmart-io-format=fmt="%d%i%s"`。如果没有该选项，编译器就不会对 IO 的使用进行假设，并确保将全功能的 IO 函数链接到最终的程序映像中。

以下面对智能 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，编译器无法从格式字符串和参数中推断出任何使用信息。在这些情况下，可以使用-msmart-io-format 选项，该选项有可能允许生成更多最佳格式化的 IO 函数，从而缩短程序的代码长度。例如，如果 fmt1 和 fmt2 所指向的格式字符串只使用"%d"、"%i"和"%s"转换说明符，则应发出-msmart-io-format=fmt="%d%i%s"选项。

fmt 字符串可以包含任何有效的 printf 样式转换规范，其中包括标志和修饰符（例如"%-13.9ls"），并且应该准确反映那些使用不明确的函数所使用的转换规范。如果未在 *fmt* 参数中包含规范，但格式化 IO 函数却使用该参数中的规范，则可能会导致代码失败。

如果 *fmt* 是一个空字符串或不包含任何可识别的转换规范，则将报错。有效转换规范以外的字符是允许使用的，但是会被忽略。

该选项可以在命令行中多次使用。每个选项使用的转换规范会进行累加。

3.6.1.14 Short-calls 选项

-mshort-call 选项用于控制如何对调用进行编码。

当针对程序存储器为 8 KB 以上的器件进行编译时，如果程序执行退出当前函数，编译器将自动使用长跳转和调用指令。这样可以使程序执行至整个存储空间，但程序将较大，需要较长的执行时间。-mshort-calls 选项将强制调用使用目标地址范围有限的 PC 相关指令，例如 rjmp 和 rcall 指令。该选项不会影响通过函数指针实现的间接跳转或调用。

请谨慎使用该选项，因为如果函数超出短指令范围，代码可能会失败。要将函数指针编码为 16 位宽（甚至在具有大容量存储器的器件上），请参见-mrelax 选项（3.6.1.10. Relax 选项）。该选项不会影响 avr2 和 avr4 架构，这两种架构的程序存储器小于 8 KB 并且始终使用短调用/跳转指令。

3.6.1.15 Strict-X 选项

-mstrict-X 选项可确保 X 寄存器（r26-r27）仅用于间接、后递增或预递减寻址。这样会限制寄存器的使用，可能会有利于缩短代码长度。

3.6.1.16 Tiny-stack 选项

-mtiny-stack 选项用于控制堆栈指针的宽度。

在一些数据 RAM 容量较小的器件上，堆栈指针的宽度只有 8 位。而在其他器件上，堆栈指针的宽度为 16 位，有时可能需要单独访问每个字节以更改堆栈指针所指向的位置。

如果器件使用 16 位堆栈指针，而堆栈位于存储区的低半地址部分且长度不超过 256 字节，则该选项将强制堆栈指针仅使用单个字节，从而减少调整堆栈指针所需的代码量。

如果器件所实现的 RAM 容量不超过 256 字节，则将自动应用该选项。

3.6.2 控制输出类型的选项

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

表 3-7. 输出类型控制选项

选项 （链接至说明部分）	产生的结果
-c	中间文件。
-E	预处理文件。
-o *file*	指定名称的输出文件。
-mprint-devices	仅芯片信息。
-S	汇编文件。
-v	详细编译。
-x *language*	假定源文件具有指定内容的输出。
-###	命令行，但不执行编译器应用程序。

.....（续）	
选项 （链接至说明部分）	产生的结果
--help	仅帮助信息。
--version	编译器版本信息。

3.6.2.1 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。

3.6.2.2 E：仅预处理

-E 选项用于生成预处理的 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 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）。

3.6.2.3 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）。

3.6.2.4 Print-devices

-mprint-devices 选项将显示编译器支持的器件的列表。

列出的名称是可以与-mcpu 选项一起使用的器件。该选项将仅显示编译器发布时官方支持的器件。可通过器件系列包（DFP）支持的其他器件不会显示在该列表中。

打印器件列表之后，编译器将会终止。

3.6.2.5 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：

```
xc8-cc -mcpu=atmega3290p -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。

3.6.2.6 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。

3.6.2.7 X：指定源语言选项

-x 语言选项用于指定以下源文件的语言。

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，直到出现会完全关闭后续文件的语言规范的下一个-x 选项或-x none 选项为止。下表列出了允许使用的语言。

表 3-8. 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
cpp-output	预处理 C 源文件
c-header	C 头文件
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：

```
xc8-cc -mcpu=atxmega32d4 -x c-header init.h
```

将创建名为 init.h.gch 的预编译头文件。

3.6.2.8 ###选项

-###选项类似于-v，但不执行命令。使用该选项，无需执行编译器即可查看编译器的命令行。

3.6.2.9 帮助

--help 选项显示有关 xc8-cc 编译器选项的信息，随后驱动程序将终止。

3.6.2.10 版本

--version 选项将打印编译器版本信息，然后退出。

3.6.3 控制 C 语言的选项

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

表 3-9. C 语言控制选项

选项 （链接至说明部分）	控制
-ansi	严格符合 ANSI 标准。
-aux-info *filename*	生成函数原型。
-fno-asm	关键字识别。
-fno-builtin -fno-builtin-*function*	内置函数的使用。
-f[no-]signed-char -f[no-]unsigned-char	普通 char 类型的符号性（为有符号还是无符号类型）。
-f[no-]signed-bitfields -f[no-]unsigned-bitfields	普通 int 位域的符号性。
-mext=*extension*	生效的语言扩展。
-std=*standard*	C 语言标准。

3.6.3.1 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。

3.6.3.2 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。采用 C 语言以外的任何语言时，会默默忽略该选项。

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，声明后的注释中还提供 K&R 样式的参数及其声明的列表。

例如，使用以下命令编译时：

```
xc8-cc -mcpu=atmega3290p -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); /* () */
```

3.6.3.3 Ext 选项

-mext=*extension* 选项用于控制编译期间使用的语言扩展。下表列出了允许使用的扩展。

表 3-10. 可接受的 C 语言扩展

扩展	C 语言说明
xc8	本机 XC8 扩展（默认）
cci	所有 MPLAB XC 编译器均可接受的通用 C 接口

使能 cci 扩展将要求编译器检查所有源代码和编译器选项，确定它们是否符合通用 C 接口（Common C Interface，CCI）。符合该接口的代码会更容易在所有 MPLAB XC 编译器之间移植。不符合 CCI 的代码或选项将通过编译器警告进行标记。

3.6.3.4 No-asm 选项

-fno-asm 选项用于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。

使用该选项时可确保 asm、inline 和 typeof 不被识别为关键字。可以改为使用含义相同的关键字__asm__、__inline__和__typeof__。

-ansi 选项隐含-fno-asm。

3.6.3.5 No-builtin 选项

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

通常，可以避免函数调用的特殊代码是针对很多内置函数生成的。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。

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。

3.6.3.6 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。

3.6.3.7 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 signed char，但如果使用了-mext=cci 选项，则等效于 unsigned char。

-fsigned-char（或-fno-unsigned-char 选项）会强制普通 char 变为有符号类型。

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。

3.6.3.8 Std 选项

-std=*standard* 选项指定编译器假定源代码将符合的 C 标准。下表列出了允许使用的标准和器件。

表 3-11. 可接受的 C 语言标准

标准	支持
c89 或 c90	ISO C90（ANSI）程序
c99	ISO C99 程序

3.6.3.9 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。

3.6.3.10 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 signed char，但如果使用了-mext=cci 选项，则等效于 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。

3.6.4 控制警告和错误的选项

警告就是诊断消息，用于报告虽不是固有错误但指示源代码或其他某种情况异常而可能导致代码运行时失败的构造。

可以使用以-W 开头的选项请求特定警告；例如，使用-Wimplicit 请求与隐式声明有关的警告。其中每个特定警告选项还具有以-Wno-开头、用于关闭警告的否定形式，例如-Wno-implicit。

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 3-12. 所有警告隐含的警告和错误选项

选项 （链接至说明部分）	控制
-f[no-]syntax-only	仅检查代码的语法错误
-pedantic	严格 ANSI C 标准所要求的警告；拒绝所有使用已禁止扩展的程序
-pedantic-errors	-pedantic 隐含的警告，只是将生成错误而非警告
-w	禁止所有警告消息
-W[no-]all	使能所有警告

.....（续）

选项 （链接至说明部分）	控制
-W[no-]address	存储器地址的可疑使用引起的警告
-W[no-]char-subscripts	字符类型数组下标引起的警告
-W[no-]comment	可疑注释引起的警告
-W[no-]div-by-zero	编译时整数被零除引起的警告
-Wformat	printf() 参数不合适引起的警告
-Wimplicit	-Wimplicit-int 和-Wimplicit-function-declaration 隐含的警告
-Wimplicit-function- declaration	使用未声明函数引起的警告
-Wimplicit-int	未指定类型的声明引起的警告
-Wmain	异常主定义引起的警告
-Wmissing-braces	缺少大括号引起的警告
-Wno-multichar	多字符常量引起的警告
-Wparentheses	缺少优先顺序引起的警告
-Wreturn-type	缺少返回类型引起的警告
-Wsequence-point	序列点违例引起的警告
-Wswitch	缺少或多余 case 值引起的警告
-Wsystem-headers	系统头文件中的代码引起的警告
-Wtrigraphs	使用三字符组引起的警告
-Wuninitialized	使用未初始化的变量引起的警告
-Wunknown-pragmas	使用未知 pragma 伪指令引起的警告
-Wunused	未使用对象和构造引起的警告
-Wunused-function	未使用静态函数引起的警告
-Wunused-label	未使用标号引起的警告
-Wunused-parameter	未使用参数引起的警告
-Wunused-variable	未使用变量引起的警告
-Wunused-value	未使用值引起的警告

表 3-13. 并非所有警告隐含的警告选项

选项	控制
-Wextra	生成附加警告消息
-Waggregate-return	返回的聚合对象引起的警告
-Wbad-function-cast	函数强制转换为非匹配类型引起的警告
-Wcast-qual	丢弃的指针限定符引起的警告
-Wconversion	可以更改值的隐式转换引起的警告
-Werror	针对可疑构造生成错误而非警告

.....（续）

选项	控制
-Winline	无法内联函数时的警告
-Wlarger-than=*len*	定义大对象时的警告
-W[no-]long-long	使用 long long 引起的警告
-Wmissing-declarations	未声明函数时的警告
-Wmissing- format-attribute	缺少格式属性引起的警告
-Wmissing-noreturn	不返回属性可能遗漏引起的警告
-Wmissing-prototypes	未使用原型声明函数时的警告
-Wnested-externs	extern 声明引起的警告
-Wno-deprecated- declarations	是否针对弃用的声明生成警告
-Wpointer-arith	获取未确定大小的类型的大小时的警告
-Wredundant-decls	冗余声明引起的警告
-Wshadow	局部对象隐藏其他对象（局部对象与其他对象同名）时的警告
-W[no-]sign-compare	有符号比较引起的警告
-Wstrict-prototypes	K&R 函数声明引起的警告
-Wtraditional	传统差异引起的警告
-Wundef	未定义标识符引起的警告
-Wunreachable-code	无法达到的代码引起的警告
-Wwrite-strings	使用非 const 字符串指针时的警告

3.6.4.1 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。

3.6.4.2 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。

3.6.4.3 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。

3.6.4.4 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。

3.6.4.5 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。

3.6.4.6 Werror 选项

-Werror 选项具有所有被视为错误的警告消息，因此在触发警告的情况依然存在时也能阻止编译完成。

3.6.4.7 Extra 选项

-Wextra 选项在以下情况下生成额外警告。

- 非易失性自动变量可通过调用 longjmp 进行更改。这些警告只能出现在优化编译时。编译器只能发现对 setjmp 的调用，无法获知 longjmp 的调用位置。实际上，信号处理程序可在代码中的任意点对其进行调用。因此，即使实际上没有问题，也可能生成警告，因为实际上无法在会引起问题的位置调用 longjmp。
- 函数可以通过 return *value*;和 return;退出。完成函数主体而未传递任何返回语句被视为 return;。
- 表达式语句或逗号表达式的左侧不包含副作用。要禁止警告，请将未使用的表达式强制转换为 void。例如，x[i,j]等表达式会引起警告，但 x[(void)i,j]不会。
- 无符号值使用<或<=与 0 进行比较。
- x<=y<=z 等比较将出现。这相当于(x<=y ? 1 : 0) <= z，这是一种不同于普通数学符号的解释。
- static 等存储类说明符并非声明中的关键所在。根据 C 标准，该用法即将过时。
- 如果还指定了-Wall 或-Wunused，则将发出有关未使用参数的警告。
- 当有符号值转换为无符号值时，比较有符号值与无符号值可能得到不正确的结果（但如果还指定了-Wno-sign-compare，则不会发出警告）。
- 聚合包含部分括上括号的初始化程序。例如，以下代码会引起此类警告，因为 x.h 的初始化程序前后缺少大括号：

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

- 聚合包含的初始化程序并不初始化所有成员。例如，以下代码会引起此类警告，因为 x.h 会隐式初始化为 0：

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

3.6.5 调试选项

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 3-14. 调试选项

选项 （链接至说明部分）	控制
-g	生成的调试信息的类型。
-mchp-stack-usage	生成堆栈使用信息和警告。
-mcodecov	插装输出以提供代码覆盖信息。
-Q	打印编译每个函数时与该函数相关的诊断信息以及编译结束时针对每次编译的统计信息
-save-temps[=*dir*]	编译后是否应保留中间文件以及保留位置。

3.6.5.1 G：生成调试信息选项

-g*format* 选项用于指示编译器生成附加信息，硬件工具可以使用这些信息来调试程序。

下表列出了支持的格式。

表 3-15. 支持的调试文件格式

格式	调试文件格式
-g*level*	生成的调试信息量，其中 *level* 是一个介于 0 和 3 之间的的值。
-gdwarf-3	ELF/DWARF 发行版 3。

默认情况下，编译器生成 DWARF 发行版 2 文件。除了指定调试格式的-g 选项之外，还可以使用该选项的-g*level* 形式。

编译器支持在使能优化器的情况下使用该选项，从而可以对经过优化的代码进行调试；但是，优化代码中采用的捷径有时可能会产生意外的结果，例如不存在的变量以及发生意外更改的流控制。

3.6.5.2 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 4.2.9. 堆栈指导。

3.6.5.3 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 4.2.8. 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。

3.6.5.4 Q：打印函数信息选项

-Q 选项用于指示编译器在编译每个函数时打印函数的名称，并在编译完成时打印每次编译的统计信息。

3.6.5.5 Save-temps 选项

-save-temps 选项用于指示编译器在编译完成后保留临时文件。

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.s 将产生 foo.i、foo.s 和 foo.o 目标文件。

-save-temps=cwd 选项与-save-temps 等效。

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。

```
xc8-gcc -save-temps=obj -c bar.c -o dir/xbar.o
```

3.6.6 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。

表 3-16. 一般优化选项

选项 （链接至说明部分）	需要许可证	编译时的优化
-O0	全部	无优化（默认）
-O1 -O	全部	优化级别 1
-O2	全部	优化级别 2
-O3	仅 PRO	针对速度的优化
-Og	全部	更好调试
-Os	仅 PRO	针对代码大小的优化
-fdata-sections	全部	每个数据对象分配给其自己的段
-f[no-]fat-lto-objects	仅 PRO	目标代码和内部表示均写入目标文件
-ffunction-sections	全部	每个函数分配给其自己的段
-flto	仅 PRO	标准链接时优化器
-flto-partition=*algorithm*	仅 PRO	运行链接时优化器时通过指定算法对目标文件进行分区

.....（续）

选项 （链接至说明部分）	需要许可证	编译时的优化
-fomit-frame-pointer	全部	优先使用堆栈指针而非帧指针
-mno-pa-on-file=*filename*	PRO	针对指定文件禁止过程抽象
-mno-pa-on-function=*function*	PRO	针对指定函数禁止过程抽象
-mno-pa-outline-calls	PRO	针对各个函数调用禁止过程抽象
-fsection-anchors	全部	相对于一个符号访问静态对象
-funroll-[all-]loops	全部	展开循环以提高执行速度
-fuse-linker-plugin	仅 PRO	支持链接时优化的链接器插件
-fwhole-program	仅 PRO	整个程序的优化
-m[no-]gas-isr-prologues	全部	针对小型中断服务程序优化了现场切换代码
-mpa-callcost-shortcall	仅 PRO	更高程度的过程抽象
-mpa-iterations=*n*	仅 PRO	指定数量的过程抽象迭代
--nofallback	全部	仅应用所选的优化级别，即使没有许可证也不会被强制回退到较低级别

3.6.6.1 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成本并使调试产生预期结果。

使用该优化级别进行编译时，各语句之间是相互独立的。如果在语句之间使用断点来停止程序，则随后可以为任何变量分配一个新值，或者将程序计数器更改至函数中的任何其他语句，最终从源代码中获得预期的结果。

编译器只在寄存器中分配使用 register 声明的变量。

3.6.6.2 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。

3.6.6.3 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。

3.6.6.4 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。

3.6.6.5 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。

3.6.6.6 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。

此外，该选项还会执行进一步的优化，旨在缩短代码长度，但可能会减缓程序执行速度，例如过程抽象优化。

只有具备编译器许可证才能使用该优化级别。

3.6.6.7 Data-sections 选项

-fdata-sections 选项用于将每个对象放入其自己的段中，以帮助垃圾收集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收集结合使用时（使用-Wl,--gc-sections 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。

3.6.6.8 Fat-lto-objects 选项

-ffat-lto-objects 选项用于请求编译器生成写入惟一 ELF 段的 fat 目标文件，其中包含目标代码和 GIMPLE（GCC 的内部表示之一）。如果库代码可能链接使用和不使用标准链接时优化器（由-flto 选项控制）的项目，此类目标文件会很有用。

该选项的-fno-fat-lto-objects 形式（未指定选项时的默认形式）用于禁止将目标代码包含到目标文件中，从而加快编译。但是，此类目标文件必须始终使用标准链接时优化器进行链接。

3.6.6.9 Function-sections 选项

-ffunction-sections 选项用于将每个函数放入其自己的段中，以帮助垃圾收集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收集结合使用时（使用-Wl,--gc-sections 使能），最终输出可能较小。使用该选项会产生更细粒度的段，这些段还可能帮助链接器更好地将相关段分配到一起，如果还使用了-mrelax 选项，则可能会进一步缩短代码长度。但是，-ffunction-sections 选项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。

3.6.6.10 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代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。

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

```
xc8-cc -c -O3 -flto -mcpu=atmega3290p foo.c
xc8-cc -c -O3 -flto -mcpu=atmega3290p bar.c
xc8-cc -o myprog.elf -flto -O3 -mcpu=atmega3290p foo.o bar.o
```

使能链接时优化的另一种（更简单）的方法是：

```
xc8-cc -o myprog.elf -flto -O3 -mcpu=atmega3290p foo.c bar.c
```

3.6.6.11 Lto-partition 选项

-flto-partition=*algorithm* 选项用于控制运行链接时优化器时对目标文件进行分区所使用的算法。参数 none 用于完全禁止分区，直接从整个程序分析（Whole Program Analysis，WPA）阶段执行链接时优化步骤。该工作模式将得出最优结果，但代价是需要更大的编译器存储器和更长的编译时间，而小程序不太可能出现这种问题。对目标文件进行分区可以提高编译性能。参数 one 指定只应使用一个分区，参数 1to1 则指定对原始源文件指定的镜像进行分区。默认参数为 balanced，用于指定尽可能划分为大小相等的块。

3.6.6.12 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。

3.6.6.13 No-pa-on-file 选项

-mno-pa-on-file=*filename* 选项用于禁止指定目标文件或库归档文件中包含的代码的过程抽象。如果指定文件未处理，则不会产生警告。

3.6.6.14 No-pa-on-function 选项

-mno-pa-on-function=*function* 选项用于禁止指定函数中包含的代码的过程抽象。如果指定函数不存在，则不会产生警告。

3.6.6.15 No-pa-outline-calls 选项

-mno-pa-outline-calls 选项用于阻止过程抽象优化概括包含函数调用的代码。

编译器会执行各种检查来确保在使能过程抽象优化时安全概括代码块。其中一项检查是为了确保块中调用的任何程序均不会访问堆栈中的对象，例如基于堆栈的参数或返回值。如果概括了对此类程序的调用，则对编译器插入的抽象程序的额外调用（将在返回地址压入堆栈时更改堆栈指针）在一些情况下可能会导致读取错误堆栈数据。编译器不会概括包含对堆栈访问程序的调用的代码，但将允许在其他时间概括调用。-mno-pa-outline-calls 选项用于禁止包含调用的代码的过程抽象，甚至在编译器确信此类优化安全时也是如此。

3.6.6.16 Section-anchors 选项

-fsection-anchors 选项允许相对于一个符号访问 static 对象。

使能该选项时生成的代码可能会将多个 static 对象作为相对于一个基址的偏移进行访问，而非单独访问每个对象。很显然，该优化只能在程序中定义了多个 static 对象时缩短代码长度。尽管该选项在没有编译器许可证的情况下也能使用，但其实它最适合只能通过 PRO 许可证实现的 s 级优化（-Os）。

3.6.6.17 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分支延时。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。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。

3.6.6.18 Use-linker-plugin 选项

-fuse-linker-plugin 选项允许在链接时优化期间使用链接器插件，通过向链接时优化器公开更多代码来提高优化的质量。-fwhole-program 选项不应与-fuse-linker-plugin 选项一起使用。

3.6.6.19 Whole-program Optimizations 选项

-fwhole-program 选项用于运行更高程度的过程间优化。

该选项假定当前编译单元代表编译的整个程序。所有公共函数和变量（main()以及通过属性 externally_visible 合并的函数和变量除外）均变为 static 函数且实际上由过程间优化器进行更高程度的优化。当使用该选项等效于针对包含单个文件的程序正确使用 static 关键字加上使用选项-flto 时，该标志可用于编译很多小型程序，因为函数和变量变为整个组合编译单元（而非单个源文件本身）的局部函数和变量。

整个程序优化不应与-fuse-linker-plugin 链接时优化选项结合使用。

3.6.6.20 ISR Prologues 选项

-m[no-]gas-isr-prologues 选项用于控制针对小型中断服务程序（Interrupt Service Routine，ISR）生成的现场切换代码。

如果未使能该功能，编译器将保存寄存器 R0、R1 和 SREG，并用与 ISR 相关的现场保存代码清零寄存器 R1 以及用现场恢复代码恢复这些寄存器。使能该功能时，汇编器将扫描 ISR 中的寄存器使用情况，并且仅在需要时保存这些寄存器。对于小型 ISR，结果将是中断代码更小且执行速度更快。该功能可以安全地与其他编译器功能结合使用，例如代码覆盖、堆栈指导和过程抽象，这些功能将继续按预期工作。

该功能在 1 级和更高级优化下自动使能（使能-Og 选项时除外）。可以使用-mno-gas-isr-prologues 选项针对整个程序禁止该功能；也可以使用包含相应 ISR 定义的 no_gccisr 函数属性针对特定 ISR 禁止该功能。

3.6.6.21 Pa-callcost-shortcall 选项

`-mpa-callcost-shortcall` 选项用于执行更高程度的过程抽象，以便链接器能够舒缓长调用并缩短代码长度。但是，如果未满足基本假设，该选项可能增大代码长度，因此需要监视其对于每个项目是否有益。

3.6.6.22 Pa-iterations 选项

`-mpa-iterations=`*n* 选项可将过程抽象迭代数从默认值 2 更改为其他值。更多迭代可能会缩短代码长度，但也可能影响代码编译时间。

3.6.6.23 Nofallback 选项

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

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。

3.6.7 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 3-17. 预处理器选项

选项 （链接至说明部分）	控制
`-C`	保留注释
`-d`*letters*	保留宏定义
`-D`*macro* `-D`*macro=defn*	定义宏
`-fno-show-column`	省略诊断中的行编号
`-H`	打印头文件名称
`-imacros` *file*	仅包含文件宏定义
`-include` *file*	包含文件
`-iquote`	指定加引号的包含文件搜索路径
`-M`	生成 make 规则
`-MD`	将相依性关系信息写入文件
`-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`	头文件搜索中忽略的系统目录
`-P`	不要生成#line 伪指令
`-trigraphs`	支持三字符组
`-U`*macro*	取消定义宏

.....（续）	
选项 （链接至说明部分）	控制
-undef	不要预定义非标准宏

3.6.7.1 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。

3.6.7.2 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。

下表列出了-d 可接受的字母参数。其他参数将被默默忽略。

表 3-18. 预处理器调试信息

字母	产生的结果
D	全部宏定义
M	仅在预处理结束时生效的宏定义
N	不含参数和内容的宏定义

3.6.7.3 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* 且指定替换文本的预处理器宏。其用途相当于在每个正在编译的模块的顶部放置#define *macro text*。

该选项的任何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 4.13.1.1. 预处理器运算。

将宏定义为 C 字符串字面值时需要对字符串使用的双引号字符（" "）进行转义。如果要包含双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，则应在字符串两端再加一对双引号。

例如，要传递 C 字符串"hello world"（替换文本中包含双引号字符和空格），应使用-DMY_STRING="\"hello world\""。另外，也可以在整个选项两端加双引号："-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双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。

3.6.7.4 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。

3.6.7.5 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。由于文件生成的输出会被丢弃，因此该选项的惟一作用就是使文件中定义的宏可用于主输入。

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 选项均按照写入的顺序处理。

3.6.7.6 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，*file* 的内容先进行编译。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 选项均按照写入的顺序处理。

3.6.7.7 Iquote 选项

-iquote *dir* 选项用于在预处理期间将目录 *dir* 添加到供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，例如#include "file"。

3.6.7.8 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。

规则列表打印到标准输出，而非预处理的 C 程序。

-M 选项隐含-E。

3.6.7.9 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，只是相依性关系信息写入文件后会继续编译。包含相依性关系信息的文件与扩展名为.d 的源文件同名。

3.6.7.10 MF：指定相依性关系文件选项

-MF *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。

3.6.7.11 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，则还必须指定-M 或-MM。-MD 或-MMD 不支持该选项。

3.6.7.12 MM：为加引号的头文件生成 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。

3.6.7.13 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。

3.6.7.14 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，这些虚假规则可以解决 make 错误。

典型输出如下：

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

3.6.7.15 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。

3.6.7.16 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。结果即为目标。-MT 选项用于将目标设置为指定的字符串。如果想要多个目标，可以将其指定为-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
```

3.6.7.17 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。

3.6.7.18 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）。

结合使用-nostdinc 与-iquote 可以将 include-file 搜索路径限定为仅显式指定的目录。

3.6.7.19 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。

3.6.7.20 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。

3.6.7.21 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。

3.6.7.22 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）。

3.6.8 汇编选项

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 3-19. 汇编器选项

选项 （链接至说明部分）	控制
-Wa,*option*	传递给汇编器的选项。
-Xassembler *option*	传递给汇编器的选项。

3.6.8.1 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。

3.6.8.2 Xassembler 选项

-Xassembler *option* 选项将 *option* 传递给汇编器，在此它将被解析为汇编器选项。该选项可用于提供编译器不知道如何识别或-Wa 选项无法解析的系统特定的汇编器选项。

3.6.9 映射汇编器选项

下表列出了常用的汇编器选项。

表 3-20. 映射汇编器选项

选项	控制
-Wa,-a=*file.lst*	生成指定的汇编列表文件

3.6.10 链接选项

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，将不会运行链接器。

表 3-21. 链接选项

选项 （链接至说明部分）	控制
-l*library*	扫描的库文件
-nodefaultlibs	库代码是否链接项目
-nostartfiles	是否链接运行时启动模块
-nostdlib	库和启动代码是否链接项目
-s	从可执行代码中移除所有符号表和重定位信息。
-u	链接库模块以定义符号。可以合法地通过不同符号多次使用-u 来强制装入额外的库模块。
-Wl,*option*	要传递给链接器的选项。
-Xlinker *option*	要传递给链接器的系统特定选项

3.6.10.1 L：指定库文件选项

-l*library* 选项用于在链接时扫描名为 library 的库中的未解析符号。

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。搜索的目录包括几个标准系统目录，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 libz.a。如果 bar.o 引用 libz.a 中的函数，则无法装入这些函数。

通常，这种方式找到的文件为库文件（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普通目标文件，则该文件以通常方式链接。

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。

3.6.10.2 Nodefaultlibs 选项

-nodefaultlibs 选项可阻止标准系统库链接到项目。只有指定的库才被传递给链接器。

3.6.10.3 Nostartfiles 选项

-nostartfiles 选项可阻止运行时启动模块链接到项目。

3.6.10.4 Nostdlib 选项

-nostdlib 选项可阻止标准系统启动文件和库链接到项目。不传递启动文件给链接器，只传递指定的库给链接器。

编译器可生成对 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。指定了此选项时，应通过某个其他机制提供这些入口点。

3.6.10.5 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。

3.6.10.6 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。

3.6.10.7 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，在此它将被解析为链接器选项。如果 *option* 包含逗号，则表示以逗号分隔的多个选项。

3.6.10.8 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。该选项可用于提供编译器不知道如何识别的系统特定的链接器选项。

3.6.11 映射链接器选项

下表列出了常用的链接器选项。

表 3-22. 映射链接器选项

选项	控制
-Wl,--[no-]data-init	生成.dinit 段和数据初始化
-Wl,--gc-sections	通过垃圾收集回收未使用的存储器
-Wl,-Map=*mapfile*	生成链接器映射文件
-Wl,--section-start=*section=addr*	将自定义命名的段置于指定地址处。它无法用于放置必须使用-Wl,-T 选项放置的标准段，例如.data、.bss 和.text。
-Wl,-T*script*	链接时使用的链接描述文件
-Wl,-T*section=address*	分配到标准段（.data、.bss 和.text）的地址（见 4.14.2. 更改和链接所分配的段）

3.6.12 目录搜索选项

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 3-23. 目录搜索选项

选项 （链接至说明部分）	控制
-idirafter *dir*	搜索系统路径后更多的头文件搜索目录
-I*dir*	预处理器包含文件搜索目录
-L*dir*	库搜索目录
-nostdinc	头文件搜索目录

3.6.12.1 Idirafter 选项

-idirafter *dir* 选项用于将目录 *dir* 添加到第二个包含路径中。主包含路径中的任何目录中均未找到头文件时，将搜索第二个包含路径中的目录（包括-I 指定的目录）。

3.6.12.2 I：指定包含文件搜索路径选项

-I*dir* 选项将目录 *dir* 添加到供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。

3.6.12.3 L：指定库搜索路径选项

-L*dir* 选项用于为已使用-l 选项指定的库文件指定一个供搜索的其他目录。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。

3.6.12.4 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）。

结合使用-nostdinc 与-iquote 可以将 include-file 搜索路径限定为仅显式指定的目录。

3.6.13 代码生成约定选项

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 3-24. 代码生成约定选项

选项 （链接至说明部分）	控制
-fshort-enums	enum 类型的长度

3.6.13.1 Short-enums 选项

-fshort-enums 选项用于将最小可能整型类型分配给 enum，以便可以容纳可能值的范围。使用该选项生成的代码与不使用该选项生成的代码无法二进制兼容。

3.7 MPLAB X IDE 集成

可以将 8 位语言工具集成到 MPLAB X IDE 中并由其控制，从而为 8-bit AVR MCU 器件系列提供了基于 GUI 的应用程序代码开发工具。

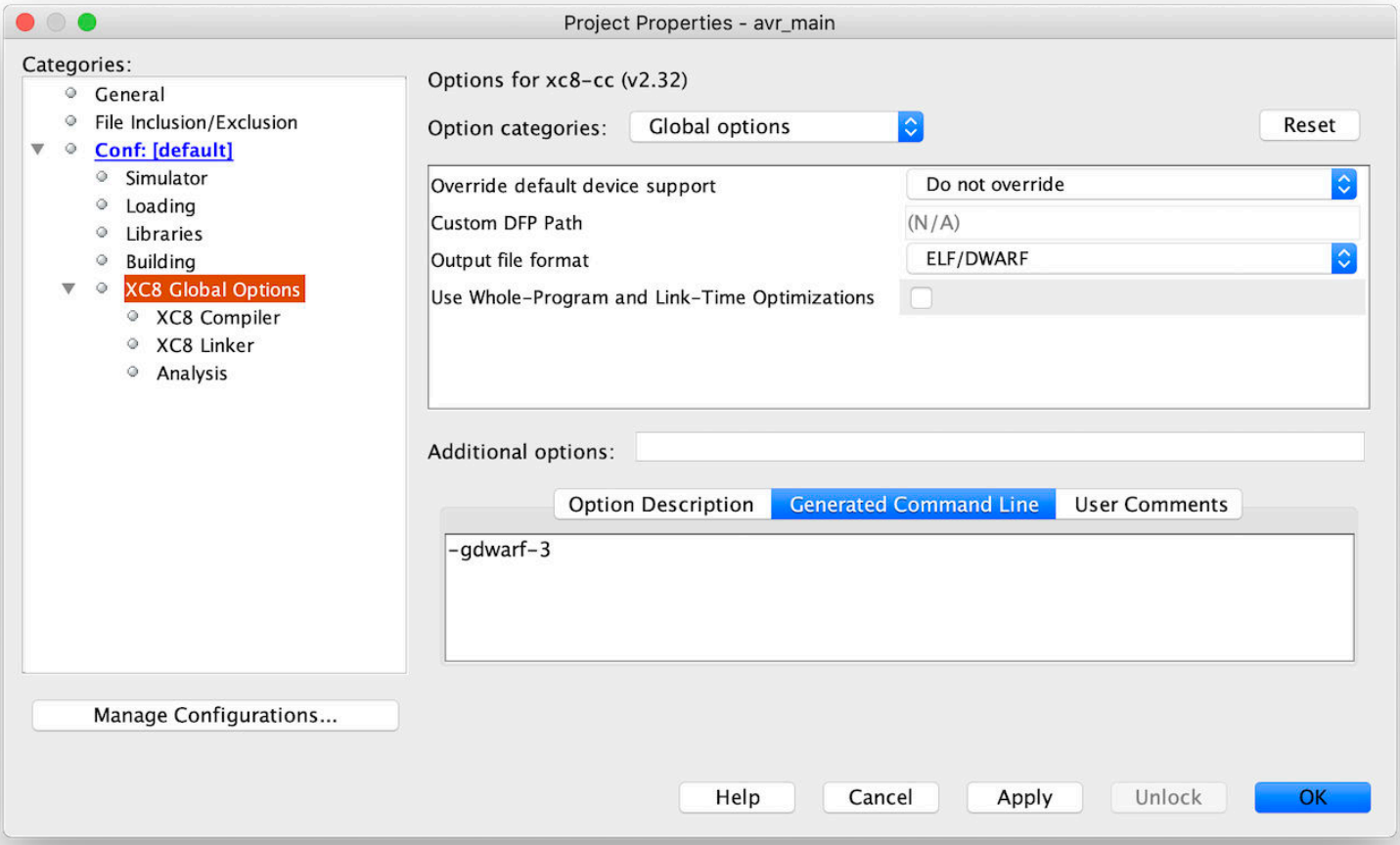
有关 IDE 安装以及创建和设置项目以使用 MPLAB XC8 C 编译器的信息，请参见《MPLAB® X IDE 用户指南》。

3.7.1 MPLAB X IDE 等效选项

下文中的说明将 MPLAB X IDE 的 **Project Properties** 中的控件与 MPLAB XC8 命令行驱动程序选项相对应。本用户指南的相关章节中提供了参考信息，可详细了解选项功能。在 IDE 中，单击任意选项时，可以看到在对话框下半部分的 **Option Description**（选项说明）字段中会显示联机帮助和示例。

3.7.1.1 XC8 Global Options（XC8 全局选项）——Global options（全局选项）

图 3-2. XC8 Global Options——Global Options



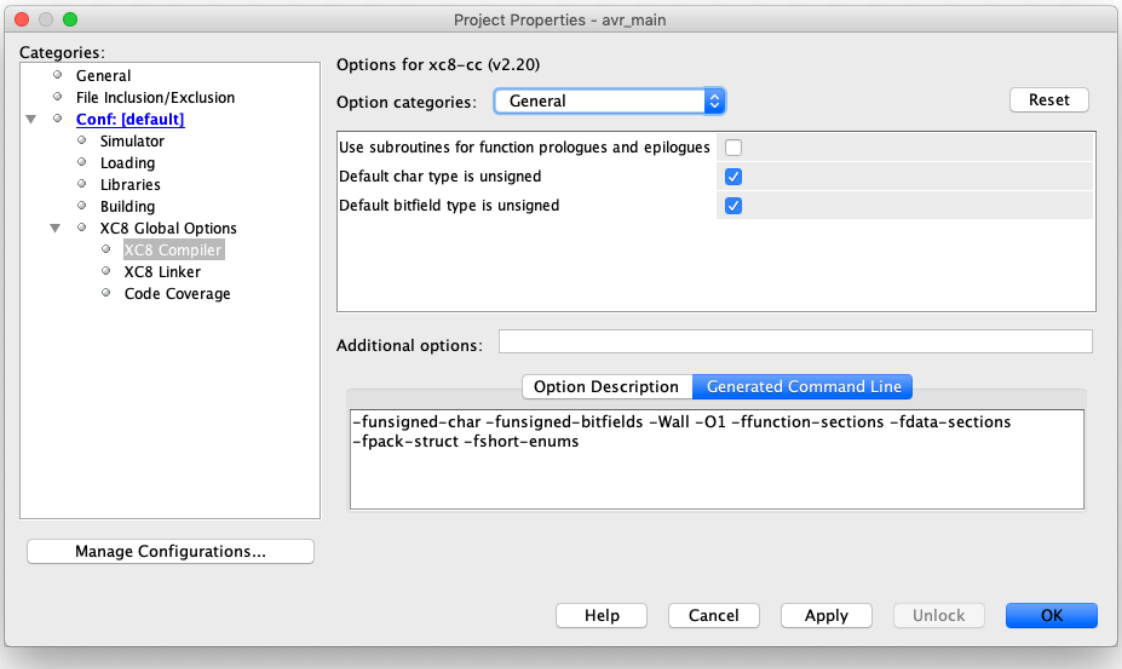
Override default device support（改写默认器件支持） 选中该复选框将使能下面的 **Custom DFP Path**（自定义 DFP 路径）字段，该字段用于指定备用器件系列包的位置。DFP 提供器件特定信息，例如项目中的代码可使用的寄存器名称和地址。

Output file format（输出文件格式） 该选项用于指定输出文件类型。请参见 3.6.5.1. G：生成调试信息选项。

Use Whole-program and Link-Time Optimizations（使用整个程序优化和链接时优化） 选中该复选框将对整个组合编译单元运行更高程度的优化。请参见 3.6.6.10. Lto 选项和 3.6.6.19. Whole-program Optimizations 选项。

3.7.1.2 XC8 Compiler（XC8 编译器）——General（常规）选项

图 3-3. XC8 Compiler——General 选项

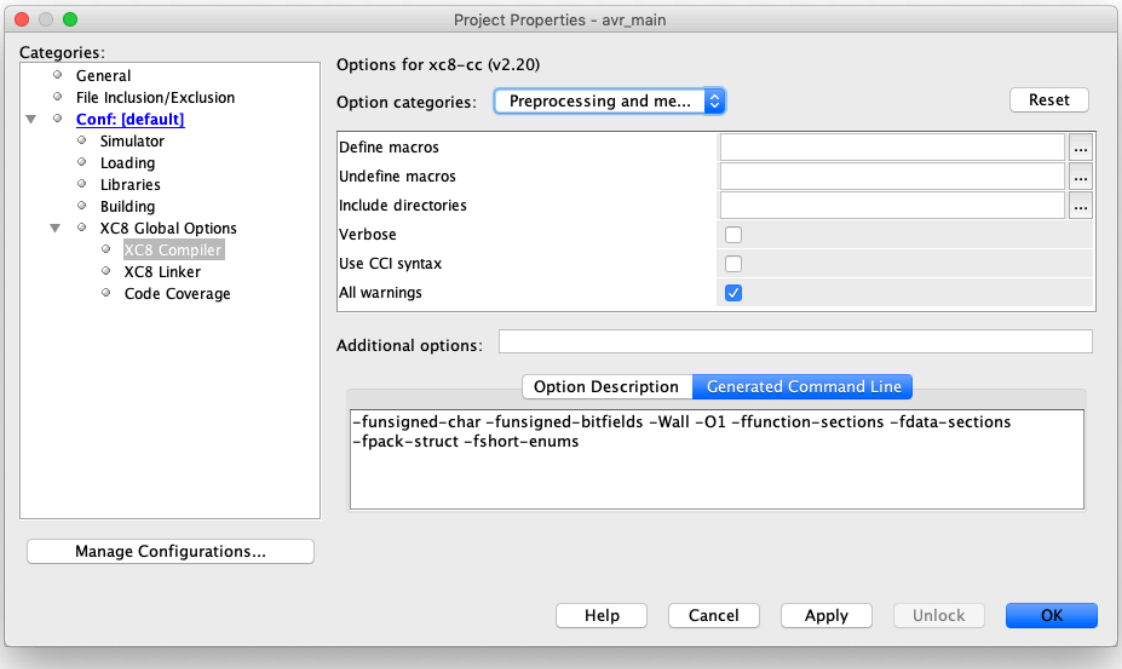


Default char type is unsigned（默认 char 类型为无符号） 使能该复选框指示普通 `char` 类型将被视为 `unsigned char`。请参见 3.6.3.7. Signed-char 选项和 3.6.3.10. Unsigned-char 选项。

Default bitfield type is unsigned（默认位域类型为无符号） 使能该复选框指示普通位域将被视为无符号对象。请参见 3.6.3.6. Signed-bitfields 选项和 3.6.3.9. Unsigned-bitfields 选项。

3.7.1.3 XC8 Compiler——Preprocessing and Messaging（预处理和消息发送）选项

图 3-4. XC8 Compiler——Preprocessing and Messaging 选项



Define macros（定义宏） 该字段可用于定义预处理器宏。请参见 3.6.7.3. D：定义宏。

Undefine macros（取消定义宏） 该字段可用于取消定义预处理器宏。请参见 3.6.7.21. U：取消定义宏。

Include directories（包含目录） 该字段可用于指定头文件的搜索目录。请参见 3.6.12.2. I：指定包含文件搜索路径选项。

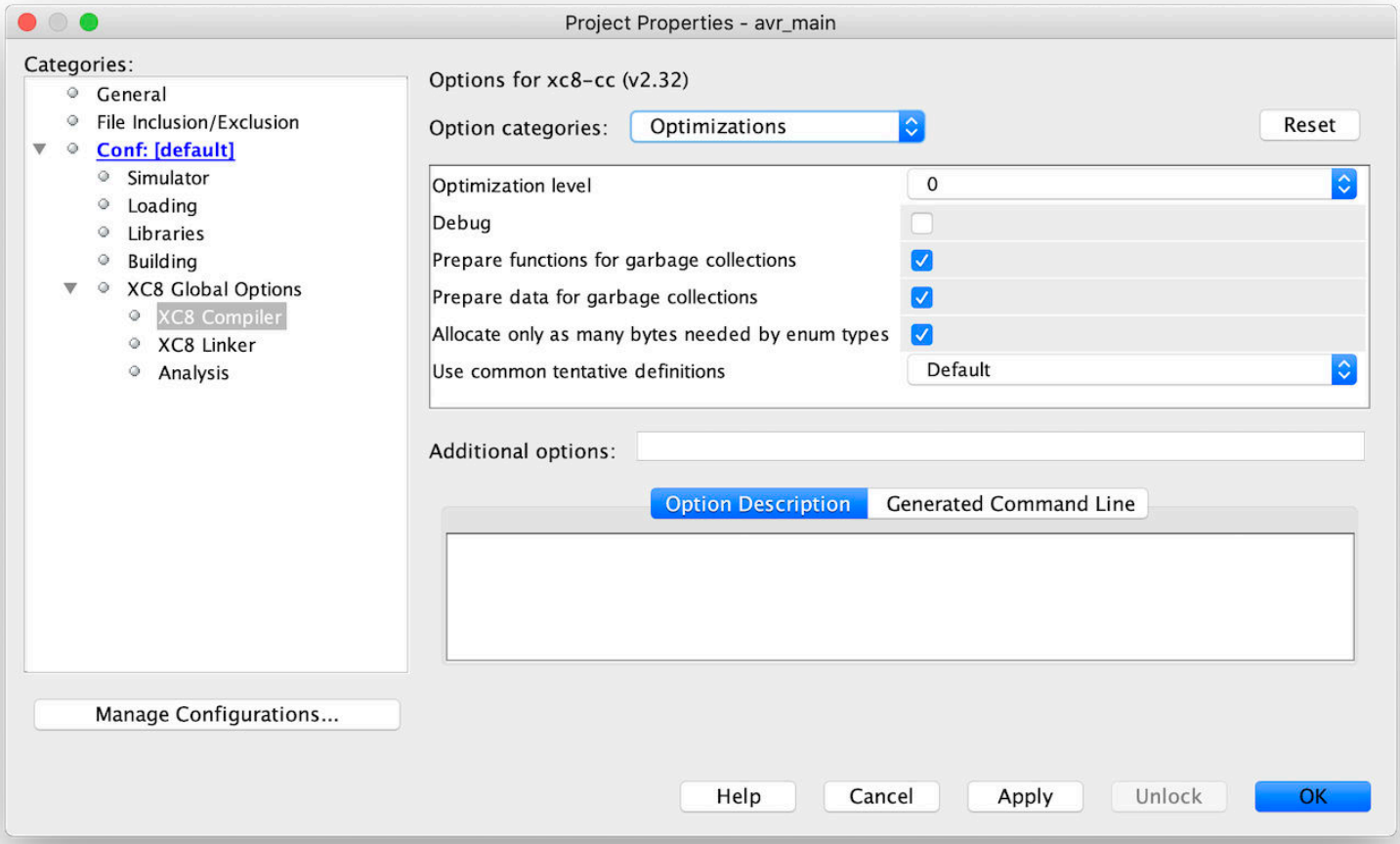
Verbose（详细） 使能该复选框将显示编译项目时所使用的命令行。请参见 3.6.2.6. V：详细编译。

Use CCI syntax（使用 CCI 语法） 使能该复选框将请求强制使用 CCI 语言扩展。请参见 3.6.3.3. Ext 选项。

All warnings（所有警告） 使能该复选框将请求在编译时不禁止编译器警告消息。请参见 3.6.4.5. Wall 选项。

3.7.1.4 XC8 Compiler——Optimizations（优化）选项

图 3-5. XC8 Compiler——Optimizations 选项



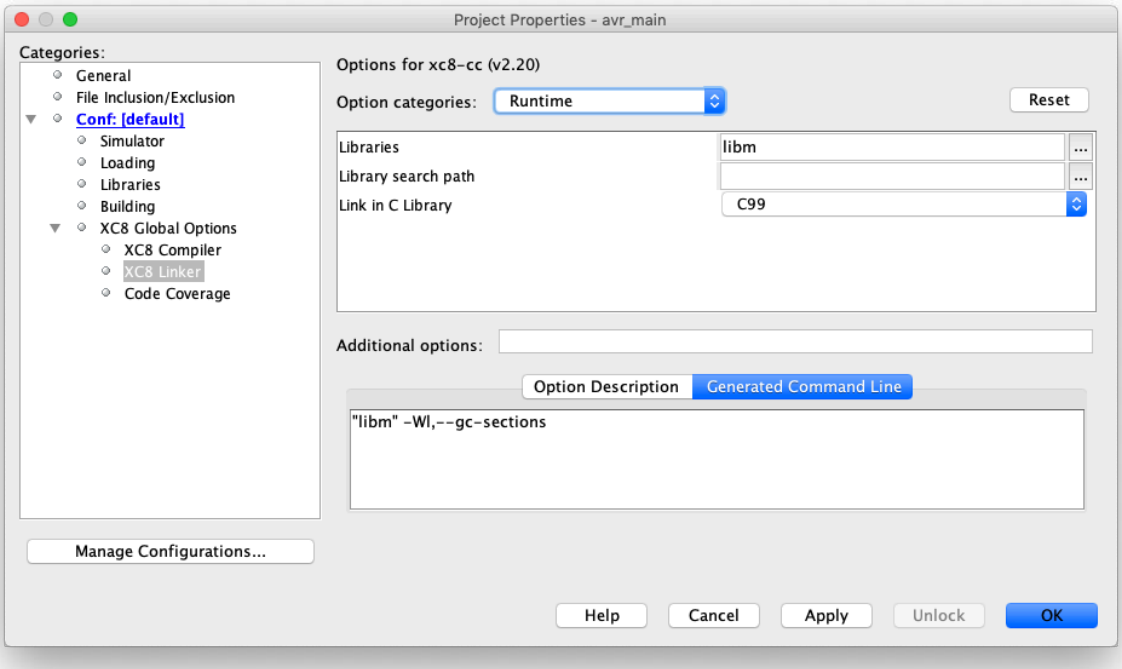
Optimization level（优化级别） 该选项用于控制程序的优化级别。请参见 3.6.6. 控制优化的选项。

Debug（调试） 该复选框用于禁止可能影响代码调试能力的高程度优化。请参见 3.6.6.5. Og：更好调试选项。

其他优化 该对话框中的其余选项用于控制可优化所生成代码的特定代码生成功能。

3.7.1.5 XC8 Linker（XC8 链接器）——Runtime（运行时）选项

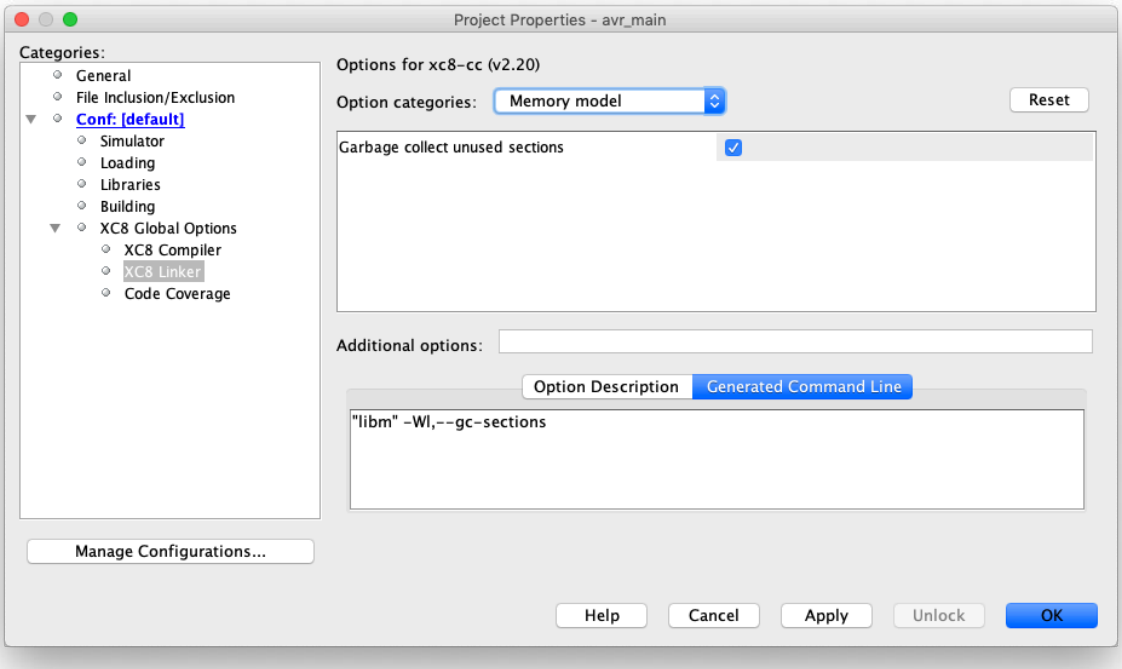
图 3-6. XC8 Linker——Runtime 选项



Libraries（库）	该字段用于指定要链接的更多库的名称。
Library search path（库搜索路径）	该字段用于指定编译器查找库文件时所使用的搜索路径。请参见 3.6.12.3. L：指定库搜索路径选项。
Link in C library（链接 C 库）	该选项用于指定是否将链接标准库。请参见 3.6.10.2. Nodefaultlibs 选项。

3.7.1.6 XC8 Linker——Memory Model（存储器模型）选项

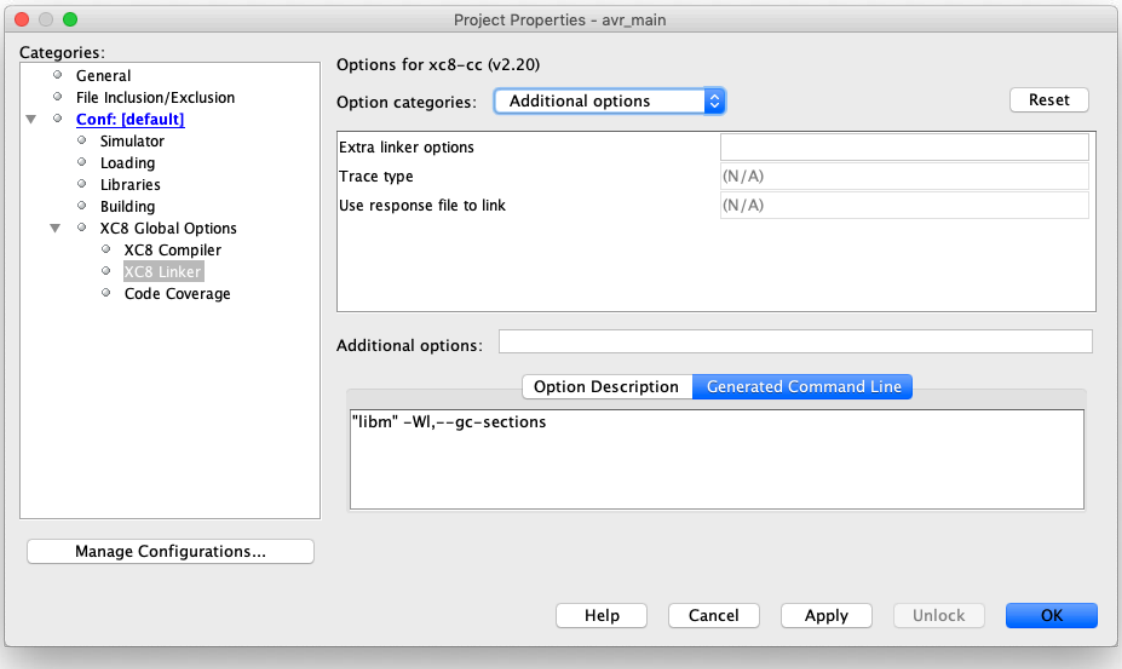
图 3-7. XC8 Linker——Memory Model 选项



Garbage collect unused sections（对未使用的段进行垃圾收集） 该选项用于指定链接器是否应将其视为未使用的段移除。

3.7.1.7 XC8 Linker——Additional options（额外选项）

图 3-8. XC8 Linker——Additional options



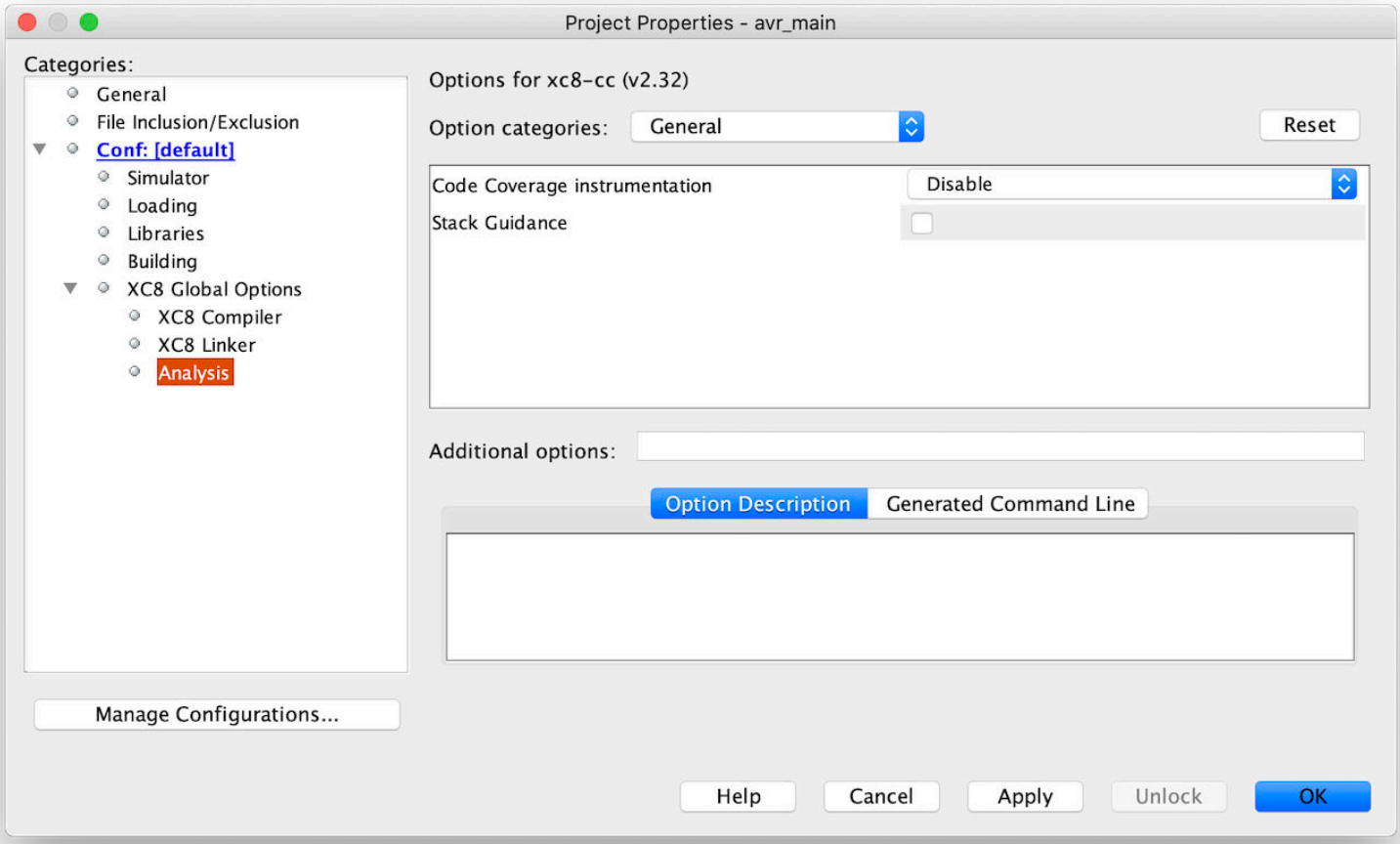
Extra linker options（额外的链接器选项） 该字段可用于指定无法从 IDE 控制的额外的链接器相关选项。请参见 3.6.10.7. Wl：将 Option 传递给链接器选项。

Trace type（跟踪类型） 该选项尚未实现。

Use response file to link（使用响应文件进行链接） 该字段允许编译器在链接步骤期间使用命令行选项文件（优先于项目属性中的其他链接步骤设置）。请参见 3.1.1.1. 长命令行。只有在 Windows 下运行 MPLAB X IDE 时，该选项才适用。

3.7.1.8 XC8 Analysis（分析）——General

图 3-9. XC8 Analysis——General 选项



Code Coverage instrumentation（代码覆盖插装） 该选项用于控制是否在程序输出中插装汇编器序列，这些序列用于记录该序列所表示的代码的执行情况，且其数据可以方便项目源代码执行程度的分析。请参见 3.6.5.3. Codecov 选项和 4.2.8. 代码覆盖。

Stack Guidance（堆栈指导） 该复选框用于为获得许可的 PRO 编译器使能编译器的堆栈指导功能。该功能可估算程序所用的任何堆栈的最大深度，并打印一个报告。请参见 3.6.5.2. 堆栈指导选项和 4.2.9. 堆栈指导。

3.8 Microchip Studio 集成

可以将 8 位语言工具集成到适用于 AVR 和 SAM 器件的 Microchip Studio 中并由其控制，从而为 8 位 AVR MCU 器件系列提供基于 GUI 的应用程序代码开发工具。

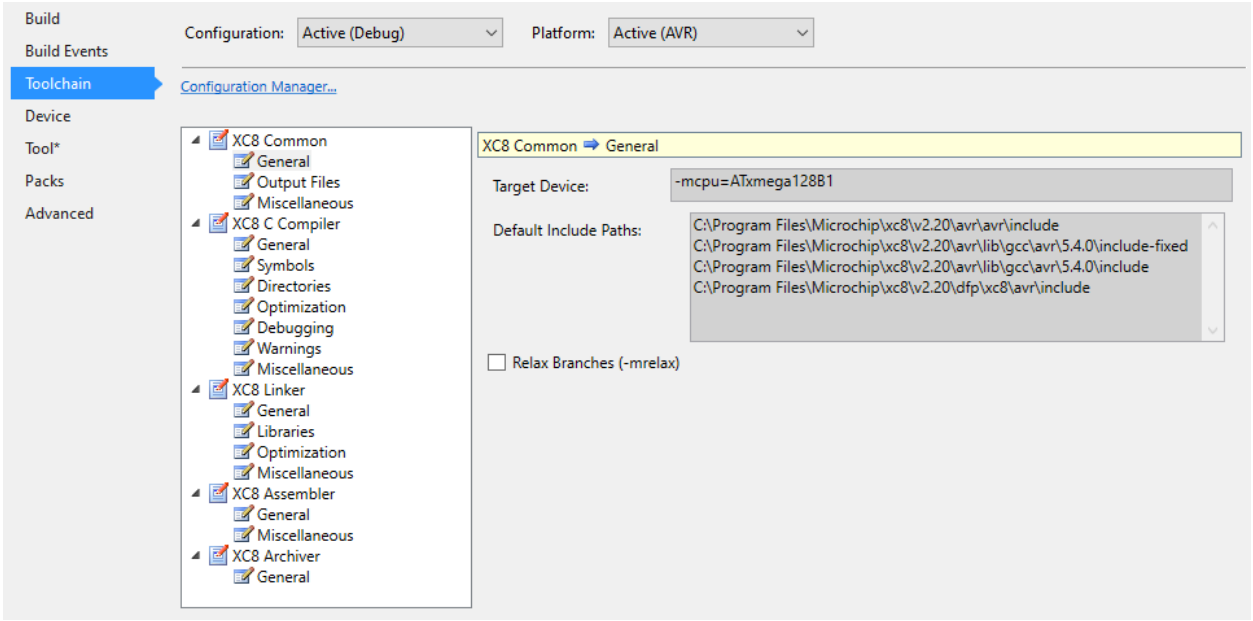
有关 IDE 安装以及创建和设置项目以使用 MPLAB XC8 C 编译器的信息，请参见 *Microchip Studio 7 User Guide*（DS50002718）。

3.8.1 Microchip Studio 等效选项

下文中的说明将 Microchip Studio 的 **Toolchain**（工具链）中的控件与 MPLAB XC8 命令行驱动程序选项相对应。本用户指南的相关章节中提供了参考信息，可详细了解选项功能。

3.8.1.1 XC8 Common（XC8 通用）——General

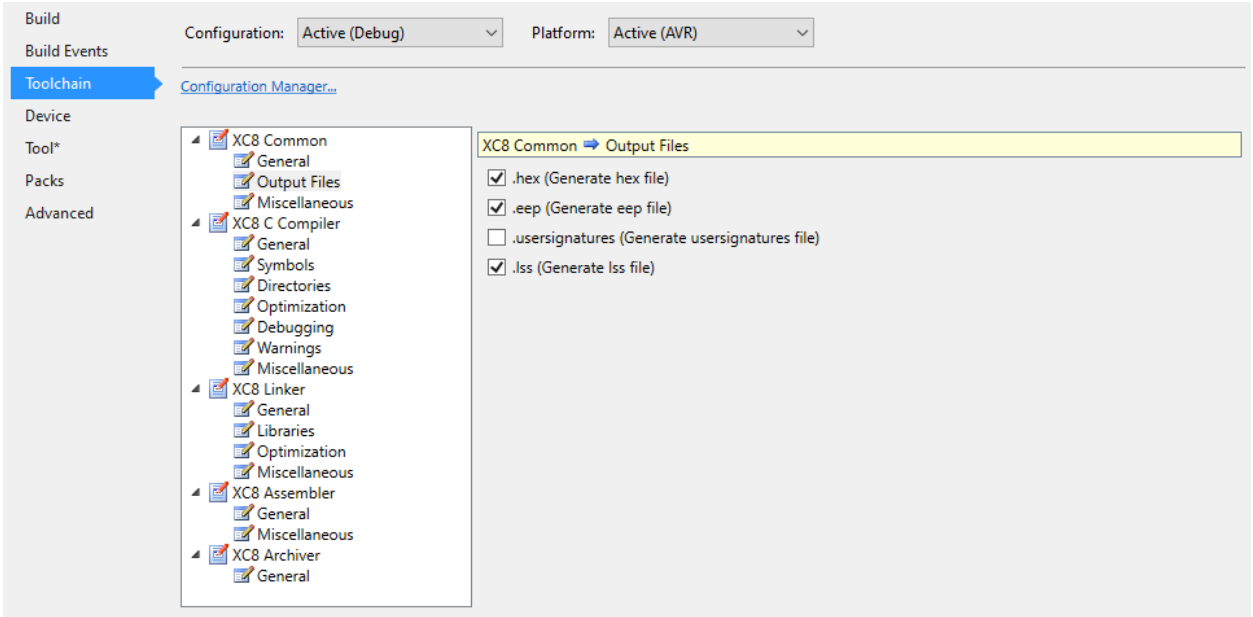
图 3-10. XC8 Common——General



Relax Branches（舒缓分支） 选中该复选框可优化长调用和跳转指令。请参见 3.6.1.10. Relax 选项。

3.8.1.2 XC8 Common——Output Files（输出文件）

图 3-11. XC8 Common——Output Files



Generate hex file（生成 hex 文件） 选中该复选框时，将在链接后调用 `avr-objcopy` 应用程序以在生成 ELF 文件之外再生成 HEX 文件。

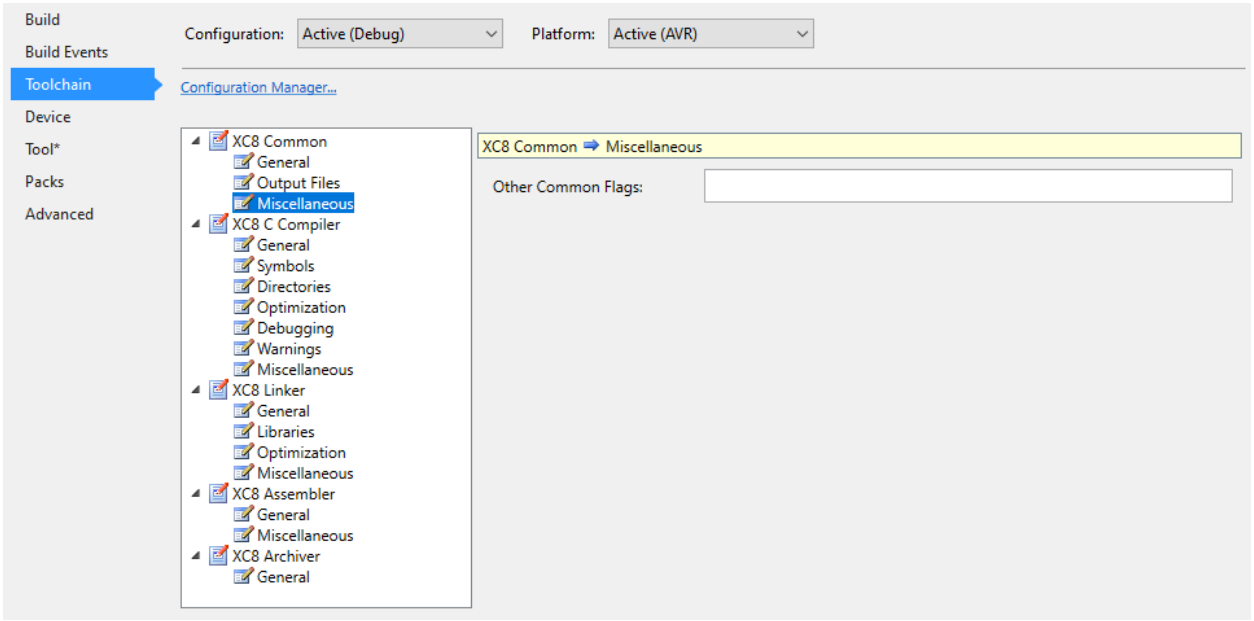
Generate eep file（生成 eep 文件） 选中该复选框时，将在链接后调用 `avr-objcopy` 应用程序以从 ELF 文件中提取 `.eeprom` 段并存储到 `.eep` 文件中。

Generate usersignatures file（生成用户签名文件） 选中该复选框时，将在链接后调用 avr-objcopy 应用程序以从 ELF 文件中提取.user_signatures 段并存储到.usersignatures 文件中。

Generate lss file（生成 lss 文件） 选中该复选框时，将在链接后调用 avr-objdump 应用程序以生成汇编列表文件（扩展名为.lss）。

3.8.1.3 XC8 Common——Miscellaneous（其他）

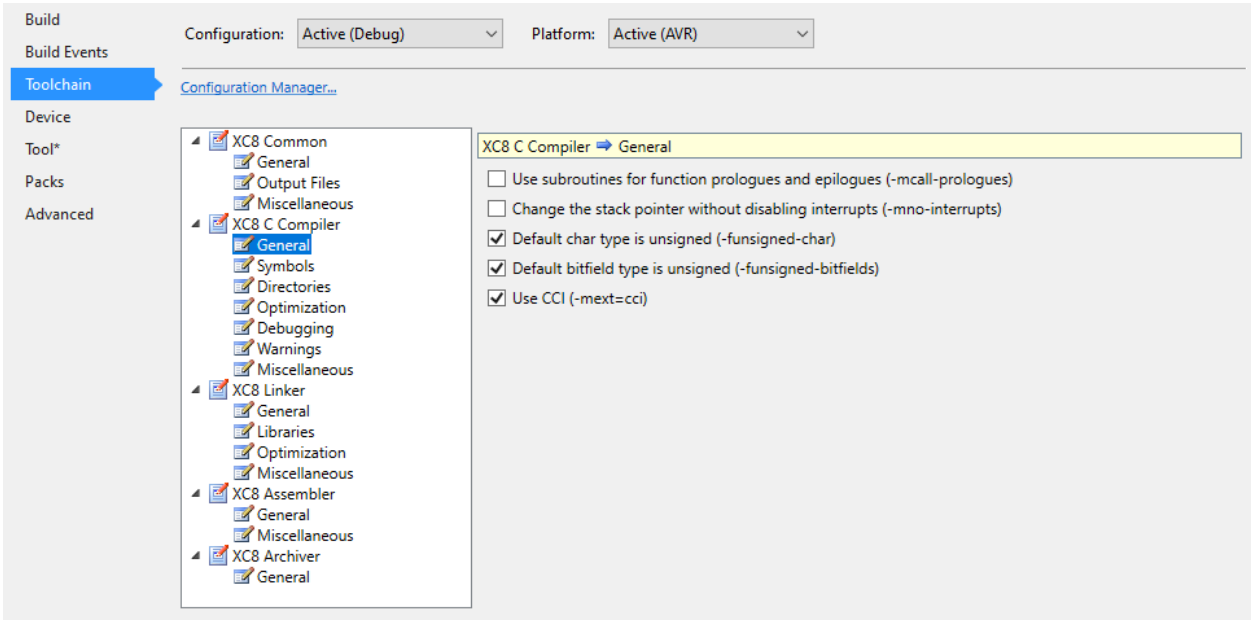
图 3-12. XC8 Common——Miscellaneous



Other Common Flags（其他通用标志） 在该字段中输入其他选项会影响编译的所有阶段。

3.8.1.4 XC8 C Compiler（XC8 C 编译器）——General

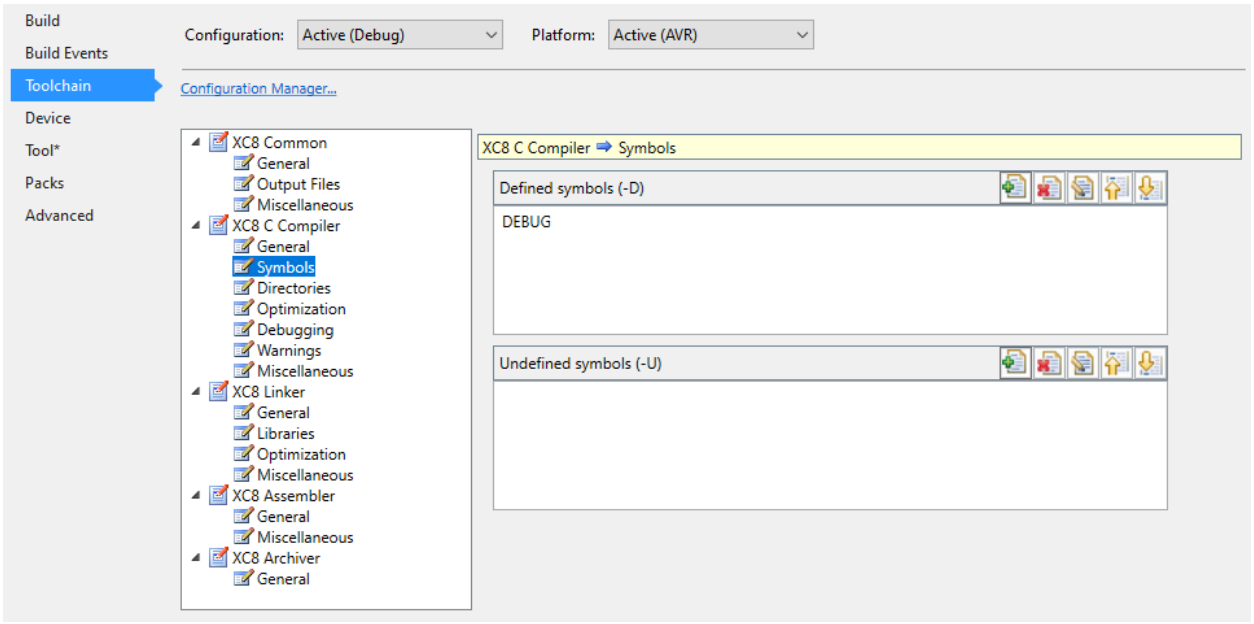
图 3-13. XC8 C Compiler——General



Use subroutines for function prologues and epilogues（为函数前言和结语使用子程序）	该选项影响函数在进入和退出时如何保存和恢复寄存器。请参见 3.6.1.3. Call-prologues 选项。
Change the stack pointer without disabling interrupts（更改堆栈指针而不禁止中断）	该选项用于控制更改堆栈指针时是否应禁止中断。请参见 3.6.1.8. No-interrupts 选项。
Default char type is unsigned	该选项用于控制普通 `char` 使用的符号性。请参见 3.6.3.7. Signed-char 选项。
Default bitfield type is unsigned	该选项用于控制普通 `int` 位域使用的符号性。请参见 3.6.3.7. Signed-char 选项。
Use CCI（使用 CCI）	选中该选项指示源代码应符合通用 C 接口标准。请参见 3.6.3.3. Ext 选项。

3.8.1.5 XC8 C Compiler——Symbols（符号）

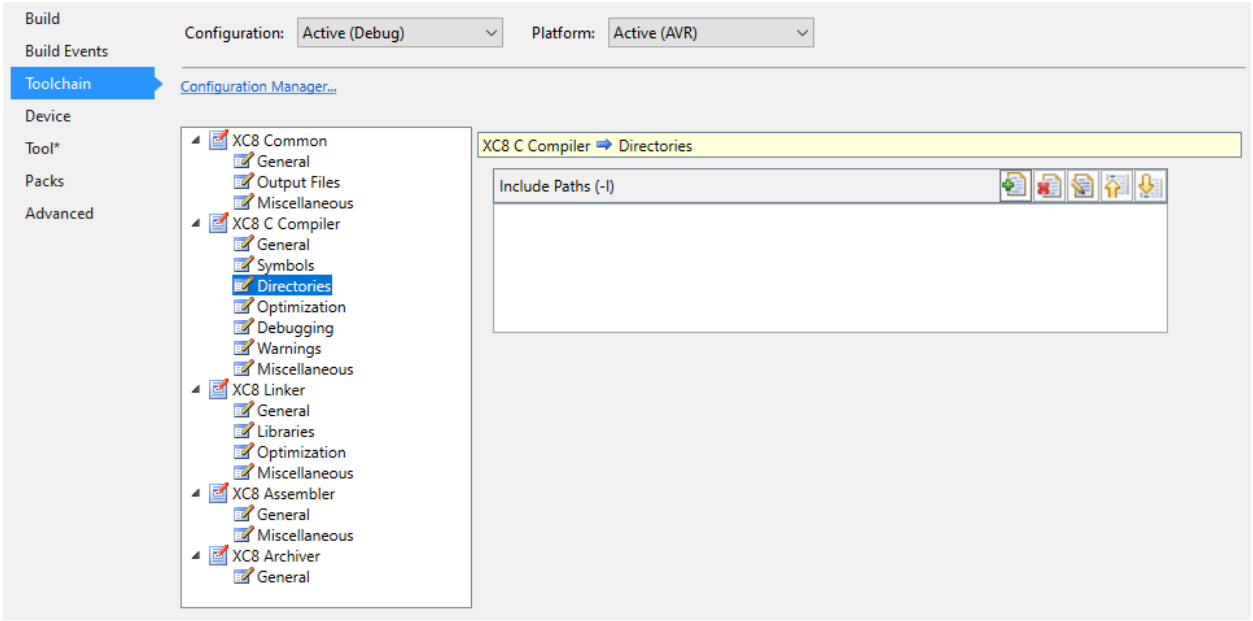
图 3-14. XC8 C Compiler——Symbols



Defined symbols（定义符号）	这些控件用于定义预处理器宏符号和值。请参见 3.6.7.3. D：定义宏。
Undefined symbols（取消定义符号）	这些控件用于取消定义预处理器宏符号和值。请参见 3.6.7.21. U：取消定义宏。

3.8.1.6 XC8 C Compiler——Directories（目录）

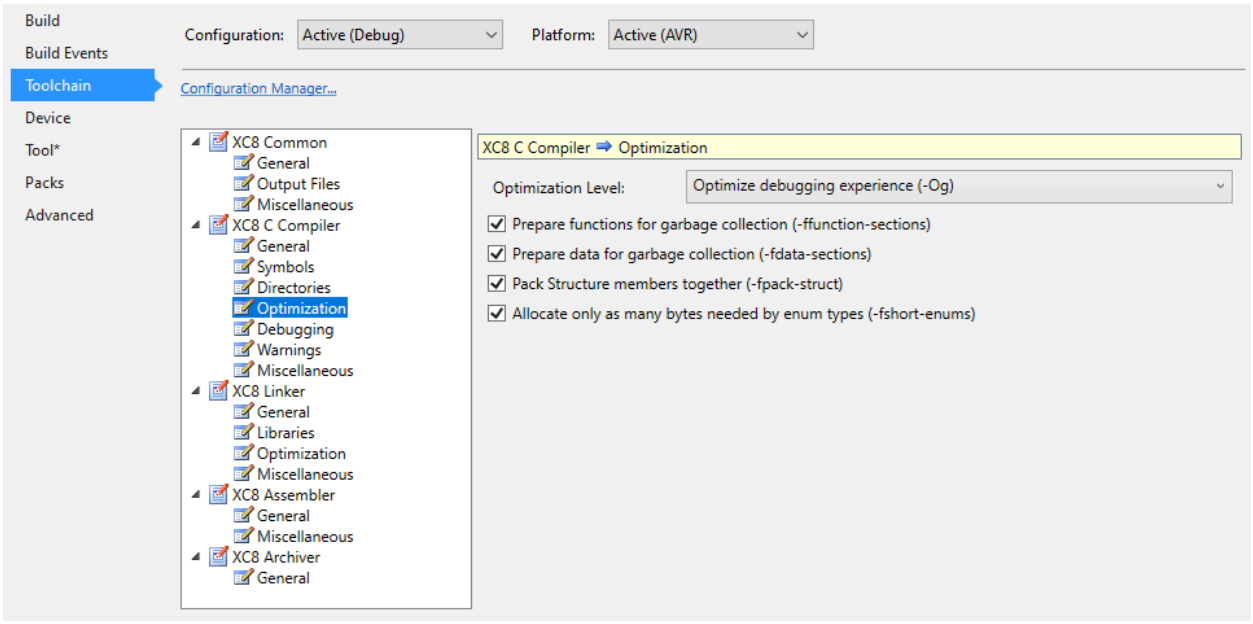
图 3-15. XC8 C Compiler——Directories



Include paths（包含路径） 这些控件用于指定包含文件的搜索路径。请参见 3.6.12.2. I：指定包含文件搜索路径选项。

3.8.1.7 XC8 C Compiler——Optimization

图 3-16. XC8 C Compiler——Optimization



Optimization level 该选项用于选择所执行优化的程度和类型。例如，请参见 3.6.6.1. O0：0 级优化。

Prepare functions for garbage collection（为垃圾收集准备函数） 选中该复选框可使函数存储器分配适用于垃圾收集。请参见 3.6.6.9. Function-sections 选项。

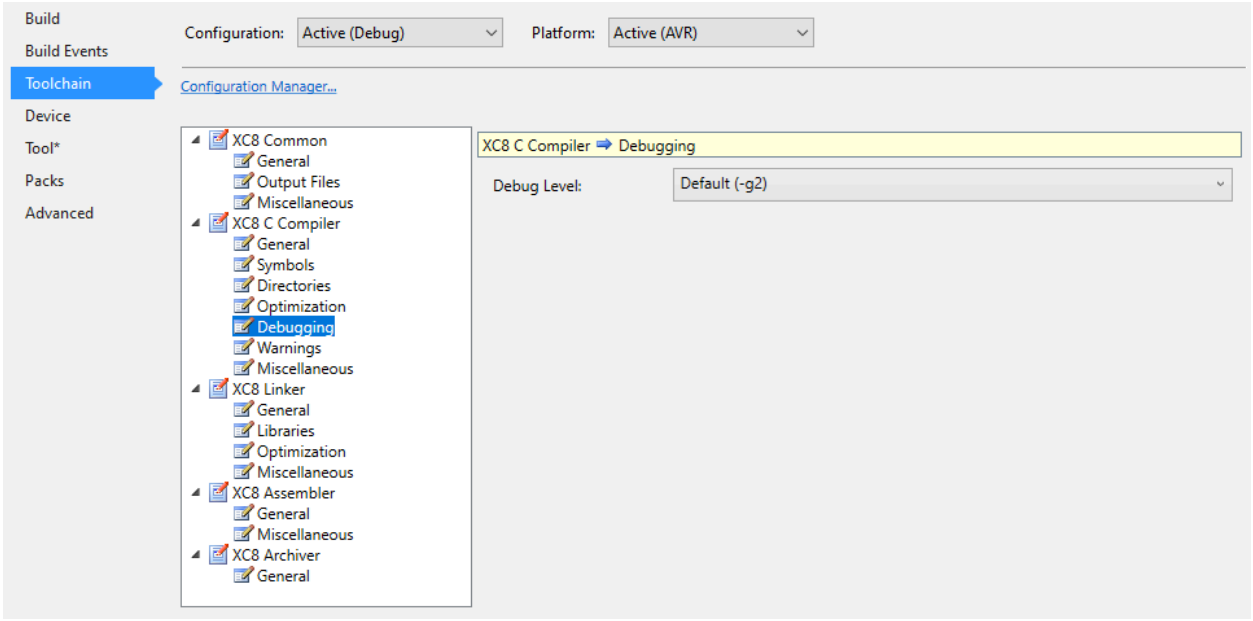
Prepare data for garbage collection（为垃圾收集准备数据） 选中该复选框可使对象存储器分配适用于垃圾收集。请参见 3.6.6.7. Data-sections 选项。

Pack structure members together（将结构成员打包在一起） 该选项对于 8 位 AVR 器件不起作用。

Allocate only as many bytes as needed by enum types（只分配枚举类型所需的字节数） 使能该选项会将最小可能整型类型分配给 `enum`。请参见 3.6.13.1. Short-enums 选项。

3.8.1.8 XC8 C Compiler——Debugging（调试）

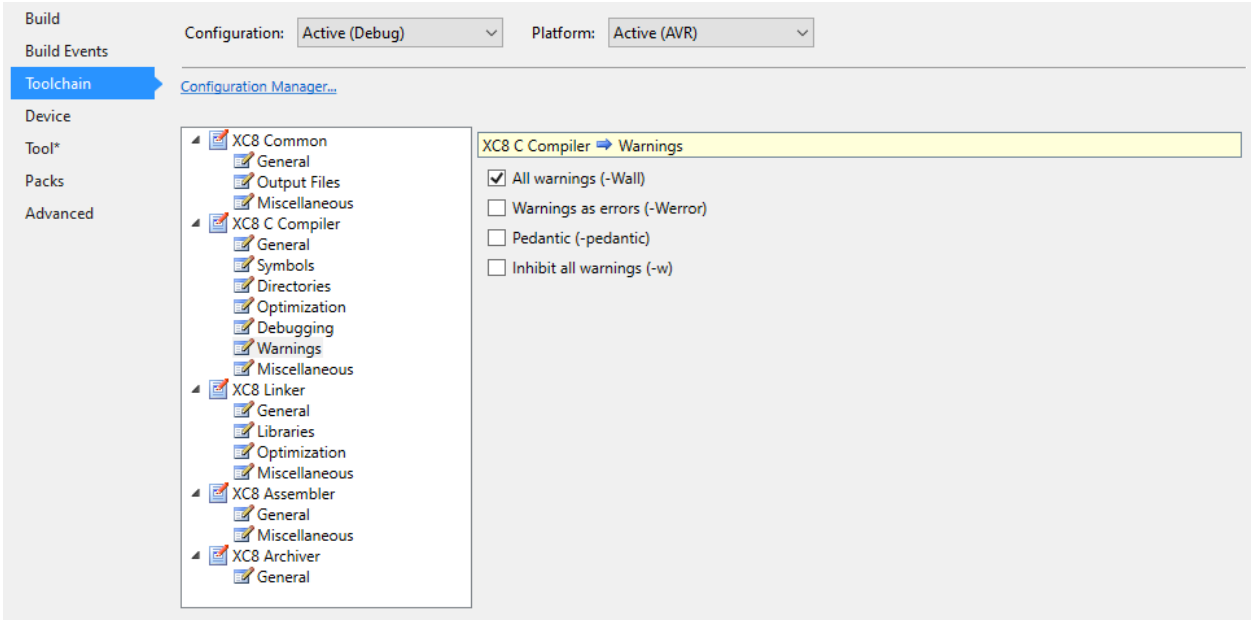
图 3-17. XC8 C Compiler——Debugging



Debug level（调试级别） 该选项用于控制生成的调试信息量。请参见 3.6.5.1. G：生成调试信息选项。

3.8.1.9 XC8 C Compiler——Warnings（警告）

图 3-18. XC8 C Compiler——Warnings



All warnings 选中该复选框可为全部有问题的代码构造使能警告。请参见 3.6.4.5. Wall 选项。

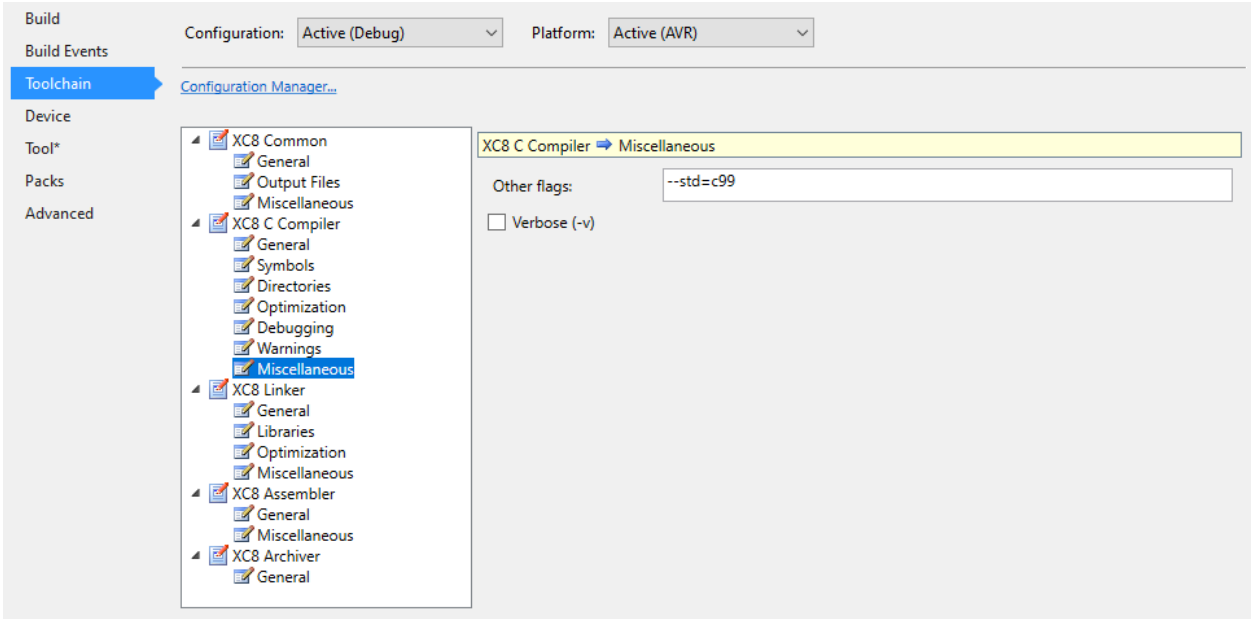
Warnings as errors（警告视为错误） 选中该复选框会将警告消息视为将阻止编译完成的错误。请参见 3.6.4.6. Werror 选项。

Pedantic（严格） 选中该复选框将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。请参见 3.6.4.1. 严格程度选项。

Inhibit all warnings（禁止所有警告） 选中该复选框将禁止所有警告消息。请参见 3.6.4.4. W：禁止所有警告选项。

3.8.1.10 XC8 C Compiler——Miscellaneous（其他）

图 3-19. XC8 C Compiler——Miscellaneous

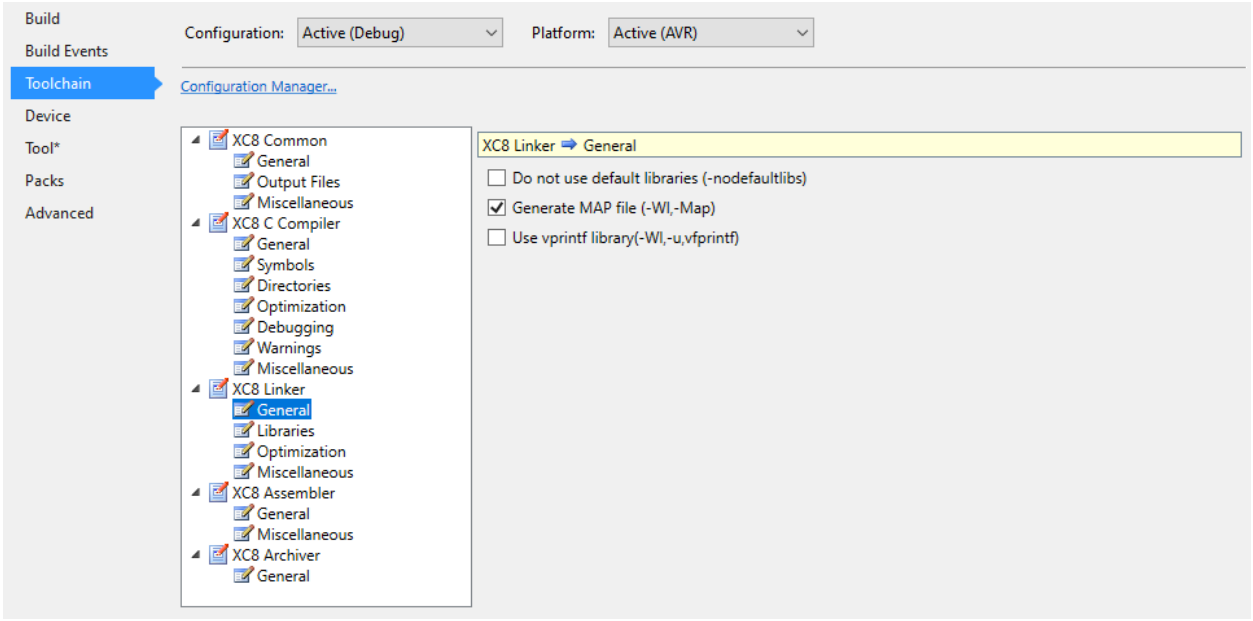


Other flags（其他标志） 在该字段中输入其他选项会影响编译的代码生成阶段。

Verbose 选中该复选框可指定详细编译，其中所有内部编译器应用程序的命令行参数都将在编译时打印。请参见 3.6.2.6. V：详细编译。

3.8.1.11 XC8 Linker——General

图 3-20. XC8 Linker——General



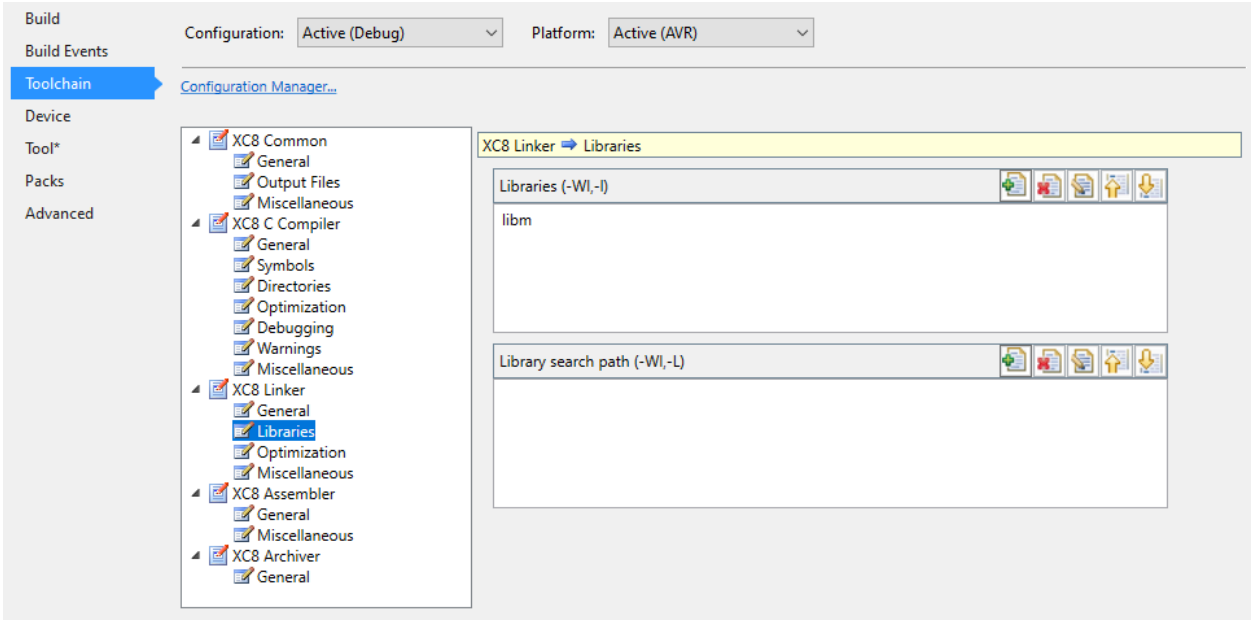
Do not use default libraries（不要使用默认库） 选中该复选框将阻止标准系统库链接到项目。请参见 3.6.10.2. Nodefaultlibs 选项。

Generate MAP file（生成映射文件） 选中该复选框可在链接后生成映射文件。请参见 3.6.11. 映射链接器选项。

Use vprintf library（使用 vprintf 库） 该选项用于强制链接更多库，允许浮点型格式说明符与 avr-libc 结合使用。

3.8.1.12 XC8 Linker——Libraries

图 3-21. XC8 Linker——Libraries

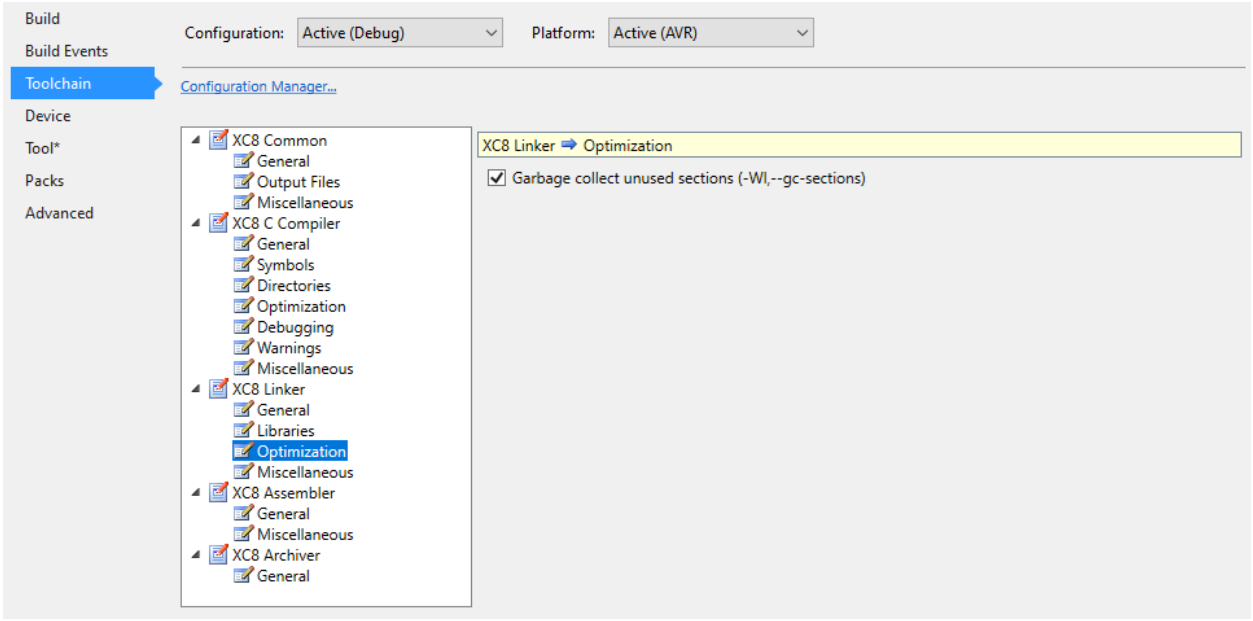


Libraries 这些控件用于指定库以在链接时在其中扫描未解析符号。请参见 3.6.10.1. L：指定库文件选项。

Library search path 这些控件用于指定更多的库文件搜索目录。请参见 3.6.12.3. L：指定库搜索路径选项。

3.8.1.13 XC8 Linker——Optimization

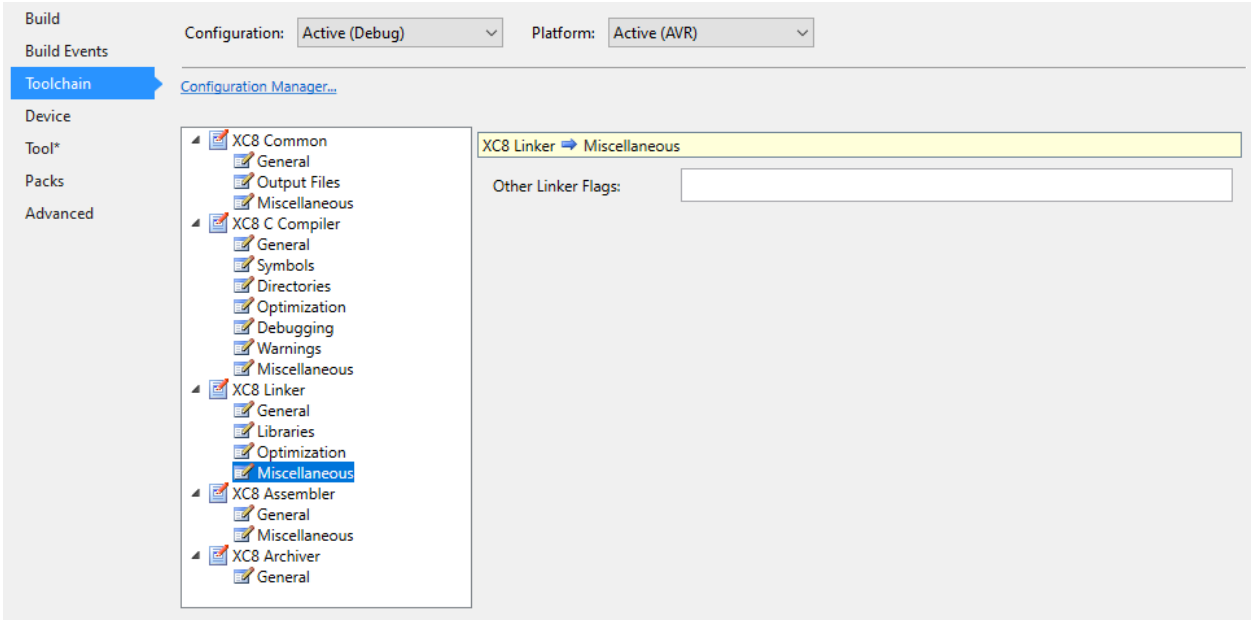
图 3-22. XC8 Linker——Optimization



Garbage collect unused sections 选中该复选框可从程序输出中移除未使用的段。请参见 3.6.11. 映射链接器选项。

3.8.1.14 XC8 Linker——Miscellaneous

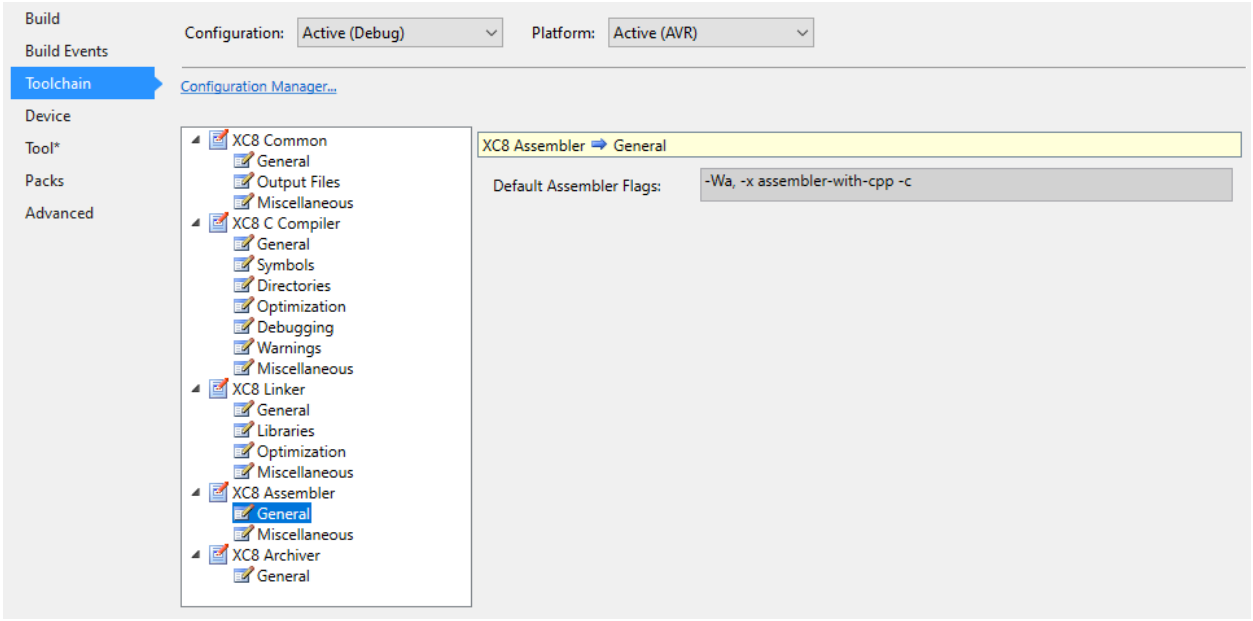
图 3-23. XC8 Linker——Miscellaneous



Other linker flags（其他链接器标志） 在该字段中输入其他选项会影响编译的链接阶段。

3.8.1.15 XC8 Assembler（XC8 汇编器）——General

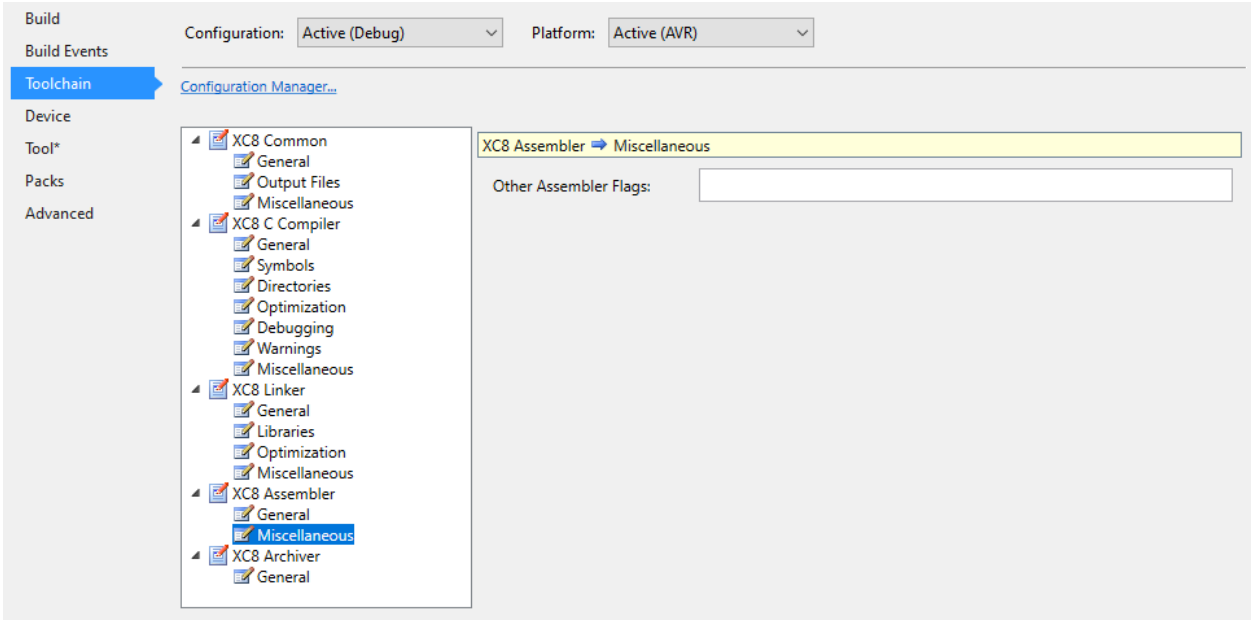
图 3-24. XC8 Assembler——General



Default Assembler Flags（默认汇编器标志） 将传递给汇编器的默认选项的汇总。

3.8.1.16 XC8 Assembler——Miscellaneous

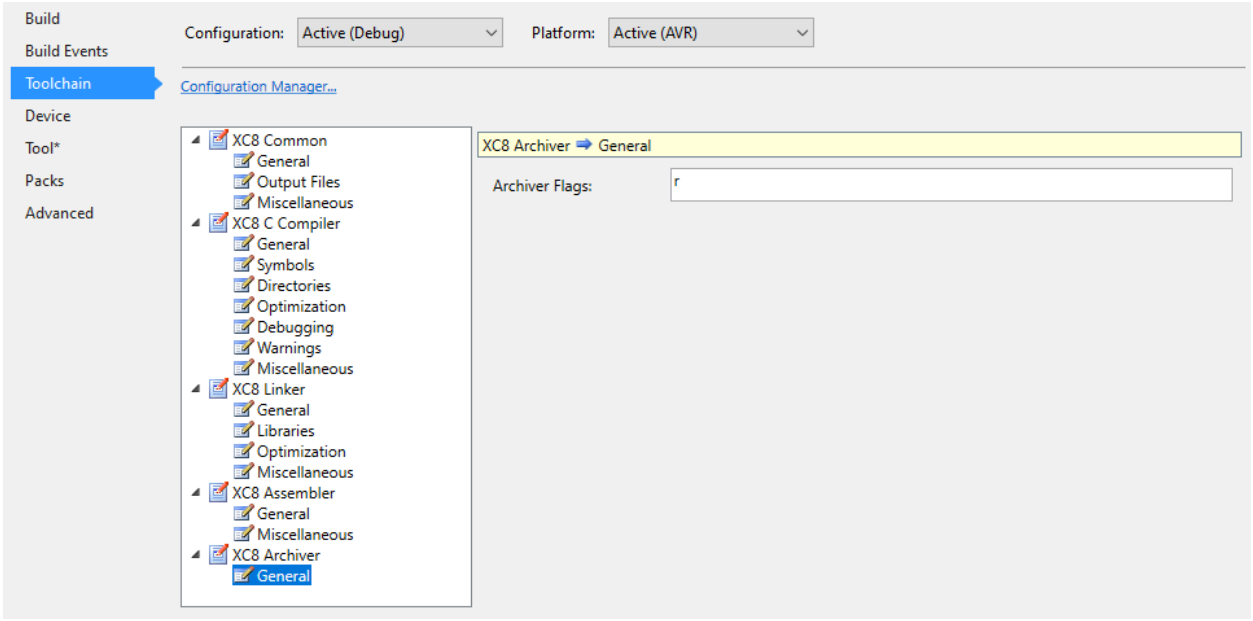
图 3-25. XC8 Assembler——Miscellaneous



Other assembler flags（其他汇编器标志） 在该字段中输入其他选项会影响编译的汇编器阶段。

3.8.1.17 XC8 Archiver（XC8 归档器）——General

图 3-26. XC8 Archiver——General



Archiver flags（归档器标志） 在该字段中输入其他选项会影响编译的归档阶段。请参见 5.1.1. 使用归档器/库管理器

4. C 语言特性

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

4.1 符合 C 标准

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）。可以使用`-std` 选项（见 3.6.3.8. Std 选项）选择程序标准。

该实现未进行任何关于基础操作系统的假设，也不支持流、文件或线程。本节讨论此编译器与这些标准存在偏差的方面。

4.1.1 通用 C 接口标准

该编译器符合 Microchip XC 编译器通用 C 接口标准（CCI），并可确认 C 源代码符合 CCI。

CCI 对 C 标准进行了进一步细化，并尝试在整个 MPLAB XC 编译器系列之间标准化实现定义的行为和非标准扩展。

可以通过使用`-mext=cci` 选项强制执行 CCI（见 3.6.3.3. Ext 选项）。

4.1.2 与 C99 标准的偏差

在 MPLAB XC8 C 编译器中实现的 C 语言与 C99 标准存在偏差，如以下各节所述。

4.1.2.1 复数支持

复数类型`_Imaginary` 不受支持（尽管允许使用`_Complex`）。`<complex.h>`头文件也不受支持。

4.1.3 实现定义的行为

ISO C 标准的某些特性具有实现定义的行为。这意味着一些 C 代码的确切行为会因编译器而不同。本手册将详细介绍编译器的确切行为，6. 实现定义的行为对其进行了全面总结。

4.2 与器件相关的特性

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

4.2.1 器件支持

MPLAB XC8 C 编译器旨在支持所有 8 位 PIC 和 AVR 器件（只有必须用汇编语言编程的 avr1 架构器件除外）。对 8 位 AVR 器件编程时应参考本用户指南；对 PIC 目标器件编程时，请参见《适用于 PIC® MCU 的 MPLAB® XC8 C 编译器用户指南》（DS50002737F_CN）。

新的 AVR 器件不断推出。有几种方法可以检查您使用的编译器是否支持特定器件。

从命令行运行要使用的编译器，并向它传递选项`-mprint-devices`（见 3.6.2.4. Print-devices）。此时会打印所有器件的列表。

您也可以在您喜欢的网络浏览器中查看支持的器件。打开文件`avr_chipinfo.html`，获取支持的所有器件的列表。该文件位于编译器安装目录下的 docs 目录中。

可以通过下载和安装发布的器件系列包（DFP）扩展编译器支持的器件的列表。这可以通过 Microchip 开发环境进行管理。

4.2.2 指令集支持

编译器支持所有 8 位 AVR 器件的所有指令集，但 avr1 架构的指令集除外。

4.2.3 堆栈

MPLAB XC8 实现了一个堆栈。该堆栈用于函数返回地址以及由函数分配的基于堆栈的对象。寄存器 r28 和 r29（Y 指针）用作帧指针，通过其可以访问基于堆栈的对象。

堆栈指针初始化为最高有效数据存储器地址。函数被调用时，会为其基于堆栈的对象分配一块存储区，堆栈在存储区中会从高地址向低地址分配。当函数退出时，为其分配的存储区可由其他函数使用。

请注意，编译器无法检测作为一个整体为堆栈保留的存储区是否溢出。不会进行任何运行时检查来确定软件堆栈是否溢出。如果软件堆栈发生溢出，可能会导致数据损坏和代码失败。

4.2.4 配置位访问

配置位（或熔丝）用于设置基本的器件操作，例如振荡器模式、看门狗定时器、编程模式和锁定位代码保护。必须正确设置这些位以确保程序正确执行。

使用具有以下格式的配置 pragma 伪指令设置器件。

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

此处，*setting* 是配置设置描述符（如 WDT），*state* 是所需状态的文本描述（如 SET）。SET 和 CLEAR 以外的状态以设置描述符为前缀，如以下示例中的 BODLEVEL_4V3。

```
#pragma config WDTON = SET
#pragma config EESAVE = CLEAR
#pragma config BODLEVEL = BODLEVEL_4V3
#pragma config LB = LB_NO_LOCK
```

与每款器件关联的设置和状态可以通过一个 HTML 指南确定。打开编译器安装目录的 docs 目录中的 avr_chipinfo.html 文件。单击您所用目标器件的链接，页面将显示适用于该 pragma 伪指令的设置和值。更多信息，请查看您所使用器件的数据手册。

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

```
#pragma config WDTON=SET, EESAVE=CLEAR, BODLEVEL=BODLEVEL_4V3, LB=LB_NO_LOCK
```

value 字段可以采用常量，而不采用描述符，如下所示。

```
#pragma config SUT_CKSEL = 0x10
```

设置/值对不由预处理器扫描，并且不会进行宏替换。不得为设置/值对加引号。

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

pragma 伪指令未指定的位将被赋予一个默认值。配置字中的所有位都应进行编程而非依赖该默认值，以防止意外的程序行为。

4.2.5 签名

编程软件可以使用签名值来验证在编程前是否针对目标器件编译了程序。

每个器件都指定了签名，只需包含<avr/signature.h>头文件即可将签名添加到程序中。该头文件将在代码中声明一个 unsigned char 常量数组，并使用器件的 I/O 头文件中定义的三个签名字节（MSB 在前）对其进行初始化。该数组随后将放入生成的链接 ELF 文件中的签名段。该头文件只应在应用程序中包含一次。

用于初始化数组的三个签名字节是器件 I/O 头文件中定义的宏（从 MSB 到 LSB）：SIGNATURE_2、SIGNATURE_1 和 SIGNATURE_0。

4.2.6 从 C 代码中使用 SFR

特殊功能寄存器（Special Function Register，SFR）是可从 C 程序中访问的存储器映射寄存器。每个寄存器都可以使用在包含<xc.h>后可用的宏进行访问。例如：

```
#include <xc.h>
if(EEDR == 0x0)
  PORTA = 0x55;
```

SFR 中的位可以通过特殊宏 _BV() 以及其他表示要访问的位的宏进行访问。 例如，要将 PORTB 中的 bit 1 置 1，可使用以下代码。

```
PORTB |= _BV(PB1);
```

要将 EECR 中的 bit 4 和 bit 5 清零，可使用以下代码。

```
EECR &= ~(_BV(EEPM4) | _BV(EEPM5));
```

在这两个示例中，编译器将尽可能使用器件的单个位置 1 和清零指令。

4.2.7 特殊的寄存器问题

一些与定时器相关的 16 位寄存器在内部使用宽度为 8 位的临时寄存器（器件数据手册中称为 TEMP）来保证对定时器进行原子访问，因为传送定时器值需要两次单独的字节传送。通常，器件将在访问当前定时器/计数器值寄存器（TCNTn）和输入捕捉寄存器（ICRn）以及写入输出比较寄存器（OCRnM）时使用该寄存器。请参见器件数据手册来确定使用 TEMP 寄存器的外设。

您的程序不能访问该临时寄存器，但有许多外设共用它，因此您的程序需要确保该寄存器不会被同样使用该寄存器的中断程序破坏。

在主干代码中，执行使用该寄存器的代码期间可以禁止中断。可以通过将代码封装到对 cli() 和 sei() 宏的调用中来实现这一目的，但如果全局中断标志的状态未知，可以使用以下示例代码。

```
unsigned int read_timer1(void)
{
  unsigned char sreg;
  unsigned int val;

  sreg = SREG;  // save state of interrupt
  cli();        // disable interrupts
  val = TCNT1;  // read timer value register; TEMP used internally
  SREG = sreg;  // restore state of interrupts

  return val;
}
```

4.2.8 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。

该功能最初可用于除 Attiny 系列（如 attiny5 和 attiny40 等）之外的所有 8 位 AVR 器件。

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器需求增加。

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。（关于如何使用新的代码覆盖功能的信息，请参见 Microchip 的分析工具套件许可网页）。

4.2.9 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。

4.2.9.1 堆栈指导信息

堆栈指导功能可估算由 MPLAB XC8 C 编译器编译的程序所使用的数据堆栈的使用情况。

下例显示了编译成功后可能显示的堆栈使用信息摘要。

```
======================== STACK USAGE GUIDANCE ========================
In the call graph beginning at 'main',
   52 bytes of stack are required.

However, the following cautions exist:

1. Recursion has been detected:
   __fp_splitA
No stack usage predictions can be made.

2. The following labels are interrupt functions:
   __vector_18 uses 15 bytes
You must add stack allowances for those functions.

3. The following labels cannot be connected to the main call graph.
This is usually caused by some indirection:
   func1 uses 2 bytes
   func2 uses 0 bytes
   __fp_zero uses 0 bytes
You must add stack allowances for those functions.
======================================================================
```

该信息将显示估算的程序堆栈使用总量。

此外，还可能显示一些警示消息。这些内容阐明了上述估算所报告的潜在堆栈溢出，并指出了确定更精确的程序堆栈使用量时必须考虑的额外堆栈使用。由于无法知道何时会使用这类额外存储器等原因，早些时候摘要中未加入这些内容。

在所述情形下可能显示以下警示。

递归函数 在主调用图中检测到递归函数调用。递归调用的迭代次数无法预测，所以无法确定软件堆栈总大小，因而无法使用堆栈指导功能。检测到的递归函数的名称在此警示中列出。

断开连接的函数 已检测到未直接调用的函数。此类函数可能已在 C 代码中间接调用或通过某种其他方式调用，也可能根本未调用。每个函数的堆栈使用量在此警示中指出，确定程序的软件堆栈使用总量时需要加以考虑。

不确定的调用 在主调用图中检测到不确定的调用。这些调用可能是间接调用或对标号的调用。调用使用的字节数在此警示中指出，确定程序的总软件堆栈使用量时需要加以考虑。

不确定的堆栈调整 在主调用图中检测到不确定的堆栈调整。 这可能是由于使用可变长度的数组、使用 `alloca` 等基于堆栈分配存储器的存储器分配函数（而非 `malloc` 等基于堆分配的函数）或者存在没有可用堆栈信息的函数。无

法报告与这些情况相关的存储器使用信息。已知要使用的额外字节的静态计算在此警示中指示。计算结果表示的是程序使用的最小额外字节数。

中断函数 即使可能准确掌握主干代码调用图和中断调用图使用的堆栈的大小，但中断会随时触发，在这种情况下，编译器无法可靠地确定程序的总堆栈使用量。此警示将提醒您存在中断函数，考虑程序的总软件堆栈使用量时，需将这种情况下的堆栈使用考虑在内。

4.3 支持的数据类型和变量

C 编程语言中的值符合多种数据类型中的一种，数据类型决定存储大小、格式和值范围。用于存储这些值的变量和对象使用同一类型集来定义。

4.3.1 标识符

标识符用于表示 C 对象和函数，必须遵守限制规则。

C 标识符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。虽然可以使用下划线开头，但这种标识符保留供编译器使用，不应由程序中的 C 源代码定义。汇编域标识符则不是这样。

标识符区分大小写，所以 `main` 不同于 `Main`。

4.3.2 整型数据类型

MPLAB XC8 编译器支持长度为 1、2、4 和 8 字节的整型数据类型。下表列出了支持的数据类型及其对应的长度和算术类型。

表 4-1. 整型数据类型

类型	长度（位）	算术类型
`signed char`	8	有符号整型
`unsigned char`	8	无符号整型
`signed short`	16	有符号整型
`unsigned short`	16	无符号整型
`signed int`	16	有符号整型
`unsigned int`	16	无符号整型
`signed long`	32	有符号整型
`unsigned long`	32	无符号整型
`signed long long`	64	有符号整型
`unsigned long long`	64	无符号整型

如果未指定类型（包括 `char` 类型）的符号性，则该类型为 `signed`。

所有整型值都使用最低有效字节（Least Significant Byte，LSB）位于器件存储器低地址的小尾数法格式表示。

有符号值以二进制补码整型值的形式存储。

可以由这些类型存放的值范围在`<limits.h>`（见 *Microchip Unified Standard Library Reference Guide*）的声明中进行了概要描述。其中的符号是一些预处理器宏，它们在源代码中包含`<limits.h>`之后可用。由于 C 标准未详细规定数据类型的长度，所以使用这些宏可以实现可移植性更高的代码，它们可以检查该实现中的类型可以存放的值的范围限制。

`<stdint.h>`中也提供了一些宏，这些宏定义与固定宽度类型（如 `int8_t` 和 `uint32_t` 等）相关联的值。

4.3.3 布尔型

编译器支持 `_Bool`，这种类型用于存储 true 和 false 值。

此类型变量存储的值不是整型值，在表达式中的行为与包含整型变量的相似表达式有所不同。如果值为 0，则转换为 _Bool 型后结果为 0（false）；如果值不为 0，结果为 1（true）。

<stdbool.h>头文件定义 true 和 false 宏，二者可用于_Bool 类型和 bool 宏（扩展至_Bool 类型）。例如：

```
#include <stdbool.h>
_Bool motorOn;
motorOn = false;
```

4.3.4 浮点型数据类型

MPLAB XC8 编译器支持 32 位浮点型。浮点型使用 IEEE 754 32 位格式实现。下表列出了浮点型数据类型及其长度。

表 4-2. 浮点型数据类型

类型	长度（位）	算术类型
float	32	实数
double	32	实数
long double	32	实数

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 LSB 位于低地址的小尾数法格式表示。

无穷大是所有运算的合法参数，表现为采用相应符号的最大可表示数。例如，表达式+inf + -inf 的结果为值 0。

下表介绍了两种浮点型的格式，其中：

- *符号*是指符号位，指示数字是正数还是负数。
- *偏移的指数*宽度为 8 位，它以加 127 的形式存储（即，指数 0 存储为 127）。
- *尾数*位于小数点右侧。在小数点左侧有一个隐含位，对于零值，隐含位为零，对于其他值，它总是为 1。零值通过零指数表示。

该数字的值为$(-1)^{符号} \times 2^{(指数-127)} \times 1.尾数$。

表 4-3. 浮点型格式

格式	符号	偏移的指数	尾数
IEEE 754 32 位	x	xxxx xxxx	xxx xxxx xxxx xxxx xxxx xxxx

下表列出了 IEEE 754 32 位格式的一个示例。请注意，尾数列的最高位（MSb，即小数点左边的位）是隐含位，除非指数为零，否则假定它为 1。

表 4-4. 浮点型格式示例 IEEE 754

格式	值	偏移的指数	1.尾数	十进制
32 位	7DA6B69Bh	11111011b	1.01001101011011010011011b	2.77000e+37
		（251）	（1.302447676659）	—

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 223（其中，23 是尾数大小），得到 0.302447676659。将该小数加上 1。然后，通过以下方式得到浮点数：

$-1^{0}x2^{124}x1.302447676659$

它约等于：

2.77000e+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对象表示。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。

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，可以表示的下一个最高值（近似）为 95000.00781。

<float.h>的声明中总结了浮点格式的特性（包含在 *Microchip Unified Standard Library Reference Guide* 中）。其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。由于 C 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。

4.3.5 结构和联合

MPLAB XC8 C 编译器 支持 struct 和 union 类型。结构和联合的惟一区别在于对每个成员应用的存储器偏移量。

这两种类型至少为 1 个字节宽。完全支持位域和_Bool 对象。

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

4.3.5.1 结构和联合限定符

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

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

在此例中，每个结构成员都将是只读的。请记住，如果结构使用 const 限定，则应对所有成员进行初始化，因为它们不能在运行时初始化。

4.3.5.2 结构中的位域

MPLAB XC8 C 编译器 完全支持结构中的位域。

位域总是分配到 8 位字中，虽然它在定义中通常使用 unsigned int 类型。所定义的第一个位将为用于存储它的字的 LSb。声明位域时，如果当前的 8 位单元可容纳它，则会将它分配到其中；否则，会在结构中分配一个新的字节。

位域可以跨 8 位分配单元之间的边界；但访问这样的位域时，代码的效率将相当低下。

例如，以下定义：

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

将产生一个占用 1 个字节的结构。

如果 foo 最终链接到地址 0x10 处，位域 lo 将为地址 0x10 的 bit 0；位域 hi 将为地址 0x10 的 bit 7。dummy 的 LSb 将为地址 0x10 的 bit 1。

注：访问包含多个位的位域效率非常低。如果代码长度和执行速度比较关键，请考虑使用 char 类型或 char 结构成员。请注意，一些 SFR 定义为位域。其中大多数位域包含一个位，但有些是包含多个位的对象。

可以通过声明未命名位域来填充控制寄存器中有效位之间的未用空间。例如，如果永远不会引用 dummy，以上结构可以声明为：

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

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

```
struct {
    unsigned        lo  : 1;
```

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

带有未命名位域的结构可以进行初始化。不应为未命名成员提供初始值，例如：

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

将正确初始化成员 lo 和 hi。

长度为 0 的位域是一种特殊情况。标准规定不能将任何其他位域存放到前一个位域（如果有）的分配单元。

4.3.5.3 匿名结构和联合

MPLAB XC8 编译器支持匿名结构和联合。它们是不具有标识符的 C11 结构，无需引用该结构的标识符即可访问其成员。匿名结构和联合必须放在其他结构或联合的内部。例如：

```
struct {
    union {
        int x;
        double y;
    };
} aaa;
aaa.x = 99;
```

此处，union 未命名，其成员可以作为结构的一部分进行访问。

4.3.6 指针类型

MPLAB XC8 C 编译器支持两种基本指针类型：

数据指针 这类指针存放可以由程序读取（也可能写入）的对象地址。

函数指针 这类指针存放可通过指针调用的可执行函数的地址。

这些指针类型不可互换使用。切勿使用数据指针（甚至通用 void *指针）存放函数地址，也不要使用函数指针存放对象地址。

4.3.6.1 组合类型限定符和指针

先回顾一下 C 标准对于指针类型定义的约定会很有帮助。

指针可以像任何其他 C 对象一样进行限定，但这样做时必须小心，因为会有两个量与指针相关联。第一个量是实际指针本身，它像任何普通 C 对象一样对待，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。指针本身（存放地址的变量）不是 volatile 类型；但是，进行指针解引用时访问的对象被视为 volatile 类型。即，可通过指针访问的目标对象可在外部进行修改。

在第二个示例中，名为 ivp 的指针是 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。

4.3.6.2 数据指针

指向仅位于数据空间中的对象的指针全部为 2 字节宽。

使用__flash 或__flash*n*（其中 *n* 的范围为 1 至 5）限定的指针可以访问程序存储器中存储的数据对象。这两种指针类型均为 16 位宽。指向__flash 的指针使用 lpm 指令访问数据；指向__flashn 的指针使用 RAMPZ 寄存器和 elpm 指令。这两种指针均无法用于访问 RAM 中的对象。

必须为这些指针赋予使用相同限定符定义的对象的地址，例如，只能为指向__flash1 的指针赋予同样使用__flash1 限定符的对象的地址。

4.3.6.2.1 指向两个存储空间的指针

任何指向 const 的指针均能够读取数据存储空间和程序存储空间中的对象。此类指针被称为混合存储空间指针，可能大于未在目标类型上使用 const 定义的常规数据指针。

例如，以下函数可以从任一存储空间中读取 int 并返回，具体取决于传递给该函数的地址。

```
int read(const int * mip) {
  return *mip;
}
```

对于 avrtiny 或 avrxmega3 系列中的任何器件，这些指针为 16 位宽，能够以数据存储器或程序存储器中的对象为目标，因为程序存储器映射到数据空间。对于其他器件，指针为 24 位宽。

24 位宽的 const *类型地址使用地址的最高位来线性化闪存和 RAM，从而确定要访问的存储空间。如果 MSb 置 1，则使用两个低字节作为地址从数据存储器中访问对象。如果地址的 MSb 清零，则通过根据地址的高字节设置的 RAMPZ 段寄存器从程序存储器中访问数据。

请注意，const int *指针类型与 const __memx int *指针类型在如何访问对象方面类似，但前者无需使用非标准关键字，并且可使用-mconst-data-in-progmem 选项（见 3.6.1.5. Const-data-in-progmem 选项）控制访问。

4.3.6.3 函数指针

MPLAB XC8 编译器完全支持指向函数的指针。它们通常用于调用存储在用户定义的 C 数组中的几个函数地址之一，该数组就如同一个查找表。

函数指针的长度为 2 字节。由于地址按字对齐，此类指针可到达最高 128 KB 的程序存储器地址范围。如果所用器件支持的程序存储器量超过该值，并且您希望间接访问大于该地址的程序，则需要使用-mrelax 选项（见 3.6.1.10. Relax 选项），该选项可保持指针的长度，但将指示链接器通过查找表使调用到达其最终目标地址。

为了在程序存储空间大于 128 KB 的器件上实现间接跳转，提供了一个名为 EIND 的特殊功能寄存器，可在执行 eicall 或 eijmp 指令时用作目标地址的最高有效部分。编译器也可能使用该寄存器来通过 ret 指令仿真间接调用或跳转。

编译器从不设置 EIND 寄存器并假定它在启动代码或程序执行期间从不更改，这意味着 EIND 寄存器不会在函数或中断服务程序前言或结语中保存或恢复。

为了适应对函数和计算跳转的间接调用，链接器将生成包含到所需地址的直接跳转的函数桩（stub 或 trampoline）。对函数桩进行间接调用和跳转，随后重定向执行到所需函数或位置。

要使函数桩正常工作，必须使用-mrelax 选项。该选项可确保链接器将使用 16 位函数指针与函数桩的组合，甚至目标地址可能在大于 128 KB 时也如此。

默认链接描述文件假定代码需要 EIND 寄存器包含 0。如果并非如此，必须使用自定义链接描述文件以将名称以.trampolines 开头的段放入与 EIND 寄存器存放的值对应的段中。

来自 libgcc.a 库的启动代码从不设置 EIND 寄存器。

用户特定的启动代码可以合法地提前设置 EIND，例如通过.init3 段中的初始化代码进行设置。运行此类代码时，需在用于初始化 RAM 并调用构造函数的通用启动代码之前，但在用于将 EIND 寄存器设置为与向量表位置对应的值的 AVR-LibC 启动代码之后。

```
#include <avr/io.h>

static void
__attribute__((section(".init3"),naked,used,no_instrument_function))
init3_set_eind (void)
{
  __asm volatile ("ldi r24,pm_hh8(__trampolines_start)\n\t"
  "out %i0,r24" :: "n" (&EIND) : "r24","memory");
}
```

__trampolines_start 符号在链接描述文件中定义。

如果满足以下两个条件，链接器会自动生成函数桩：

- 通过 gs 汇编器修改程序（生成函数桩的简称）获取标号的地址，如下所示：

  ```
  LDI r24, lo8(gs(func))
  LDI r25, hi8(gs(func))
  ```

- 该标号的最终位置是函数桩所在段外面的代码段。

在以下情况下，编译器将为代码标号发出 gs 修改程序：

- 获取函数或代码标号的地址时
- 生成计算跳转时
- 使用由前言保存的函数时（见 3.6.1.3. Call-prologues 选项）
- 生成 switch/case 分配表时（可通过指定-fno-jump-tables 选项（见 3.6.1.9. No-jump-tables 选项）禁止这些分配表）
- 启动/关闭期间调用的 C 和 C++构造函数/解构函数

不支持跳转到绝对地址，如以下示例所示：

```
int main (void)
{
  /* Call function at word address 0x2 */
  return ((int(*)(void)) 0x2)();
}
```

必须改为通过符号（以下示例中的 func_4）调用函数，以便设置函数桩：

```
int main (void)
{
  extern int func_4 (void);
  /* Call function at byte address 0x4 */
  return func_4();
}
```

应通过-Wl,--defsym,func_4=0x4 链接项目。或者，也可以在链接描述文件中定义 func_4。

4.3.7 常量类型和格式

C 代码中的常量是一个立即数值，可用多种格式指定且会被赋予一种类型。

4.3.7.1 整型常量

整型常量的格式会指定其基数。 MPLAB XC8 支持标准基数说明符，以及用于在 C 代码中指定二进制常量的说明符。

下表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 4-5. 基数格式

基数	格式	示例
二进制	0b*number* 或 0B*number*	0b10011010
八进制	0*number*	0763
十进制	*number*	129
十六进制	0x*number* 或 0X*number*	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。下表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，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 4-6. 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赋予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;
shifter = 20;
result = 1 << shifter;   // oops!
```

常量 1 将被赋予 int 类型，因此移位操作的结果将为 int 类型。虽然将该结果赋予 long 变量 result，它的长度永远不会大于 int 的长度，无论对该常量进行了多少位的移位。在此例中，值 1 被左移 20 位将产生结果 0，而不是 0x100000。

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

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

4.3.7.2 浮点型常量

浮点型常量具有 double 类型，除非加上后缀 f 或 F，在后面这种情况下它是一个 float 常量。后缀 l 或 L 用于指定 long double 类型，MPLAB XC8 将其视为与 double 相同的类型。

浮点型常量可用小数点和/或指数指定为十进制数字。如果使用 C99，则浮点型常量可表示为以 p 或 P 开始的十六进制数字和二进制指数。例如：

```
myFloat = -123.98E12;
myFloat = 0xFFEp-22;  // C99 float representation
```

4.3.7.3 字符和字符串常量

字符常量引上单引号字符’，例如'a’。字符常量具有 int 类型，虽然编译器之后可能会将其优化为 char 类型。

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

```
const char name[] = "Bj\370rk";
printf("%s's Resum\351", name);     \\ prints "Bjørk's Resumé"
```

该实现不支持多字节字符常量。

字符串常量或字符串字面值引上双引号字符"，例如"hello world"。字符串常量的类型为 const char *，构成字符串的字符存储在程序存储器中，与使用 const 限定的所有对象一样。

将无法修改的字符串字面值赋予一个未指定 const 目标的指针时，会产生一条常见的相关警告，例如：

```
char * cp = "hello world\n";
```

请参见 4.3.6.1. 组合类型限定符和指针，将指针限定如下。

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

定义并使用字符串初始化数组（即，不是指针）属于例外情况。例如：

```
char ca[]= "hello world\n";
```

实际上会将字符串字符复制到 RAM 数组中，而不是像前面的示例一样将字符的地址赋予指针。字符串字面值保持为只读，但该数组是可读写的。

对于具有相同字符序列的字符串，MPLAB XC8 编译器将使用相同的存储单元和标号（用于初始化驻留在数据空间中的数组的字符串除外）。例如，在以下代码片段中

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

字符串"world"中的字符和字符串"hello world"中的最后 6 个字符（最后一个字符为空终止符）将由存储器中的相同字符表示。字符串"hello"不会与字符串"hello world"中的相同字符重叠，因为两者中的空字符的位置不同。

4.3.8 标准类型限定符

编译器支持标准限定符 const 和 volatile，它们对于嵌入式应用程序开发十分重要。

4.3.8.1 const 类型限定符

const 类型限定符用于指示编译器，某个对象是只读的，不应被修改。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。

假设未使用其他存储限定符，则 const 限定对象（不包括自动和参数对象）默认链接到程序空间，但可以使用-mno-const-data-in-progmem 选项（见 3.6.1.5. Const-data-in-progmem 选项）更改。此类对象也可设为绝对对象，以便轻松置于程序存储器中的特定地址处，请参见 4.4.4.1. 程序存储器中的绝对对象。

const 限定对象按照__memx 限定对象的方式访问，请参见 4.3.9.1. Memx 地址空间限定符。如果 const 限定对象的定义还使用了__flash 或__flashn（见 4.3.9.2. Flash 限定符和 4.3.9.3. Flashn 限定符）等存储限定符，则这些限定符隐含的读访问策略优先于 const 隐含的读访问策略。

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 定义为只读 int 变量，用于存放值 3。

4.3.8.2 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。该信息可以防止优化器删除对声明为 volatile 的对象看起来多余的引用，因为这些引用可能会改变程序的行为。

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：

```
#include <xc.h>
volatile static unsigned int TACTL __at(0x800160);
```

volatile 限定符并不保证所有访问都是原子操作，由于 8 位 AVR 架构通常只能每条指令访问 1 字节数据，因此通常不是原子操作。

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 关键字的常见用途是防止一些未使用的变量被删除。如果某个非 volatile 变量从不使用，或者其使用方式没有任何影响，那么它可能会在编译器生成代码之前被删除。

如果某条 C 语句仅包含一个 volatile 变量的标识符，则会生成读取变量的存储单元并丢弃结果的代码。例如，如果 PORTB;为完整语句，则会生成用于读取 PORTB 的汇编代码。

4.3.9 特殊的类型限定符

MPLAB XC8 C 编译器支持一些特殊的类型限定符，使用户可以控制将具有静态存储持续时间的对象放入特定的地址空间。

4.3.9.1 Memx 地址空间限定符

结合使用__memx 与 const 限定符指示对象将被放入程序空间。现在，这种将对象放入闪存的方法很大程度上是冗余的，因为只需使用 const 限定符便可执行相同的任务，请参见 4.3.8.1. const 类型限定符。针对任何将程序存储器映射到其数据存储空间的器件（例如 avrxmega3 和 avrtiny 器件）进行编译时，均不需要__memx 限定符。

使用该限定符的指针可以按照与指向 const 的指针类似的方式引用数据存储空间和程序存储空间，请参见 4.3.6.2.1. 指向两个存储空间的指针。

4.3.9.2 Flash 限定符

结合使用__flash 与 const 限定符指示对象应位于不同的程序存储段中。该段可使用项目的链接描述文件重新定位（需要时），但必须完全链接到最低 64 KB 闪存段中。与__flash 限定符关联的段默认位于最低闪存段（段 0）中，占用最低 64 KB 地址范围。

对于没有存储器映射闪存的器件，将使用 lpm 指令读取数据。

使用该限定符的指针可引用程序存储器，请参见 4.3.6.2. 数据指针。

4.3.9.3 Flashn 限定符

结合使用__flash*n* 与 const 限定符（其中 *n* 的范围为 1 至 5）指示对象应位于不同的程序存储段中。为了使程序正常运行，这些段必须使用项目的链接描述文件置于正确的闪存段中。在目标器件实现了闪存段的前提下，__flash1 限定符与应位于 64 KB 至 128 KB 地址范围的闪存段中的闪存段 1 相关联，__flash2 限定对象放入与 128 KB 至 196 KB 范围的段相关联的段，依此类推。很明显，并非所有__flash*n* 限定符均适用于所有器件。

对于没有存储器映射闪存的器件，将使用 RAMPZ 寄存器和 elpm 指令读取数据，从而读取整个程序存储器。

使用这些限定符的指针可引用程序存储器，请参见 4.3.6.2. 数据指针。

4.3.10 属性

编译器关键字 __attribute__() 用于指定对象或结构字段的特殊属性。在该关键字后面的括号内放置一个由相关属性组成的逗号分隔列表，例如：

```
__attribute__((unused))
```

属性可放在对象定义中的任何位置，但通常按以下示例所示来放置。

```
char __attribute__((weak)) input;
char input __attribute__((weak));
```

注：变量属性在整个项目的使用务必一致。例如，如果某个变量在一个具有 aligned 属性的文件中定义并在另一个没有 aligned 属性的文件中声明了 extern，则可能导致链接错误。

4.3.10.1 Absdata 属性

absdata 属性指示可通过接受绝对地址的 lds 和 sts 指令访问对象。只有 ATtiny40 等精简型 AVR Tiny 内核才支持该属性。

必须确保相应的数据位于地址范围 0x40-0xbf 中，以避免出现超出范围错误。一种实现方法是使用适当的链接描述文件。

4.3.10.2 Address 属性

address(*addr*) 属性用于将对象标识符的别名设为可能超出 IO 地址范围的外设地址。此类标识符随后可用于寻址这些存储器映射外设。例如：

```
volatile int porta __attribute__((address (0x600)));
```

将在内部使 porta 等于地址 0x600。

该属性不影响存储器中任何普通对象的分配。要将对象置于普通数据存储器中的指定地址处，请使用 __at() 说明符（见 4.4.4. 绝对变量）。

4.3.10.3 Aligned 属性

aligned(*n*) 属性用于将对象的地址与下一个 *n* 字节边界对齐，其中的 *n* 为该属性的数字参数。如果使能了 CCI（见 3.6.3.3. Ext 选项），则可使用更易移植的宏 __align(*n*)（注意拼写不同）。

该属性也可在结构成员上使用。此类成员将与结构内的指定边界对齐。

如果忽略了对齐参数，变量的对齐将设置为 1（基本数据类型的最大对齐值）。

请注意，aligned 属性用于增大（而非减小）变量的对齐。要减小变量的对齐值，请使用 packed 属性。

4.3.10.4 Deprecated 属性

deprecated 属性将在使用指定对象时生成一条警告。如果选项字符串参数存在，它将打印到警告中。如果使能了 CCI（见 3.6.3.3. Ext 选项），则可使用更易移植的宏 __deprecate（注意拼写不同）。

4.3.10.5 Io 属性

使用 io(*address*) 属性定义的对象表示 I/O 空间中指定地址处的存储器映射外设。示例：

```
volatile int porta __attribute__((io(0x22)));
```

如果未使用地址，则不会为对象赋予地址，但编译器仍将在适用时使用 in 和 out 指令，前提是某个其他模块将为对象赋予地址。例如：

```
extern volatile int porta __attribute__((io));
```

4.3.10.6 Io_low 属性

io_low(*address*) 属性与 io(*address*) 属性类似，但可额外告知编译器对象位于 I/O 区域的低半部分，允许使用 cbi、sbi、sbic 和 sbis 指令。该属性还具有未指定地址的 io_low 形式。

4.3.10.7 Packed 属性

packed 属性强制对象或结构成员采用尽可能小的对齐方式，即，不会为声明分配对齐填充存储。packed 与 aligned 属性结合使用时，可针对变量或结构成员的类型设置大于或小于默认对齐的任意对齐限制。

如果使能了 CCI（见 3.6.3.3. Ext 选项），则可使用更易移植的宏__pack（注意拼写不同）。

4.3.10.8 Persistent 属性

persistent 属性用于指示运行时启动代码不应清零对象。

默认情况下，在启动时会清零未显式初始化的具有静态存储持续时间的 C 对象。这与 C 语言的定义是一致的。但是，有些情况下希望在复位时保存一些数据。persistent 属性将具有静态存储持续时间的对象存储到不会被运行时启动代码更改的单独存储区中。

例如，以下代码可确保启动时不会清零变量 intvar 和 mode：

```
int __attribute__((persistent)) mode;

void test(void)
{
  static int __attribute__((persistent)) intvar;   /* must be static in this context */
  ...
}
```

如果使能了 CCI（见 3.6.3.3. Ext 选项）且包含了<xc.h>头文件，则可使用更易移植的宏__persistent。例如，以下符合 CCI 标准的代码与上述代码类似：

```
#include <xc.h>
__persistent int mode;

void test(void)
{
  static __persistent int intvar;   /* must be static in this context */
  ...
}
```

4.3.10.9 Progmem 属性

结合使用 progmem 属性与 const 限定符可将对象放入程序存储器。或者，也可使用<avr/pgmspace.h>定义、映射到该属性的更易移植的 PROGMEM 宏。例如：

```
#include <avr/pgmspace.h>
const unsigned char PROGMEM romChar = 0x55;
```

在支持命名地址空间的 AVR GCC 编译器之前，这是惟一可将对象放入闪存的方法，因此，如今该方法是为了兼容传统项目或提高从其他平台移植的代码的可移植性。

与程序存储器中可直接读取的 const 限定对象不同，使用 progmem 属性定义的对象必须使用适当的库函数（如 pgm_read_byte_near()）读取。编程器负责使用正确的库函数；但用于访问以该方式定义的对象的代码通常很高效，因为编译器可更准确地获知对象的位置。

4.3.10.10 Section 属性

section(*section*)属性用于将对象分配到用户指定的段，以免编译器将其放入默认的段。如果使能了 CCI（见 3.6.3.3. Ext 选项），则可使用更易移植的宏__section(*section*)。关于使用该限定符的完整信息，请参见 4.14.2. 更改和链接所分配的段。

例如，以下符合 CCI 标准的代码将 foobar 放入名为 myData 的惟一段：

```
int __section("myData") foobar;
```

4.3.10.11 Unused 属性

unused 属性用于向编译器指示可能未使用对象，并且在检测到未使用对象时不应生成警告。

4.4 存储器分配和访问

定义的对象会自动被分配到某个存储区。在一些情况下，可更改该分配。后续章节将讨论存储区和分配。

4.4.1 地址空间

大多数 8 位 AVR 器件都采用哈佛架构，该架构具有独立的数据存储空间（RAM）和程序存储空间。在一些器件上，程序存储空间映射到数据存储空间且可从数据存储空间访问。一些器件还实现了映射到存储器的 EEPROM。

通用 RAM 与 SFR 共用相同的数据空间；但 SFR 出现在可通过用于访问 I/O 空间的指令（如 `in` 和 `out` 指令）访问的地址范围（在器件数据手册中称为 I/O 空间）内。如果器件中的 SFR 数多于指令可寻址的 SFR 数，则寄存器位于高地址处并通过 `st` 和 `ld` 指令组访问。

程序存储空间主要用于可执行代码，但也可在其中存放数据。不同器件系列使用几种方法来在该存储器中存放和读取数据，但存放到其中的所有对象都是只读的。

4.4.2 数据存储器中的对象

大多数变量最终会被存放到数据存储器。由于自动对象和静态存储持续时间对象存储器分配方式根本上的不同，以下将分别对它们进行讨论。

4.4.2.1 静态存储持续时间对象

并非基于堆栈分配空间的对象（除了自动对象、参数对象和 `const` 限定对象以外的所有对象）具有静态（永久）存储持续时间，并由编译器放入数据存储器中。

分配分两个步骤执行。编译器将每个对象放入特定的段，然后链接器将这些段放入相关存储区。放入后，可以全面解析对象在这些段中的地址。

编译器会考虑两类对象，这两类都与程序开始执行时对象应包含的值有关。每个对象类别都有一系列相应的段（见 4.14.1. 编译器生成的段），具体在下面列出。

bss 这些段包含任何未初始化的对象，运行时启动代码会清零这些对象。

data 这些段包含已初始化对象的 RAM 映像，运行时启动代码会将这些对象的非零值复制到这些段中。

有关运行时启动代码如何运行的信息，请参见 4.9. main、运行时启动和复位。

4.4.2.1.1 static 对象

所有 `static` 对象都有静态存储持续时间，即使是在某个函数内部定义的、作用域限制为该函数的局部 `static` 对象也如此。即使局部 `static` 对象也可被指针引用并且程序将确保这些对象在定义它们的函数的调用之间保留其值，除非通过一个指针显式地进行修改。

属于 `static` 类型的对象在程序执行过程中仅进行一次初始值赋值。因此，它们会生成效率高于已初始化 `auto` 对象的代码，后者在每次定义它们的块开始执行时都需要进行赋值。但与 `auto` 对象不同，`static` 对象的初始值必须为常量表达式。

同时被指定为 `const` 的所有 `static` 变量都将存储在程序存储器中。

4.4.2.1.2 更改默认分配

可通过以下任一方法为具有静态存储持续时间的对象更改默认存储器分配：

- 使用说明符
- 将对象设为绝对对象
- 将对象放入其自己的段中并显式链接该段

可以使用 `__memx` 说明符（见 4.3.9.1. Memx 地址空间限定符）将变量放入组合闪存和数据段中。

如果只有几个对象需要定位到数据存储空间中的特定地址处，则可以将这些对象设为绝对对象（如 4.4.4. 绝对变量所述）。将变量设为绝对变量之后，它们的地址将硬编码到生成的输出代码中，它们将不再放入段中，不遵循正常的存储器分配过程。

通过更改默认的链接器选项，可以将分配了不同类别的静态存储持续时间对象的 `.bss` 和 `.data` 段作为一个整体进行移动。例如，可以移动所有持久性变量。

此外，也可在使能 CCI（见 3.6.3.3. Ext 选项）后通过使用__section()说明符（见 4.3.10.10. Section 属性）将对象放在特定位置，以将这些对象分配到惟一的段中，然后通过一个选项将该段链接到所需地址。关于更改段的默认链接器选项的更多信息，请参见 4.14.2. 更改和链接所分配的段。

4.4.2.2 自动存储持续时间对象

具有自动存储持续时间的对象（如自动对象、参数对象和临时变量）基于编译器实现的堆栈分配空间。临时对象也可能放入堆栈中。4.2.3. 堆栈介绍了 MPLAB XC8 和 8 位 AVR 器件使用的堆栈。

因为具有自动存储持续时间的对象不是在程序的整个执行时间中都存在，所以可以在对象不存在时回收存储器，并将其分配给程序中的其他对象。通常此类对象存储在某种形式的动态数据堆栈中，动态数据堆栈可以由每个函数简便地分配然后释放存储器。由于该堆栈用于在调用函数时创建新的函数对象实例，因此所有函数均可重入。

标准 const 限定符可用于自动对象，不会影响这些对象在存储器中的定位方式。这表示 const 限定局部对象仍为自动对象，将在数据存储空间的堆栈中为其分配存储空间。

4.4.2.2.1 对象长度限制

具有自动存储持续时间的对象无法设为大于对象出现时可用的堆栈空间。因此，允许的最大长度不固定，将取决于程序的执行。

4.4.2.2.2 更改默认分配

所有具有自动存储持续时间的对象均位于堆栈中，因此无法单独移动。这些对象无法设为绝对对象，也无法使用__section()说明符被赋予惟一的段。

4.4.3 程序空间中的对象

具有静态存储持续时间且使用 const 与__flash、__flash*n*、__memx 中的任意一个限定符或者使用 const 与 progmem 属性定义的对象总是放入程序存储器。当使能编译器的 const-in-progmem 功能（默认状态，或使用-mconst-data-in-progmem 选项设为显式状态，请参见 3.6.1.5. Const-data-in-progmem 选项）时，仅使用 const 类型限定符定义的对象也放入程序存储器。

-mno-const-data-in-progmem 选项用于禁止 const-in-progmem 功能，并强制仅使用 const 类型限定符定义的所有对象改为放入数据存储器。

avrtiny 和 avrxmega3 器件系列可轻松访问程序存储器对象，因为该存储器映射到数据地址空间中。对于其他器件系列，程序存储器有所不同，通过不同代码序列进行访问。

-mconst-data-in-config-mapped-progmem 选项（见 3.6.1.4. Const-data-in-config-mapped-progmem 选项）可用于支持该存储器映射功能的器件，使链接器将所有 const 限定数据放入一个 32 KB 段中并自动初始化相关 SFR 寄存器以确保这些对象映射到数据存储器中，这些对象在数据存储器中的访问会更有效率。

4.4.3.1 程序存储器对象的长度限制

仅使用 const 或者结合使用 const 和__memx 定义的对象仅受可用程序存储器限制，使用 progmem 属性定义的对象可跨多个闪存段，但必须使用来自 pgm_read_xxx_far()系列的函数访问它们。使用__flash 或__flashn 限定符定义的对象被限制在闪存的一个 64 KB 段中可用的存储空间，不得跨越段边界。

请注意，对于未将程序存储器映射到数据空间中的器件，除了数据本身以外，还需要使用额外的代码来读取程序存储器中的数据。该代码可能是仅包含一次的库代码；但用于读取程序存储器的代码序列通常比用于读取 RAM 的代码序列要长。

4.4.3.2 更改程序存储器对象的默认分配

可通过以下任一方法为程序存储器中的对象更改默认存储器分配：

- 将对象设为绝对对象
- 将对象放入其自己的段中并显式链接该段

如果只有几个程序存储器对象需要定位到程序存储空间中的特定地址处，则可以将对象设为绝对对象。4.4.4. 绝对变量对绝对变量进行了介绍。

此外，也可在使能 CCI（见 3.6.3.3. Ext 选项）后通过使用__section()说明符（见 4.3.10.10. Section 属性）将程序存储器中的对象放在特定位置，以将这些对象分配到惟一段中，然后通过一个选项将该段链接到所需地址。关于更改段的默认链接器选项的更多信息，请参见 4.14.2. 更改和链接所分配的段。

4.4.4 绝对变量

对象可以通过在其声明之后加上__at(*address*)构造而定位到特定地址处；其中，*address* 是对象在存储器中定位到的位置。要编译该构造，必须使能 CCI（见 3.6.3.3. Ext 选项）且必须包含<xc.h>。此类对象称为绝对对象。

将变量设为绝对变量是将对象放入用户定义的存储单元的最简单方法，但这种方法仅允许放置在必须在编译前已知且必须针对要重定位的每个对象指定的地址处。

4.4.4.1 程序存储器中的绝对对象

任何具有静态存储持续时间以及具有文件作用域的 const 限定对象均可置于程序存储器中的绝对地址处。

例如：

```
const int settings[] __at(0x200) = { 1, 5, 10, 50, 100 };
```

会将数组 settings 放置在程序存储器中的地址 0x200 处。

4.4.5 EEPROM 中的变量

对于带有片上 EEPROM 的器件，编译器提供了几种方法来访问该存储器，如以下章节所述。

4.4.5.1 Eeprom 变量

可以通过使用 section 属性指定将对象放入.eeprom 段来将对象放入 EEPROM 中。如果所选器件不支持该属性，将生成一条警告。请参见器件数据手册来了解器件是否支持该存储器。

宏 EEMEM 在<avr/eeprom.h>中定义，还可用于简化 EEPROM 中的对象的定义。例如，以下两个定义均可创建将存储到 EEPROM 中的对象。

```
int serial __attribute__((section(.eeprom)));
char EEMEM ID[5] = { 1, 2, 3, 4, 5 };
```

该段中的对象会被清零或初始化（需要时），就如同普通的基于 RAM 的对象；但是，初始化过程不由运行时启动代码执行。初始值会被放入 HEX 文件中，并在对器件编程时烧写到 EEPROM 中。因此，如果在程序执行期间修改 EEPROM，然后复位器件，这些对象将不会包含启动时在代码中指定的初始值。

请注意，eeprom 段中的对象必须全部使用 const 类型限定符或全部不使用该限定符。

4.4.5.2 Eeprom 访问函数

必须使用在包含<avr/eeprom.h>后可访问的特殊库程序（如 eeprom_read_byte()和 eeprom_write_word()）访问 EEPROM 中的对象。

用于访问基于 EEPROM 的对象的代码会比用于访问基于 RAM 的对象的代码更长和更慢。如果某些值需要在复杂计算中多次使用，可考虑使用这些程序将这些值从 EEPROM 复制到基于 RAM 的常规对象中。

4.4.6 寄存器中的变量

可以定义一个变量并将其与指定寄存器相关联；但是，通常建议由编译器进行寄存器分配以实现最优结果并避免代码失败。

寄存器变量是通过使用 register 关键字并指示所需寄存器来定义的，如以下示例所示：

```
register int input asm("r12");
```

必须使用加双引号的有效 AVR 器件寄存器名称作为 asm()的参数。此类定义放在函数内部或外部均可，但是无法将变量设为 static。对局部寄存器变量的支持被限制为在调用扩展嵌入汇编时为输入和输出操作数指定寄存器。

编译器将在当前编译单元的持续时间内保留分配的寄存器，但库程序可能会破坏分配的寄存器，因此建议分配通常由函数调用保存和恢复的寄存器（4.6. 寄存器使用所述的由调用保存的寄存器）。

无法获取 register 变量的地址。

4.4.7 动态存储器分配

动态存储器分配是程序在执行期间手动请求和释放任意大小存储器块的一种方法，在 MPLAB XC 编译器支持的大多数器件中以某种形式提供。

程序可调用 malloc()或 calloc()库函数在运行时分配存储器块。与对象的静态或动态分配不同，所请求的大小无需为常量表达式（可变长度的自动数组（支持时）除外）。包含所有已分配存储器块的连续存储区统称为堆。

不再需要的已分配存储器块可由使用 free()库函数的程序释放。已释放的存储器有可能在之后重新分配。如果需要更改已分配存储器块的大小，可使用 realloc()函数。

根据所选的目标器件和选项，MPLAB XC 编译器可能实现完整和/或简化的动态存储器分配方案，也可能完全不允许动态存储器分配。下节介绍了使用该编译器进行动态存储器分配的确切操作。有关动态存储器分配函数的语法及其使用，请参见 *Microchip Unified Standard Library Reference Guide*。

4.4.7.1 AVR 器件的动态存储器分配

MPLAB XC8 C 编译器针对所有目标器件实现了一种不受限制的动态存储器分配方案；但是，avrtiny 器件等小型器件不太可能有足够的程序存储器来容纳所需库函数，也没有足够的数据存储器使动态存储器分配发挥作用。

动态存储器分配方案允许使用所有标准存储器分配库函数。最初，存储器可按需分配，仅受限于为堆保留的最大大小。分配后释放的存储器块与相邻空闲存储器合并，并归入类似大小的 bin 中。该存储器将被视为在后续请求额外存储器时重新分配。

尽管分配方案会尝试高效利用存储器，但总是会发生碎片化。请谨慎分配和释放存储器。程序设计人员应尽可能谨慎，避免使用动态存储器分配。

尽管成功调用存储器分配函数将返回大小足以存储所请求字节数的块，但分配的各个块的总大小可能大于请求的大小，前者取决于程序中是否调用了 free()函数。

如果从未调用 free()，则存储器分配函数将尽可能分配所请求存储器大小的块并返回指向该块的指针。在这种情况下，如果存在足够的空闲存储器，则用于分配 1 字节存储器的请求（如 malloc(1)）将分配 1 字节块。

如果程序中调用了 free()，则编译器将随分配的每个块包含额外的存储器以管理该存储器。在这种情况下，分配的每个块将包含 2 字节宽的报头，并且将额外使用 2 个字节作为空闲列表指针，因此可为块分配的最小存储器量为 4 字节。由于地址在成功分配后返回的存储器之前的 2 个字节包含报头信息，因此程序永远不应假设该存储器可能包含分配的用户数据。

无论是否调用了 free()函数，分配 0 字节存储器的请求（如 malloc(0)）都将被视为分配 1 字节存储器的请求，如果成功，将返回指向分配的存储器的指针。

可通过调整一些变量来自定义分配函数（如 malloc()）的行为。应先更改这些变量再分配存储器，请记住，调用的任何库函数均可能使用动态存储器。

变量__malloc_heap_start 和__malloc_heap_end 可用于限制由 malloc()和 calloc()函数分配的存储器。这两个变量以静态方式初始化为分别指向__heap_start 和__heap_end。__Heap_start 变量设置为紧随在静态对象分配后由最佳适应分配器使用的最后一个数据存储器地址后的地址，__heap_end 设置为 0，该值将堆置于堆栈之下。

如果堆位于外部 RAM 中，必须相应地调整__malloc_heap_end。为此，可在运行时直接写入该变量来完成，或者在链接时调整符号__heap_end 的值自动完成。

以下示例给出的选项可将映射到链接描述文件中的.data 输出段的输入段重定位到外部 RAM 中的存储单元 0x1100。由于它们是 RAM 中的地址，在指定地址中 MSb 将置 1。堆将扩展至地址 0xffff。

```
-Wl,-Tdata=0x801100,--defsym=__heap_end=0x80ffff
```

如果普通变量保留在内部 RAM 中时堆应位于外部 RAM 中，可以使用以下选项。请注意，在本示例中，堆与堆栈之间的存储空间有一个始终无法通过普通变量或动态存储器分配访问的“孔”。

```
-Wl,--defsym=__heap_start=0x802000,--defsym=__heap_end=0x803fff
```

如果__malloc_heap_end 为 0，存储器分配程序会尝试检测栈底以避免扩展堆时发生堆栈-堆冲突。它们分配的存储空间不会超出由__malloc_margin 字节的缓冲区指定的当前堆栈限制。因此，可能中断当前函数的中断程序的所有可能堆栈帧以及所有后续的嵌套函数调用均不得请求更多的堆栈空间，否则它们将面临与数据段发生冲突的风险。__malloc_margin 的默认值设置为 32。

4.4.8 存储器模型

MPLAB XC8 C 编译器 不使用固定存储器模型来改变变量的存储器分配。存储器分配是完全自动的，不存在任何存储器模型控制。

4.5 操作符和语句

MPLAB XC8 C 编译器支持所有 ANSI 操作符，其中一些操作符的行为以实现定义的方式表现出来，请参见 6. 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。

4.5.1 整型提升

整型提升更改了一些表达式值的类型。MPLAB XC8 C 编译器始终按照 C 标准执行整型提升，但此操作会使未预料到此类行为的人员感到困惑。

当操作符具有多个操作数时，这些操作数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。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。

整型提升指的是将枚举类型、char、short int 或位域类型的有符号或无符号形式隐式转换为 signed int 或 unsigned int 类型。如果转换结果可以通过 signed int 表示，则它就是目标类型，否则则转换为 unsigned int。

假设存在以下示例：

```
unsigned char count, a=0, b=50;
if(a - b < 10)
  count++;
```

a - b 的 unsigned char 结果为 206（它大于 10），但在执行减法运算之前，a 和 b 都会通过整型提升转换为 signed int。对这些数据类型执行减法运算的结果为-50（它小于 10），因此会执行 if() 语句的主体。

如果减法运算的结果为无符号量，则应用强制类型转换，如以下示例所示，强制以 unsigned int 类型进行比较：

```
if((unsigned int)(a - b) < 10)
  count++;
```

经常发生的另一个问题是使用按位取反操作符~。该操作符会翻转值中的每个位。 假设存在以下代码。

```
unsigned char count, c;
c = 0x55;
if( ~c == 0xAA)
  count++;
```

如果 c 包含值 0x55，则通常假定~c 会产生 0xAA；但是，对于使用 16 位 int 的编译器，其结果为 0xFFAA，对于使用 32 位 int 的编译器，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 类型的操作数执行的。但在某些情况下，无论操作数是 char 类型还是 int 类型，运算的结果都是相同的。在这些情况下，编译器不会执行整型提升，以提高代码效率。假设存在以下示例。

```
unsigned char a, b, c;
a = b + c;
```

严格地说，该语句要求将 b 和 c 的值提升为 unsigned int 类型，执行加法运算，将加法运算的结果强制转换为 a 的类型，然后进行结果赋值。在这种情况下，无论是以 int 还是 char 执行加法运算，a 的赋值均相同，因此编译器可能会编码前者。

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 C 标准。

4.5.2 循环移位

C 语言未规定循环移位操作符；但是，它允许进行移位。可以按照下面的 C 代码示例针对 16 位整型执行循环移位。

```
unsigned char c;
unsigned int u;
c = (c << 1) | (c >> 7); // rotate left, one bit
u = (u >> 2) | (u << 14); // rotate right, two bits
```

4.5.3 switch 语句

默认情况下，跳转表用于优化 `switch()` 语句。`-fno-jump-tables` 选项用于阻止使用这些跳转表，改为使用由比较序列组成的序列。跳转表的执行速度通常更快，但对于大多数跳转以 `default` 标号为目标的 `switch()` 语句，跳转表可能会产生过量的代码。

跳转表使用有限范围的 `lpm` 汇编指令。当针对程序存储器大于 64 KB 的器件编译自举程序时，务必使用`-fno-jump-tables` 选项。

4.6 寄存器使用

编译器基于 C 源代码生成的汇编代码将使用 AVR 寄存器组中的一些特定寄存器。假定一些寄存器通过函数调用存放其值。

编译器可为函数内的值分配由调用使用的寄存器（r18-r27 和 r30-r31）。函数无需保存这些寄存器的内容。这些寄存器可在手写汇编子程序中使用。由于这些程序调用的任何 C 函数均可能破坏这些寄存器，因此调用程序必须确保将其内容保存下来，以便后续在适当时进行恢复。

此外，编译器还可为局部数据分配由调用保存的寄存器（r2-r17 和 r28-r29）；但 C 函数必须保存这些寄存器。手写汇编子程序负责保存这些寄存器并在需要时进行恢复。即使编译器已为传递的参数分配了寄存器，也必须保存这些寄存器。

临时寄存器 r0 可能被 C 函数破坏，但中断处理程序会保存其内容。

编译器会假定零寄存器 r1 总是包含值 0。该寄存器可在手写汇编程序中用于中间值，但使用后必须清零（如使用 `clr r1`）。请注意，乘法指令在 r1-r0 寄存器对中返回其结果。进入中断处理程序时保存 r1 并将其清零，退出时恢复 r1（非零时）。

中断程序负责保存和恢复自身使用的所有寄存器（见 4.8.4. 现场切换）。

下表列出了在整个程序中具有专用功能的寄存器。

表 4-7. 专用寄存器

寄存器名称	适用器件
r0	临时寄存器
r1（__zero_reg__）	零寄存器（存放 0 值）
r28 和 r29	帧指针（Y 指针）

4.7 函数

函数可以根据 C 语言按照通常的方式编写。以下几节介绍了与函数相关联的实现特性。

4.7.1 函数说明符

除了影响函数链接的标准 C 说明符 `static` 以外，还有几种非标准函数说明符，以下几节对此进行了介绍。

4.7.1.1 inline 说明符

`inline` 函数说明符用于建议编译器尽可能将对指定函数的调用替换为函数的主体。这样可以提高程序的执行速度。

任何具有内部链接的函数（使用 `static` 说明符的函数）均可作为内联函数，但 C99 语言标准对具有外部链接的函数如何使用 `inline` 说明符施加了限制。使用 `inline` 函数说明符（而非 `extern`）的函数的文件作用域声明称为内联

定义，仅提供该函数的外部定义的替代定义。可以在其他模块中提供该函数的附加外部定义，也可以通过提供使用 extern 的声明在同一模块中将函数的定义设为外部定义。

以下给出了一个将函数设为内联函数的示例。

```
extern int combine(int x, int y);  // make this an external definition

inline int combine(int x, int y) {
  return 2 * x - y;
}
```

编译器可自行编码对内联定义（内联函数）或外部定义的调用。代码不应做出关于是否进行内联的任何假设。-Winline 选项可用于在标记内联的函数无法替换时发出警告，并给出失败原因。

对内联定义的所有函数调用都会被编码为如同调用被替换为所调用函数主体的形式。这将在汇编代码级执行。只有使能优化器（1 级或更高）时，才会发生内联，但可以通过使用 always_inline 属性要求某个函数总是内联。

如果发生了内联，则程序的执行速度会提高，因为它可以消除与调用相关联的调用与返回序列。如果与内联函数主体关联的汇编代码极短，则代码长度会减小；但如果内联函数的主体大于它替换的调用/返回序列，则代码的长度会增大。只有对于生成少量汇编代码的函数，才应考虑使用该说明符。请注意，函数主体中的 C 代码量并不是其生成汇编代码长度的良好指标。可以考虑使用预处理器宏作为内联函数的替代方法。

4.7.1.2 Nopa 说明符

如果使能了 CCI（见 3.6.3.3. Ext 选项），__nopa 说明符将扩展为__attribute__((nopa,noinline))，从而禁止过程抽象以及相关函数的内联。这将确保不会对内联代码进行过程抽象。

4.7.2 函数属性

编译器关键字__attribute__()用于指定函数的特殊属性。在该关键字后面的括号内放置一个由相关属性组成的逗号分隔列表，例如：

```
__attribute__((weak))
```

属性可放在对象定义中的任何位置，但通常按以下示例所示来放置。

```
char __attribute__((weak)) input(int mode);
char input(int mode) __attribute__((weak));
```

4.7.2.1 Naked 属性

naked 属性允许编译器构造必要的函数声明，同时允许函数主体为汇编代码。指定函数不会包含编译器生成的前言和结语序列。只有基本 asm()语句能够安全地包含在 naked 函数中。不要使用扩展 asm()或者基本 asm()与 C 代码组合，因为它们不受支持。

4.7.2.2 no_gccisr 属性

no_gccisr 属性用于禁止与中断函数相关联的现场切换代码的所有优化。另请参见 3.6.6.20. ISR Prologues 选项。

4.7.2.3 Nopa 属性

nopa 属性指示不该将过程抽象优化应用于函数。

4.7.2.4 Os_main/os_task 属性

在 AVR 上，具有 OS_main 或 OS_task 属性的函数不会通过其前言/结语保存/恢复任何由调用保存的寄存器。

如果保证在进入函数时禁止中断，可以使用 OS_main 属性。这样，当必须通过更改堆栈指针为局部变量设置帧时，可以节省资源。

如果不能保证在进入函数（例如，多线程操作系统中的任务函数）时禁止中断，可以使用 OS_task 属性。在这种情况下，由全局中断允许标志的保存/清零/恢复来保护堆栈指针寄存器的更改。

该属性与 naked 函数属性的区别如下：

- naked 函数没有返回指令，而 OS_main 和 OS_task 函数有 ret 或 reti 返回指令。
- naked 函数不会设置帧（针对局部变量）或帧指针，而 OS_main 和 OS_task 则会按需设置。

4.7.2.5 Section 属性

`section("section")`属性用于将函数分配到用户指定的段，以免编译器将其放入默认的段。如果使能了 CCI（见 3.6.3.3. Ext 选项），则可使用更易移植的说明符`__section(section)`。关于使用该说明符的完整信息，请参见 4.14.2. 更改和链接所分配的段。

例如，以下符合 CCI 标准的代码会将与 `readInput` 函数相关联的代码放入名为 `myText` 的惟一段中：

```
int __section("myText") readInput(int port)
{ ... }
```

4.7.2.6 Weak 属性

`weak` 属性用于使声明作为弱符号发出。弱符号指示如果同一符号的全局版本可用，则应改为使用该版本。

如果对外部符号引用应用 `weak` 属性，则链接时无需使用该符号。例如：

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

在以上程序中，如果有一些其他模块未定义 `s`，程序仍将进行链接但不会为 `s` 分配地址。 有条件地验证是否已定义 `s`（并在其值存在时返回值）。 其他情况下返回 0。该功能有很多用途，主要是为了提供可与可选库链接的通用代码。

4.7.3 可执行代码的分配

与 C 函数相关联的代码总是放置在链接到目标器件的程序存储器的`.text` 段中。

4.7.4 更改默认的函数分配

可以通过以下任一方法更改函数的默认存储器分配：

- 将函数设为绝对函数
- 将函数放入其自己的段中并链接该段

显式将各个函数置于已知地址处的最简单方法是，按照用于绝对变量的类似方式使用`__at()`构造将这些函数设为绝对函数。必须使能 CCI 以接受该语法，并且必须包含<xc.h>。

如果和绝对函数关联的代码与来自其他绝对函数的代码发生重叠，则编译器将发出一条警告。编译器不会将与普通函数关联的代码定位到绝对函数顶部。

以下给出了将函数放置在地址 400h 处的绝对函数示例：

```
int __at(0x400) mach_status(int mode)
{
  /* function body */
}
```

如果对于中断函数使用该构造，它只会影响与中断函数主体关联的代码的位置。与中断向量关联的中断现场切换代码不会重定位。

可以使用`__section()`说明符将函数分配到用户定义的 psect 中（见第 3.15.2 节“更改和链接所分配的段”），以便将这个新段链接到所需地址。这种方法最为灵活，可将函数置于固定地址处、其他段之后或某个地址范围内的任一位置处。当绝对函数与中断函数结合使用时，只会影响中断函数主体的位置。

无论如何定位函数，请谨慎选择其地址。尽可能避免存储器碎片化，防止链接器出现错误的可能性提高。

4.7.5 函数长度限制

对于所有器件，常规函数的生成代码仅受可用程序存储器限制。更多详细信息，请参见 4.7.3. 可执行代码的分配。

4.7.6 函数参数

MPLAB XC8 使用固定约定来向函数传递参数。用于传递参数的方法取决于所涉及参数的长度和数量。

注：名称“实参”（argument）和“形参”（parameter）通常可以互换，但实参通常是指传递给函数的值，形参则是指由函数定义的用于存储实参的变量。

参数通过寄存器传递给函数，并根据存放对象的需要占用相应数量的寄存器。但是，寄存器是成对分配的，因此至少将为每个参数分配两个寄存器，即使参数为单字节也是如此。这样可以更高效地使用 AVR 指令集。

可用的第一个寄存器对是 r24-r25，其后的寄存器对可低至寄存器 r8。例如，如果调用原型为

```
int map(unsigned long a, char b);
```

的函数，则 4 字节形参 a 的实参将传递到寄存器 r22 至 r25 中，其中 r22 存放最低有效字节，r25 存放最高有效字节；形参 b 的实参将分配到寄存器 r20 和 r21。

如果所有可用寄存器完成分配后还有参数要传递，则这些参数基于堆栈进行传递。

具有可变参数列表的函数（printf()等）的参数均基于堆栈传递。

4.7.7 函数返回值

函数的返回值通常返回到寄存器中。

单字节返回值返回到 r24 中。多字节返回值返回到所需数量的寄存器中，最高寄存器为 r25。因此，16 位值返回到 r24-r25 中，32 位值返回到 r22-r25 中，依此类推。

4.7.8 调用函数

函数使用 rcall 指令进行调用。如果目标器件的程序存储器大于 8 KB，则将使用更大的调用指令，以便能够到达程序存储器中任意位置处的任何函数。

如果能够保证所有调用目标均位于 rcall 指令的范围内，则可以使用-mshort-calls 选项（见 3.6.1.14. Short-calls 选项）请求短调用。

4.8 中断

MPLAB XC8 编译器包含了允许完全从 C 代码中处理中断的特性。中断函数通常称为中断服务程序或 ISR。

下面是使用中断需遵循的一般步骤。有关这些步骤的更多详细信息，请参见以下部分。

- 编写所需数量的中断函数。可以考虑使用一个或多个额外的函数来处理未用中断源的意外触发。
- 在代码中的适当点，允许所需的中断源。
- 在代码中的适当点，使能全局中断标志。

中断函数不能直接从 C 代码中调用（因为使用了不同的返回指令），但可以调用其他函数，例如用户定义的函数和库函数。

中断代码指的是由于发生中断而执行的任何代码，包括从 ISR 中调用的函数和库代码。中断代码在执行相应的中断返回指令时完成。这与主干代码形成对比，后者是一个独立的应用程序，通常是在复位后执行的程序的主要部分。

4.8.1 编写中断服务程序

编写普通 ISR 时，请遵循以下准则。

- 通过使能 CCI（3.6.3.3. Ext 选项）和使用__interrupt()说明符编写每个 ISR 原型。这样创建的函数具有适当的名称、原型和属性。
- 如果需要，在处理完源代码后将相关中断标志清零，但通常不需要此操作。
- 只有绝对必要时才在 ISR 主体内重新允许中断。ISR 返回时将自动重新允许中断。
- 使 ISR 尽可能简短。复杂代码通常将使用更多寄存器，这将增大现场切换代码的长度。

编译器会以不同于其他函数的方式处理中断函数，即生成代码来保存和恢复函数使用的所有寄存器，并使用一条特殊的指令返回。

硬件将在执行中断时全局禁止中断。

通常，每个中断源都有相应的中断标志位，可在控制寄存器中访问。这些标志置 1 时，表示已满足特定中断条件。中断标志通常在处理中断的过程中清零（在调用中断处理程序时清零或通过读取特定硬件寄存器清零）；但是，在一些实例中必须通过代码手动将标志清零。否则可能导致在当前 ISR 返回后立即再次触发中断。

SFR 中的标志位具有惟一的属性，可通过向这些标志位写入逻辑 1 来将其清零。要利用该属性，应直接写入该寄存器而非使用可能执行读-修改-写操作的任何指令序列。因此，要将 TC0 中断标志寄存器中的 TOV0 定时器溢出标志清零，请使用以下代码：

```
TIFR = _BV(TOV0);
```

可保证将 TOV0 位清零，同时使剩余位保持不变。

下面给出了中断函数的示例。

```
void __interrupt(SPI_STC_vect_num) spi_Isr(void) {
  process(SPI_ClientReceive());
  return;
}
```

请注意，`__interrupt()`说明符的参数是一个向量编号，可在程序中包含<xc.h>后用作以 `vect_num` 为结尾的宏。

可以使用由<avr/interrupt.h>定义的宏和属性创建更复杂的中断函数定义，如以下示例所示。

如果没有要针对中断源执行的代码但想要确保程序将在中断意外触发时继续正常工作，则可以使用 `EMPTY_INTERRUPT()`宏和中断源参数创建空 ISR。

```
#include <avr/interrupt.h>
EMPTY_INTERRUPT(INT2_vect);
```

特殊中断源符号 `BADISR_vect` 可与 `ISR()`宏结合使用来定义可处理任何未定义中断的函数。如果未定义该函数，则未定义中断将触发器件复位。例如：

```
ISR(BADISR_vect) {
  // place code to process undefined interrupts here
  return;
}
```

如果希望允许嵌套中断，可以手动将嵌入 `sei` 指令添加到 ISR 以重新使能全局中断标志；但是，也可为 `ISR()`宏使用一个参数来通过编译器将该指令添加到中断程序开头。例如：

```
ISR(IO_PINS_vect, ISR_NOBLOCK)
{ ... }
```

`ISR_NOBLOCK` 扩展为`__attribute__((interrupt))`，后者可代替前者使用。

如果一个 ISR 将与多个中断向量结合使用，则可通过常规方式（使用`__interrupt()`）为一个向量定义该 ISR，然后将该 ISR 重用于使用 `ISR_ALIASOF()`参数的其他向量定义。

```
void __interrupt(PCINT0_vect_num)
{ ... }
ISR(PCINT1_vect, ISR_ALIASOF(PCINT0_vect));
```

在某些情况下，ISR 执行的编译器生成的现场切换代码可能并非最优。在此类情形下，可以请求编译器忽略该现场切换代码，改为自行提供。为此，可以使用 `ISR_NAKED` 参数，如以下示例所示。

```
ISR(TIMER1_OVF_vect, ISR_NAKED)
{
  PORTB |= _BV(0); // results in SBI which does not affect SREG
  reti();
}
```

请注意，编译器不会生成任何现场切换代码（包括中断指令的最终返回），因此必须编写所有相关切换代码和 `reti` 指令。在 SREG 寄存器被 ISR 修改时必须对其进行手动保存，并且编译器隐含的`__zero_reg__`总是为 0 的假设可能是错误的，例如，`mul` 指令后立即产生中断时。

4.8.2 更改默认的中断函数分配

如果要移动中断函数本身，可以使用__at()说明符（见 4.7.4. 更改默认的函数分配）。这样不会更改向量表的位置，而相应的表项仍将指向函数移动后的正确地址。

4.8.3 指定中断向量

如果定义了中断函数，则中断向量单元的填充过程是完全自动的（如 4.8.1. 编写中断服务程序所示）。编译器会自动将每个 ISR 入口点链接到适当的固定向量单元。

无法在运行时更改中断向量的存储单元，也无法更改链接到向量的代码。即，不能为同一向量定义备用中断函数，并在程序执行期间选择哪个函数将处于活动状态。如果为同一向量定义多个中断函数，将产生错误。

通过定义使用 BADISR_vect 作为其向量的中断函数，可以针对未在项目中显式指定的中断向量分配默认函数地址。

4.8.4 现场切换

编译器会在发生中断时自动将代码链接到项目中，保存当前状态，然后在中断返回时恢复该状态。

所有调用使用的寄存器都将由编译器生成的中断代码保存其内容。这即为现场保护或现场切换代码。

在中断函数返回之前，软件会自动恢复由软件保存的所有对象。恢复的顺序与保存现场时所用的顺序相反。

4.8.5 允许中断

包含<xc.h>之后，将有两个宏可供使用，它们控制所有可用中断的屏蔽。这两个宏为 ei()（用于允许或取消屏蔽所有中断）和 di()（用于禁止或屏蔽所有中断）。

在所有器件上，它们都会影响状态寄存器 SREG 中的 I 位。在允许了程序中需要的中断对应的中断允许位之后，应使用这些宏。

例如：

```
TIMSK = _BV(TOIE1);
ei(); // enable all interrupts
// ...
di(); // disable all interrupts
```

注：通常，不应在中断函数自身内部重新允许中断。在执行 reti 指令时，硬件会自动重新允许中断。在中断函数内重新允许中断可能在处理不当时导致代码失败。

除了全局允许中断，每个器件的特定中断都需要单独允许（如果需要该器件的中断）。当一些器件使其寄存器中的中断允许位保持置 1 时，外部中断和定时器中断具有系统级配置寄存器。

要修改 TIMSK 寄存器，请使用 timer_enable_int(*ints*)。通过 *ints* 传递的值应是中断允许位的位掩码，该值是特定于器件的。

要修改 GIMSK 寄存器（使用 AVR Mega 器件时为 EIMSK 寄存器，其他情况下为 GICR 寄存器），请使用 enable_external_int(*mask*)。如果未定义其中任何寄存器，该宏将不可用。

例如：

```
// Enable timer 1 overflow interrupts
timer_enable_int(_BV(TOIE1));
// Do some work...
// Disable all timer interrupts
timer_enable_int(0);
```

4.8.6 从中断程序访问对象

如果从中断程序读/写对象时其他函数也在访问这些对象，则会不安全。

建议使用 volatile 说明符（见 4.3.8.2. volatile 类型限定符）显式标记在中断和主干代码中访问的对象。编译器将限制对 volatile 对象执行的优化。

即使对象标记为 volatile 时，编译器也无法保证将以原子方式访问它们。对多字节对象执行的操作尤其如此。

除非可以保证以原子方式进行访问，否则应在修改中断函数所用对象的任何主干代码前后禁止中断。<avr/atomic.h>中提供了一些宏来帮助访问这些对象。

4.9 main、运行时启动和复位

退出复位时，程序将先执行由编译器添加的运行时启动代码，之后将控制转移到函数 main()。以下几节将介绍此序列。

4.9.1 main 函数

标识符 main 是一个特殊的标识符。在程序中必须总是有且仅有一个名为 main() 的函数。这是在程序中执行的第一个 C 函数。

由于程序不是由主机调用，编译器会在 main() 的末尾插入特殊的代码，该代码在该函数结束时（即执行了 main() 内的 return 语句，或者代码执行到达 main() 的终止右大括号）执行。这段特殊代码会导致执行跳转到地址 0，即所有 8 位 AVR 器件的复位向量。这实质上是执行一种软件复位。请注意，软件复位后的寄存器状态可能不同于硬件复位后。

建议 main() 函数不要结束。在 main() 中的代码前后或代码末尾添加一个永远不会终止的循环（如 while(1)），从而使该函数的执行永远不会终止。例如，

```
int main(void)
{
    // your code goes here
    // finished that, now just wait for interrupts
    while(1)
        continue;
}
```

4.9.2 运行时启动代码

C 程序要求先初始化一些特定的对象，并且器件处于某种特定状态，之后才会开始执行其 main() 函数。执行这些任务是运行时启动代码的工作，特别是（没有特定的顺序）：

- 对在定义时赋值的静态存储持续时间对象进行初始化
- 将非初始化静态存储持续时间对象清零
- 对寄存器或器件状态进行常规设置
- 调用 main() 函数

提供运行时启动代码的几个运行时启动代码目标文件之一将链接到程序中。

运行时启动代码会假定器件刚刚退出复位，寄存器将存放它们的上电复位值。请注意，当看门狗或 RST_SWRST_bm 复位器件时，寄存器将复位为其已知的默认设置；而跳转到复位向量将不会更改寄存器，寄存器将保持其之前的状态。

下表列出了用于存放运行时启动代码的段。

表 4-8. 在 main 之前使用的运行时启动代码段

段名	说明
.init0	弱绑定到__init()，请参见 4.9.3. 上电程序。如果用户定义了__init()，则复位后会立即跳转到它。
.init1	未使用。用户可定义。
.init2	在 C 程序中，弱绑定到用于初始化堆栈和清零__zero_reg__（r1）的代码。
.init3	未使用。用户可定义。
.init4	该段包含 libgcc.a 中用于将.data 的内容从程序存储器复制到数据存储器的代码，以及用于清零.bss 段的代码。
.init5	未使用。用户可定义。

.....（续）	
段名	说明
.init6	C 程序未使用。
.init7	未使用。用户可定义。
.init8	未使用。用户可定义。
.init9	调用 main() 函数。

main() 函数将返回到运行时启动代码也会提供的代码中。通过将代码放入下表列出的段中，可以在 main() 返回后执行代码。

表 4-9. 在 main 之后使用的运行时启动代码段

段名	说明
.fini9	未使用。用户可定义。
.fini8	未使用。用户可定义。
.fini7	在 C 程序中，弱绑定以初始化堆栈和清零__zero_reg__（r1）。
.fini6	C 程序未使用。
.fini5	未使用。用户可定义。
.fini4	未使用。用户可定义。
.fini3	C 程序未使用。
.fini2	未使用。用户可定义。
.fini1	未使用。用户可定义。
.fini0	在程序终止和完成任何_exit()代码（.fini9 至.fini1 段中的代码）后，转到无限循环。

4.9.2.1 对象的初始化

运行时启动代码的一个任务就是确保任何静态存储持续时间对象在程序开始执行之前包含其初始值。下例中的 input 就是一个例证。

```
int input = 88;
```

在上面的代码中，初始值（0x88）将作为数据存储在程序存储器中，并被复制到由运行时启动代码为 input 所保留的存储器中。为提高效率，初始值存储为数据块，并通过循环复制。

可以使用-Wl,--no-data-init 禁止对象初始化；但依赖于包含其初始值的对象的代码将失败。

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，并且运行时启动代码不会考虑这些对象。

注：已初始化的 auto 变量可能会影响代码性能，特别是对象长度较大时。可以考虑改用 static 局部对象。

在复位时需要保存内容的对象应使用__persistent 属性进行标记。这种对象链接到存储器的不同区域，运行时启动代码不会更改它。

相关信息

4.3.10.8. Persistent 属性

4.9.2.2 清零对象

对于具有静态存储持续时间且未赋值的对象，必须通过运行时启动代码在 main() 函数开始执行前清零，例如：

```
int output;
```

运行时启动代码会清零由未初始化对象占用的所有存储单元，因此它们在 main() 执行之前将包含零。

可以使用-Wl,--no-data-init 禁止对象清零；但依赖于包含其初始值的对象的代码将失败。

在复位时需要保存内容的对象应使用__persistent 进行限定。这种对象链接到存储器的不同区域，运行时启动代码不会更改它。

相关信息

4.3.10.8. Persistent 属性

4.9.3 上电程序

一些硬件配置需要进行特殊的初始化，通常是在复位之后的前几个指令周期内进行。为此，您可以在运行时启动代码期间执行自己的上电程序。

如果将所需代码写入运行时启动代码使用的.initn 段之一，编译器会负责将代码链接到相应位置，而无需调整链接描述文件。4.9.2. 运行时启动代码列出了这些段。

例如，下面是放入.init1 段中的小型汇编序列，将在复位后不久、调用 main() 之前执行。

```
#include <avr/io.h>

 .section .init1,"ax",@progbits
   ldi r16,_BV(SRE) | _BV(SRW)
   out _SFR_IO_ADDR(MCUCR),r16
```

将该程序放入汇编源文件中进行汇编，然后将输出与程序中的其他文件进行链接。

请记住，这些段中的代码在所有运行时启动代码执行前执行，因此没有可用的堆栈，并且__zero_reg__（r1）可能尚未初始化。最好使__stack 保持其默认值（位于内部 SRAM 的末尾，因为这种做法更加快速，一些器件（如 ATmega161）上需要通过它来解决已知勘误），并使用-Wl,-Tdata,0x801100 选项来启动堆栈以上的数据段。

4.10 库

MPLAB XC8 C 编译器提供可辅助代码开发的函数库、宏、类型和对象。

该编译器提供 C 语言标准库以及一些特定于器件的库资源。

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。

4.10.1 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型打印的基本功能。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*。

4.10.1.1 AVR 器件的智能 IO

使用 MPLAB XC8 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）。在此类情况下，-msmart-io-format="*fmt*”选项可用于指定非字面格式字符串中已使用的转换规范，以链接更优库。必须确保程序仅使用列出的转换规范，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 所指向的格式字符串只共同使用%d、%i 和%s 转换说明符，则应发出-msmart-io-format=fmt="%d%i%s"选项。

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。

4.10.2 标准库

Microchip 统一标准库包括由 C99 语言规范提供的标准 C 语言头文件所定义的函数，以及任意类型和简化其使用所需的预处理器宏。所有 Microchip C 编译器均提供该库。

文档 *Microchip Unified Standard Library Reference Guide* 介绍了库函数的行为和接口以及库类型和宏的预期使用。

4.10.3 用户定义的库

可以创建用户定义的库，并将其与程序进行链接。库文件比很多源文件更易于管理，因而可以加快编译速度。但是，库必须与特定项目的目标器件和选项兼容。用户可能需要创建和维护几个版本的库，以使它能够用于不同的项目。

可以使用库管理器 avr-ar（见 5.1. 归档器/库管理器）创建库。

编译后，即可在命令行及源文件中使用用户定义的库。可以向 IDE 项目中添加更多库，或者通过选项指定。

搜索未解析的符号时，会先扫描命令行指定的库文件（在扫描完所有项目模块之后），然后再扫描 C 标准库，因此库文件中的内容可以重新定义 C 标准库中所定义的任何内容。

4.10.4 使用库程序

已经引用的库函数和对象将自动链接到您的程序，前提是库文件包含在项目中。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。

程序需要声明其使用的任何库函数或符号。标准库随标准 C 头文件（.h 文件）提供，可包含到源文件中。有关各个库函数对应的头文件，请参见您爱读的 C 语言教材或 7.1. 库示例代码。如果您自行创建库文件，通常要编写库头文件。

头文件不是库文件。库文件包含预编译的代码，通常为函数和变量定义；头文件提供库中的函数、变量和类型的声明（而不是定义）。头文件也可以定义预处理器宏。

4.11 混合使用 C 代码和汇编代码

通过使用两种不同的方法，可以将汇编语言与 C 代码混合使用：

- 汇编代码放入单独的汇编源模块中。
- 汇编代码嵌入到 C 代码中。

注：项目中包含的汇编代码越多，维护它的难度越高，所花时间越多。如果编译器进行了更新，由于经过更新的编译器的工作方式可能存在差异，汇编代码可能需要进行修改。这些因素不会影响以 C 编写的代码。如果必须添加汇编代码，则最好将它编写为处于独立汇编模块中的自包含程序，而不是将它嵌入到 C 代码中。

4.11.1 集成汇编语言模块

整个函数可以用汇编语言进行编码，作为独立的.s（或.S）源文件包含到项目中。它们将被汇编并由链接器组合到输出镜像中。

以下是编写可通过 C 代码调用的汇编程序时必须遵循的准则。

- 将<xc.h>头文件包含在代码中。如果是使用#include 来包含，应确保源文件使用的扩展名为.S，从而确保文件进行了预处理。
- 为可执行汇编代码选择或定义适合的段（关于介绍指南，请参见 4.14.1. 编译器生成的段）。
- 为程序选择名称（标号）
- 确保程序的标号可从其他模块进行访问
- 使用_SFR_IO_ADDR 等宏来获取与可访问 IO 存储空间的指令结合使用的正确 SFR 地址。
- 为程序选择相应的 C 等效原型，基于它来确定参数传递的模型。
- 如果需要向程序传递值或从程序返回值，请使用适当的寄存器来传递参数。

以下示例给出了用于获取 int 参数的 atmega103 器件汇编程序，将其添加到 PORTD 的内容中，然后作为 int 返回。

```
#include <xc.h>
.section .text
.global plus ; allow routine to be externally used
plus:
  ; int parameter in r24/5
  in r18, _SFR_IO_ADDR(PORTD) ; read PORTD
  add r24, r18 ; add to parameter
  adc r25, r1 ; add zero to MSB
  ; parameter registers are also the return location, so ready to return
  ret
.end
```

代码已放入.text 段中，因此将自动放在存储器内为代码留出的区域，无需调整默认链接器选项。

已使用_SFR_IO_ADDR 宏来确保针对用于读取 IO 存储空间的指令指定了正确的地址。

由于使用了 C 预处理器#include 伪指令和预处理器宏，因此必须预处理汇编文件来确保该文件在编译时使用扩展名.S。

要从 C 代码中调用某个汇编程序，必须提供该程序的声明。以下给出了先声明汇编程序操作，然后调用程序的 C 代码片段。

```
// declare the assembly routine so it can be correctly called
extern int plus(int);
void main(void) {
  volatile unsigned int result;
  result = plus(0x55); // call the assembly routine
}
```

4.11.2 嵌入汇编

汇编指令可以通过 asm();语句直接嵌入到 C 代码中。嵌入汇编有两种形式：简单和扩展。

在简单形式中，汇编指令使用以下语法编写：

```
asm("instruction");
```

其中，instruction 是有效汇编语言构造，例如：

```
asm("sei");
```

可以用一个字符串编写多条指令，但应将每条指令放入新行并使用换行符和制表符来确保这些指令在汇编列表文件中的格式正确。

```
asm ("nop\n\t"
 "nop\n\t"
 "nop\n\t"
 "nop\n\t");
```

在使用 asm()的扩展汇编指令中，指令的操作数使用 C 表达式来表示。后续章节中介绍的扩展语法具有如下通用格式：

```
asm("template" [ : [ "constraint"(output-operand) [ , ... ] ]
[ : [ "constraint"(input-operand) [ , ... ] ]
[ "clobber" [ , ... ] ]
 ] ]);
```

例如，

```
asm("in %0, %1" : "=r" (value) : "I" (_SFR_IO_ADDR(PORTD)) );
```

template（模板）指定输入和输出操作数的指令助记符和可选占位符（表示为百分号后跟一个数字），下节中将对其进行介绍。编译器会替换这些助记符和可选占位符以及模板中引用输入、输出和转移标号的其他标记，然后将得到的字符串输出到汇编器。

4.11.2.1 输入和输出操作数

模板后面是由零或多个输出操作数组成的逗号分隔列表，用于指示经过汇编代码和输入操作数修改的 C 对象的名称，使得来自 C 变量和表达式的值可用于汇编代码。

每个操作数有多个组成部分，具体如下所述：

```
[ [asmSymbolicName] ] constraint (Cexpression)
```

其中，*asmSymbolicName* 是操作数的可选符号名称，*constraint* 是指定操作数放置限制的字符串，*Cexpression* 是操作数使用的 C 变量和表达式，这一部分用括号括起来。

第一个（最左侧）输出操作数的编号为 0，后面所有输出操作数的编号均为前一个操作数的编号加 1，输入操作数也采用这种编号方式。

下表列出了支持的限制字母（有关操作数修饰符的信息，请参见本节后面部分的表）。

表 4-10. 输入和输出操作数限制

字母	限制	范围
a	简单高编号寄存器	r16 至 r23
b	基本指针寄存器对	r28 至 r32（Y 和 Z）
d	高编号寄存器	r16 至 r31
e	指针寄存器对	r26 至 r31（X、Y 和 Z）
l	低编号寄存器	r0 至 r15

.....（续）

字母	限制	范围
q	堆栈指针寄存器	SPH:SPL
r	任何寄存器	r0 至 r31
t	临时寄存器	r0
w	adiw 指令中可使用的特殊高编号寄存器对	r24、r26、r28 和 r30
x	指针寄存器对 X	r27:r26（X）
y	指针寄存器对 Y	r29:r28（Y）
z	指针寄存器对 Z	r31:r30（Z）
G	浮点型常量	0.0
I	6 位正整型常量	0 至 63
J	6 位负整型常量	-63 至 0
K	整型常量	2
L	整型常量	0
M	8 位整型常量	0 至 255
N	整型常量	-1
O	整型常量	8、16 和 24
P	整型常量	1
Q	基于 Y 或 Z 指针的存储器地址（带位移）	
Cm2	整型常量	-2
C0*n*	整型常量，其中 *n* 的范围为 0 至 7	n
Ca*n*	能够在没有破坏寄存器的情况下进行逻辑“与”运算的 n 字节整型常量，其中 *n* 的范围为 2 至 4	
Co*n*	能够在没有破坏寄存器的情况下进行逻辑“或”运算的 n 字节整型常量，其中 *n* 的范围为 2 至 4	
Cx*n*	能够在没有破坏寄存器的情况下进行逻辑“异或”运算的 n 字节整型常量，其中 *n* 的范围为 2 至 4	
Csp	整型常量	-6 至 6
Cxf	至少有一个 0xF 半字节的 4 字节整型常量	
C0f	没有 0xF 半字节的 4 字节整型常量	
Ynn	编译时已知的定点常量	
Y0*n*	定点或整型常量，其中 *n* 的范围为 0 至 2	n
Ym*n*	定点或整型常量，其中 *n* 的范围为 1 至 2	-n
YIJ	定点或整型常量	-0x3F 至 0x3F

所选限制应与适合 AVR 指令操作数的寄存器或常量匹配。编译器将对照 C 表达式检查限制；但如果使用了错误的限制，运行时代码可能会失败。例如，如果使用 ori 指令指定限制 r，则编译器可自由地为该操作数选择任何寄存器（r0 至 r31）。如果编译器选择 r2 至 r15 范围内的寄存器，则将失败。在这种情况下，正确的限制为 d。另一方面，如果使用限制 M，编译器将确保您仅使用 8 位立即数操作数。

下表列出了所有需要操作数的 AVR 汇编器助记符，以及其中每个操作数的相关限制。

表 4-11. 指令和操作数限制

助记符	限制	助记符	限制
adc	r,r	add	r,r
adiw	w,I	and	r,r
andi	d,M	asr	r
bclr	I	bld	r,I
brbc	I,label	brbs	I,label
bset	r,I	bst	r,I
cbi	I,I	cbr	d,I
com	r	cp	r,r
cpc	r,r	cpi	d,M
cpse	r,r	dec	r
elpm	t,z	eor	r,r
in	r,I	inc	r
ld	r,e	ldd	r,b
ldi	d,M	lds	r,label
lpm	t,z	lsl	r
lsr	r	mov	r,r
movw	r,r	mul	r,r
neg	r	or	r,r
ori	d,M	out	I,r
pop	r	push	r
rol	r	ror	r
sbc	r,r	sbci	d,M
sbi	I,I	sbic	I,I
sbiw	w,I	sbr	d,M
sbrc	r,I	sbrs	r,I
ser	d	st	e,r
std	b,r	sts	label,r
sub	r,r	subi	d,M
swap	r		

限制字符前面可能加上一个限制修饰符。不带修饰符的限制用于指定只读操作数。下表列出了限制修饰符。

表 4-12. 输入和输出限制修饰符

字母	限制
=	只写操作数，通常用于所有输出操作数。

.....（续）	
字母	**限制**
+	读-写操作数
&	寄存器应仅用于输出

因此在以下示例中：

```
asm("in %0, %1" : "=r" (value) : "I" (_SFR_IO_ADDR(PORTD)) );
```

汇编指令通过模板"in %0, %1"定义。%0 标记引用第一个输出操作数"=r" (value)，%1 引用第一个输入操作数"I" (_SFR_IO_ADDR(PORTD))。本例中未指示被破坏的寄存器。

编译器可能按如下形式编码以上嵌入汇编：

```
lds r24,value
/* #APP */
in r24, 12
/* #NOAPP */
sts value,r24
```

编译器添加了注释，以告知汇编器包含的指令是手写的。在本例中，编译器选择了寄存器 r24 来存储从 PORTD 读取的值；但是，基于编译器的优化策略，该寄存器可能不会显式装载或存储值，也根本不会包含汇编代码。例如，如果从未在 C 程序的其余部分使用变量值，则除非关闭优化器，否则编译器可能删除嵌入汇编代码。为了避免这种情况，可以将 volatile 属性添加到 asm()语句中，如下所示：

```
asm volatile("in %0, %1" : "=r" (value) : "I" (_SFR_IO_ADDR(PORTD)));
```

如有需要，可为操作数命名。名称在操作数列表中的限制前面用方括号括起，并且对已命名操作数的引用将在%号后面使用括上方括号的名称而非一个数字。因此，以上示例也可编写为：

```
asm("in %[retval], %[port]" :
 [retval] "=r" (value) :
 [port] "I" (_SFR_IO_ADDR(PORTD)) );
```

破坏列表（clobber list）主要用于告知编译器由汇编代码执行的修改。该语句段可以忽略，但所有其他段均为必需项。使用冒号进行分隔，但如果未使用输入或输出，则将操作数字段留空，例如：

```
asm volatile(“cli”::);
```

输出操作数必须是只写的，并且 C 表达式结果必须为左值（即，在赋值左侧有效）。请注意，编译器不会检查操作数的类型对于汇编指令中使用的操作类型是否合理。输入操作数是只读的。

在输入和输出需要相同操作数时，读-写操作数不受支持，但可以通过限制字符串中的一个数字指示哪个操作数的寄存器用作输入寄存器。此处给出了一个示例：

```
asm volatile("swap %0" : "=r" (value) : "0" (value));
```

该语句将交换名为 value 的 8 位变量的半字节。 约束"0"指示编译器使用第一个操作数所用的输入寄存器。但请注意，这并非自动隐含相反的情况。

编译器可以为输入和输出选择同一寄存器，即使未收到此指示也是如此。如果使用输入操作数之前汇编代码修改了输出操作数，这会成为一个问题。如果代码取决于输入和输出操作数使用的不同寄存器，则必须为输出操作数使用限制修饰符&，如以下示例所示。

```
asm volatile("in %0,%1" "\n\t"
 "out %1,%2" "\n\t"
 : "=&r" (result)
 : "I" (_SFR_IO_ADDR(port)), "r" (source)
 );
```

在本例中，将从一个端口读取一个值，然后再向该端口写入一个值。如果编译器为输入和输出选择同一寄存器，则输出值将被第一条汇编指令破坏；但使用&限制修饰符可以防止编译器在为输出值选择一个寄存器后再将该寄存器用于任何输入操作数。

此处提供了另一个交换 16 位值的高低字节的示例：

```
asm volatile("mov __tmp_reg__, %A0" "\n\t"
 "mov %A0, %B0" "\n\t"
 "mov %B0, __tmp_reg__" "\n\t"
 : "=r" (value)
 : "0" (value)
 );
```

请注意寄存器__tmp_reg__的使用，使用该寄存器时不必保存其内容。标记中用来代表指令操作数的字母 A 和 B 引用多字节寄存器中的字节部分，A 引用最低有效字节，B 引用接下来的最高有效字节，依此类推。

以下示例用于交换 32 位值的字节，其中使用 4 字节量的 C 和 D 部分，而非列出同一操作数作为输入和输出操作数（通过将"0"作为输入操作数限制），也可通过使用"+r"作为输出限制将操作数声明为读-写操作数。

```
asm volatile("mov __tmp_reg__, %A0" "\n\t"
 "mov %A0, %D0" "\n\t"
 "mov %D0, __tmp_reg__" "\n\t"
 "mov __tmp_reg__, %B0" "\n\t"
 "mov %B0, %C0" "\n\t"
 "mov %C0, __tmp_reg__" "\n\t"
 : "+r" (value)
 );
```

如果操作数不能装入单个寄存器中，编译器将自动分配足够的寄存器来存放整个操作数。这也意味着，通常需要将输入操作数的类型强制转换为所需大小。

如果输入操作数限制指示指针寄存器对（如"e" (ptr)），并且编译器选择寄存器 Z（r30:r31），则必须使用%a0（小写 a）来引用 Z 寄存器（当在如下现场中使用时）：

```
ld r24,Z
```

4.11.2.2 破坏操作数

被破坏寄存器的列表是可选的；但如果指令修改未被指定为操作数的寄存器，则需要向编译器告知所做的修改。

通常可以安排汇编，这样便无需指定被破坏的寄存器。指示某个寄存器已被破坏将强制编译器在汇编指令前存储其值并在汇编指令后重载这些值，而且将限制编译器优化代码的能力。

以下示例将执行原子递增。它会先禁止中断，然后使指针变量所指向的 8 位值递增。请注意，由于需要在允许中断前存储递增的值，因此将使用指针。

```
asm volatile(
 "cli" "\n\t"
 "ld r24, %a0" "\n\t"
 "inc r24" "\n\t"
 "st %a0, r24" "\n\t"
 "sei" "\n\t"
 :
 : "e" (ptr)
 : "r24"
);
```

针对以上情况，编译器可能产生以下代码：

```
cli
ld r24, Z
inc r24
st Z, r24
sei
```

为了避免该序列破坏寄存器 r24，请使用编译器定义的特殊临时寄存器__tmp_reg__。

```
asm volatile(
 "cli" "\n\t"
 "ld __tmp_reg__, %a0" "\n\t"
 "inc __tmp_reg__" "\n\t"
 "st %a0, __tmp_reg__" "\n\t"
 "sei" "\n\t"
 :
 : "e" (ptr)
);
```

编译器始终将在需要时重载临时寄存器。

以上代码会无条件地重新允许中断（可能并非所需行为）。为了使代码更通用，可以将当前状态存储到由编译器选择的寄存器中。

```
{
  uint8_t s;
  asm volatile(
  "in %0, __SREG__" "\n\t"
  "cli" "\n\t"
  "ld __tmp_reg__, %a1" "\n\t"
  "inc __tmp_reg__" "\n\t"
  "st %a1, __tmp_reg__" "\n\t"
  "out __SREG__, %0" "\n\t"
  : "=&r" (s)
  : "e" (ptr)
  );
}
```

此处的汇编代码将修改 ptr 所指向的变量，因此 ptr 的定义应使用 volatile 说明符指示其目标可能发生意外更改，例如：

```
volatile uint8_t *ptr;
```

特殊破坏存储器将告知编译器，汇编代码可能修改任何存储单元。它会强制编译器先更新内容当前保存在寄存器中的所有变量，然后再执行汇编代码。

将存储器破坏功能与汇编指令结合使用时，可以确保先前对 volatile 对象的所有访问都将在该指令执行前完成，并且在该指令后执行的 volatile 访问尚未开始。但是，它不会阻止编译器跨越存储器破坏指令所创建的屏障移动非 volatile 相关指令，因为此类指令可能是允许或禁止中断的指令。

4.11.3 汇编代码和 C 代码之间的交互

MPLAB XC8 C 编译器 包含了几项功能，这些功能旨在使 C 代码可以符合用户定义的汇编代码的要求。此外也必须采取一些预防措施，以确保汇编代码不会干扰基于 C 代码生成的汇编代码。

4.11.3.1 等效汇编符号

默认情况下，AVR-GCC 在 C 代码和汇编代码中使用相同的函数或对象符号名称。汇编代码中的 C 语言符号前没有前导下划线字符。

可以使用特殊形式的 asm() 语句为汇编代码指定不同的名称：

```
unsigned long value asm(“clock”) = 3686400;
```

该语句指示编译器使用符号名称 clock 而非 value。这只对具有静态存储持续时间的对象有意义，因为基于堆栈的对象在汇编代码中没有符号名称，这些对象可以缓存到寄存器中。

通过编译器，可以指定使用特定寄存器：

```
void Count(void)
{
 register unsigned char counter asm("r3");
 // ... some code...
 asm volatile("clr r3");
```

```
 // ... more code...
}
```

汇编指令 clr r3 将清零变量计数器。编译器不会完全保留指定寄存器，可能会被重用。编译器无法检查指定寄存器的使用是否与任何其他预定义寄存器发生冲突。建议不要通过该方式保留过多寄存器。

为了更改函数的汇编名称，需要使用原型声明，因为编译器不会接受函数定义中的 asm() 关键字。例如：

```
extern long calc(void) asm ("CALCULATE");
```

在 C 代码中调用函数 calc() 将生成用于调用 CALCULATE 函数的汇编指令。

4.11.3.2 从汇编代码中访问寄存器

在汇编代码中，SFR 定义不会自动可访问。可以通过包含头文件<xc.h>来访问这些寄存器的定义。

该头文件中寄存器的符号与 C 域中使用的符号相同；但是，应使用适当的 I/O 宏来确保将正确地址编码到用于访问 I/O 存储空间的指令中，例如，将以下内容写入 TCNT0 寄存器：

```
out _SFR_IO_ADDR(TCNT0), r19
```

寄存器中的位具有与其相关联的宏，可以直接与需要位编号（0 至 7）的指令结合使用，也可在基于该位在 SFR 中的位置而需要位掩码时与 _BV() 宏结合使用，例如：

```
sbic _SFR_IO_ADDR(PORTD), PD4
ldi r16, _BV(TOIE0)
```

4.12 优化

MPLAB XC8 编译器可执行各种优化。优化可使用-O 选项（如 3.6.6. 控制优化的选项所述）来控制。在免费模式下，其中一些优化会被禁止。即使使能了它们，也只有在满足非常特定的条件时才可能应用这些优化。因此，您可能会看到一些代码行进行了优化，而其他相似行则没有。调试代码时，可能希望降低优化级别来确保程序流符合预期。

4.13 预处理

在编译之前，需要对所有 C 源文件进行预处理。预处理文件在编译后会被删除，但可以通过使用-E 选项（见 3.6.2.2. E：仅预处理）检查该文件。

如果汇编源文件使用 .S 扩展名，则会对其进行预处理。

4.13.1 预处理器伪指令

除了标准伪指令之外，XC8 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 4-13. 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 的缩写。	请参见#ifdef
#else	根据条件包含源代码行。	请参见#if
#endif	终止条件包含源代码。	请参见#if

.....（续）

伪指令	含义	示例
#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

带参数的宏扩展可以使用#字符将参数转换为一个字符串，以及使用##序列来连接参数。如果要连接两个表达式，可以考虑使用两个宏，以处理其中任一表达式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扩展的表达式。请记住，对某个宏标识符进行扩展之后，如果它在连接之后出现，将不会再次对它进行扩展。

4.13.1.1 预处理器运算

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。当宏进行了文本扩展且预处理完成后，扩展即形成了一个 C 表达式，代码生成器以及其他 C 代码将对其求值。扩展 C 表达式中的标记继承了一种类型，其值会以通常方式进行整型提升和类型转换。

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（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 XC8 C 编译器，此长度为 64 位。

4.13.2 预定义的宏

编译器驱动程序会为预处理器定义一些特定的符号，从而允许基于芯片类型等条件进行条件编译。下表列出了驱动程序定义的较常用符号。除非另有说明，否则每个符号（如定义）都等于 1。

表 4-14. 预定义的宏

符号	说明
__AVR_Device__	在-mcpu 选项指定器件而非架构时设置。指示器件（例如针对 atmega8 进行编译时）将设置宏__AVR_ATmega8__。
__AVR_DEVICE_NAME__	在-mcpu 选项指定器件而非架构时设置。指示器件（例如针对 atmega8 进行编译时）将该宏定义为 atmega8。
__AVR_ARCH__	指示器件架构。可能值如下：avr2、avr25、avr3、avr31、avr35、avr4、avr5、avr51 和 avr6 架构分别为 2、25、3、31、35、4、5、51 和 6；avrtiny、avrxmega2、avrxmega3、avrxmega4、avrxmega5、avrxmega6 和 avrxmega7 架构分别为 100、102、103、104、105、106 和 107。
__AVR_ASM_ONLY__	指示所选器件只能用汇编语言编程。
__AVR_CONST_DATA_IN_CONFIG_MAPPED_PROGMEM__	指示 const 限定对象将放入将映射到数据存储器中的 32 KB 程序段中。
__AVR_CONST_DATA_IN_PROGMEM__	指示 const 限定对象将放入程序存储器中。
__AVR_ERRATA_SKIP__ __AVR_ERRATA_SKIP_JMP_CALL__	指示所选器件（AT90S8515 和 ATmega103）不得因为硬件勘误而跳过（SBRS、SBRC、SBIS、SBIC 和 CPSE 指令）32 位指令。只有另外设置了__AVR_HAVE_JMP_CALL__时，才定义第二个宏。
__AVR_HAVE_EIJMP_EICALL__	指示所选器件具有大于 128 KB 的程序存储器、3 字节宽的程序计数器以及 eijmp 和 eicall 指令。
__AVR_HAVE_ELPM__	指示所选器件具有 elpm 指令。
__AVR_HAVE_ELPMX__	指示器件具有 elpm Rn,Z 和 elpm Rn,Z+指令。
__AVR_HAVE_JMP_CALL__	指示所选器件具有 jmp 和 call 指令以及大于 8 KB 的程序存储器。
__AVR_HAVE_LPMX__	指示所选器件具有 lpm Rn,Z 和 lpm Rn,Z+指令。
__AVR_HAVE_MOVW__	指示所选器件具有用于执行 16 位寄存器-寄存器传送操作的 movw 指令。
__AVR_HAVE_MUL__ __AVR_HAVE_MUL__	指示所选器件具有硬件乘法器。
__AVR_HAVE_RAMPD__ __AVR_HAVE_RAMPX__ __AVR_HAVE_RAMPY__ __AVR_HAVE_RAMPZ__	分别指示器件具有 RAMPD、RAMPX、RAMPY 或 RAMPZ 特殊功能寄存器。
__AVR_HAVE_SPH__ __AVR_SP8__	分别指示器件具有 16 位或 8 位堆栈指针。这些宏的定义受所选器件影响，适用于 avr2 和 avr25 架构。
__AVR_HAVE_8BIT_SP__ __AVR_HAVE_16BIT_SP__	分别指示编译器使用 8 位还是 16 位堆栈指针。-mtiny-stack 选项将影响定义哪些宏
__AVR_ISA_RMW__	指示所选器件具有读-修改-写指令（xch、lac、las 和 lat）。
__AVR_MEGA__	指示所选器件具有 jmp 和 call 指令。
__AVR_PM_BASE_ADDRESS__=*addr*	指示地址空间为线性且程序存储器映射到数据存储器。赋给该宏的值是所映射存储器的起始地址。

.....（续）

符号	说明
__AVR_SFR_OFFSET__=*offset*	指示可直接访问 SFR 的指令（如 in、out 和 sbi）要从数据存储器地址中减去的偏移量。
__AVR_SHORT_CALLS__	指示使用了-mshort-calls 选项，该选项会影响所用调用指令且可自动设置。
__AVR_TINY__	指示所选器件或架构属于 TINY 系列。
__AVR_TINY_PM_BASE_ADDRESS__=*addr*	已弃用；请使用__AVR_PM_BASE_ADDRESS__。指示 TINY 器件地址空间为线性且程序存储器映射到数据存储器。
__AVR_XMEGA__	指示所选器件或架构属于 XMEGA 系列。
__AVR_2_BYTE_PC__	指示所选器件具有最高 128 KB 的程序存储器且程序计数器的宽度为 2 字节。
__AVR_3_BYTE_PC__	指示所选器件具有至少 128 KB 的程序存储器且程序计数器的宽度为 3 字节。
__BUILTIN_AVR_*name*	指示所选器件可使用指定的内置功能。
__CODECOV	使能代码覆盖时，值__CC_RAM (1)。
__DEBUG	执行调试编译并且使用 MPLAB X IDE 时。
__FLASH*n*	基于所选器件上的闪存段数定义__FLASH、__FLASH1 和__FLASH2 等。
__LINE__	指示源代码行号。
__MEMX	指示所选器件可使用__memx 说明符。
__NO_INTERRUPTS__	指示使用了-mno-interrupts 选项，该选项会影响堆栈指针的更改方式。
__DATE__	指示当前日期，例如 May 21 2004。
__FILE__	指示正在预处理的源文件。
__TIME__	指示当前时间，例如 08:06:31。
__XC	指示正在使用 Microchip 的 MPLAB XC 编译器。
__XC8	指示正在使用用于 Microchip 8 位器件的 MPLAB XC 编译器。
__XC8_VERSION	指示编译器的版本号乘以 1000，例如，v1.00 将表示为 1000。

4.13.3 pragma 伪指令

MPLAB XC8 只接受一条 AVR 特定的 pragma 伪指令，即 config pragma 伪指令，4.2.4. 配置位访问对其进行了介绍。

4.14 链接程序

除非请求编译器在编译序列中提前停止，否则编译器将会自动调用链接器。

链接器将使用从命令行驱动程序获取的选项运行，并使用链接器描述文件来指定存储区以及放置段的位置。

用户可以调整传递给链接器的链接器选项，但只有在一些特殊情况下才需要这么做（有关更多信息，请参见 3.6.10.7. WI：将 Option 传递给链接器选项）。

链接器会创建一个映射文件，其中将详细说明分配给各个段和代码中一些对象的存储器。映射文件是查找存储器信息的最佳途径。

4.14.1 编译器生成的段

代码生成器会将代码和数据放置到具有标准名称的段中，它们随后通过默认的链接描述文件进行定位。在汇编代码中，可以通过使用.section 汇编器伪指令来创建段。如果不确定哪个段存放项目中的某个对象或代码，可以生成并检查相关汇编列表文件。

4.14.1.1 程序空间段

下面介绍了程序存储器中的常见段的内容。

.text 这些段包含所有不需要特殊链接位置的可执行代码。

.initn 这些段用于定义从复位开始一直执行到调用 main()的运行时启动代码。这些段中的代码按照从 init0 到 init9 的顺序执行。

.finin 这些段用于定义通过返回或调用 exit()在 main()终止后执行的退出代码。.finin 段中的代码按照从.fini9 到.fini0 的降序执行。

4.14.1.2 数据空间段

下面介绍了数据存储器中的常见段的内容。

.bss 该段包含具有静态存储持续时间且尚未初始化的所有对象。

.data 该段包含具有静态存储持续时间且已使用相应值完成初始化的所有对象的 RAM 映像。

.rodata 这些段存放只读数据。

4.14.2 更改和链接所分配的段

函数和对象所在的默认段的位置可以通过驱动程序选项更改。4.14.1. 编译器生成的段列出了编译器用于存放对象和代码的默认段。

__section()说明符允许将对象或函数重定向到用户定义的段中。这样便可重定位各个对象或函数。

使用__section()说明符的对象将由运行时启动代码按常规方式清零或初始化。

以下给出了将对象和函数分配到非默认段的示例。要使用__section()说明符，必须使能 CCI（见 3.6.3.3. Ext 选项）。

```
int __section("myBss") foobar;
int __section("myText") helper(int mode) { /* ... */ }
```

编译（链接）程序时，可以使用-Wl,--section-start=*section*=*addr* 选项链接这些段，前提是链接描述文件已定义了一个同名输出段。如果未定义，则可能需要复制默认的链接描述文件，对其进行适当修改并使用-Wl,-T*script* 驱动程序选项进行指定。修改操作必须定义一个与指定段同名的输出段。以下示例会将所有名为 myBss 的输入段分配给同名输出段。

```
myBss :
  {
    KEEP(*(myBss))
  }
```

请注意，数据空间中的所有地址都需要使用 0x800000 的偏移量。 例如，假设想要将上面创建的新 myBss 段置于 SRAM 地址 0x300 处，应使用类似下面的选项：

```
-Wl,--section-start=myBss=0x800300
```

-Wl,-T*section*=*addr* 选项可用于定位标准段（如.text、.data 和.bss 段）；但由于 MPLAB XC8 使用最佳适应分配器（Best Fit Allocator，BFA），因此编译器通常不会使用这些段来存储对象或代码。

尽管-Wl,-Ttext 选项将按照指定的偏移量移动.text 段，但该选项还是会为链接器设置一个起始地址，链接器将从该起始地址开始为所有代码（甚至是由 BFA 处理的代码）分配程序存储器。因此，使用该选项时，.text 段将以指定的确切偏移量开始；其他代码段将位于其后。类似地，-Wl,-Tdata 选项也会为链接器设置一个起始地址，链接器将从该起始地址开始分配数据（无论是否已完成初始化）。

4.14.3 链接描述文件

链接描述文件用于指示链接器如何定位存储器中的段。下表列出了这些链接描述文件的五种不同形式。这些形式基于传递到链接器的选项来选择。

表 4-15. 链接描述文件形式

描述文件扩展名	控制链接器选项	链接器操作
.x	默认	
.xr	-r	不执行重定位
.xu	-Ur	解析对构造函数的引用
.xn	-n	将文本设置为只读
.xbn	-N	将文本段和数据段设置为可读写。

4.14.3.1 链接描述文件符号

编译器的链接描述文件使用多个符号来定位段所在的存储区的起始地址和最大结束地址。当可以使用-Wl,--defsym,*symbol=address* 选项来重定义这些符号时，通常会有一些编译器选项更容易实现预期目标。

例如，__TEXT_REGION_LENGTH__和__DATA_REGION_LENGTH__符号分别定义用于存放程序代码和基于 RAM 的对象的存储区的最大长度。相较于直接调整，其实可以考虑使用-mreserve 选项（3.6.1.11. Reserve 选项），使用该选项在链接时无需考虑存储器范围。例如，要将具有 8 KB 程序存储器的器件上的可用程序存储空间减小为 6 KB 的低地址部分，可使用选项-mreserve=rom@0x1800:0x2000。

__TEXT_REGION_ORIGIN__和__DATA_REGION_ORIGIN__符号分别定义用于存放程序代码和基于 RAM 的对象的存储区的起始地址。相较于直接调整，其实可以考虑分别使用-Wl,-Ttext 和-Wl,-Tdata 选项（4.14.2. 更改和链接所分配的段）。例如，要使所有代码从地址 0x100 处开始，可使用选项-Wl,-Ttext=0x100。

4.14.4 替换库模块

对于弱库函数（见 4.7.2.6. Weak 属性），可以使用自定义程序替换同名的库程序，而不必使用库管理器 avr-ar。只需将该程序的定义包含到项目中即可。

5. 实用程序

本章介绍一些与编译器一起捆绑的实用应用程序。

本章讨论的是较为常用的应用程序，但是通常不需要直接执行它们。这些应用程序的某些功能由基于命令行参数或 MPLAB X IDE 项目属性选择的命令行驱动程序间接调用。

5.1 归档器/库管理器

归档器/库管理器程序具有将几个中间文件组合为单个文件（称为库归档文件）的功能。库归档文件相比于其中包含的各个文件更易于管理，并且占用的磁盘空间可能会更小。

归档器可以构建编译器所需的所有库归档文件类型，并且可以检测现有归档文件的格式。

5.1.1 使用归档器/库管理器

归档器程序名为 avr-ar，用于创建和编辑库归档文件。它具有以下基本命令格式：

```
avr-ar [options] file.a [file.o ...]
```

其中，*file*.a 表示正在创建或编辑的库归档文件。

归档文件之后的文件（如果需要）是指定命令所需的目标（.o）模块。

options 是零个或多个用于控制程序的选项，如下表所示。

表 5-1. 归档器命令行选项

选项	作用
-d *modules*	删除模块
-m *modules*	重新排序模块
-p	列出模块
-r *modules*	替换模块
-t	列出模块与符号
-x *modules*	提取模块
--target *device*	指定目标器件

在替换或提取模块时，必须指定要替换或提取的模块的名称。如果未提供任何名称，将替换或提取归档中的所有模块。

通过请求归档器替换归档中的模块来创建归档文件或将文件添加到现有归档。由于模块并不存在，所以会将该模块追加到归档的末尾。

归档器将按模块在命令行上的顺序使用模块来创建库归档。在更新归档时，将会保留模块的顺序。添加到归档的任何模块都将追加到末尾。

模块在归档中的顺序对于链接器是有意义的。如果归档中包含的某个模块引用同一个归档中另一个模块中定义的符号，定义该符号的模块应位于引用符号的模块之后。

使用-d 选项时，将从归档中删除指定的模块。在这种情况下，不提供任何模块名称会产生错误。

-p 选项将列出归档文件中的模块。

-m 选项可接受模块名称的列表，并会对归档文件中的匹配模块重新排序，使它们的顺序与命令行上列出的模块顺序相同。未列出的模块将保留其现有顺序，并将出现在重新排序的模块之后。

avr-ar 归档器不适用于仅使用 LTO 数据编译（即，使用-fno-fat-lto-objects 选项编译）的目标文件。对于此类目标文件，请改用 avr-gcc-ar 归档器。

5.1.1.1 归档器示例

此处提供了一些使用归档器的示例。以下命令：

```
xc8-ar -r myAvrLib.a ctime.o init.o
```

会创建一个名为 myAvrLib.a 的库归档，其中包含模块 ctime.o 和 init.o。以下命令会从库归档 lcd.a 中删除目标模块 a.o：

```
xc8-ar -d lcd.a a.o
```

5.2 Objdump

avr-objdump 应用程序可显示有关目标文件的各种信息。

该工具的命令行的一般格式如下：

```
avr-objdump [options] objfiles
```

其中 *objfiles* 可以是任何目标文件，包括归档文件或输出文件。该工具能够确定所指定文件的格式。

--help 选项用于显示该应用程序可使用的所有命令。

仅对 AVR ELF 文件而言，-Pmem-usage 选项将以字节数和可用存储空间的百分比来显示程序和数据存储器使用情况。

该工具的常见用途是获取整个程序的完整列表文件。为此，需在编译项目时使用编译器的-g 选项，然后使用类似下面的命令调用 avr-objdump 应用程序。

```
avr-objdump -S -l a.out > avr.lst
```

这将基于默认编译器输出文件来创建 avr.lst 列表文件，列表中显示原始 C 源代码和行号信息。

6. 实现定义的行为

本节说明编译器的行为选项，其中 C 标准表明行为是实现定义的行为。

6.1 概述

ISO C 需要一个符合标准的实现来记录在标准中定义为“实现定义的”行为的选项。以下各节列出了全部此类领域内容、针对编译器所做的选择以及在 ISO/IEC 9899:1999（又称 C99）标准（或 ISO/IEC 9899:1990（又称 C90））中对应的章节编号。

6.2 转换

ISO 标准	实现
“如何标识一条诊断消息（3.10 和 5.1.1.3）。”	
	默认情况下，在命令行上进行编译时，将使用以下格式。字符串（warning）仅针对警告消息显示。 *filename*:*line*:*column*:{error/warning}: *message*
“在转换阶段 3，新行之外的各个非空空白字符序列是保留还是替换为一个空格字符（5.1.1.2）”	
	编译器将每个前导或穿插在中间的空白字符序列替换为一个空格。尾随空白字符序列替换为新行。

6.3 环境

ISO 标准	实现
“转换阶段 1 中实际源文件多字节字符与源字符集之间的映射关系（5.1.1.2）。”	
“程序在独立环境中启动时调用的函数的名称和类型（5.1.2.1）。”	
	`int main (void);`
“在独立环境中终止程序的效果（5.1.2.1）。”	
	中断被禁止，程序无限循环
“`main` 函数的备用定义方式（5.1.2.2.1）。”	
	`void main (void);`
“赋给 `main` 的 argv 参数所指向的字符串的值（5.1.2.2.1）。”	
	没有参数传递给 `main`。未定义 `argc` 或 `argv` 引用。
“交互式设备由什么构成（5.1.2.3）。”	
	由应用程序定义。
“信号集、其语义和默认处理（7.14）。”	
	信号未实现。
“除 `SIGFPE`、`SIGILL` 和 `SIGSEGV` 之外的对应于计算异常的信号值（7.14.1.1）。”	
	信号未实现。
“在程序启动时执行 `signal(sig, SIG_IGN);` 的等效代码所针对的信号（7.14.1.1）。”	

.....（续）

ISO 标准	实现
	信号未实现。
“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（7.20.4.5）。”	
	主机环境是应用程序定义的。
“system 函数执行字符串的方式（7.20.4.6）。”	
	主机环境是应用程序定义的。

6.4 标识符

ISO 标准	实现
“哪些额外的多字节字符可能会出现在标识符及其通用字符名称对应项中（6.4.2）。”	
	无。
“标识符中的有效初始字符数（5.2.4.1 和 6.4.2）。”	
	所有字符均有效。

6.5 字符

ISO 标准	实现
“字节中的位数（C90 3.4 和 C99 3.6）。”	
	8.
“执行字符集成员的值（C90 和 C99 5.2.1）。”	
	执行字符集为 ASCII。
“针对每个标准字母转义序列所生成的执行字符集成员的惟一值（C90 和 C99 5.2.2）。”	
	执行字符集为 ASCII。
“char 对象的值，在该对象中存储了除基本执行字符集成员之外的任何字符（C90 6.1.2.5 和 C99 6.2.5）。”	
	char 对象的值是源字符集中字符的 8 位二进制表示。即，未执行转换。
“哪些 signed char 或 unsigned char 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）。”	
	默认情况下，signed char 与普通 char 在功能上等效。如果指定了 CCI，则默认使用 unsigned char。可以使用-funsigned-char 和-fsigned-char 选项显式指定类型。
“源字符集（字符常量和字符串字面值）成员与执行字符集成员之间的映射（C90 6.1.3.4、C99 6.4.4.4、C90 和 C99 5.1.1.2）。”	
	为执行字符集保留了源字符集的二进制表示。
“包含多个字符或者包含一个不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）。”	
	前一个值左移 8，下一个字符的位模式被屏蔽。最终结果为 int 类型。如果结果大于 int 所能表达的值，会发出一条警告诊断，值被截断为 int 大小。

.....（续）

ISO 标准	实现
“宽字符常量（包含多个多字节字符，或者包含一个未在扩展执行字符集中表示的多字节字符或转义序列）的值（C90 6.1.3.4 和 C99 6.4.4.4）。”	
	源文件中不支持多字节字符。
“用于将宽字符常量（包含一个与扩展执行字符集的一个成员映射的多字节字符）转换成对应的宽字符代码的当前区域设置（C90 6.1.3.4 和 C99 6.4.4.4）。”	
	源文件中不支持多字节字符。
“用于将宽字符串字面值转换成对应的宽字符代码的当前区域设置（C90 6.1.4 和 C99 6.4.5）。”	
	不支持宽字符串。
“包含一个未在执行字符集中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）。”	
	源文件中不支持多字节字符。

6.6 整型

ISO 标准	实现
“存在于实现中的扩展整型类型（C99 6.2.5）。”	
	关键字 `__int24` 和 `__uint24` 分别标识有符号和无符号 24 位整型类型。
“有符号整型类型是否使用符号和量值、二进制补码或二进制反码来表示，异常值是陷阱表示还是普通值（C99 6.2.6.2）。”	
	所有整型类型均表示为二进制补码，所有位模式均为普通值。
“与另外一个精度相同的扩展整型类型相关的任意扩展整型类型的等级（C99 6.3.1.1）。”	
	没有精度相同的扩展整型类型。
“将整型值转换为有符号整型类型（当该值无法在此类型对象中表示时）的结果或引发的信号（C90 6.2.1.2 和 C99 6.3.1.3）。”	
	将值 X 转换成宽度为 N 的类型时，结果的值为 X 的二进制补码形式的最低有效 N 位。即，X 被截断为 N 位。没有引发信号。
“有符号整型的一些按位运算的结果（C90 6.3 和 C99 6.5）。”	
	右移操作符（>>操作符）符号扩展为有符号值。因此，具有 `signed int` 值 0x0124 的对象右移一位的结果为值 0x0092，值 0x8024 右移一位的结果为值 0xC012。右移无符号整型值将总是清零结果的 MSb。左移（<<操作符）有符号或无符号值将总是清零结果的 LSb。其他按位运算将操作数视为无符号进行操作。

6.7 浮点型

ISO 标准	实现
“浮点运算的精度及<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）。”	
	精度未知。
“通过 `FLT_ROUNDS` 的非标准值表征的舍入行为（C90 和 C99 5.2.4.2.2）。”	
	未使用此类值。

.....（续）

ISO 标准	实现
“通过 FLT_EVAL_METHOD 的非标准负值表征的求值方法（C90 和 C99 5.2.4.2.2）。”	
	未使用此类值。
“当整型转换为不能精确地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）。”	
	整型值舍入到最接近它的浮点型表达式。
“当浮点数转换为较短的浮点数时的舍入方向（C90 6.2.1.4 和 6.3.1.5）。”	
	当浮点数转换为短浮点值时会向下舍入。
“如何为某些浮点常量选择最接近的可表示值或与最接近的可表示值紧邻的更大或更小的可表示值（C90 6.1.3.1 和 C99 6.4.4.2）。”	
	不适用；FLT_RADIX 是 2 的幂。
“在不被 FP_CONTRACT pragma 伪指令禁止的情况下，是否及如何压缩浮点表达式（C99 6.5）。”	
	该 pragma 伪指令未实现。
“FENV_ACCESS pragma 伪指令的默认状态（C99 7.6.1）。”	
	此 pragma 伪指令未实现。
“更多的浮点异常、舍入模式、环境、分类及其宏名称（C99 7.6 和 7.12）。”	
	不支持。
“FP_CONTRACT pragma 伪指令的默认状态（C99 7.12.2）。”	
	此 pragma 伪指令未实现。
“当舍入结果实际上等于符合 IEC 60559 的实现中的数学结果时，是否会引发“不精确”的浮点异常（C99 F.9）。”	
	不会引发异常。
“在符合 IEC 60559 的实现中，当结果很小但并非不精确时，是否会引发“下溢”（和“不精确”）浮点异常（C99 F.9）。”	
	不会引发异常。

6.8 数组和指针

ISO 标准	实现
将指针强制转换为整型（或反之）的结果（C90 6.3.4 和 C99 6.3.2.3）。	
	从整型到指针（或反之）的强制转换结果使用源类型的二进制表示，并根据目标类型进行适当的重新解释。 如果源类型大于目标类型，则丢弃最高有效位。从指针强制转换为整型时，如果源类型小于目标类型，则结果进行符号扩展。从整型强制转换为指针时，如果源类型小于目标类型，则根据源类型的符号性扩展结果。
“减去指向同一数组元素的两个指针后的结果大小（C90 6.3.6 和 C99 6.5.6）。”	
	有符号整型结果将与减法中的指针操作数的大小相同。

6.9 提示

ISO 标准	实现
“通过使用 register 存储类说明符进行的的建议的有效程度（C90 6.5.1 和 C99 6.7.1）。”	
	register 存储类可用于在寄存器中定位某些对象（参见关于*寄存器使用*的章节）。
“通过使用 inline 函数说明符进行的建议的有效程度（C99 6.7.4）。”	
	如果获得许可的编译器将优化器设置为 2 级或更高级别，则可能会内联函数。在其他情况下，函数不会被内联。

6.10 结构、联合、枚举和位域

ISO 标准	实现
“普通 int 位域是视为 signed int 位域还是 unsigned int 位域（C90 6.5.2、C90 6.5.2.1、C99 6.7.2 和 C99 6.7.2.1）。”	
	普通 char 被视为无符号整型。-fsigned-bitfields 选项可用于将位域视为有符号。
“除_Bool、signed int 和 unsigned int 之外的允许的位域类型（C99 6.7.2.1）。”	
	允许所有整型类型。
“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）。”	
	位域可以跨越存储单元。
“位域在单元中的分配顺序（C90 6.5.2.1 和 C99 6.7.2.1）。”	
	结构中定义的第一个位域将分配到存储单元中的 LSb 位置。随后的位域将分配到更高位。
“结构的非位域成员对齐（C90 6.5.2.1 和 C99 6.7.2.1）。”	
	不执行对齐。
“与每个枚举类型兼容的整型类型（C90 6.5.2.2 和 C99 6.7.2.2）。”	
	可以选择 signed int 或 unsigned int 来表示枚举类型。

6.11 限定符

ISO 标准	实现
“什么构成对 volatile 限定类型的对象的访问（C90 6.5.3 和 C99 6.7.3）。”	
	每次引用 volatile 限定对象的标识符都构成对该对象的一次访问。

6.12 预处理伪指令

ISO 标准	实现
“两种形式的头文件名称中的序列如何映射到头文件或外部源文件名称（C90 6.1.7 和 C99 6.4.7）。”	
	定界符之间的字符序列被认为是一个字符串，它是主机环境的文件名。

.....（续）

ISO 标准	实现
“控制条件包含的常量表达式中字符常量的值是否与执行字符集中相同字符常量的值匹配（C90 6.8.1 和 C99 6.10.1）。”	
	是。
“控制条件包含的常量表达式中单字符字符常量的值是否可为负值（C90 6.8.1 和 C99 6.10.1）。”	
	是。
“搜索通过< >定界符包含的头文件的位置以及如何指定位置或识别头文件（C90 6.8.2 和 C99 6.10.2）。”	
	预处理器搜索使用-I 选项指定的任何目录，然后，在未使用-nostdinc 选项的前提下，标准编译器将包含目录<install directory>/avr/avr/ include。
“如何在通过" "定界符包含的头文件中搜索指定的源文件（C90 6.8.2 和 C99 6.10.2）。”	
	编译器首先在含有包含文件的目录中搜索指定文件，然后在搜索通过< >定界符包含的头文件的目录中搜索。
“将预处理标记组合成头文件名称的方法（C90 6.8.2 和 C99 6.10.2）。”	
	所有标记（包括空白）都被视为头文件名的组成部分。不对定界符内的标记执行宏扩展。
“用于#include 处理的嵌套限制（C90 6.8.2 和 C99 6.10.2）。”	
	没有限制。
“#操作符是否在字符常量或字符串字面值中的通用字符名称开头位置的\字符之前插入一个\字符（6.10.3.2）。”	
	否。
“针对每个识别的非 STDC #pragma 伪指令的行为（C90 6.8.6 和 C99 6.10.6）。”	
	请参见有关 *Pragma 伪指令*的章节。
“转换的日期和时间不可用时，__DATE__和__TIME__的定义（C90 6.8.8 和 C99 6.10.8）。”	
	转换日期和时间始终可用。

6.13 库函数

ISO 标准	实现
“独立程序可用的任何库工具，第 4 条要求的最低限度工具除外（5.1.2.1）。”	
	请参见适用于您的目标器件的编译器用户指南。
“由 assert 宏打印的诊断的格式（7.2.1.1）。”	
	Assertion failed: *expression* (*file*: *func*: *line*)
	Assertion failed: (*message*), function *function*, file *file*, line *line*.\n. 如果__func__不可用，function *function* 部分将被跳过。
“fegetexceptflag 函数所存储的浮点异常标志的表示（7.6.2.2）。”	
“除了上溢或下溢异常外，feraiseexcept 函数是否还引发不精确异常（7.6.2.3）。”	
	未实现。
“"C”和""”之外的可作为 setlocale 函数的第二个参数传递的字符串（7.11.1.1）。”	

.....（续）

ISO 标准	实现
	无。
“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）。”	
	未实现。
“数学函数的域错误，不符合此国际标准的要求（7.12.1）。”	
	无。
“在发生域错误时数学函数所返回的值（7.12.1）。”	
	在发生域错误时，errno 变量设置为 EDOM
“在发生上溢和/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）。”	
	是
“当 fmod 函数的第二个参数为零时，是发生域错误还是返回零（7.12.10.1）。”	
	返回第一个参数。
“remquo 函数在减小商值时所使用的模数的以 2 为底的对数（7.12.10.3）。”	
	未实现。
“是否在调用信号处理程序之前执行 signal(sig, SIG_DFL);的等效代码，否则将执行信号的阻塞（7.14.1.1）。”	
	信号未实现。
“宏 NULL 扩展成的空指针常量（7.17）。”	
	((void*)0)
“文本流的最后一行是否需要终止换行符（7.19.2）。”	
	流未实现。
“在读入时，是否显示向紧邻换行符之前的文本流中写出的空格字符（7.19.2）。”	
	流未实现。
“可以附加到写入二进制流的数据的空字符数（7.19.2）。”	
	流未实现。
“附加模式流的文件位置指示符最初是位于文件的开头还是结尾（7.19.3）。”	
	流未实现。
“对文本流写入是否会导致相关文件在该点后面被截断（7.19.3）。”	
	流未实现。
“文件缓冲的特性（7.19.3）。”	
	文件处理未实现。
“零长度文件是否实际存在（7.19.3）。”	
	文件处理未实现。
“有效文件名的构成规则（7.19.3）。”	
	文件处理未实现。

.....（续）

ISO 标准	实现
“同一个文件是否可以多次打开（7.19.3）。”	
	文件处理未实现。
“用于文件中多字节字符的编码的性质和选择（7.19.3）。”	
	文件处理未实现。
“删除功能对打开文件的影响（7.19.4.1）。”	
	文件处理未实现。
“在调用重命名函数之前存在具有新名称的文件的效果（7.19.4.2）。”	
	文件处理未实现。
“是否在程序异常终止时删除打开的临时文件（7.19.4.3）。”	
	文件处理未实现。
“当 tmpnam 函数调用次数超过 TMP_MAX 时会发生什么情况（7.19.4.4）。”	
	文件处理未实现。
“允许哪些模式更改（如果有）以及在什么情况下允许（7.19.5.4）。”	
	文件处理未实现。
“用于打印无穷大或 NaN 的样式，以及该样式是为 NaN 打印时 n-char-sequence 的含义（7.19.6.1 和 7.24.2.1）。”	
	NaN 打印为 nan，不打印字符序列。无穷大打印为[-/+]inf。
“fprintf 或 fwprintf 函数中%p 转换的输出（7.19.6.1 和 7.24.2.1）。”	
	功能上与%lx 等效。
“在 fscanf 或 fwscanf 函数中%[转换的 scanlist 内，对于既不是第一个也不是最后一个字符，且当^为首字符时不是第二个字符的-字符的解释（7.19.6.2 和 7.24.2.2）。”	
	流未实现。
“通过 fscanf 或 fwscanf 函数中的%p 转换匹配的序列集（7.19.6.2 和 7.24.2.2）。”	
	流未实现。
“失败时由 fgetpos、fsetpos 或 ftell 函数将宏 errno 设置成的值（7.19.9.1、7.19.9.3 和 7.19.9.4）。”	
	流未实现。
“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）。”	
	该序列没有含义。
“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）。”	
	否。
“当请求的大小为零时，calloc、malloc 和 realloc 函数返回一个空指针还是一个指向所分配对象的指针（7.20.3）。”	
	请求的大小增加到 1 字节。如果可以成功分配，则返回指向该空间的指针；否则返回 NULL。

.....（续）	
ISO 标准	**实现**
“当调用中止函数时，打开的输出流是否刷新，打开的流是否关闭，或者临时文件是否被删除（7.20.4.1）。”	
	流未实现。
“由中止函数返回给主机环境的终止状态（7.20.4.1）。”	
	主机环境是应用程序定义的。
“系统函数在其参数不是空指针时返回的值（7.20.4.5）。”	
	主机环境是应用程序定义的。
“本地时区和夏令时（7.23.1）。”	
	由应用程序定义。
“时间的范围和精度可以用 `clock_t` 和 `time_t` 表示（7.23）”	
	`time_t` 类型用于存放秒数，定义为 `long` 类型；`clock_t` 定义为 `unsigned long`。
“时钟函数的时代（7.23.2.1）。”	
	由应用程序定义。
“"C”时区设置中 `strftime`、`strfxtime`、`wcsftime` 和 `wcsfxtime` 函数的`%Z` 说明符的替代字符串（7.23.3.5、7.23.3.6、7.24.5.1 和 7.24.5.2）。”	
“三角函数、双曲线函数、以*e* 为底的指数函数、以*e* 为底的对数函数、误差函数和对数伽马函数是否或何时在符合 IEC 60559 的实现中引发不精确异常（F.9）。”	
	否。
“<math.h>中的函数是否遵从舍入方向模式（F.9）。”	
	不强制舍入模式。

6.14 架构

ISO 标准	**实现**
“赋给头文件<float.h>、<limits.h>和<stdint.h>中指定的宏的值或表达式（C90 和 C99 5.2.4.2、C99 7.18.2 及 7.18.3）。”	
	请参见 *Microchip Unified Standard Library Reference Guide* 中的<float.h>、<limits.h>和<stdint.h>部分。
“在标准中未明确指定时，任何对象中字节的数量、顺序和编码（C99 6.2.6.1）。”	
	小尾数法，先从最低有效字节填充。
“`sizeof` 运算符的结果的值（C90 6.3.3.4 和 C99 6.5.3.4）。”	

7. 库函数

MPLAB XC8 C 编译器随附多个库，其中包括方便进行程序开发的函数、宏和类型。

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。

本章将介绍特定于 MPLAB XC8 的库函数和宏。

7.1 库示例代码

大多数库函数都给出了示例代码。这些示例说明了如何调用这些函数，有的可能还提供了其他方面的使用信息，但不一定完整，也不一定实用。示例代码在不同的目标器件上可能以不同的方式编码，并且在运行时也可能以不同的方式运行。

这些示例可以在软件模拟器中运行，例如 MPLAB X IDE 中的软件模拟器。或者，也可以在硬件上运行，但需要针对正在使用的器件和硬件设置进行修改。示例中未显示在硬件上执行代码时所需的器件配置位，因为这些配置位因器件而异。如果使用的是 MPLAB X IDE，请利用其内置工具生成初始化配置位所需的代码，这些代码可以复制并粘贴到项目的源代码中。有关 Configuration Bits（配置位）窗口的说明和使用，请参见《MPLAB® X IDE 用户指南》。

许多库示例使用 `printf()` 函数。除了示例中显示的代码之外，可能还需要相应的代码将该函数打印到所选的外设。

在 MPLAB X IDE 软件模拟器中运行示例时，可以将 `printf()` 函数的输出发送到 USART（对于某些器件，该外设称为 UART）并显示在窗口中。为此，必须：

- 在 MPLAB X IDE 中使能 USART IO 功能（IDE 可能提供 USART 选项）。
- 确保项目代码初始化并使能 IDE 使用的同一 USART。
- 确保项目代码定义“打印字节”函数，用于将一个字节发送给相关的 USART。
- 确保 `printf()` 函数将调用相关的打印字节函数。

一些编译器可能提供已实现如上所述的 USART 初始化和打印字节函数的通用代码。对于其他工具，通常可以使用 Microchip 代码配置器（Microchip Code Configurator，MCC）来生成此代码。检查 MCC 是否适用于目标器件。即使不适用，也可以改写用于类似器件的 MCC 输出。通常，MCC 中的默认 USART 设置可与软件模拟器一起使用，但这些设置可能不适合最终应用。配置 USART 后，除了 `printf()` 之外，还可以使用写入 `stdout` 的任何标准 IO 库函数。

一些库示例可能还使用 `scanf()` 函数。除了示例中显示的代码之外，可能还需要相应的代码让该函数读取所选的外设。

在 MPLAB X IDE 软件模拟器中运行示例时，可以使用 `scanf()` 函数来读取从文本文件获取输入的 USART。为此，必须：

- 在 MPLAB X IDE 中使能 USART IO 功能。
- 确保项目代码初始化并使能 IDE 使用的同一 USART。
- 确保项目代码定义“读取字节”函数，用于从相关的 USART 读取一个字节。
- 确保 `scanf()` 函数将调用相关的读取字节函数。
- 提供包含所需输入的文本文件，并通过寄存器注入将该文件的内容传递给与 IDE 使用的 USART 相关联的接收寄存器。

一些编译器可能提供已实现如上所述的 USART 初始化和读取字节函数的通用代码。对于其他工具，通常可以使用 Microchip 代码配置器（MCC）来生成此代码。通常，MCC 使用的默认 USART 设置可与软件模拟器一起使用，但这些设置可能不适合最终应用。配置 USART 后，除了 `scanf()` 之外，还可以使用读取 `stdin` 的任何标准 IO 库函数。

有关 MPLAB X IDE 的更多信息，请参见《MPLAB® X IDE 用户指南》；有关 MCC 工具的更多信息，请参见 *MPLAB® Code Configurator v3.xx User's Guide*。

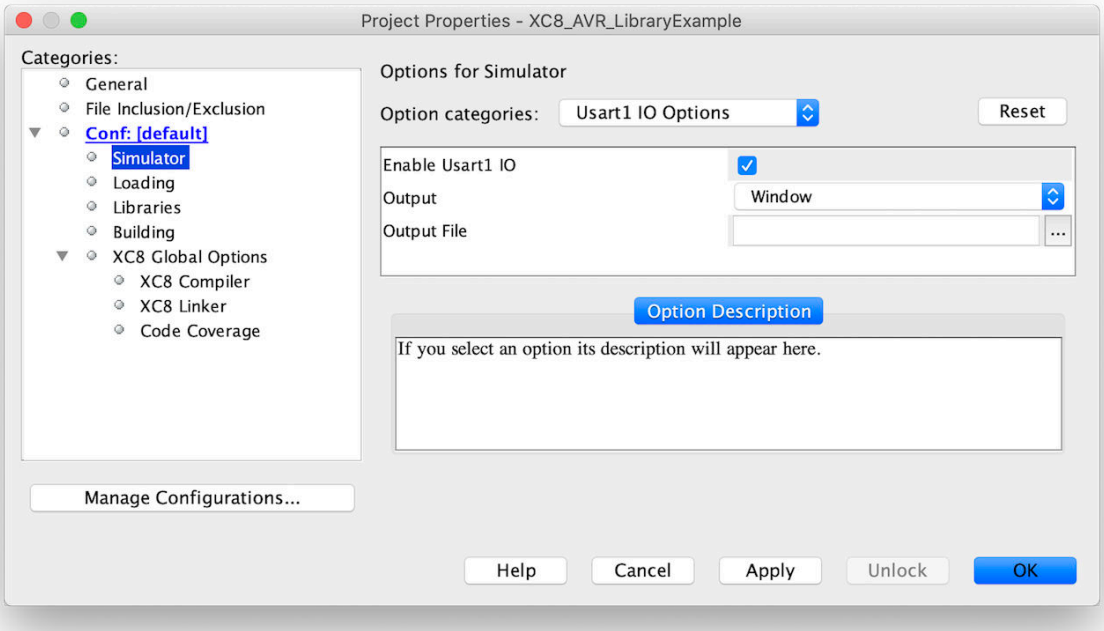
以下章节将更详细地讨论上述特定于编译器的实现。

7.1.1 8 位 AVR MCU 的示例代码

MPLAB X IDE 中的 UART IO 功能可用于查看 `stdout` 流的输出。正确配置后，在软件模拟器中运行程序时，可以从 IDE 查看 `printf()` 和其他函数的输出。

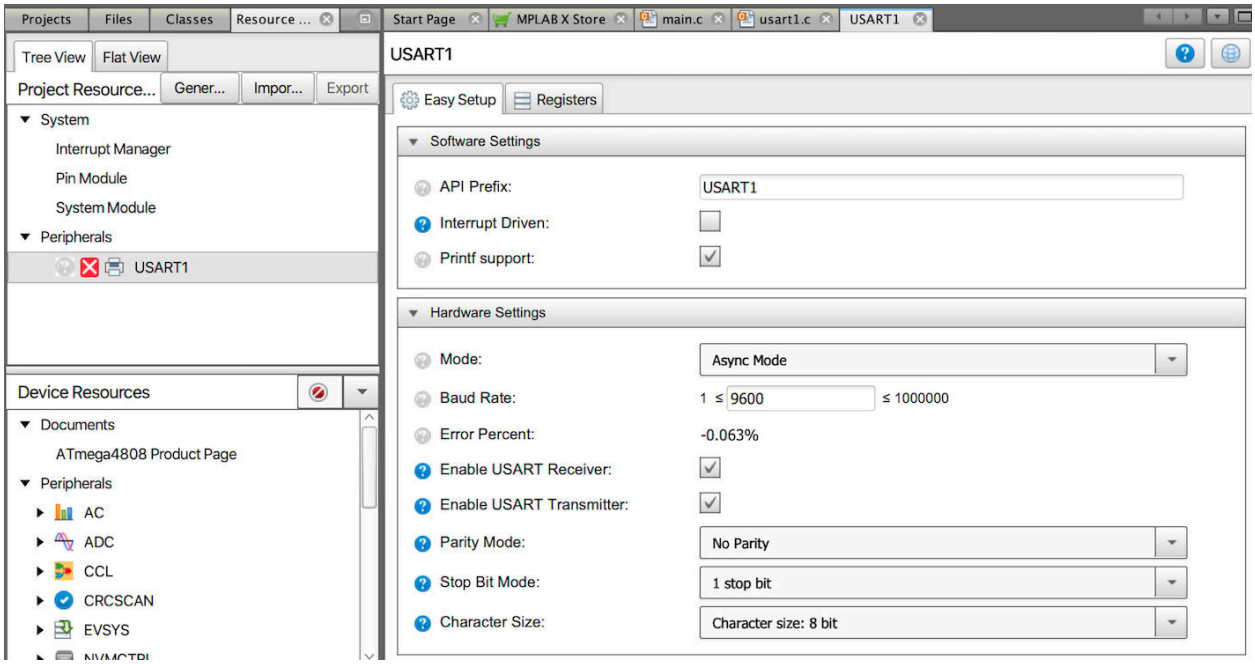
该功能通过 **Project properties > Simulator > USARTx IO Options**（项目属性 > 软件模拟器 > USARTx IO 选项）使能。可能会有多个 USART 供您选择。请选择您的代码将写入的 USART。输出可以显示在 IDE 的窗口中，或者发送到主机上的文件中，具体取决于您在对话框中所做的选择。

图 7-1. 在 MPLAB X IDE 中使能 USART IO 功能



通过 MCC 生成的 USART 初始化和打印字节函数可在软件模拟器中使用 `printf()`。如果使能 USART 对话框中的 **Printf support**（Printf 支持）复选框，MCC 将通过 `FDEV_SETUP_STREAM()` 宏生成必要映射来确保 `printf()` 调用正确的打印字节函数。应可在软件模拟器中使用默认通信设置，但如果要在硬件上使用 USART，则可能需要更改这些设置。

图 7-2. 使用 MCC 初始化 USART



7.2 <boot.h>自举程序函数

该模块中的宏为一些 AVR 器件上的自举程序支持功能提供 C 语言接口。这些宏适用于所有大小的闪存。

这些宏不会自动禁止全局中断，而是由编程人员来进行。有关在写入闪存期间允许全局中断的注意事项，另请参见所用处理器数据手册。

7.2.1 boot_is_spm_interrupt Macro

检查是否已允许存储程序存储器中断。

包含文件

<avr/boot.h>

原型

```
int boot_is_spm_interrupt(void);
```

备注

该宏在 SPM 中断允许位置 1 时返回 1；在其他情况下则返回 0。

示例

```
#include <avr/boot.h>
int main(void)
{
  if(boot_is_spm_interrupt())
    disableMode();
}
```

7.2.2 boot_lock_bits_set Macro

将自举程序锁定位置 1。

包含文件

<avr/boot.h>

原型

```
int boot_lock_bits_set(unsigned char mask);
```

备注

该宏将 SPM 控制寄存器中的位掩码指定的位置 1。自举程序锁定位置 1 后，只能通过全片擦除清零，而此操作又将擦除自举程序本身。

示例

```
#include <avr/boot.h>
int main(void)
{
  boot_lock_bits_set(_BV(BLB11)|_BV(BLB12));
}
```

7.2.3 boot_lock_bits_set_safe Macro

将自举程序锁定位置 1。

包含文件

<avr/boot.h>

原型

```
int boot_lock_bits_set_safe(unsigned char mask);
```

备注

该宏用于在确保 EEPROM 和 SPM 操作完成后将 SPM 控制寄存器中的位掩码指定的位置 1。自举程序锁定位置 1 后，只能通过全片擦除清零，而此操作又将擦除自举程序本身。

示例

```
#include <avr/boot.h>
int main(void)
{
  boot_lock_bits_set_safe(_BV(BLB11)|_BV(BLB12));
}
```

7.2.4 boot_lock_fuse_bits_get Macro

读取锁定位或熔丝位。

包含文件

<avr/boot.h>

原型

```
int boot_lock_fuse_bits_get(unsigned int address);
```

备注

该宏返回指定地址处的锁定位或熔丝位的值。*address* 参数可以是 GET_LOW_FUSE_BITS、GET_LOCK_BITS、GET_EXTENDED_FUSE_BITS 或 GET_HIGH_FUSE_BITS 中的一个。返回的位为物理值，如果某个位位置为 0，则表示相应位置已编程。

示例

```
#include <avr/boot.h>
int main(void)
{
  boot_lock_bits_set(_BV(BLB11)|_BV(BLB12));
}
```

7.2.5 boot_page_erase Macro

擦除包含地址的闪存页。

包含文件

<avr/boot.h>

原型

```
void boot_page_erase(unsigned int address);
```

备注

擦除包含指定字节地址的闪存页。

示例

```
#include <avr/boot.h>
int main(void)
{
  boot_page_erase(0x100);
}
```

7.2.6 boot_page_erase_safe Macro

擦除包含地址的闪存页。

包含文件

<avr/boot.h>

原型

```
void boot_page_erase_safe(unsigned int address);
```

备注

在确保 EEPROM 和 SPM 操作完成后擦除包含指定字节地址的闪存页。

示例

```
#include <avr/boot.h>
int main(void)
{
  boot_page_erase_safe(0x100);
}
```

7.2.7 boot_page_fill Macro

将一个字放入闪存地址对应的自举程序临时页缓冲区。

包含文件

<avr/boot.h>

原型

```
void boot_page_fill(unsigned int address, unsign int value);
```

备注

该宏将指定值放入闪存地址对应的自举程序临时页缓冲区。地址为字节地址；但 AVR 器件将字写入缓冲区，因此对于要写入的每个 16 位字，将地址递增 2。数据的 LSB 写入低地址；数据的 MSB 写入高地址。

示例

```
#include <avr/boot.h>
int main(void)
{
```

```
  boot_page_fill(0x100, 0x55);
}
```

7.2.8 boot_page_fill_safe Macro

将一个字放入闪存地址对应的自举程序临时页缓冲区。

包含文件

<avr/boot.h>

原型

```
void boot_page_fill_safe(unsigned int address, unsign int value);
```

备注

该宏用于在确保 EEPROM 和 SPM 操作完成后将指定值放入闪存地址对应的自举程序临时页缓冲区。地址为字节地址；但 AVR 器件将字写入缓冲区，因此对于要写入的每个 16 位字，将地址递增 2。数据的 LSB 写入低地址；数据的 MSB 写入高地址。

示例

```
#include <avr/boot.h>
int main(void)
{
  boot_page_fill_safe(0x100, 0x55);
}
```

7.2.9 boot_page_write Macro

将自举程序临时缓冲区的内容写入闪存页。

包含文件

<avr/boot.h>

原型

```
void boot_page_write(unsigned int address);
```

备注

该宏用于将自举程序临时缓冲区的内容写入指定地址对应的闪存页。地址为字节地址。

示例

```
#include <avr/boot.h>
int main(void)
{
  boot_page_write(0x55);
}
```

7.2.10 boot_page_write_safe Macro

将自举程序临时缓冲区的内容写入闪存页。

包含文件

<avr/boot.h>

原型

```
void boot_page_write_safe(unsigned int address);
```

备注

该宏用于在确保 EEPROM 和 SPM 操作完成后将自举程序临时缓冲区的内容写入指定地址对应的闪存页中。地址为字节地址。

示例

```
#include <avr/boot.h>
int main(void)
{
  boot_page_write_safe(0x55);
}
```

7.2.11 boot_rww_busy Macro

检查边写边读（Read-While-Write，RWW）段是否繁忙。

包含文件

<avr/boot.h>

原型

```
int boot_rww_busy(void);
```

备注

该宏在边写边读段繁忙位置 1 时返回 1；在其他情况下则返回 0。

示例

```
#include <avr/boot.h>
int main(void)
{
  while(boot_rww_busy())
    waitRWW();
}
```

7.2.12 boot_rww_enable Macro

使能边写边读（RWW）段。

包含文件

<avr/boot.h>

原型

```
void boot_rww_enable(void);
```

备注

该宏用于使能边写边读段，这样当擦除或写入该段时，仍可从非边写边读（No-Read-While-Write，NRWW）段执行代码。

示例

```
#include <inttypes.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>

void boot_program_page (uint32_t page, uint8_t *buf) {
  uint16_t i;
  uint8_t sreg;

  sreg = SREG;
  cli();
  eeprom_busy_wait ();
  boot_page_erase (page);
  boot_spm_busy_wait ();      // Wait until the memory is erased.
  for (i=0; i<SPM_PAGESIZE; i+=2) {
    // Set up little-endian word.
    uint16_t w = *buf++;
    w += (*buf++) << 8;
    boot_page_fill (page + i, w);
  }
```

```
  boot_page_write (page);     // Store buffer in flash page.
  boot_spm_busy_wait();       // Wait until the memory is written.
  // Reenable RWW-section again. We need this if we want to jump back
  // to the application after bootloading.
  boot_rww_enable ();
  // Re-enable interrupts (if they were ever enabled).
  SREG = sreg;
}
```

7.2.13 boot_rww_enable_safe Macro

使能边写边读（RWW）段。

包含文件

<avr/boot.h>

原型

```
void boot_rww_enable_safe(void);
```

备注

该宏用于在确保 EEPROM 和 SPM 操作完成后使能边写边读段。使能并擦除或写入边写边读段时，仍可从非边写边读（NRWW）段执行代码。

示例

```
#include <inttypes.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>

void boot_program_page (uint32_t page, uint8_t *buf) {
  uint16_t i;
  uint8_t sreg;

  sreg = SREG;
  cli();
  eeprom_busy_wait ();
  boot_page_erase (page);
  boot_spm_busy_wait ();      // Wait until the memory is erased.
  for (i=0; i<SPM_PAGESIZE; i+=2) {
    // Set up little-endian word.
    uint16_t w = *buf++;
    w += (*buf++) << 8;
    boot_page_fill (page + i, w);
  }
  boot_page_write (page);     // Store buffer in flash page.
  boot_spm_busy_wait();       // Wait until the memory is written.
  // Reenable RWW-section again.We need this if we want to jump back
  // to the application after bootloading.
  boot_rww_enable_safe();
  // Re-enable interrupts (if they were ever enabled).
  SREG = sreg;
}
```

7.2.14 boot_signature_byte_get Macro

读取地址处的签名行字节。

包含文件

<avr/boot.h>

原型

```
unsigned char boot_signature_byte_get(unsigned int address);
```

备注

该宏返回指定地址处的签名行字节。对于一些器件，该宏可获取出厂时存储的振荡器校准字节。地址参数可以是 0-0x1f（如数据手册中所述）。

示例

```
#include <avr/boot.h>
int main(void)
{
  unsigned char signature;

  signature = boot_signature_byte_get(0x1f);
}
```

7.2.15 boot_spm_busy Macro

检查 SPM 是否繁忙。

包含文件

<avr/boot.h>

原型

```
int boot_spm_busy(void);
```

备注

该宏在 SPM 允许位置 1 时返回 1；在其他情况下则返回 0。

示例

```
#include <avr/boot.h>
int main(void)
{
  if(boot_spm_busy())
    altMode();
}
```

7.2.16 boot_spm_busy_wait 宏

在 SPM 繁忙时等待。

包含文件

<avr/boot.h>

原型

```
int boot_spm_busy_wait(void);
```

备注

等待至 SPM 变为未使能状态。

示例

```
#include <avr/boot.h>
int main(void)
{
  boot_spm_busy_wait();
}
```

7.2.17 boot_spm_interrupt_disable 宏

禁止 SPM 中断。

包含文件

<avr/boot.h>

原型

```
void boot_spm_interrupt_disable(void);
```

备注

该宏用于禁止与 SPM 相关的中断。

示例

```
#include <avr/boot.h>

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

7.2.18 boot_spm_interrupt_enable 宏

禁止 SPM 中断。

包含文件

<avr/boot.h>

原型

```
void boot_spm_interrupt_enable(void);
```

备注

该宏用于允许与 SPM 相关的中断。

示例

```
#include <avr/boot.h>

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

7.3 <cpufunc.h> CPU 相关函数

头文件 cpufunc.h 包含与指令执行相关的函数。

7.3.1 _MemoryBarrier 宏

实现读/写存储器屏障。

包含文件

<avr/cpufunc.h>

原型

```
void _MemoryBarrier(void);
```

备注

存储器屏障会指示编译器不要缓存寄存器中任何超出屏障的存储器数据。与使用 volatile 限定符声明某个对象来阻止特定优化相比，此操作有时会更为有效。

示例

```
#include <avr/cpufunc.h>
int main(void)
{
  data = readMode();
  _MemoryBarrier();
  processData(&data);
}
```

7.3.2 _NOP 宏

该宏不执行任何操作。

包含文件

<avr/cpufunc.h>

原型

```
void _NOP(void);
```

备注

该宏执行 nop 指令。

该宏不得用于实现延时。最好使用来自<util/delay_basic.h>或<util/delay.h>的函数实现这一目的。由于 nop 指令没有被优化掉，因此能够可靠地用作调试断点的位置。

示例

```
#include <avr/cpufunc.h>
int main(void)
{
  while(1)
    _NOP();
}
```

7.4 <delay.h>延时函数

头文件 delay.h 包含可在程序执行中产生延时的函数。

7.4.1 _delay_ms 函数

延迟一段指定的时间。

包含文件

<util/delay.h>

原型

```
void _delay_ms(double ms);
```

参数

ms 要延迟的时间（毫秒数）。

备注

该函数使用_delay_loop_2()函数来延迟执行。宏 F_CPU 应定义为指定 CPU 时钟频率（Hz）的常量。必须使能编译器优化器才能实现精确的延时。

最大可能延时为 4294967.295 ms/F_CPU（MHz）。如果请求的值大于最大可能延时，则将导致溢出并产生 0 µs 的延时。

将请求的毫秒数转换为时钟周期数无法总是得到整数值。默认情况下，时钟周期数向上舍入到下一个整数。这样能够确保至少达到请求的延时量。

通过在包含该头文件前定义宏__DELAY_ROUND_DOWN__或__DELAY_ROUND_CLOSEST__，可分别将算法设置为向下舍入或舍入到最接近的整数。

示例

```
#define F_CPU 4000000UL
#include <util/delay.h>

int main(void)
```

```
{
  _delay_ms(20);  // delay for 20 milli seconds
}
```

7.4.2 _delay_us 函数

延迟一段指定的时间。

包含文件

<util/delay.h>

原型

```
void _delay_us(double us);
```

参数

µs 要延迟的时间（微秒数）。

备注

该函数使用_delay_loop_1()函数来延迟执行。宏 F_CPU 应定义为指定 CPU 时钟频率（Hz）的常量。必须使能编译器优化器才能实现精确的延时。

最大可能延时为 4294967.295 µs/F_CPU（MHz）。如果请求的值大于最大可能延时，则将导致溢出并产生 0 µs 的延时。

将请求的微秒数转换为时钟周期数无法总是得到整数值。默认情况下，时钟周期数向上舍入到下一个整数。这样能够确保至少达到请求的延时量。

通过在包含该头文件前定义宏__DELAY_ROUND_DOWN__或__DELAY_ROUND_CLOSEST__，可分别将算法设置为向下舍入或舍入到最接近的整数。

示例

```
#define F_CPU 4000000UL
#include <util/delay.h>

int main(void)
{
  _delay_us(20);  // delay for 20 micro seconds
}
```

7.5 <pgmspace.h>

<pgmspace.h>头文件用于声明与读取程序存储器相关的宏。这些功能中的大多数主要是针对传统代码而包含的，因为使用 const-data-in-progmem 功能可以更轻松地读取程序存储器。

7.5.1 pgm_get_far_address 宏

获取对象的 far 地址。

包含文件

<avr/pgmspace.h>

原型

```
uint_farptr_t pgm_get_far_address(object);
```

备注

获取 *object* 的 far（32 位）地址。

示例

```
#include <avr/pgmspace.h>
const unsigned char PROGMEM romObj = 0x55;
int main(void)
{
  uint_farptr_t * cp;
  cp = pgm_get_far_address(&romObj);
}
```

7.5.2 pgm_read_byte 宏

从具有 near 地址的程序空间读取一个字节。

包含文件

<avr/pgmspace.h>

原型

```
unsigned char pgm_read_byte(unsigned int);
```

备注

从具有 16 位（near）地址的程序空间读取一个字节。

示例

```
#include <avr/pgmspace.h>
const unsigned char PROGMEM romObj = 0x55;
int main(void)
{
  unsigned char val;
  val = pgm_read_byte(&romObj);
}
```

7.5.3 pgm_read_byte_far 宏

从具有 far 地址的程序空间读取一个字节。

包含文件

<avr/pgmspace.h>

原型

```
unsigned char pgm_read_byte_far(unsigned long int);
```

备注

从具有 32 位（far）地址的程序空间读取一个字节。这些函数使用 elpm 指令，因此可以访问程序存储器中的任何地址。

示例

```
#include <avr/pgmspace.h>
const unsigned char PROGMEM romObj = 0x55;
int main(void)
{
  unsigned char val;
  val = pgm_read_byte_far(&romObj);
}
```

7.5.4 pgm_read_byte_near 宏

从具有 near 地址的程序空间读取一个字节。

包含文件

<avr/pgmspace.h>

原型

```
unsigned char pgm_read_byte_near(unsigned int);
```

备注

从具有 16 位（near）地址的程序空间读取一个字节。

示例

```
#include <avr/pgmspace.h>
const unsigned char PROGMEM romObj = 0x55;
int main(void)
{
  unsigned char val;
  val = pgm_read_byte_near(&romObj);
}
```

7.5.5 pgm_read_dword_near 宏

从具有 near 地址的程序空间读取一个双倍宽度字。

包含文件

<avr/pgmspace.h>

原型

```
unsigned long int pgm_read_dword_near(unsigned int);
```

备注

从具有 16 位（near）地址的程序空间读取一个 32 位字。

示例

```
#include <avr/pgmspace.h>
const unsigned long PROGMEM romObj = 0x55;
int main(void)
{
  unsigned long val;
  val = pgm_read_dword_near(&romObj);
}
```

7.5.6 pgm_read_dword_far 宏

从具有 far 地址的程序空间读取一个双倍宽度字。

包含文件

<avr/pgmspace.h>

原型

```
unsigned long int pgm_read_byte_far(unsigned long int);
```

备注

从具有 32 位（far）地址的程序空间读取一个 32 位字。这些函数使用 elpm 指令，因此可以访问程序存储器中的任何地址。

示例

```
#include <avr/pgmspace.h>
const unsigned long int PROGMEM romObj = 0x55;
int main(void)
{
  unsigned long int val;
```

```
  val = pgm_read_dword_far(&romObj);
}
```

7.5.7 pgm_read_dword_near 宏

从具有 near 地址的程序空间读取一个双倍宽度字。

包含文件

<avr/pgmspace.h>

原型

```
unsigned long int pgm_read_dword_near(unsigned int);
```

备注

从具有 16 位（near）地址的程序空间读取一个 32 位字。

示例

```
#include <avr/pgmspace.h>
const unsigned long PROGMEM romObj = 0x55;
int main(void)
{
  unsigned long val;
  val = pgm_read_dword_near(&romObj);
}
```

7.5.8 pgm_read_float 宏

从具有 near 地址的程序空间读取一个浮点对象。

包含文件

<avr/pgmspace.h>

原型

```
float pgm_read_word(unsigned int);
```

备注

从具有 16 位（near）地址的程序空间读取一个 `float` 对象。

示例

```
#include <avr/pgmspace.h>
const float PROGMEM romObj = 1.23;
int main(void)
{
  float val;
  val = pgm_read_float(&romObj);
}
```

7.5.9 pgm_read_float_far 宏

从具有 far 地址的程序空间读取一个浮点对象。

包含文件

<avr/pgmspace.h>

原型

```
float pgm_read_byte_far(unsigned long int);
```

备注

从具有 32 位（far）地址的程序空间读取一个 float 对象。这些函数使用 elpm 指令，因此可以访问程序存储器中的任何地址。

示例

```
#include <avr/pgmspace.h>
const float PROGMEM romObj = 1.23;
int main(void)
{
  float val;
  val = pgm_read_float_far(&romObj);
}
```

7.5.10 pgm_read_float_near 宏

从具有 near 地址的程序空间读取一个浮点对象。

包含文件

<avr/pgmspace.h>

原型

```
float pgm_read_word_near(unsigned int);
```

备注

从具有 16 位（near）地址的程序空间读取一个 float 对象。

示例

```
#include <avr/pgmspace.h>
const float PROGMEM romObj = 1.23;
int main(void)
{
  float val;
  val = pgm_read_float_near(&romObj);
}
```

7.5.11 pgm_read_ptr 宏

从具有 near 地址的程序空间读取一个指针。

包含文件

<avr/pgmspace.h>

原型

```
void * pgm_read_ptr(unsigned int);
```

备注

从具有 16 位（near）地址的程序空间读取一个 16 位通用指针。

示例

```
#include <avr/pgmspace.h>
unsigned int input;
const unsigned int PROGMEM * romPtr = &input;
int main(void)
{
  unsigned int * val;
  val = (unsigned int *)pgm_read_ptr(&romPtr);
}
```

7.5.12 pgm_read_ptr_far 宏

从具有 far 地址的程序空间读取一个指针。

包含文件

<avr/pgmspace.h>

原型

```
void * pgm_read_byte_far(unsigned long int);
```

备注

从具有 32 位（far）地址的程序空间读取一个 16 位通用指针。这些函数使用 elpm 指令，因此可以访问程序存储器中的任何地址。

示例

```
#include <avr/pgmspace.h>
unsigned int input;
const unsigned int PROGMEM * romPtr = &input;
int main(void)
{
  unsigned int * val;
  val = pgm_read_ptr_far(&romPtr);
}
```

7.5.13 pgm_read_ptr_near 宏

从具有 near 地址的程序空间读取一个指针。

包含文件

<avr/pgmspace.h>

原型

```
void * pgm_read_ptr_near(unsigned int);
```

备注

从具有 16 位（near）地址的程序空间读取一个 16 位通用指针。

示例

```
#include <avr/pgmspace.h>
unsigned int input;
const unsigned int PROGMEM * romPtr = &input;
int main(void)
{
  unsigned int * val;
  val = (unsigned int *)pgm_read_ptr_near(&romPtr);
}
```

7.5.14 pgm_read_word 宏

从具有 near 地址的程序空间读取一个字。

包含文件

<avr/pgmspace.h>

原型

```
unsigned int pgm_read_word(unsigned int);
```

备注

从具有 16 位（near）地址的程序空间读取一个 16 位字。

示例

```
#include <avr/pgmspace.h>
const unsigned int PROGMEM romObj = 0x55;
int main(void)
{
  unsigned int val;
  val = pgm_read_word(&romObj);
}
```

7.5.15 pgm_read_word_far 宏

从具有 far 地址的程序空间读取一个字。

包含文件

<avr/pgmspace.h>

原型

```
unsigned int pgm_read_byte_far(unsigned long int);
```

备注

从具有 32 位（far）地址的程序空间读取一个 16 位字。这些函数使用 elpm 指令，因此可以访问程序存储器中的任何地址。

示例

```
#include <avr/pgmspace.h>
const unsigned int PROGMEM romObj = 0x55;
int main(void)
{
  unsigned int val;
  val = pgm_read_word_far(&romObj);
}
```

7.5.16 pgm_read_word_near 宏

从具有 near 地址的程序空间读取一个字。

包含文件

<avr/pgmspace.h>

原型

```
unsigned int pgm_read_word_near(unsigned int);
```

备注

从具有 16 位（near）地址的程序空间读取一个 16 位字。

示例

```
#include <avr/pgmspace.h>
const unsigned int PROGMEM romObj = 0x55;
int main(void)
{
  unsigned int val;
  val = pgm_read_word_near(&romObj);
}
```

7.5.17 PSTR 宏

获取指向程序空间中的字符串的指针。

包含文件

<avr/pgmspace.h>

原型

```
const PROGMEM char * PSTR(string);
```

备注

获取指向程序空间中的字符串的指针。

示例

```
#include <avr/pgmspace.h>
int main(void)
{
 const PROGMEM char * cp;
  cp = PSTR("hello");
}
```

7.6 <sfr_defs.h>

头文件 sfr_defs.h 包含有助于访问特殊功能寄存器的函数。

7.6.1 _BV 宏

允许位操作的宏。

包含文件

<avr/sfr_defs.h>

原型

```
void _BV(bit_number);
```

备注

该宏用于将位编号转换为字节值，从而允许访问地址中的位。

编译器将使用硬件 sbi 或 cbi 指令在适当时执行访问，否则执行读或写操作。

示例

```
#include <xc.h>
#include <avr/sfr_defs.h>
int main(void)
{
  PORTB |= _BV(PB1);  // set bit #1 of PORTB
  EECR &= ~(_BV(EEPM4) | _BV(EEPM5));  // clear bits #4 and #5 in EECR
}
```

7.6.2 Bit_is_clear 宏

在 SFR 中的位清零时返回 true 的宏。

包含文件

<avr/sfr_defs.h>

原型

```
int _bit_is_clear(sfr, bit_number);
```

备注

该宏用于测试 *sfr* 中的指定位是否清零，清零时返回 true；否则返回 0。

示例

```
#include <xc.h>
#include <avr/sfr_defs.h>
int main(void)
{
  if(bit_is_clear(EIMSK, PCIE2))
    proceed = 1;
}
```

7.6.3 bit_is_set 宏

在 SFR 中的位置 1 时返回 true 的宏。

包含文件

```
<avr/sfr_defs.h>
```

原型

```
int _bit_is_set(sfr, bit_number);
```

备注

该宏用于测试 *sfr* 中的指定位是否置 1，置 1 时返回 true；否则返回 0。

示例

```
#include <xc.h>
#include <avr/sfr_defs.h>
int main(void)
{
  if(bit_is_set(EIMSK, PCIE2))
    proceed = 0;
}
```

7.6.4 loop_until_bit_is_clear 宏

等待直到 SFR 中的位清零的宏。

包含文件

```
<avr/sfr_defs.h>
```

原型

```
int loop_until_bit_is_clear(sfr, bit_number);
```

备注

该宏循环至 *sfr* 中的指定位清零为止。

示例

```
#include <xc.h>
#include <avr/sfr_defs.h>
int main(void)
{
  loop_until_bit_is_clear(PINA, PINA3);
  process();
}
```

7.6.5 loop_until_bit_is_set 宏

等待直到 SFR 中的位置 1 的宏。

包含文件

```
<avr/sfr_defs.h>
```

原型

```
int loop_until_bit_is_set(sfr, bit_number);
```

备注

该宏循环至 *sfr* 中的指定位置 1 为止。

示例

```
#include <xc.h>
#include <avr/sfr_defs.h>
int main(void)
{
  startConversion();
  loop_until_bit_is_set(ADCSRA, ADIF);
}
```

7.7 <sleep.h>

可使 8 位 AVR 器件进入不同的休眠模式。有关这些模式的详细信息，请参见器件数据手册。

7.7.1 set_sleep_mode 宏

指定器件在休眠模式下的行为。

包含文件

<avr/sleep.h>

原型

```
void set_sleep_mode(mode);
```

备注

该宏通过设置器件寄存器中的相应位来指定器件在休眠模式下的行为。模式参数可以是包含<avr/sleep.h>后定义的相应宏，例如 SLEEP_MODE_IDLE、SLEEP_MODE_PWR_DOWN、SLEEP_MODE_PWR_SAVE 和 SLEEP_MODE_STANDBY 等。

示例

```
#include <xc.h>
#include <avr/sleep.h>
int main(void)
{
  set_sleep_mode(SLEEP_MODE_IDLE);
  cli();
  if (some_condition)
  {
    sleep_enable();
    sei();
    sleep_cpu();
    sleep_disable();
  }
  sei();
}
```

7.7.2 sleep_bod_disable 宏

禁止在休眠前进行欠压检测。

包含文件

<avr/sleep.h>

原型

```
void sleep_bod_disable(void);
```

备注

该宏可用于禁止在器件进入休眠模式前进行欠压检测。该宏将生成嵌入汇编代码来正确实现用于禁止欠压检测器（Brown Out Detector，BOD）的定时序列。但是，禁止 BOD 后，器件进入休眠模式的周期数有限制，否则不会真正禁止 BOD。

并非所有器件均支持该功能。

示例

```
#include <xc.h>
#include <avr/sleep.h>
int main(void)
{
  initSystem();
  sleep_enable();
  sleep_bod_disable();
  sei();
  sleep_cpu();
  sleep_disable();
}
```

7.7.3 sleep_cpu 宏

使器件休眠。

包含文件

```
<avr/sleep.h>
```

原型

```
void sleep_cpu(void);
```

备注

该宏用于使器件休眠。必须先将休眠使能位置 1 才能使器件休眠。

示例

```
#include <xc.h>
#include <avr/sleep.h>
int main(void)
{
  cli();
  if (some_condition)
  {
    sleep_enable();
    sei();
    sleep_cpu();
    sleep_disable();
  }
  sei();
}
```

7.7.4 sleep_disable 宏

使器件退出休眠模式。

包含文件

```
<avr/sleep.h>
```

原型

```
void sleep_disable(void);
```

备注

该宏用于将休眠使能位清零，从而防止器件进入休眠模式。

示例

```
#include <xc.h>
#include <avr/sleep.h>
int main(void)
{
  initSystem();
  sleep_enable();
  sei();
  sleep_cpu();
  sleep_disable();
}
```

7.7.5 sleep_enable 宏

使器件进入休眠模式。

包含文件

<avr/sleep.h>

原型

```
void sleep_enable(void);
```

备注

该宏通过将休眠使能位置 1 来使器件进入休眠模式，从而可使用 sleep_cpu() 在需要时将器件置于休眠模式。

器件如何退出休眠模式取决于通过 set_sleep_mode() 函数选择的特定模式。

示例

```
#include <xc.h>
#include <avr/sleep.h>
int main(void)
{
  cli();
  if (some_condition)
  {
    sleep_enable();
    sei();
    sleep_cpu();
    sleep_disable();
  }
  sei();
}
```

7.7.6 sleep_mode 宏

使能休眠模式，使器件进入休眠状态。

包含文件

<avr/sleep.h>

原型

```
void sleep_mode(void);
```

备注

该宏用于将休眠使能位清零，使器件进入休眠模式并在之后禁止休眠使能位。由于该宏在某些情况下可能引发竞争，因此可以改用单独执行上述各项操作的宏，从而确保在器件进入休眠模式前立即允许中断。

示例

```
#include <xc.h>
#include <avr/sleep.h>
int main(void)
{
  initSystem();
  sleep_mode();
}
```

7.8 内置函数

编译器支持大量内置函数，这些函数可像调用常规库函数一样使用，但通常扩展为在使用点插入的嵌入汇编代码序列。由于它们不需要调用返回序列，因此使用起来较为高效，而且相较于以类似方式定义的预处理器宏来说更为稳健。

7.8.1 builtin_avr_cli 内置函数

插入用于允许全局中断的 `cli` 指令。

原型

```
void __builtin_avr_cli(void);
```

备注

`di()`宏（在代码中包含<xc.h>头文件后可用）执行类似任务。

示例

```
unsigned int read_timer1(void)
{
  unsigned int val;

  __builtin_avr_cli(); // disable interrupts
  val = TCNT1;         // read timer value register; TEMP used internally
  __builtin_avr_sei(); // re-enable interrupts
  return val;
}
```

7.8.2 builtin_avr_delay_cycles 内置函数

插入简单的延时指令序列。

原型

```
void __builtin_avr_delay_cycles(unsigned long tick)
```

备注

该内置函数将生成用于延迟数个节拍周期的代码。节拍操作数必须为常量表达式，无法考虑可能延长延时的中断的作用。

示例

```
void toggle(void) {
  PORTB = 0x0F;
  __builtin_avr_delay_cycles(10);
  PORTB = 0x00;
}
```

7.8.3 builtin_avr_flash_segment 内置函数

返回完整地址的闪存段。

原型

```
char __builtin_avr_flash_segment(const __memx void * mp)
```

备注

该内置函数用于返回假定为 24 位__memx 字节地址的参数的闪存段（64 KB 块）编号。如果地址未指向闪存，则内置函数返回-1。

示例

```
const __memx int myObject = 22;

int main(void) {
  char segment;

  segment = __builtin_avr_flash_segment(&myObject);
}
```

7.8.4 builtin_avr_fmul 内置函数

插入 fmul 小数乘法指令序列。

原型

```
unsigned int __builtin_avr_fmul(unsigned char x, unsigned char y)
```

备注

该内置函数将生成相应的代码以将操作数装入适当寄存器、对 **x** 和 **y** 执行小数无符号乘法并存储结果。

示例

```
#include <stdint.h>

int main(void) {
  uint16_t result;
  uint8_t a, b;

  a = 128;
  b = 3;
  result = __builtin_avr_fmul(a, b);  // result will be assigned 768
}
```

7.8.5 builtin_avr_fmuls 内置函数

插入 fmuls 小数乘法指令序列。

原型

```
int __builtin_avr_fmuls(char x, char y)
```

备注

该内置函数将生成相应的代码以将操作数装入适当寄存器、对 **x** 和 **y** 执行小数有符号乘法并存储结果。

示例

```
#include <stdint.h>

int main(void) {
  int16_t result;
  int8_t a, b;

  a = 128;
  b = 3;
  result = __builtin_avr_fmuls(a, b);  // result will be assigned -768
}
```

7.8.6 builtin_avr_fmulsu 内置函数

插入 fmulsu 小数乘法指令序列。

原型

```
int __builtin_avr_fmulsu(char x, unsigned char y)
```

备注

该内置函数将生成相应的代码以将操作数装入适当寄存器、对 **x** 和 **y** 执行小数有符号和无符号乘法并存储结果。

示例

```
#include <stdint.h>

int main(void) {
  int16_t result;
  int8_t a;
  uint8_t b;

  a = 128;
  b = 3;
  result = __builtin_avr_fmulsu(a, b);  // result will be assigned -768
}
```

7.8.7 builtin_avr_insert_bits 内置函数

插入 nop 指令。

原型

```
uint8_t __builtin_avr_insert_bits (uint32_t map, uint8_t bits, uint8_t val)
```

备注

按照 map 指定的方式将 bits 中的位插入 val，并返回得到的值。

map 中的每个半字节（4 位，共 8 个）控制返回值中的一个位，其位位置与该半字节在 map 中的半字节位置相同。例如，map 的最高有效半字节控制针对返回值最高有效位的插入操作，依此类推。如果 map 中的半字节为 0xF，则在返回值中对应的位是 val 的相应位（未经修改）。值为 0 至 7 的半字节指示在返回值中的相应位是从 bits 内相应位位置的位获取的。

示例

```
#include <stdint.h>
#include <stdio.h>

int main(void) {
  uint8_t  result, myValue, myBits;
  volatile char dummy;

  myValue = 0xF6;
  myBits  = 0x3A;

  result = __builtin_avr_insert_bits(0x01234567, myBits, myValue);
  printf("result is 0x%X\n", result);
  result = __builtin_avr_insert_bits(0x5566FFFF, myBits, myValue);
  printf("result is 0x%X\n", result);
}
```

示例输出

```
result is 0x5C
result is 0xC6
```

示例说明

上例中第一次使用内置函数时，map 值中的第一个半字节（0x0）指示结果的最高有效位将来自 myBits 中的 bit 0。次高有效位将来自 bit 1，依此类推。给定上例中的映射半字节时，第一次执行内置函数将返回 myBits 的值（各个位以相反的顺序出现）。

第二次使用内置函数时，map 的四个低半字节为 0xF，表示结果值的低四位将是 myValue 的低四位（未经修改）。map 的两个最高有效半字节为 0x5，表示返回值的两个最高有效位将是 myBits 的 bit 5（即 1）。同样，结果的下两位将是 myBits 的 bit 6（即 0）。

7.8.8 builtin_avr_nop 内置函数

插入一条 nop（无操作）指令。

原型

```
void __builtin_avr_nop(void);
```

备注

示例

```
int spikeD(void)
{
  PORTD = 0x0F;
  __builtin_avr_nop();
  __builtin_avr_nop();
  PORTD = 0x00;
}
```

7.8.9 builtin_avr_sei 内置函数

插入用于允许全局中断的 sei 指令。

原型

```
void __builtin_avr_sei(void);
```

备注

ei()宏（在代码中包含<xc.h>头文件后可用）执行类似任务。

示例

```
unsigned int read_timer1(void)
{
  unsigned int val;

  __builtin_avr_cli(); // disable interrupts
  val = TCNT1;         // read timer value register; TEMP used internally
  __builtin_avr_sei(); // re-enable interrupts
  return val;
}
```

7.8.10 builtin_avr_sleep 内置函数

插入用于将器件置于休眠模式的 sleep 指令。

原型

```
void __builtin_avr_sleep(void);
```

示例

```
void operationMode(unsigned char channel)
{
  unsigned char val, status=GO;

  while(status) {
```

```
    val = readData(channel);
    __builtin_avr_wdr();
    status = processData(val);
    if(status == WAIT) {
      __builtin_avr_sleep();
      status = GO;
    }
    __builtin_avr_wdr();
  }
}
```

7.8.11 builtin_avr_swap 内置函数

插入 swap 半字节交换指令序列。

原型

```
unsigned char __builtin_avr_swap(unsigned char)
```

备注

该内置函数将生成相应的代码以将操作数装入适当寄存器、执行半字节交换并存储结果。

示例

```
int main(void) {
  uint8_t a;
  uint8_t result;

  a = 0x81;
  result = __builtin_avr_swap(a);  // result will be 0x18
}
```

7.8.12 builtin_avr_wdr 内置函数

插入用于复位看门狗定时器的 wdr 指令。

原型

```
void __builtin_avr_wdr(void);
```

示例

```
void operationMode(unsigned char channel)
{
  unsigned char val, status=GO;

  while(status) {
    val = readData(channel);
    __builtin_avr_wdr();
    status = processData(val);
    if(status == WAIT) {
      __builtin_avr_sleep();
      status = GO;
    }
    __builtin_avr_wdr();
  }
}
```

8. 文档版本历史

版本 A（2018 年 3 月）
本文档的初始版本，改编自 MPLAB XC8 C Compiler User's Guide（DS50002053）。

版本 B（2019 年 3 月）
- 增加了与程序存储器中的 `const` 限定对象相关的信息
- 增加了有关新代码覆盖功能的信息
- 增加了与 chipinfo HTML 文件相关的信息
- 增加了与编译器命令行选项对应的 MPLAB X IDE 项目属性对话框的说明和屏幕截图
- 更新了配置位信息
- 阐明了与绝对对象相关的信息
- 更新了预定义宏表
- 其他更正和改进

版本 C（2020 年 3 月）
- 本指南已迁移到新的撰写和发布系统；与旧版本相比，格式上可能会有差异
- 更新了标准库的文档
- 更新了与 DFP 的结构相关的信息
- 阐明并扩展了与优化相关的信息

版本 D（2020 年 9 月）
- 更新并新增了 MPLAB X IDE 项目属性对话框的屏幕截图
- 增加了 Microchip Studio 项目属性对话框的屏幕截图
- 新增了用于控制新的数据初始化功能的`-Wl,--no-data-init` 选项
- 记录了更多选项，包括：`-Werror`、`-fdata-sections`、`-ffunction-sections` 和`-g`*`level`*

版本 E（2021 年 8 月）
- 删除了标准 C 库函数；现在这些函数在单独的 *Microchip Universal Standard Library Reference Guide* 文档中介绍
- 增加了有关智能 IO 功能的信息
- 新增了`-m[no-]gas-isr-prologues` 选项和 `no_gccisr` 属性
- 新增了`-mcall-isr-prologues` 选项
- 增加了有关代码覆盖功能的信息
- 增加了有关堆栈指导功能的信息
- 扩展并更新了信息链接部分
- 更新并新增了 MPLAB X IDE 项目属性对话框的屏幕截图

版本 F（2022 年 6 月）
- 更正了代码示例并扩展了有关写入中断程序的信息
- 增加了介绍`-msmart-io` 和`-msmart-io-format` 选项的部分，这两个选项用于控制执行格式化 IO 的库代码的功能集
- 增加了介绍新的`-mreserve` 选项的部分，该选项用于防止链接器填充存储区
- 增加了介绍`-mno-pa-on-file` 和`-mno-pa-on-function` 选项以及新的`-mno-pa-outline-calls` 选项的部分，这些选项全部用于控制过程抽象优化

- 增加了介绍-f[no-]fat-lto-objects、-flto-partition、-fomit-frame-pointer 和-funroll-[all-]loops 优化选项的部分
- 增加了映射的链接器选项-Wl,--section-start
- 增加了缺少的-MF、-P 和-U 驱动程序选项部分。
- 阐明了有关 const-in-program-memory 功能的信息
- 增加了介绍支持的内置函数的部分
- 调整了如何使用-D 选项（已在编译器中更改）传递加双引号字符串的说明
- 提及了代码覆盖功能所需的分析工具套件许可证（取代代码覆盖许可证）
- 增加了__CODECOV 预处理器宏并指示何时对其进行定义
- 阐明了__nopa 说明符只有在使能 CCI 时可用
- 增加了讨论链接描述文件符号的部分

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 https://www.microchip.com/en-us/support/design-help/client-supportservices。

商标

ISBN： 978-1-6683-1894-2

质量管理体系

有关 Microchip 的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 **亚特兰大** 德卢斯，佐治亚州 电话：678-957-9614 传真：678-957-1455 **奥斯汀，德克萨斯州** 电话：512-257-3370 **波士顿** 韦斯特伯鲁，马萨诸塞州 电话：774-760-0087 传真：774-760-0088 **芝加哥** 艾塔斯卡，伊利诺伊州 电话：630-285-0071 传真：630-285-0075 **达拉斯** 阿迪森，德克萨斯州 电话：972-818-7423 传真：972-818-2924 **底特律** 诺维，密歇根州 电话：248-848-4000 **休斯顿，德克萨斯州** 电话：281-894-5983 **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 **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 **罗利，北卡罗来纳州** 电话：919-844-7510 **纽约，纽约州** 电话：631-435-6000 **圣何塞，加利福尼亚州** 电话：408-735-9110 电话：408-436-4270 **加拿大 - 多伦多** 电话：905-695-1980 传真：905-695-2078	**澳大利亚 - 悉尼** 电话：61-2-9868-6733 **中国 - 北京** 电话：86-10-8569-7000 **中国 - 成都** 电话：86-28-8665-5511 **中国 - 重庆** 电话：86-23-8980-9588 **中国 - 东莞** 电话：86-769-8702-9880 **中国 - 广州** 电话：86-20-8755-8029 **中国 - 杭州** 电话：86-571-8792-8115 **中国 - 香港特别行政区** 电话：852-2943-5100 **中国 - 南京** 电话：86-25-8473-2460 **中国 - 青岛** 电话：86-532-8502-7355 **中国 - 上海** 电话：86-21-3326-8000 **中国 - 沈阳** 电话：86-24-2334-2829 **中国 - 深圳** 电话：86-755-8864-2200 **中国 - 苏州** 电话：86-186-6233-1526 **中国 - 武汉** 电话：86-27-5980-5300 **中国 - 西安** 电话：86-29-8833-7252 **中国 - 厦门** 电话：86-592-2388138 **中国 - 珠海** 电话：86-756-3210040	**印度 - 班加罗尔** 电话：91-80-3090-4444 **印度 - 新德里** 电话：91-11-4160-8631 **印度 - 浦那** 电话：91-20-4121-0141 **日本 - 大阪** 电话：81-6-6152-7160 **日本 - 东京** 电话：81-3-6880-3770 **韩国 - 大邱** 电话：82-53-744-4301 **韩国 - 首尔** 电话：82-2-554-7200 **马来西亚 - 吉隆坡** 电话：60-3-7651-7906 **马来西亚 - 槟榔屿** 电话：60-4-227-8870 **菲律宾 - 马尼拉** 电话：63-2-634-9065 **新加坡** 电话：65-6334-8870 **台湾地区 - 新竹** 电话：886-3-577-8366 **台湾地区 - 高雄** 电话：886-7-213-7830 **台湾地区 - 台北** 电话：886-2-2508-8600 **泰国 - 曼谷** 电话：66-2-694-1351 **越南 - 胡志明市** 电话：84-28-5448-2100	**奥地利 - 韦尔斯** 电话：43-7242-2244-39 传真：43-7242-2244-393 **丹麦 - 哥本哈根** 电话：45-4485-5910 传真：45-4485-2829 **芬兰 - 埃斯波** 电话：358-9-4520-820 **法国 - 巴黎** 电话：33-1-69-53-63-20 传真：33-1-69-30-90-79 **德国 - 加兴** 电话：49-8931-9700 **德国 - 哈恩** 电话：49-2129-3766400 **德国 - 海尔布隆** 电话：49-7131-72400 **德国 - 卡尔斯鲁厄** 电话：49-721-625370 **德国 - 慕尼黑** 电话：49-89-627-144-0 传真：49-89-627-144-44 **德国 - 罗森海姆** 电话：49-8031-354-560 **以色列 - 若那那市** 电话：972-9-744-7705 **意大利 - 米兰** 电话：39-0331-742611 传真：39-0331-466781 **意大利 - 帕多瓦** 电话：39-049-7625286 **荷兰 - 德卢内市** 电话：31-416-690399 传真：31-416-690340 **挪威 - 特隆赫姆** 电话：47-72884388 **波兰 - 华沙** 电话：48-22-3325737 **罗马尼亚 - 布加勒斯特** 电话：40-21-407-87-50 **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