TMS320C28x 优化 C/C++ 编译器 v22.6.0.LTS

User's Guide

Literature Number: ZHCU876Z
JULY 2001 – REVISED OCTOBER 2023

目录

插图清单

表格清单

This page intentionally left blank.

关于本手册

TMS320C28x 优化 C/C++ 编译器用户指南 说明如何使用下列德州仪器 (TI) 代码生成编译器工具：

- 编译器
- 链接后优化器
- 库构建实用程序
- C++ 名称还原器

TI 编译器支持符合国际标准化组织 (ISO) 这些语言标准的 C 和 C++ 代码。编译器支持 1989、1999 和 2011 版本的 C 语言以及 2003 版本的 C++语言。

本用户指南讨论了 TI C/C++ 编译器的特性。本手册假设您已了解如何编写 C/C++ 程序。由 Brian W. Kernighan 和 Dennis M. Ritchie 所著的 *C 程序设计语言* (第二版) 介绍了基于 ISO C 标准的 C 语言。您可以使用 Kernighan 和 Ritchie (以下简称为 K&R) 一书作为本手册的补充。本手册中对 K&R C (相对于 ISO C) 的引用是指由 Kernighan 和 Ritchie 所著的 *C 程序设计语言* 第一版中定义的 C 语言。

标记规则

本文档使用以下规则：

- 程序列表、程序示例和交互式显示用特殊字体显示。交互式显示采用粗体形式的特殊字体来区分输入的命令与系统显示的项目（如提示符、命令输出、错误消息等）。C 代码示例如下所示：

```
#include <stdio.h>
main()
{    printf("Hello World\n");
}
```

- 在语法描述中，指令、命令和说明为**粗体**，参数为*斜体*。语法中粗体显示的部分应按所示方式输入；语法中斜体显示的部分描述了应输入信息的类型。
- 方括号（[和]）用于标识可选参数。如果使用可选参数，需要在括号内指定信息。除非方括号是**粗体**，否则不要输入方括号本身。下面是一个具有可选参数的命令的示例：

cl2000 [*options*] [*filenames*] [**--run_linker** [*link_options*] [*object files*]]

- 大括号（{ 和 }）表明必须选择大括号内的参数之一，不要输入大括号本身。这是一个带有大括号的命令的示例，大括号并不包含在实际语法中，但表明您必须指定 --rom_model 或 --ram_model 选项：

cl2000 --run_linker {**--rom_model** | **--ram_model**} *filenames*

[**--output_file=** *name.out*] **--library=** *libraryname*

- 在汇编器语法语句中，最左侧列被预留给标签或符号的第一个字符。如果标签或符号是可选的，则通常不会显示。如果标签或符号是必需参数，则从框的左边距开始显示，如下例所示。除了符号或标签外，任何指令、命令、说明或参数都不能从最左侧列开始。

symbol **.usect** "*section name*", *size in bytes*[, *alignment*]

- 有些指令的参数数量可变。例如，.byte 指令。此语法显示为 [, ..., *parameter*]。
- TMS320C2800™ 内核被称为 TMS320C28x 或 C28x。

相关文档

以下书籍可以作为本用户指南的补充：

ANSI X3.159-1989, Programming Language - C (Alternate version of the 1989 C Standard), American National Standards Institute

ISO/IEC 9899:1989, International Standard - Programming Languages - C (The 1989 C Standard), International Organization for Standardization

ISO/IEC 9899:1999, International Standard - Programming Languages - C (The 1999 C Standard), International Organization for Standardization

ISO/IEC 9899:2011, International Standard - Programming Languages - C (The 2011 C Standard), International Organization for Standardization

ISO/IEC 14882-2003, International Standard - Programming Languages - C++ (The 2003 C++ Standard), International Organization for Standardization

The C Programming Language (second edition), by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Englewood Cliffs, New Jersey, 1988

The Annotated C++ Reference Manual, Margaret A. Ellis and Bjarne Stroustrup, published by Addison-Wesley Publishing Company, Reading, Massachusetts, 1990

C：A Reference Manual (fourth edition), by Samuel P. Harbison, and Guy L. Steele Jr., published by Prentice Hall, Englewood Cliffs, New Jersey

Programming Embedded Systems in C and C++, by Michael Barr, Andy Oram (Editor), published by O'Reilly & Associates; ISBN: 1565923545, February 1999

Programming in C, Steve G. Kochan, Hayden Book Company

The C++ Programming Language (second edition), Bjarne Stroustrup, published by Addison-Wesley Publishing Company, Reading, Massachusetts, 1990

Tool Interface Standards (TIS) DWARF Debugging Information Format Specification Version 2.0, TIS Committee, 1995

DWARF Debugging Information Format Version 3, DWARF Debugging Information Format Workgroup, Free Standards Group, 2005 (http://dwarfstd.org)

DWARF Debugging Information Format Version 4, DWARF Debugging Information Format Workgroup, Free Standards Group, 2010 (http://dwarfstd.org)

System V ABI specification (http://www.sco.com/developers/gabi/)

德州仪器 (TI) 提供的相关文档

有关 TI 代码生成工具的更多信息，请参阅以下资源：

- Code Composer Studio 文档概述
- 德州仪器 (TI) E2E 软件工具论坛

以下文档可作为对本用户指南的补充：

SPRAAB5 ***DWARF 对 TI 目标文件的影响。*** 介绍了德州仪器 (TI) 对 DWARF 规范的扩展。

SPRU513 ***TMS320C28x 汇编语言工具用户指南*** 介绍了用于 TMS320C28x 器件的汇编语言工具（用于开发汇编语言代码的汇编器和其他工具）、汇编器指令、宏命令、通用目标文件格式和符号调试指令。

SPRU430 ***TMS320C28x DSP CPU 和指令集参考指南*** 介绍了 TMS320C28x 定点数字信号处理器 (DSP) 的中央处理器 (CPU) 和汇编语言指令。它还描述了这些 DSP 上可用的仿真特性。

SPRU566 ***TMS320x28xx 外设参考指南*** 介绍了 28x 数字信号处理器 (DSP) 的外设参考指南。

SPRUHS1 ***TMS320C28x 扩展指令集技术参考手册*** 介绍了 TMU、VCRC、VCU-II、FPU32 和 FPU64 加速器的架构、流水线和指令集。

SPRAC71 ***TMS320C28x 嵌入式应用二进制接口 (EABI) 应用报告***。为德州仪器 (TI) 的 TMS320C28x 系列处理器提供基于 ELF 的嵌入式应用二进制接口 (EABI) 的规范。EABI 定义了程序、程序组件和执行环境（如果存在操作系统，还包括操作系统）之间的低级别接口。

SPRUEX3 ***TI SYS/BIOS 实时操作系统用户指南。***SYS/BIOS 使应用开发人员能够开发嵌入式实时软件。SYS/BIOS 是一个可扩展的实时内核。它适用于需要实时调度和同步或实时检测的应用。SYS/BIOS 提供抢占式多线程、硬件抽象、实时分析和配置工具。

商标

TMS320C2800™, TMS320C28x™, and Code Composer Studio™ are trademarks of Texas Instruments.
所有商标均为其各自所有者的财产。

章节 1

软件开发工具简介

TMS320C28x™ 由一套软件开发工具支持，其中包括优化 C/C++ 编译器、汇编器、链接器以及各种实用程序。

本章概述了这些工具，并介绍了优化 C/C++ 编译器的特性。在 *TMS320C28x 汇编语言工具用户指南* 中详细论述了汇编器和链接器。

1.1 软件开发工具概述

图 1-1 阐述软件开发流程。图中阴影部分突出了 C 语言程序最常见的软件开发路径。其他部分是增强开发流程的外围功能。

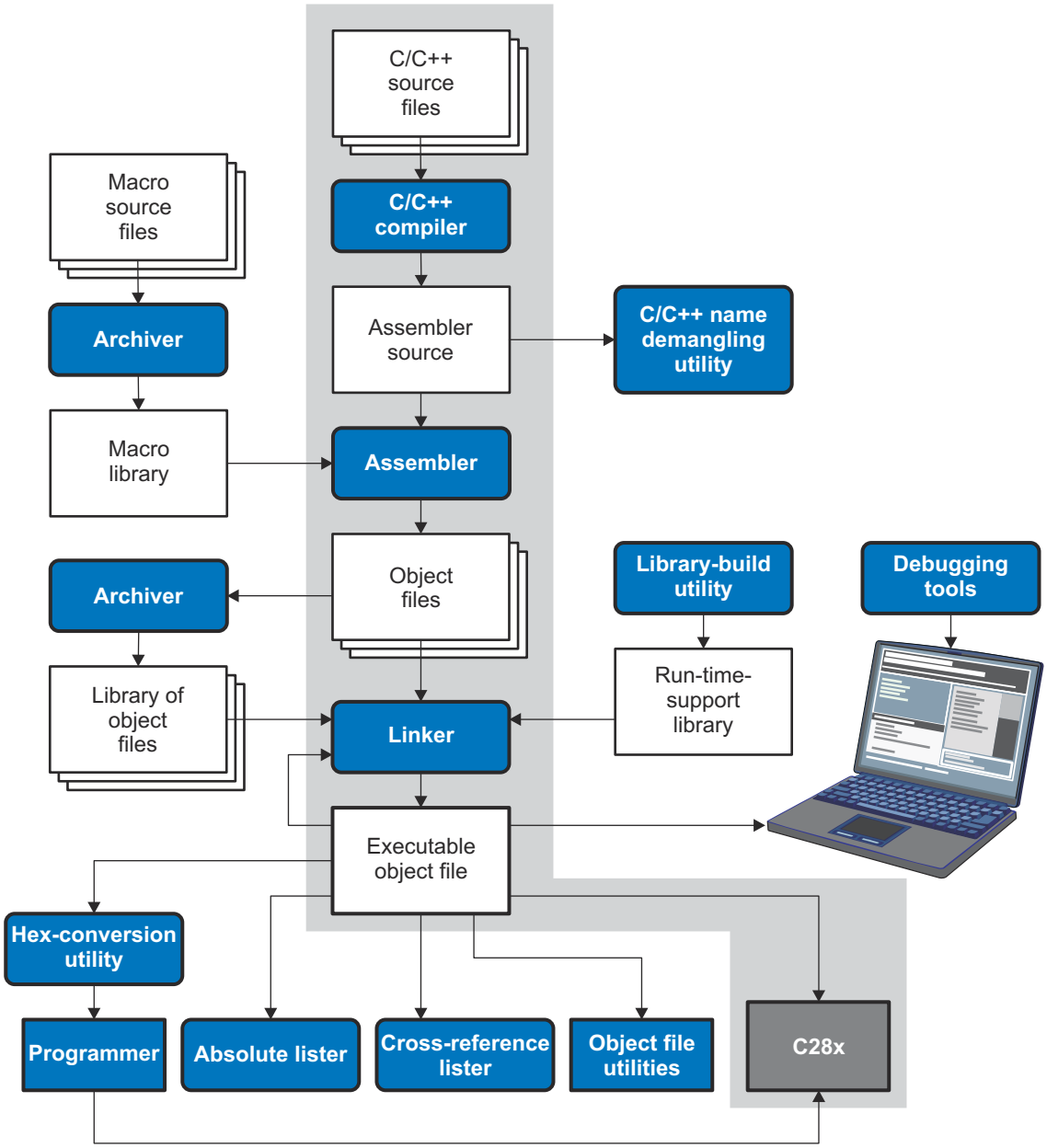


图 1-1. TMS320C28x 软件开发流程

以下列表描述了图 1-1 中显示的工具：

- **编译器**接受 C/C++ 源代码，生成 C28x 汇编语言源代码。请参阅章节 2。
- **汇编器**将汇编语言源文件转换成机器语言可重定位的目标文件。请参阅 *TMS320C28x 汇编语言工具用户指南*。
- **链接器**将可重定位的目标文件组合成单个绝对可执行的目标文件。在创建可执行文件时，会执行重定位并解析外部引用。链接器接受可重定位的目标文件和对象库作为输入。有关链接器的概览信息，请参阅章节 4。请参阅 *TMS320C28x 汇编语言工具用户指南*，了解详细信息。

- **归档器**允许将一组文件收集到一个称为*库*的单个存档文件中。归档器允许通过删除、替换、提取或添加成员来修改这种库。归档器最有用的应用之一是构建目标文件库。请参阅 *TMS320C28x 汇编语言工具用户指南*。
- **运行时支持库**包含编译器支持的标准 ISO C 和 C++ 库函数、编译器实用程序函数、浮点算术函数和 C I/O 函数。请参阅章节 8。

  如果编译器和链接器选项需要自定义的库版本，**库构建实用程序**将自动构建运行时支持库。请参阅节 8.5。C 和 C++ 的标准运行时支持库函数的源代码位于编译器安装目录的 lib\src 子目录中提供。
- **十六进制转换实用程序**将目标文件转换为其他目标格式。可将转换后的文件下载到 EPROM 编程器。请参阅 *TMS320C28x 汇编语言工具用户指南*。
- **绝对列表器**接受链接的目标文件作为输入并创建 .abs 文件作为输出。可以组合这些 .abs 文件以生成包含绝对地址而不是相对地址的列表。如果没有绝对列表器，生成这样的列表将是冗长而乏味的，并且需要许多手动操作。请参阅 *TMS320C28x 汇编语言工具用户指南*。
- **交叉引用列表器**使用目标文件生成交叉引用列表，其中显示符号、其定义以及其在链接的源文件中的引用。请参阅 *TMS320C28x 汇编语言工具用户指南*。
- **C++ 名称还原器**是一种调试辅助工具，可将编译器改编的名称转换回其在 C++ 源代码中声明的原始名称。如图 1-1 所示，可对编译器输出的汇编文件使用 C++ 名称还原器；还可对汇编器列表文件和链接器映射文件使用此实用程序。请参阅章节 9。
- **链接后优化器**删除或修改汇编语言指令以生成更好的代码。必须使用编译器 -plink 选项运行链接后优化器。请参阅章节 5。
- **反汇编器**解码目标文件以显示它们所表示的汇编指令。请参阅 *TMS320C28x 汇编语言工具用户指南*。
- 此开发流程的主要产品是可执行的目标文件，其可以在 **TMS320C28x** 器件上执行。

1.2 编译器接口

编译器是名为 cl2000 的命令行程序。此程序可以一步编译、优化、汇编和链接程序。在 Code Composer Studio ™ 中，编译器自动运行以执行构建项目所需的步骤。

更多有关程序编译的信息，请参阅节 2.1。

编译器具有直接调用约定，因此可以编写相互调用的汇编和 C 函数。更多有关调用约定的信息，请参阅章节 7。

1.3 ANSI/ISO 标准

编译器支持 1989、1999 和 2011 版本的 C 语言以及 2003 版本的 C++ 语言。编译器中的 C 和 C++ 语言特征是按照下述 ISO 标准实现的：

- **ISO 标准 C：**C 编译器支持 989、1999 和 2011 版本的 C 语言。
  - **C89。**使用 --c89 选项编译会使编译器符合 ISO/IEC 9899:1990 C 标准，该标准先前被批准为 ANSI X3.159-1989。“C89”和“C90”指的是同一种编程语言。本文档中使用了“C89”。
  - **C99。**使用 --c99 选项编译会使编译器符合 ISO/IEC 9899:1999 C 标准。
  - **C11。**使用 --c11 选项编译会使编译器符合 ISO/IEC 9899:2011 C 标准。

  Kernighan 和 Ritchie 的《*C 程序设计语言*》(K&R) 第二版中也介绍了 C 语言。
- **ISO 标准 C++：**编译器使用 C++03 版本的 C++ 标准。请参阅 C++ 标准 ISO/IEC 14882:2003。 Ellis 和 Stroustrup 的《*C++ 参考手册注解*》(ARM) 中也介绍了该语言，但该语言不是标准语言。有关*不受支持的* C++ 特性的说明，请参阅节 6.2。
- **ISO 标准运行时支持：**编译器工具附带一个扩展的运行时库。除非另有说明，否则库函数符合 ISO C/C++ 库标准。该库包括标准输入和输出函数、字符串操作函数、动态内存分配函数、数据转换函数、计时函数、三角函数以及指数和双曲线函数。不包括信号处理函数，因为这些函数是特定于目标系统的。如需更多信息，请参阅 章节 8。

如需了解命令行选项以选择代码所使用的 C 或 C++ 标准，请参阅 节 6.14 。

1.4 输出文件

以下类型的输出文件由编译器创建：

- **COFF 目标文件。**通用目标文件格式 (COFF) 提供基本的模块化（单独编译）编译功能，例如重定位。
- **ELF 目标文件。**可执行和可链接格式 (ELF) 支持早期模板实例化和内联函数导出等现代语言功能。ELF 是 System V 应用程序二进制接口 (ABI) 的一部分。用于 C28x 的 ELF 格式由 C28x 嵌入式应用程序二进制接口 (EABI) 扩展，相关信息请参阅 SPRAC71 文档。

1.5 实用程序

这些功能是编译器实用程序：

- **库构建实用程序**

  库构建实用程序允许您从源代码中为运行时模型的任何组合自定义构建对象库。有关更多信息，请参阅 节 8.5。
- **C++ 名称还原器**

  C++ 名称还原器 (dem2000) 是一种调试辅助工具，可将在编译器生成的汇编代码、反汇编输出或编译器诊断消息中检测到的每个已改编的名称转换为在 C++ 源代码中找到的原始名称。有关更多信息，请参阅 章节 9。
- **十六进制转换实用程序**

  对于独立的嵌入式应用程序，编译器能够将所有代码和初始化数据放入 ROM 中，从而允许 C/C++ 代码从复位开始运行。编译器输出的 COFF 或 ELF 文件可以使用十六进制转换实用程序转换为 EPROM 编程器数据文件，如《*TMS320C28x 汇编语言工具用户指南*》所述。

章节 2

使用 C/C++ 编译器

编译器将您的源程序转换成 TMS320C28x 可执行的机器语言目标代码。源代码必须经过编译、汇编和链接才能创建可执行文件。所有这些步骤都是通过使用编译器一次性执行的。

2.1 关于编译器

编译器可一步完成编译、优化、汇编和选择性链接。编译器在一个或多个源代码模块上执行以下步骤：

- **编译器**接受 C/C++ 源代码和汇编代码。编译器会生成目标代码。

  可以在单命令中编译 C、C++ 和汇编文件。编译器使用文件扩展名来区分不同的文件类型。有关更多信息，请参阅节 2.3.9。
- **链接器**会组合目标文件以创建可执行或可链接文件。链接步骤是可选的，因此您可以独立编译和汇编许多模块，然后再链接这些模块。有关如何链接文件，请参阅章节 4。

备注

调用链接器

默认情况下，编译器不会调用链接器。可以使用 --run_linker (-z) 编译器选项调用链接器。有关详细信息，请参阅节 4.1.1。

有关汇编器和链接器的完整说明，请参阅《*TMS320C28x 汇编语言工具用户指南*》。

2.2 调用 C/C++ 编译器

要调用编译器，请输入：

cl2000 [*options*] [*filenames*] [**--run_linker** [*link_options*] *object files*]]

cl2000	用于运行编译器和汇编器的命令。
options	影响编译器对输入文件的处理方式的选项。表 2-6 到表 2-28 列出了这些选项。
filenames	一个或多个 C/C++ 源文件和 汇编语言源文件。
--run_linker (**-z**)	调用链接器的选项。--run_linker 选项的缩写形式为 -z。有关更多信息，请参阅章节 4。
link_options	控制链接过程的选项。
object files	链接过程的目标文件的名称。

编译器的参数分为三种类型：

- 编译器选项
- 链接选项
- 文件名

--run_linker 选项指示待执行的链接。如果使用 --run_linker 选项，则任何编译器选项都必须位于 --run_linker 选项之前，并且所有链接选项都必须位于 --run_linker 选项之后。

源代码文件名必须位于 --run_linker 选项之前。其他目标文件的文件名可以放置在 --run_linker 选项之后。

例如，如果要编译两个名为 symtab.c 和 file.c 的文件，则汇编第三个名为 seek.asm 的文件，并通过链接创建一个名为 myprogram.out 的可执行程序，则需要输入：

```
cl2000 symtab.c file.c seek.asm --run_linker --library=lnk.cmd
       --output_file=myprogram.out
```

2.3 使用选项更改编译器的行为

选项控制编译器的运行。本部分说明选项约定和选项摘要表。此外，还提供常用选项（包括用于类型检查和汇编的选项）的详细说明。

如需查看选项的帮助屏幕摘要，请在命令行上输入不带参数的 **cl2000**。

下述原则适用于编译器选项：

- 通常有两种方法来指定给定的选项。“长格式”使用两个连字符前缀，通常是更具描述性的名称。“短格式”使用单个连字符前缀以及并不总是直观的字母与数字的组合。
- 选项通常区分大小写。
- 单个选项不能组合。
- 带参数的选项应在参数前用等号指定，以清楚地将参数与选项关联起来。例如，用于取消定义常量的选项可以表示为 --undefine=*name*。同样，用于指定最大优化量的选项可以表示为 -O=3。还可以在某些选项后直接指定参数，例如 -O3 与 -O=3 相同。选项与可选参数之间不允许有空格，因此不接受 -O 3。
- 文件和除 --run_linker 选项外的选项可以按任何顺序出现。--run_linker 选项必须跟在所有编译器选项之后且在任何链接器选项之前。

可以使用 C2000_C_OPTION 环境变量为编译器定义默认选项。有关环境变量的详细说明，请参阅节 2.4.1。

表 2-1 到表 2-28 汇总了所有选项（包括链接选项）。使用表中的参考资料以获取更完整的选项说明。

表 2-1. 处理器选项

选项	别名	效果	段
--silicon_version=28	-v28	指定 TMS320C28x 架构。默认值（也是唯一接受的值）为 28。不再需要此选项。	节 2.3.4
--abi={coffabi|eabi}		选择应用程序二进制接口。默认为 coffabi。也为 eabi 提供支持。	节 2.3.4
--cla_support[=cla0|cla1|cla2]		为类型 0、类型 1 或类型 2 指定 TMS320C28x CLA 加速器支持。默认为 cla0。仅当目标硬件提供这一功能时才使用此选项。	节 2.3.4
--float_support={ fpu32 | fpu64 | softlib }		指定使用 TMS320C28x 32 位或 64 位硬件浮点支持。默认为 softlib。仅当目标硬件提供这一功能时才使用此选项。	节 2.3.4
--idiv_support={none | idiv0}		使用硬件扩展来支持快速整数除法，以提供一组指令来加速 16 位、32 位和 64 位值的整数除法。如果此硬件可用，请使用 --idiv_support=idiv0 来使用这些指令。仅当目标硬件提供这一功能时才使用此选项。默认为 none。（仅限 EABI）	节 2.3.4
--lfu_reference_elf=*path*	-lfu=*path*	为了创建与实时固件更新 (LFU) 兼容的二进制可执行文件，请指定前一个 ELF 二进制可执行文件的路径以用作参考，从中获取全局和静态符号的内存地址列表。这个先前的二进制文件可以是兼容 LFU 的二进制文件，但这不是必需的。（LFU 仅支持 EABI）	节 2.15
-lfu_default[=none | preserve]		如果全局和静态符号地址在新的可执行文件中没有更新属性或保留属性，则指定参考 ELF 可执行文件中的全局和静态符号地址的默认处理方式。在实时固件更新（“热启动”）期间使用这些处理方式。 如果 --lfu_default=preserve（默认值），则编译器保留在参考 ELF 可执行文件中找到的所有全局和静态符号地址，除非为符号指定了 __attribute__((update))。 如果 --lfu_default=none，则编译器只保留指定了 __attribute__((preserve)) 的符号的地址。重新初始化指定了 __attribute__((update)) 的符号。链接器可以将所有其他全局和静态符号分配到任何内存地址，但不会在发生热启动时重新初始化这些符号。 （LFU 仅支持 EABI）	节 2.15
--silicon_errata_fpu1_workaround =on|off		启用此选项可防止在某些指令期间可能发生的 FPU 寄存器写入冲突。编译器会在这类指令之前添加 NOP 指令以防止冲突。仅当目标硬件提供 FPU 功能时才使用此选项。	节 2.3.4

表 2-1. 处理器选项（续）

选项	别名	效果	段
--tmu_support[=tmu0 | tmu1]		支持三角数学单元 (TMU)。使用此选项还会支持 FPU32（与 --float_support=fpu32 一样）。如果使用此选项但未指定值，则默认值为 tmu0。tmu1 选项支持所有 tmu0 功能以及 LOG2F32 和 IEXP2F32 指令。仅当目标硬件提供这一功能时才使用此选项。（TMU1 仅支持 EABI）	节 2.3.4
--vcu_support[=vcu0|vcu2|vcrc]		指定 C28x VCU 协处理器支持类型 0、类型 2 或 VCRC。仅当目标硬件提供这一功能时才使用此选项。默认为 vcu0。	节 2.3.4
--unified_memory	-mt	为统一内存模型生成代码。	节 2.3.4

表 2-2. 优化选项(1)

选项	别名	效果	段
--opt_level=off		禁用所有优化（默认值）。	节 3.1
--opt_level=*n*	-O*n*	级别 0 (-O0) 仅优化寄存器使用情况。 级别 1 (-O1) 使用级别 0 优化并在本地进行优化。 级别 2 (-O2) 使用级别 1 优化并在本地进行优化。 级别 3 (-O3) 使用级别 2 优化并对文件进行优化。 级别 4 (-O4) 使用级别 3 优化并执行链接时间优化。	节 3.1、节 3.3、节 3.6
--opt_for_space=*n*	-ms	在四个级别（0、1、2 和 3）上控制代码大小。	节 3.13
--opt_for_speed[=*n*]	-mf	控制大小和速度之间的权衡（0-5 范围）。如果此选项未指定 *n*，则默认值为 4。如果未指定此选项，则默认设置为 2。	节 3.2

(1) **注意：**机器专用选项（参阅表 2-12）也会影响优化。

表 2-3. 高级优化选项(1)

选项	别名	效果	段
--auto_inline=[*size*]	-oi	设置自动内联大小（仅限 --opt_level=3）。如果未指定 *size*，则默认值为 1。	节 3.5
--call_assumptions=*n*	-op*n*	级别 0 (-op0) 指定了模块包含从提供给编译器的源代码外部调用或修改的函数和变量。 级别 1 (-op1) 指定了模块包含从提供给编译器的源代码外部修改的变量，但不使用从源代码外部调用的函数。 级别 2 (-op2) 指定了模块不包含从提供给编译器的源代码外部调用或修改的函数或变量（默认值）。 级别 3 (-op3) 指定了模块包含从提供给编译器的源代码外部调用的函数，但不使用从源代码外部修改的变量。	节 3.4.1
--disable_inlining		防止发生任何内联。	节 2.11
--fp_mode={relaxed|strict}		启用或禁用宽松浮点模式。	节 2.3.3
--fp_reassoc={on|off}		启用或禁用浮点算术的重新关联。	节 2.3.3
--fp_single_precision_constant		使所有未添加后缀的浮点常量都被视为单精度值（而非双精度常量）。	节 2.3.3
--gen_opt_info=*n*	-on*n*	级别 0 (-on0) 禁用优化信息文件。 级别 1 (-on2) 生成优化信息文件。 级别 2 (-on2) 生成详细的优化信息文件。	节 3.3.1
--isr_save_vcu_regs={on|off}		为中断例程生成 VCU 寄存器保存/恢复到栈，以便 VCU 代码可以重入。	节 3.14
--optimizer_interlist	-os	交叉列出优化器注释与汇编语句。	节 3.10
--program_level_compile	-pm	组合源文件以执行程序级优化。	节 3.4
--sat_reassoc={on|off}		启用或禁用饱和算术的重新关联。默认为 --sat_reassoc=off。	节 2.3.3
--aliased_variables	-ma	通知编译器传递给函数的地址可能会由被调用函数中的别名修改。	节 3.9.2.2

(1) **注意：**机器专用选项（参阅表 2-12）也会影响优化。

表 2-4. 调试选项

选项	别名	效果	段
--symdebug:dwarf	-g	默认行为。启用符号调试。调试信息的生成不会影响优化。因此，默认情况下会生成调试信息。	节 2.3.5 节 3.12
--symdebug:dwarf_version=2|3|4		指定 DWARF 格式版本。COFF ABI 的默认版本为 3，EABI 的默认版本为 4。	节 2.3.5
--symdebug:none		禁用所有符号调试。	节 2.3.5 节 3.12
--symdebug:profile_coff		使用备用 STABS 调试格式启用分析。仅 COFF ABI 支持 STABS。	节 2.3.5
--symdebug:skeletal		（已弃用；无效。）	

表 2-5. Include 选项

选项	别名	效果	段
--include_path=*directory*	-I	将指定的目录添加到 #include 搜索路径。	节 2.5.2.1
--preinclude=*filename*		在编译开始时包含 *filename*。	节 2.3.3

表 2-6. 控制选项

选项	别名	效果	段
--compile_only	-c	禁用链接（否定 --run_linker）。	节 4.1.3
--help	-h	打印（在标准输出设备上）编译器理解的选项的说明。	节 2.3.2
--run_linker	-z	导致从编译器命令行调用链接器。	节 2.3.2
--skip_assembler	-n	编译 C/C++ 源文件，从而生成汇编语言输出文件。汇编器不会运行，也不会生成目标文件。	节 2.3.2

表 2-7. 语言选项

选项	别名	效果	段
--c89		根据 ISO C89 标准处理 C 文件。	节 6.14
--c99		根据 ISO C99 标准处理 C 文件。	节 6.14
--c11		根据 ISO C11 标准处理 C 文件。	节 6.14
--c++03		根据 ISO C++03 标准处理 C++ 文件。	节 6.14
--cla_background_task ={on|off}		启用或禁用 CLA 后台任务。	节 10.1.1
--cla_default		将 .c 和 .cla 文件均作为 CLA 文件处理。	节 10.1.1
--cla_signed_compare_workaround ={on|off}		如果使用整数比较可能会导致错误的答案，故在编译 CLA 比较时自动使用浮点比较功能。默认为 off。	节 10.1.1
--cpp_default	-fg	将所有带有 C 扩展名的源文件作为 C++ 源文件处理。	节 2.3.7
--exceptions		启用 C++ 异常处理。	节 6.6
--extern_c_can_throw		允许外部 C 函数传播异常。（仅限 EABI）	--
--float_operations_allowed ={none|all|32|64}		限制允许的浮点运算类型。	节 2.3.3
--gen_cross_reference_listing	-px	生成交叉引用列表文件 (.crl)。	节 2.9
--pending_instantiations=#		指定在任何给定时间内可能正在进行的模板实例化的数量。使用 0 指定一个不受限制的数字。	节 2.3.4
--printf_support={nofloat|full|minimal}		支持更小、有限版本的 printf 函数系列（sprintf、fprintf 等）和 scanf 函数系列（sscanf、fscanf 等）运行时支持函数。	节 2.3.3
--relaxed_ansi	-pr	启用宽松模式；忽略严格的 ISO 违规。默认设置为 on。要禁用此模式，请使用 --strict_ansi 选项。	节 6.14.3
--rtti	-rtti	启用 C++ 运行时类型信息 (RTTI)。	--

表 2-7. 语言选项（续）

选项	别名	效果	段
--strict_ansi	-ps	启用严格的 ANSI/ISO 模式（适用于 C/C++，不适用于 K&R C）。在此模式下，禁用与 ANSI/ISO C/C++ 冲突的语言扩展。在严格的 ANSI/ISO 模式下，大多数 ANSI/ISO 违规都会报告为错误。被视为酌情处理的违规行为可能会报告为警告。	节 6.14.3

表 2-8. 解析器预处理选项

选项	别名	效果	段
--preproc_dependency[=*filename*]	-ppd	仅执行预处理，但不写入预处理的输出，而是写入适合于输入到标准 make 实用程序的依赖行列表。	节 2.5.8
--preproc_includes[=*filename*]	-ppi	仅执行预处理，但不写入预处理的输出，而是写入 #include 指令中包含的文件列表。	节 2.5.9
--preproc_macros[=*filename*]	-ppm	仅执行预处理。将预定义和用户定义的宏列表写入与输入同名但扩展名为 .pp 的文件。	节 2.5.10
--preproc_only	-ppo	仅执行预处理。将预处理的输出写入与输入同名但扩展名为 .pp 的文件。	节 2.5.4
--preproc_with_comment	-ppc	仅执行预处理。将预处理的输出（保留注释）写入与输入同名但扩展名为 .pp 的文件。	节 2.5.6
--preproc_with_compile	-ppa	使用任何通常会禁用编译的 -pp<x> 选项在预处理后继续编译。	节 2.5.5
--preproc_with_line	-ppl	仅执行预处理。将带有行控制信息（#line 指令）的预处理的输出写入与输入同名但扩展名为 .pp 的文件。	节 2.5.7

表 2-9. 预定义宏选项

选项	别名	效果	段
--define=*name*[=*def*]	-D	预定义 *name*。	节 2.3.2
--undefine=*name*	-U	未定义 *name*。	节 2.3.2

表 2-10. 诊断消息选项

选项	别名	效果	段
--advice:performance[=all,none]		为性能提高方法提供建议。默认为 all。	节 2.3.3
--compiler_revision		打印出编译器发布版本并退出。	--
--diag_error=*num*	-pdse	将 *num* 标识的诊断分类为错误。	节 2.7.1
--diag_remark=*num*	-pdsr	将 *num* 标识的诊断分类为备注。	节 2.7.1
--diag_suppress=*num*	-pds	抑制 *num* 标识的诊断。	节 2.7.1
--diag_warning=*num*	-pdsw	将 *num* 标识的诊断分类为警告。	节 2.7.1
--diag_wrap={on|off}		使诊断消息换行（默认为 on）。请注意，此命令行选项不能在 Code Composer Studio IDE 中使用。	
--display_error_number	-pden	显示诊断的标识符及其文本。请注意，此命令行选项不能在 Code Composer Studio IDE 中使用。	节 2.7.1
--emit_warnings_as_errors	-pdew	将警告视为错误。	节 2.7.1
--issue_remarks	-pdr	发出备注（非严重警告）。	节 2.7.1
--no_warnings	-pdw	抑制诊断警告（仍会发出错误）。	节 2.7.1
--quiet	-q	抑制进度消息（静默）。	--
--set_error_limit=*num*	-pdel	将错误限制设置为 *num*。在在错误达到这个数量后，编译器放弃编译。（默认为 100。）	节 2.7.1
--super_quiet	-qq	超级静默模式。	--
--tool_version	-version	显示每个工具的版本号。	--
--verbose		显示横幅和函数进度信息。	--
--verbose_diagnostics	-pdv	提供详细的诊断消息，以自动换行的方式显示原始源代码。请注意，此命令行选项不能在 Code Composer Studio IDE 中使用。	节 2.7.1

表 2-10. 诊断消息选项（续）

选项	别名	效果	段
--write_diagnostics_file	-pdf	生成诊断消息信息文件。编译器唯一选项。请注意，此命令行选项不能在 Code Composer Studio IDE 中使用。	节 2.7.1

表 2-11. 补充信息选项

选项	别名	效果	段
--gen_preprocessor_listing	-pl	生成原始列表文件 (.rl)。	节 2.10

表 2-12. 运行时模型选项

选项	别名	效果	段
--gen_data_subsections={on|off}		将所有聚合数据（数组、结构和联合体）放入子段中。这使得链接器可以更好地控制在最终链接步骤期间删除未使用的数据。有关默认设置的详细信息，请参阅右侧的链接。	节 4.2.2
--gen_func_subsections={on|off}	-mo	将每个函数放在目标文件的一个单独子段中。如果未使用此选项，则默认为 off。有关默认设置的详细信息，请参阅右侧的链接。	节 4.2.1
--no_rpt	-mi	禁止生成 RPT 指令。	节 2.3.4
--protect_volatile	-mv	启用易失性引用保护。	节 2.3.4
--ramfunc={on|off}		如果设置为 on，则指定所有函数都应放置在位于 RAM 中的 .TI.ramfunc 段中。	节 2.3.4
--rpt_threshold=*k*		生成迭代 *k* 次或更少次数的 RPT 循环。（k 是 0 到 256 之间的常数。）	节 2.3.4

表 2-13. 入口/出口挂钩选项

选项	别名	效果	段
--entry_hook[=*name*]		启用入口挂钩。	节 2.14
--entry_parm={none|name|address}		将函数的参数指定给 --entry_hook 选项。	节 2.14
--exit_hook[=*name*]		启用出口挂钩。	节 2.14
--exit_parm={none|name|address}		将函数的参数指定给 --exit_hook 选项。	节 2.14
--remove_hooks_when_inlining		删除自动内联函数的入口/出口挂钩。	节 2.14

表 2-14. 反馈选项

选项	别名	效果	段
--analyze=codecov		从配置文件数据生成分析信息。	节 3.8.2.2
--analyze_only		仅生成分析。	节 3.8.2.2
--gen_profile_info		生成检测代码以收集配置文件信息。	节 3.7.1.3
--use_profile_info=*file1*[, *file2*,...]		指定配置文件信息文件。	节 3.7.1.3

表 2-15. 汇编器选项

选项	别名	效果	段
--keep_asm	-k	保留汇编语言 (.asm) 文件。	节 2.3.11
--asm_listing	-al	生成汇编列表文件。	节 2.3.11
--c_src_interlist	-ss	交叉列出 C 源代码和汇编语句。	节 2.12 节 3.10
--src_interlist	-s	交叉列出优化器注释（如果可用）和汇编源语句；否则交叉列出 C 语言和汇编源语句。	节 2.3.2
--absolute_listing	-aa	启用绝对列表。	节 2.3.11
--asm_cross_reference_listing	-ax	生成交叉引用文件。	节 2.3.11
--asm_define=*name*[=*def*]	-ad	设置 *name* 符号。	节 2.3.11
--asm_dependency	-apd	执行预处理；仅列出程序集依赖项。	节 2.3.11

表 2-15. 汇编器选项（续）

选项	别名	效果	段
--asm_includes	-api	执行预处理；仅列出包含的 #include 文件。	节 2.3.11
--issue_remarks		发出包含附加汇编时间检查的备注（非严重警告）。	节 2.3.11
--asm_undefine=*name*	-au	不对预定义的常量 *name* 进行定义。	节 2.3.11
--flash_prefetch_warn		汇编器警告 F281X BF 闪存预取问题。	节 2.3.11
--include_file=*filename*	-ahi	包含汇编模块的指定文件。	节 2.3.11
--preproc_asm	-mx	预处理汇编源，扩展汇编宏。	节 2.3.11

表 2-16. 文件类型说明符选项

选项	别名	效果	段
--asm_file=*filename*	-fa	不管其扩展名为何，都将 *filename* 标识为汇编源文件。默认情况下，编译器和汇编器将 .asm 文件视为汇编源文件。	节 2.3.7
--c_file=*filename*	-fc	不管其扩展名为何，都将 *filename* 标识为 C 源文件。默认情况下，编译器将 .c 文件视为 C 源文件。	节 2.3.7
--cpp_file=*filename*	-fp	不管其扩展名为何，都将 *filename* 标识为 C++ 文件。默认情况下，编译器将 .C、.cpp、.cc 和 .cxx 文件视为 C++ 文件。	节 2.3.7
--obj_file=*filename*	-fo	不管其扩展名为何，都将 *filename* 标识为目标代码文件。默认情况下，编译器和链接器将 .obj 文件视为目标代码文件，包括 *.c.obj 和 *.cpp.obj 文件。	节 2.3.7

表 2-17. 目录说明符选项

选项	别名	效果	段
--abs_directory=*directory*	-fb	指定绝对列表文件目录。默认情况下，编译器使用目标文件目录。	节 2.3.10
--asm_directory=*directory*	-fs	指定汇编文件目录。默认情况下，编译器使用当前目录。	节 2.3.10
--list_directory=*directory*	-ff	指定汇编列表文件和交叉引用列表文件目录。默认情况下，编译器使用目标文件目录。	节 2.3.10
--obj_directory=*directory*	-fr	指定目标文件目录。默认情况下，编译器使用当前目录。	节 2.3.10
--output_file=*filename*	-fe	指定编译输出文件名；可以覆盖 --obj_directory。	节 2.3.10
--pp_directory=*dir*		指定预处理器文件目录。默认情况下，编译器使用当前目录。	节 2.3.10
--temp_directory=*directory*	-ft	指定临时文件目录。默认情况下，编译器使用当前目录。	节 2.3.10

表 2-18. 默认文件扩展名选项

选项	别名	效果	段
--asm_extension=[.]*extension*	-ea	设置汇编源文件的默认扩展名。	节 2.3.9
--c_extension=[.]*extension*	-ec	设置 C 源文件的默认扩展名。	节 2.3.9
--cpp_extension=[.]*extension*	-ep	设置 C++ 源文件的默认扩展名。	节 2.3.9
--listing_extension=[.]*extension*	-es	设置列表文件的默认扩展名。	节 2.3.9
--obj_extension=[.]*extension*	-eo	设置目标文件的默认扩展名。	节 2.3.9

表 2-19. 命令文件选项

选项	别名	效果	段
--cmd_file=*filename*	-@	将文件内容解释为命令行的扩展。可以使用多个 -@ 实例。	节 2.3.2

2.3.1 链接器选项

以下各表列出了链接器选项。有关这些选项的详细信息，请参阅本文档的章节 4 以及《*TMS320C28x 汇编语言工具用户指南*》。

表 2-20. 链接器基本选项

选项	别名	说明
--run_linker	-z	启用链接。
--output_file=*file*	-o	为可执行输出文件命名。默认文件名为 .out file。
--map_file=*file*	-m	生成输入和输出段（包括空位）的映射或列表，并将列表放置在 *file* 中。
--stack_size=*size*	[-]-stack	将 C 系统栈大小设为 *size* 字，并定义全局符号来指定栈大小。默认值 = 1K 字。
--heap_size=*size*	[-]-heap	将堆大小（对于 C 中的动态存储器分配）设为 *size* 字，并定义全局符号来指定栈大小。默认值 = 1K 字。
--warn_sections	-w	创建未定义的输出段时显示一条消息。

表 2-21. 文件搜索路径选项

选项	别名	说明
--library=*file*	-l	将存档库或链接命令 *file* 命名为链接器输入。
--disable_auto_rts		禁止自动选择运行时支持库。请参阅节 4.3.1.1。
--priority	-priority	满足由包含该符号定义的第一个库实现的未解析引用。
--reread_libs	-x	强制重新读取库，以解析反向引用。
--search_path=*pathname*	-I	在查找默认位置之前，更改库搜索算法以查找用 *pathname* 命名的目录。此选项必须出现在 --library 选项之前。

表 2-22. 命令文件预处理选项

选项	别名	说明
--define=*name*=*value*		将 *name* 预定义为预处理器宏命令。
--undefine=*name*		删除预处理器宏命令 *name*。
--disable_pp		禁用命令文件预处理。

表 2-23. 诊断消息选项

选项	别名	说明
--diag_error=*num*		将由 *num* 标识的诊断分类为错误。
--diag_remark=*num*		将由 *num* 标识的诊断分类为备注。
--diag_suppress=*num*		抑制由 *num* 标识的诊断。
--diag_warning=*num*		将由 *num* 标识的诊断分类为警告。
--display_error_number		显示诊断的标识符及其文本。
--emit_references:file[=*file*]		发出包含段信息的文件。这些信息包括段大小、定义的符号和对符号的引用。
--emit_warnings_as_errors	-pdew	将警告视为错误。
--issue_remarks		发出备注（非严重警告）。
--no_demangle		禁止解码诊断消息中的符号名称。
--no_warnings		抑制诊断警告（仍会发出错误）。
--set_error_limit=*count*		将错误限制设置为 *count*。在达到此错误数量后，链接器将放弃链接。（默认为 100。）
--verbose_diagnostics		提供详细的诊断消息，以换行方式显示原始源代码。

表 2-24. 链接器输出选项

选项	别名	说明
--absolute_exe	-a	生成绝对可执行目标文件。这是默认设置；如果 --absolute_exe 和 --relocatable 均未指定，链接器的行为就像指定了 --absolute_exe 一样。
--ecc={ on | off }		启用由链接器生成的错误校正码 (ECC)。默认关闭。
--ecc:data_error		将指定的错误注入到输出文件中进行测试。
--ecc:ecc_error		将指定的错误注入到错误校正码 (ECC) 中进行测试。
--mapfile_contents=*attribute*		控制映射文件中包含的信息。
--relocatable	-r	生成不可执行的、可重定位输出目标文件。
--run_abs	-abs	生成绝对列表文件。

表 2-24. 链接器输出选项（续）

选项	别名	说明
--xml_link_info=*file*		生成结构良好的 XML *file*，其中包含有关链接结果的详细信息。

表 2-25. 符号管理选项

选项	别名	说明
--entry_point=*symbol*	-e	定义一个全局符号，用于指定可执行目标文件的主要入口点。
--globalize=*pattern*		将与 *pattern* 匹配的符号的符号链接更改为全局型。
--hide=*pattern*		隐藏与指定 *pattern* 匹配的符号。
--localize=*pattern*		将与指定 *pattern* 匹配的符号设为局部型。
--make_global=*symbol*	-g	将 *symbol* 设为全局型（覆盖 -h）。
--make_static	-h	将所有全局符号设为静态型。
--no_sym_merge	-b	禁止合并 COFF 目标文件中的符号调试信息。
--no_symtable	-s	从可执行目标文件中去除符号表信息和行号条目。
--retain		保留原本应丢弃的段列表。（仅限 EABI）
--scan_libraries	-scanlibs	扫描所有库中的重复符号定义。
--symbol_map=*refname*=*defname*		指定符号映射；对 *refname* 符号的引用被替换为对 *defname* 符号的引用。与 --opt_level=4 一同使用时，支持 --symbol_map 选项。
--undef_sym=*symbol*	-u	将 *symbol* 作为未解析符号添加到符号表中。
--unhide=*pattern*		排除与指定 *pattern* 匹配的符号，使其不被隐藏。

表 2-26. 运行时环境选项

选项	别名	说明
--arg_size=*size*	--args	为 argc/argv 存储器区域保存 *size* 个字节。
--cinit_compression[=type]		指定应用于 C 自动初始化数据的压缩类型。如果此选项没有指定 *type*，则默认为 lzss，表示 Lempel-Ziv-Storer-Szymanski 压缩。（仅限 EABI）
--copy_compression[=type]		压缩由链接器复制表复制的数据。如果此选项没有指定 *type*，则默认为 lzss，表示 Lempel-Ziv-Storer-Szymanski 压缩。（仅限 EABI）
--fill_value=*value*	-f	为输出段中的空穴设置默认填充值
--ram_model	-cr	在加载时初始化变量。有关详细信息，请参阅节 4.3.5。
--rom_model	-c	在运行时自动初始化变量。有关详细信息，请参阅节 4.3.5。

表 2-27. 链接时优化选项

选项(1)	别名	说明
--keep_asm		保留任何由 -plink 选项生成的链接后文件 (.pl) 和 .absolute 列表文件 (.abs)。这样用户便可查看链接后优化器所做的任何更改。（需要使用 -plink）
--no_postlink_across_calls	-nf	禁用跨函数的链接后优化。（需要使用 -plink）
--plink_advice_only		如果出于流水线方面的考虑而无法安全地进行更改（例如启用浮动支持或 VCU 支持时），则使用注释功能对汇编代码进行评注。（需要使用 -plink）
--postlink_exclude	-ex	从链接后传递中排除文件。（需要使用 -plink）
--postlink_opt	-plink	链接后优化（仅在 -z 后）。

(1) 有关详细信息，请参阅节 5.6。

表 2-28. 其他选项

选项	别名	说明
--compress_dwarf[=off|on]		积极减少输入目标文件中 DWARF 信息的大小。默认为 off。
--disable_clink	-j	禁止对 COFF 目标文件进行条件链接。（仅限 COFF）
--linker_help	[-]-help	显示有关语法和可用选项的信息。
--preferred_order=*function*		为函数放置设定优先级。

表 2-28. 其他选项（续）

选项	别名	说明
--unused_section_elimination[=off|on]		消除可执行模块中不需要的段。默认为 on。（仅限 EABI）
--zero_init=[off|on]		控制对未初始化的变量的预初始化。默认为 on。如果使用了 --ram_model，则始终为 off。（仅限 EABI）

2.3.2 常用选项

以下是对可能会经常使用的选项的详细说明：

--c_src_interlist 调用交叉列出功能，该功能使原始 C/C++ 源代码与编译器生成的汇编语言交织在一起。交叉列出的 C 语句可能看起来是乱序的。可通过组合 --optimizer_interlist 和 --c_src_interlist 选项，将 交叉列出功能与优化器结合使用。请参阅节 3.10。--c_src_interlist 选项可能会对性能和/或代码大小产生负面影响。

--cmd_file=*filename* 将文件的内容附加到选项集。使用此选项可避免操作系统对命令行长度或 C 样式注释的限制。使用# 或 ; 在命令文件中的一行的开头包含注释。可以用 /* 和 */ 括起来添加注释。如需指定选项，请用引号将连字符括起来。例如，"--"quiet。可以多次使用 --cmd_file 选项来指定多个文件。例如，以下代码表示 file3 应编译为源文件，而 file1 和 file2 是 --cmd_file 文件：

```
cl2000 --cmd_file=file1 --cmd_file=file2 file3
```

--compile_only 抑制链接器并覆盖用于指定链接的 --run_linker 选项。--compile_only 选项的缩写形式为 -c。在 C2000_C_OPTION 环境变量中指定了 --run_linker 但又不希望链接时，请使用此选项。请参阅节 4.1.3。

--define=*name[=def]* 预定义预处理器的常量 *name*。这相当于在每个 C 源文件的顶部插入 #define *name def*。如果省略可选的 [=*def*]，则 *name* 设置为 1。此选项的缩写形式是 -D。
如需定义带引号的字符串并保留引号，请执行以下操作之一：

- 对于 Windows，请使用 --define=*name*="\"*string def*\""。例如，--define=car="\"sedan\""
- 对于 UNIX，请使用 --define=*name*="'*string def*'"。例如，--define=car="'sedan'"
- 对于 CCS，请在文件中输入定义并使用 --cmd_file 选项包含该文件。

--help 显示调用编译器的语法并列出可用选项。如果 --help 选项后跟另一个选项或词组，则显示有关该选项或词组的详细信息。例如，要查看有关调试选项的信息，请使用 --help debug。

--include_path=*directory* 将 *directory* 添加到编译器搜索 #include 文件的目录列表中。--include_path 选项的缩写形式为 -I。可以多次使用此选项来定义几个目录；请确保用空格分隔 --include_path 选项。如果未指定目录名称，预处理器将忽略 --include_path 选项。请参阅节 2.5.2.1。

--keep_asm 保留编译器或汇编优化器的汇编语言输出。通常，编译器在汇编完成后会删除输出的汇编语言文件。此选项的缩写形式是 -k。

--quiet 抑制来自所有工具的横幅和进度信息。仅输出源文件名和错误消息。--quiet 选项的缩写形式为 -q。

--run_linker 在指定的目标文件上运行链接器。--run_linker 选项及其参数跟随命令行上的所有其他选项。--run_linker 后面的所有参数都传递给链接器。--run_linker 选项的缩写形式为 -z。请参阅节 4.1。

--skip_assembler 仅编译。指定的源文件已被编译但不会被汇编或链接。此选项的缩写形式为 -n。此选项将覆盖 --run_linker。输出为编译器的汇编语言输出。

--src_interlist 调用 交叉列出功能，该功能使优化器注释*或* C/C++ 源代码与汇编源代码交织在一起。如果调用优化器 (--opt_level=*n* 选项) ，优化器注释将与编译器的汇编语言输出交织在一起，这可能会明显地重新排列代码。如果未调用优化器，C/C++ 源代码语句将与编译器的汇编语言输出交织在一起，这样就可以检查为每条 C/C++ 语句生成的代码。--src_interlist 选项意味着 --keep_asm 选项。--src_interlist 选项的缩写形式为 -s。

--tool_version 打印编译器中每个工具的版本号。未发生编译。

--undefine=*name* 不对预定义的常量 *name* 进行定义。此选项覆盖指定常量的任何 --define 选项。--undefine 选项的缩写形式为 -U。

--verbose 编译时显示进度信息和工具集版本。重置 --quiet 选项。

2.3.3 其他有用的选项

以下是其他选项的详细说明：

--advice:performance[=all|none]
如果启用三角函数加速器 (TMU) 支持 (--tmu_support) 且 --fp_mode=strict，则当编译器遇到可以在 --fp_mode=relaxed 下用 TMU 硬件指令替换的函数时，会生成建议。这些函数调用包括浮点除法、sqrt、sin、cos、atan 和 atan2。
此外，当使用--float_support=fpu32 对 EABI 进行编译时，启用此选项会导致在检测到双精度浮点运算提供建议。由于 EABI 双精度数为 64 位，请考虑将双精度数更改为浮点数，以提高 FPU32 模式下的性能。

--float_operations_allowed={none|all|32|64}
限制允许的浮点运算类型。默认为 all。如果设置为 none、32 或 64，则检查应用程序是否将在运行时执行运算。例如，如果在命令行上指定了 --float_operations_allowed=32，则编译器将在生成双精度运算时发出错误消息。这可以用来确保双精度运算不会意外地被引入到应用程序中。检查是在进行宽松模式优化后执行的，因此完全删除非法运算不会产生任何诊断消息。

--fp_mode={relaxed|strict}
默认的浮点模式为 strict。要启用宽松浮点模式，请使用 --fp_mode=relaxed 选项。宽松浮点模式会使双精度浮点计算和存储在可能的情况下转换为单精度浮点。这种行为不符合 ISO 要求，但会加快代码速度，准确性会有降低。宽松模式下会发生以下具体的变化：

- 除以一个常数被转换为逆乘法。
- 某些 C 标准浮点函数（例如 sqrt、sin、cos、atan、atan2 和 fmodf）会被重定向到优化的内联函数（如有可能）。
- 如果使用 --tmu_support 选项来启用对三角函数加速器 (TMU) 的支持并启用宽松浮点模式，则 RTS 库调用将被替换为相应的 TMU 硬件指令，用于以下浮点运算：浮点除法、sqrt、sin、cos、atan 和 atan2。请注意，TMU 硬件指令和库例程之间存在算法差异，因此运算结果可能略有不同。
- 如果 --tmu_support=tmu1 与 --fp_mode=relaxed 结合使用，则使用以下 32 位 RTS 数学函数的特殊“宽松”版本：exp2f()、expf()、log2f()、logf() 和 powf()。请注意，未提供适用于 double 类型的宽松版本。

--fp_reassoc={on|off}
启用或禁用浮点算术的重新关联。默认为 on。

因为浮点值的精度有限，并且浮点运算是四舍五入的，所以浮点算术既不具有结合性，也不具有分配性。。例如，(1 + 3e100) - 3e100 不等于 1 + (3e100 - 3e100)。如果严格遵循 IEEE 754，编译器通常不能重新关联浮点运算。使用 --fp_reassoc=on 时，允许编译器重新关联代数，但代价是某些运算的精度会降低。

当 --fp_reassoc=on 时，可能会生成 RPT MACF32 指令。由于 RPT MACF32 指令计算两个部分和，然后将它们加在一起来计算整个累加值，因此在串行浮点乘法累加循环中，结果的精度可能有所不同。

--fp_single_precision_constant
致使所有未添加后缀的浮点常量都被视为单精度值。默认情况下，如果未使用此选项，则此类常量会按照 EABI 输出的预期隐式转换为双精度常量。如果浮点常量始终符合 32 位浮点数所支持的范围，那么将它们视为此类常量可以提高性能。
此选项可与 --fp_mode 和 -float_support 选项的任何设置一起使用。

--preinclude=*filename*
在编译开始时包含 *filename* 的源代码。这可用于建立标准的宏定义。在包含搜索列表上的目录中搜索文件名。文件按照指定的顺序进行处理。

--printf_support={full|nofloat|minimal}
支持更小、有限版本的 printf 函数系列（sprintf、fprintf 等）和 scanf 函数系列（sscanf、fscanf 等）运行时支持函数。有效值为：

- full：支持所有格式说明符。这是默认设置。
- nofloat：不支持打印和扫描浮点值。支持除 %a、%A、%f、%F、%g、%G、%e 和 %E 之外的所有格式说明符。
- minimal：支持打印和扫描没有宽度或精度标志的整数、字符或字符串值。具体来说，仅支持 %%、%d、%o、%c、%s 和 %x 格式说明符。

没有运行时错误检查来检测是否使用了未包含支持的格式说明符。--printf_support 选项位于 --run_linker 选项之前，并且必须在执行最终链接时使用。

--sat_reassoc={on|off}
启用或禁用饱和算术的重新关联。

2.3.4 运行时模型选项

这些选项专用于 TMS320C28x 工具集。有关更多信息，请参阅参考的章节。节 2.3.11 中列出了 TMS320C28x 专用汇编器选项。

C28x 编译器同时支持 COFF ABI 和嵌入式应用程序二进制接口 (EABI) ABI。EABI 使用 ELF 目标文件格式和 DWARF 调试格式。

--abi={eabi|coffabi}

指定应用程序二进制接口 (ABI)。默认的 ABI 为 COFF。还支持 EABI。请参阅节 2.13。请参阅《*C28x 嵌入式应用程序二进制接口应用报告*》(SPRAC71)。

EABI 应用程序中的所有代码都必须为 EABI 构建。在将现有的 COFF ABI 系统迁移到 EABI 之前，请确保所有库都在 EABI 模式下可用。

--cla_support={cla0|cla1|cla2}

指定 TMS320C28x 控制律加速器 (CLA) 支持 类型 0、类型 1 或类型 2 。此选项用于编译或汇编为 CLA 编写的代码。此选项在链接时不需要任何特殊的库支持；支持/不支持 FPU 的 C28x 所使用的库应该足够了。

--float_support={ fpu32 | fpu64 | softlib }

指定使用 TMS320C28x 32 位或 64 位硬件浮点支持。使用 --float_support=fpu32 指定具有 32 位硬件浮点支持的 C28x 架构。使用 --float_support=fpu64 指定具有 64 位硬件浮点支持的 C28x 架构。仅当使用 EABI 时才支持 FPU64。

如果使用 --tmu_support 选项来支持三角函数加速器，则 --float_support 选项会自动设置为 fpu32。默认值为 softlib，它会在没有特殊硬件支持的情况下执行浮点计算。

--idiv_support={ none | idiv0 }

使用硬件扩展来支持快速整数除法，以提供一组加速整数除法的指令。如果此硬件可用，请使用 --idiv_support=idiv0 来使用这些指令。默认为 none。包含此硬件的器件的数据表中含有“支持快速整数除法 (FINTDIV)”字样。（仅限 EABI。）

启用此选项后，内置整数除法和模运算符（“/”和“%”）使用适当的速度更快的指令。有关此类情况的更多信息，请参阅节 7.8.2。

启用此选项后，还可以使用节 7.6.4 中介绍的快速整数除法内在函数。为了使用这些内在函数，代码必须包含 `stdlib.h` 头文件。

--no_rpt

阻止编译器生成重复 (RPT) 指令。默认情况下，会为某些 memcpy、除法和乘法累加运算生成重复指令。但是，重复指令是不可中断的。

--pending_instantiations=#

指定在任何给定时间内可能正在进行的模板实例化的数量。使用 0 指定一个不受限制的数字。

--protect_volatile=*num*

启用易失性引用保护。已声明为易失性的非局部变量之间可能会发生流水线冲突。当写入一个易失性变量，然后读取另一个 易失性变量时，可能会发生冲突。--protect_volatile 选项允许在两个易失性引用之间至少放置 *num* 条指令，以确保写入操作发生在读取操作之前。num 为可选。如果未指定 num，则默认值为 2。例如，如果使用了 --protect_volatile=4，则易失性写入和易失性读取至少受到 4 条指令的保护。

外设流水线保护硬件保护所有内部外设和 XINTF 区域 1。如果将外设连接到 Xintf 区域 0、2、6、7，则可能需要使用 --protect_volatile 选项。内存不需要硬件保护或使用此选项。

--ramfunc={on|off}

如果设置为 on，则指定所有函数都应放置在位于 RAM 中的 .TI.ramfunc 段中。如果设置为 off，则只有具有 ramfunc 函数属性的函数才会以此种方式被处理。请参阅节 6.15.2。

较新的 TI 链接器命令文件通过在 .TI.ramfunc 段中放置函数来自动支持 --ramfunc 选项。如果链接器命令文件不包含 .TI.ramfunc 段的段规格，则可以修改链接器命令文件以将此段放在 RAM 中。有关段放置位置的详细信息，请参阅《*TMS320C28x 汇编语言工具用户指南*》。

--rpt_threshold=*k*

生成迭代 *k* 次或次数更少的 RPT 循环（k 是 0 和 256 之间的常数）。如果迭代次数大于 k 并且代码大小没有增加太多，则可以为同一个循环生成多个 RPT。在优化代码大小时使用此选项禁止为迭代次数可能大于 k 的循环生成 RPT 循环。

请注意，通过带有寄存器操作数的 RPT，内联的 memcpy 调用现在支持超过 255 个字。因此，可支持内联最多 65535 个字的 memcpy。如果设置 --no_rpt 或 --rpt_threshold 选项，则分别禁用或减少此类内联。可使用 --rpt_threshold 指定的最大值仍然是 256。

--silicon_errata_fpu1_workaround=on|off

启用此选项可防止在某些指令期间可能发生的 FPU 寄存器写入冲突。在 FRACF32、F32TOUI32 或 UI16TOF32 指令期间不能发生 CPU 到 FPU 寄存器写入。如果启用此选项，编译器会在这些指令之前增加五条 NOP 指令以防止冲突。

如果启用了以下任一选项，则默认情况下会禁用此选项：--float_support=fpu64、--tmu_support 或 --vcu_support=vcu2|vcrc。

--silicon_version=28

为 TMS320C28x 架构生成代码。唯一接受的值是 28。这是默认值，因此命令行上不再需要此选项。

--unified_memory

如果内存映射配置为单个统一空间，则使用 --unified_memory (-mt) 选项；此选项允许编译器为大多数 memcpy 调用和结构赋值生成 RPT PREAD 指令。这也使得 MAC 指令得以生成。--unified_memory 选项还允许使用更高效的数据内存指令来访问切换表。即使使用统一的内存，一些外设的内存以及与这些外设关联的 RAM 也只在数据内存中分配。

如果启用了 –unified_memory，可以通过将符号声明为易失性来阻止程序内存地址访问特定的符号。易失性和非易失性符号之间的结构体赋值会对所使用的指令产生不同的影响，具体取决于赋值的方向。从非易失性到易失性的结构体赋值可以将 RPT 与 PREAD 搭配使用，其中 PREAD 使用程序总线读取非易失性源操作数。从易失性分配到非易失性的结构体赋值可以将 RPT 与 PWRITE 搭配使用，其中 PWRITE 使用程序总线写入非易失性目标操作数。

--tmu_support[=tmu0|tmu1]

支持三角数学单元 (TMU) 。使用此选项会自动启用 FPU32 支持 (与 --float_support=fpu32 选项一样) 。当启用 TMU 支持时，可使用内在函数在 TMU 上执行三角函数指令。

TMU 硬件指令和库例程之间存在算法差异，因此运算结果可能略有不同。

tmu1 设置仅适用于 EABI。除了 tmu0 设置支持的内在函数之外，tmu1 设置还增加了对 LOG2F32 和 IEXP2F32 内在函数的支持。

在宽松浮点模式下，RTS 库调用被替换为相应的 TMU 硬件指令，用于以下浮点运算：浮点除法、sqrt、sin、cos、atan 和 atan2。此外，如果 --tmu_support=tmu1 选项与 --fp_mode=relaxed 结合使用，则使用下述 32 位浮点数学函数的特殊版本：exp2f()、expf()、log2f()、logf() 和 powf()。未提供适用于 EABI 64 位 double 类型的宽松版本。

--vcu_support[=vcu0|vcu2|vcrc]

vcu0 和 vcu2 设置指定支持 Viterbi、复数数学和 CRC 单元 (VCU) 的类型 0 或类型 2。请注意，没有 VCU 类型 1。默认值为 vcu0。

vcrc 设置指定仅支持循环冗余校验 (CRC) 算法。仅当使用 FPU32 或 FPU64 时，才支持 vcrc 。

仅当源代码是为 VCU 编写的汇编代码时，此选项才有用。对于 C/C++ 代码，此选项被忽略。此选项在链接时不需要任何特殊的库支持；支持/不支持 VCU 的 C28x 所使用的库应该足够了。同样，请注意，没有 VCU 类型 1。

2.3.5 符号调试和分析选项

下述选项用于选择符号调试或分析：

--symdebug:dwarf （默认）生成 C/C++ 源代码级调试器使用的指令，并在汇编器中启用汇编源代码调试。--symdebug:dwarf 选项的缩写形式为 -g。请参阅节 3.12。有关 DWARF 格式的详细信息，请参阅 *DWARF 调试标准*。

--symdebug:dwarf_version={2|3|4} 在指定 --symdebug:dwarf（默认值）时，指定待生成的 DWARF 调试格式版本（2、3 或 4）。默认情况下，编译器为 COFF ABI 生成 DWARF 版本 3 的调试信息，为 EABI 生成版本 4 的调试信息。可以安全地混合使用 DWARF 版本 2、3 和 4。使用 DWARF 4 时，类型信息放置在 .debug_types 段中。链接时删除重复的类型信息。这种类型合并的方法优于 DWARF 2 或 3，并能生成更小的可执行文件。此外，与 DWARF 3 相比，DWARF 4 减小了中间目标文件的大小。有关 TI 扩展至 DWARF 语言的更多信息，请参阅《*DWARF 对 TI 目标文件的影响*》(SPRAAB5)。

--symdebug:none 禁用所有符号调试输出。不建议使用此选项；其阻止了调试和大多数性能分析功能。

--symdebug:profile_coff 将必要的调试指令添加到分析器所需的目标文件中，从而允许在对优化（如果使用）影响最小的情况下进行函数级分析。此选项不会阻碍优化。
可以在 Code Composer Studio 中的函数级边界上设置断点和分析文件，但不能像具有完整调试能力那样单步调试代码。（仅限 COFF； EABI 不支持。）

--symdebug:skeletal 已弃用。没有作用

2.3.6 指定文件名

在命令行中指定的输入文件可以是 C 源文件、C++ 源文件、汇编源文件或目标文件。编译器使用文件扩展名来确定文件类型。

扩展名	文件类型
.asm、.abs 或 .s*（扩展名以 s 开头）	汇编源文件
.c	C 源文件
.C	取决于操作系统
.cpp、.cxx、.cc	C++ 源文件
.obj .c.obj .cpp.obj .o* .dll .so	对象

备注

文件扩展名区分大小写：文件扩展名是否区分大小写取决于您的操作系统。如果您的操作系统不区分大小写，带有 .C 扩展名的文件将被解释为 C 文件。如果您的操作系统区分大小写，带有 .C 扩展名的文件将被解释为 C++ 文件。

有关如何更改编译器解释各个文件名的方式的信息，请参阅节 2.3.7。有关如何更改编译器解释和命名汇编源文件和目标文件扩展名的方式的信息，请参阅节 2.3.10。

可使用通配符来编译或汇编多个文件。通配符规范因系统而异；请使用操作系统手册中列出的适当格式。例如，要编译扩展名为 .cpp 的目录中的所有文件，请输入以下命令：

```
cl2000 *.cpp
```

备注

假定源文件没有默认扩展名：如果在命令行中列出名为 example 的文件名，则编译器会假定定整个文件名是 example 而不是 example.c。不会向不包含扩展名的文件添加默认扩展名。

2.3.7 更改编译器解释文件名的方式

可以使用选项来更改编译器解释文件名的方式。如果使用的扩展名与编译器识别的扩展名不同，可以使用文件名选项来指定文件类型。可以在选项和文件名之间插入一个可选空格。为需要指定的文件类型选择合适的选项：

--asm_file=*filename*	用于汇编语言源文件
--c_file=*filename*	用于 C 源文件
--cpp_file=*filename*	用于 C++ 源文件
--obj_file=*filename*	用于目标文件

例如，如果有一个名为 file.s 的 C 源文件和一个名为 assy 的汇编语言源文件，请使用 --asm_file 和 --c_file 选项强制进行正确解释：

```
cl2000 --c_file=file.s --asm_file=assy
```

无法对文件名选项使用通配符规范。

备注

编译器创建的目标文件的默认文件扩展名已被更改，以防止当 C 和 C++ 文件具有相同名称时发生冲突。从 C 源文件生成的目标文件具有 .c.obj 扩展名。从 C++ 源文件生成的目标文件具有 .cpp.obj 扩展名。

2.3.8 更改编译器处理 C 文件的方式

--cpp_default 选项使编译器将 C 文件作为 C++ 文件进行处理。默认情况下，编译器将扩展名为 .c 的文件视为 C 文件。更多有关文件名扩展名约定的信息，请参阅节 2.3.9。

2.3.9 更改编译器解释和命名扩展名的方式

可以使用选项来更改编译器程序解释文件扩展名的方式，并为编译器程序创建的文件的扩展名命名。文件扩展名选项必须位于它们在命令行上应用的文件名之前。可以对这些选项使用通配符规范。扩展名最长可达九个字符。为需要指定的扩展类型选择合适的选项：

--asm_extension=*new extension*	用于汇编语言文件
--c_extension=*new extension*	用于 C 源文件
--cpp_extension=*new extension*	用于 C++ 源文件
--listing_extension=*new extension*	设置列表文件的默认扩展名
--obj_extension=*new extension*	用于目标文件

以下示例将汇编文件 fit.rrr ，并创建名为 fit.o 的目标文件：

```
cl2000 --asm_extension=.rrr --obj_extension=.o fit.rrr
```

扩展名中的句点 (.) 是可选的。 以上实例也可以写成：

```
cl2000 --asm_extension=rrr --obj_extension=o fit.rrr
```

2.3.10 指定目录

默认情况下，编译器程序将其创建的目标文件、汇编文件和临时文件放置在当前目录中。如果希望编译器程序将这些文件放置在不同的目录中，请使用以下选项：

--abs_directory=*directory*
指定绝对列表文件的目标目录。默认是使用与目标文件目录相同的目录。例如：

```
cl2000 --abs_directory=d:\abso_list
```

--asm_directory=*directory*
指定汇编文件的目录。例如：

```
cl2000 --asm_directory=d:\assembly
```

--list_directory=*directory*
指定汇编列表文件和交叉引用列表文件的目标目录。默认是使用与目标文件目录相同的目录。例如：

```
cl2000 --list_directory=d:\listing
```

--obj_directory=*directory*
指定目标文件的目录。例如：

```
cl2000 --obj_directory=d:\object
```

--output_file=*filename*
指定编译输出文件名；可以覆盖 --obj_directory。例如：

```
cl2000 --output_file=transfer
```

--pp_directory=*directory*
指定目标文件的预处理器文件目录 (默认为 .) 。例如：

```
cl2000 --pp_directory=d:\preproc
```

--temp_directory=*directory*
指定临时中间文件的目录。例如：

```
cl2000 --temp_directory=d:\temp
```

2.3.11 汇编器选项

以下是能够与编译器一起使用的汇编器选项。有关更多信息，请参阅《*TMS320C28x 汇编语言工具用户指南*》。

--absolute_listing
生成具有绝对地址而不是段偏移的列表。

--asm_define=*name*[=*def*]
为汇编器预定义常量 *name*，为常量生成 .set 指令，或为字符串生成 .arg 指令。如果省略可选的 [=*def*]，则 *name* 设置为 1。如果要定义带引号的字符串并保留引号，请执行以下操作之一：

- 对于 Windows，请使用 --asm_define=*name*="\"*string def*\""。例如：`--asm_define=car="\"sedan\""`
- 对于 UNIX，请使用 --asm_define=*name*="'*string def*'"。例如：`--asm_define=car='"sedan"'`
- 对于 Code Composer Studio，请在文件中输入定义，并使用 --cmd_file 选项包含该文件。

--asm_dependency
对执行汇编文件进行预处理，但不是写入预处理输出，而是将适合于输入的依赖行列表写入标准 make 实用程序。该列表将写入与源文件同名但扩展名为 .ppa 的文件中。

--asm_includes
对汇编文件进行预处理，但不是写入预处理后的输出，而是写入 #include 指令包含的文件列表。该列表将写入与源文件同名但扩展名为 .ppa 的文件中。

--asm_listing
生成汇编列表文件。

--issue_remarks
发出备注（非严重警告）。对于汇编器，启用附加的汇编时间检查。如果 .ebss 分配大小大于 64 个字，或者 16 位立即数操作数值位于 -32 768 到 65 535 范围之外，则会生成备注。

--asm_undefine=*name*
不对预定义的常量 *name* 进行定义。此选项覆盖指定名称的任何 --asm_define 选项。

--asm_cross_reference_listing
在列表文件中生成符号交叉引用。

--flash_prefetch_warn
如果程序数据访问指令跟在 BF 或 SBF 指令的 8 个字内，则启用汇编器警告。正如《*TMS320C281X/TMS320F281X DSP 器件勘误表*》(SPRZ193) 关于“Flash 和 OTP 预取缓冲溢出”的建议所述，如果在启用闪存预取缓冲器的情况下从闪存或一次性可编程 (OTP) 存储器执行此指令序列，则闪存预取缓冲器可能会溢出。溢出是否真的发生取决于指令序列、闪存等待状态和 CPU 流水线停顿。如果发生溢出，将导致执行无效的操作码。使用程序内存寻址的指令包括 MAC/XMAC、DMAC/XMACD、QMACL、IMACL、PREAD/XPREAD 和 PWRITE/XPWRITE。

--include_file=*filename*
包含汇编模块的指定文件；类似于 .include 指令。该文件包含在源文件语句之前。包含的文件不会显示在汇编列表文件中。

--preproc_asm
扩展汇编文件中的宏并汇编扩展的文件。宏扩展有助于调试汇编文件。--preproc_asm 选项仅影响汇编文件。当使用 --preproc_asm 时，编译器首先使用 调用汇编器以生成宏扩展的源 .exp 文件。然后汇编 .exp 文件以生成目标文件。调试器使用 .exp 文件进行调试。.exp 文件是一个中间文件，对该文件进行的任何更新都将丢失。需要对原始汇编文件进行更新。

2.3.12 已弃用的选项

几个编译器选项已被弃用、删除或重命名。编译器继续接受一些已弃用的选项，但不建议使用它们。

2.4 通过环境变量控制编译器

环境变量是由用户定义并向其分配字符串的系统符号。如果您希望重复运行编译器而不重新输入选项、输入文件名或路径名，则设置环境变量非常有用。

备注

C_OPTION 和 C_DIR -- 已弃用 C_OPTION 和 C_DIR 环境变量。请使用器件专用环境变量。

2.4.1 设置默认编译器选项 (C2000_C_OPTION)

您可能会发现，使用 C2000_C_OPTION 环境变量来设置编译器、汇编器和链接器默认选项很有用。如果这样做，编译器将在每次运行编译器时使用命名为 C2000_C_OPTION 的默认选项和/或输入文件名。

当希望使用相同的一组选项和/或输入文件来重复运行编译器时，使用这些环境变量来设置默认选项非常有用。编译器读取命令行和输入文件名后，查找 C2000_C_OPTION 环境变量并进行处理。

下表展示了如何设置 C2000_C_OPTION 环境变量。为操作系统选择命令：

操作系统	输入
UNIX (Bourne shell)	**C2000_C_OPTION="** *option*$_1$ [*option*$_2$ ...]**"; export C2000_C_OPTION**
Windows	**set C2000_C_OPTION=** *option*$_1$ [*option*$_2$ ...]

环境变量选项的指定方式以及含义与它们在命令行中的相同。例如，如果您想始终安静地运行（--quiet 选项）、启用 C/C++ 源代码交叉列出功能（--src_interlist 选项），并为 Windows 链接（--run_linker 选项），请设置 C2000_C_OPTION 环境变量，如下所示：

```
set C2000_C_OPTION=--quiet --src_interlist --run_linker
```

命令行或 C2000_C_OPTION 中位于 --run_linker 后面的所有选项都将传递给链接器。因此，可使用 C2000_C_OPTION 环境变量来指定默认编译器和链接器选项，然后在命令行上指定其他编译器和链接器选项。如果在环境变量中设置了 --run_linker 并且只希望进行编译，请使用编译器 --compile_only 选项。以下附加示例假设 C2000_C_OPTION 设置如上所示：

```
cl2000 *c                          ; compiles and links
cl2000 --compile_only *.c          ; only compiles
cl2000 *.c --run_linker lnk.cmd    ; compiles and links using a command file
cl2000 --compile_only *.c --run_linker lnk.cmd
        ; only compiles (--compile_only overrides --run_linker)
```

有关编译器选项的详细信息，请参阅节 2.3。有关编译器选项的详细信息，请参阅 *TMS320C28x 汇编语言工具用户指南*》中的链接器说明一章。

2.4.2 命名一个或多个备用目录 (C2000_C_DIR)

链接器使用 C2000_C_DIR 环境变量来命名包含对象库的备用目录。分配环境变量的命令语法是：

操作系统	输入
UNIX (Bourne shell)	**C2000_C_DIR="** *pathname*$_1$ **;** *pathname*$_2$ **;...";** **export C2000_C_DIR**
Windows	**set C2000_C_DIR=** *pathname*$_1$ **;** *pathname*$_2$ **;...**

pathnames 是包含输入文件的目录。路径名 (pathnames) 必须遵循以下约束：

- 路径名必须用分号分隔。
- 忽略路径开头或结尾处的空格或制表符。例如，忽略下面分号前后的空格：

```
set C2000_C_DIR=c:\path\one\to\tools ; c:\path\two\to\tools
```

- 允许在路径中使用空格和制表符来容纳包含空格的 Windows 目录。例如，下述路径名是有效的：

```
set C2000_C_DIR=c:\first path\to\tools;d:\second path\to\tools
```

环境变量保持设置状态，直到您重新启动系统或通过输入以下命令来重置变量：

操作系统	输入
UNIX (Bourne shell)	`unset C2000_C_DIR`
Windows	`set C2000_C_DIR=`

2.5 控制预处理器

本节介绍了控制预处理器的特性，预处理器是解析器的一部分。K&R 第 A12 节对 C 预处理进行了一般性描述。C/C++ 编译器包含标准 C/C++ 预处理函数，这些函数内置于编译器的第一轮中。预处理器处理：

- 宏定义和扩展
- #include 文件
- 条件编译
- 各种预处理器指令，在源文件中指定为以 # 字符开头的行

预处理器生成自解释的错误消息。出现错误的行号和文件名与诊断消息一同打印。

2.5.1 预先定义的宏名称

编译器维护并识别表 2-29 中列出的预定义宏名称。

表 2-29. 预定义 C28x 宏名称

宏名称	说明
__DATE__ (1)	以 *mmm dd yyyy* 形式扩展到编译日期
__FILE__ (1)	扩展到当前源文件名
_INLINE	如果使用了优化（--opt_level 或 -O 选项），则扩展为 1；否则未定义。
__LINE__ (1)	扩展到当前行号
__little_endian__	始终定义为 1。
__PTRDIFF_T_TYPE__	定义为 ptrdiff_t 类型
__SIZE_T_TYPE__	定义为 size_t 类型
__STDC__ (1)	定义为 1 以表示编译器符合 ISO C 标准。有关 ISO C 标准的例外情况，请参阅节 6.1。
__STDC_VERSION__	C 标准宏。
__STDC_HOSTED__	C 标准宏。始终定义为 1。
__STDC_NO_THREADS__	C 标准宏。始终定义为 1。
__TI_COMPILER_VERSION__	已定义为 7-9 位整数，具体取决于 X 是 1、2 还是 3 位。该数字不包含小数。例如，版本 3.2.1 表示为 3002001。去掉前导零以防止数字被解释为八进制。

表 2-29. 预定义 C28x 宏名称（续）

宏名称	说明
__TI_EABI__	如果使用了 --abi=eabi，则定义为 1。
__TI_GNU_ATTRIBUTE_SUPPORT__	如果启用了 GCC 扩展（这是默认设置），则定义为 1
__TI_STRICT_ANSI_MODE__	如果启用了严格的 ANSI/ISO 模式（使用了 --strict_ansi 选项），则定义为 1；否则定义为 0。
__TI_STRICT_FP_MODE__	如果使用了 --fp_mode=strict（默认设置），则定义为 1；否则定义为 0。
__TIME__ (1)	以“*hh:mm:ss*”形式扩展到编译时间
__TMS320C2000__	为 C28x 处理器定义
__TMS320C28XX__	如果目标是 C28x，则已定义
__TMS320C28XX_CLA__	如果使用了任意 --cla_support 选项并且源文件是 .cla 文件，则定义为 1。
__TMS320C28XX_CLA0__	如果使用了 --cla_support=cla0 选项并且源文件是 .cla 文件，则定义为 1。
__TMS320C28XX_CLA1__	如果使用了 --cla_support=cla1 选项并且源文件是 .cla 文件，则定义为 1。
__TMS320C28XX_CLA2__	如果使用了 --cla_support=cla2 选项并且源文件是 .cla 文件，则定义为 1。
__TMS320C28XX_FPU32__	如果使用了 --float_support=fpu32 或 fpu64 选项，则定义为 1。
__TMS320C28XX_FPU64__	如果使用了 --float_support=fpu64 选项，则定义为 1。
__TMS320C28XX_IDIV__	如果使用了 --idiv_support=idiv0 选项，则定义为 1。
__TMS320C28XX_TMU__	如果 --tmu_support 选项与任意设置一起使用，则定义为 1。
__TMS320C28XX_TMU0__	如果 --tmu_support 选项与任意设置一起使用，则定义为 1。
__TMS320C28XX_TMU1__	如果使用了 --tmu_support=tmu1 选项，则定义为 1。
__TMS320C28XX_VCU0__	如果 --vcu_support 选项与任意设置一起使用，则定义为 1。
__TMS320C28XX_VCU2__	如果使用了 --vcu_support=vcu2 选项，则定义为 1。
__TMS320C28XX_VCRC__	如果使用了 --vcu_support=vcrc 选项，则定义为 1。
__WCHAR_T_TYPE__	定义为 wchar_t 类型。

(1) 由 ISO 标准指定

可以按照与任何其他已定义名称相同的方式使用表 2-29 中列出的名称。例如，

```
printf ( "%s %s" , __TIME__ , __DATE__);
```

转换为类似如下行：

```
printf ("%s %s" , "13:58:17", "Jan 14 1997");
```

2.5.2 #include 文件的搜索路径

#include 预处理器指令告诉编译器从另一个文件读取源语句。指定该文件时，可将文件名用双引号或尖括号括起来。文件名可以是完整的路径名、部分路径信息或不带路径信息的文件名。

- 如果将文件名括在双引号 (" ") 中，编译器将按下述顺序搜索文件：
  1. 包含 #include 指令的文件目录以及包含该文件的任何文件的目录。
  2. 使用 --include_path 选项命名的目录。
  3. 使用 C2000_C_DIR 环境变量设置的目录。
- 如果将文件名括入尖括号 (< >) 中，编译器将按下述顺序在以下目录中搜索文件：
  1. 使用 --include_path 选项命名的目录。
  2. 使用 C2000_C_DIR 环境变量设置的目录。

有关使用 --include_path 选项的信息，请参阅节 2.5.2.1。有关输入文件目录的更多信息，请参阅节 2.4.2。

2.5.2.1 在 #include 文件搜索路径 (--include_path 选项) 中新增目录

--include_path 选项命名了包含 #include 文件的备用目录。--include_path 选项的缩写形式为 `-I`。--include_path 选项的格式为：

--include_path=*directory1* [**--include_path=** *directory2* ...]

每次调用编译器时，--include_path 选项的数量没有限制；每个 --include_path 选项命名一个 *directory*。在 C 源代码中，可以使用 #include 指令而不指定文件的任何目录信息；相反，可以使用 --include_path 选项指定目录信息。

例如，假设当前目录中有一个名为 source.c 的文件。文件 source.c 包含以下指令语句：

```
#include "alt.h"
```

假设 alt.h 的完整路径名是：

UNIX	/tools/files/alt.h
Windows	c:\tools\files\alt.h

下表显示了如何调用编译器。选择适用操作系统的命令：

操作系统	输入
UNIX	`cl2000 --include_path=/tools/files source.c`
Windows	`cl2000 --include_path=c:\tools\files source.c`

备注

在尖括号中指定路径信息：如果在尖括号中指定了路径信息，编译器会应用与-include_path 选项和 C2000_C_DIR 环境变量指定的路径信息相关的信息。

例如，如果使用以下命令设置 C2000_C_DIR：

```
C2000_C_DIR "/usr/include;/usr/ucb"; export C2000_C_DIR
```

或使用以下命令调用编译器：

```
cl2000 --include_path=/usr/include file.c
```

且 file.c 包含以下行：

```
#include <sys/proc.h>
```

结果是包含的文件位于以下路径中：

```
/usr/include/sys/proc.h
```

2.5.3 支持#warning 和 #warn 指令

在严格的 ANSI 模式下，TI 预处理器允许使用 #warn 指令使预处理器发出警告并继续预处理。#warn 指令等效于 GCC、IAR 和其他编译器支持的 #warning 指令。

如果使用 --relaxed_ansi 选项（默认值为 on），则同时支持 #warn 和 #warning 预处理器指令。

2.5.4 生成预处理列表文件（--preproc_only 选项）

--preproc_only 选项允许您生成扩展名为 .pp 的源文件的预处理版本。编译器的预处理函数对源文件执行以下操作：

- 每个以反斜杠 (\) 结尾的源代码行都与下一行联接。
- 扩展三字符序列。
- 删除注释。
- 将 #include 文件复制到文件中。
- 处理宏定义。
- 扩展所有宏。
- 扩展所有其他预处理指令，包括 #line 指令和条件编译。

在为技术支持案例创建源文件或询问有关代码的问题时，--preproc_only 选项很有用。该选项允许将测试用例减少到单个源文件，因为#include 文件是在预处理器运行时会合并的。

2.5.5 预处理后继续编译（--preproc_with_compile 选项）

如果要进行预处理，预处理器只进行预处理，而不会编译源代码。要覆盖此特征并在预处理源代码后继续编译，请使用 --preproc_with_compile 和其他预处理选项。例如，使用 --preproc_with_compile 和 --preproc_only 执行预处理，将预处理的输出写入扩展名为 .pp 的文件，并编译源代码。

2.5.6 生成带有注释的预处理列表文件（--preproc_with_comment 选项）

--preproc_with_comment 选项会执行除删除注释之外的所有预处理功能，并生成带有 .pp 扩展名的源文件的预处理版本。如果要保留注释，请使用 --preproc_with_comment 选项而非 --preproc_only 选项。

2.5.7 生成带有行控制详细信息的预处理列表（--preproc_with_line 选项）

默认情况下，预处理的输出文件不包含预处理器指令。要包含 #line 指令，请使用 --preproc_with_line 选项。--preproc_with_line 选项仅执行预处理并将带有行控制信息（#line 指令）的预处理输出写入名为源文件扩展名为.pp 的文件中。

2.5.8 为 Make 实用程序生成预处理输出（--preproc_dependency 选项）

--preproc_dependency 选项仅执行预处理。此选项不写入预处理输出，而是写入适合输入到标准 make 实用程序的依赖行列表。如果不提供可选的文件名，则列表将写入与源文件同名但扩展名为 .pp 的文件中。

2.5.9 生成包含#include 在内的文件列表（--preproc_includes 选项）

--preproc_includes 选项仅执行预处理，但不写入预处理输出，而是写入包含 #include 指令在内的文件列表。如果不提供可选的文件名，则列表将写入与源文件同名但扩展名为 .pp 的文件中。

2.5.10 在文件中生成宏列表（--preproc_macros 选项）

--preproc_macros 选项生成所有预定义宏和用户定义宏的列表。如果不提供可选的文件名，则列表将写入与源文件同名但扩展名为 .pp 的文件中。

输出仅包括那些被源文件直接包含的文件。首先列出预定义宏，并由注释 /* 预定义 */ 指示。接着列出用户定义宏，并由源文件名指示。

2.6 将参数传递给 main()

一些程序通过 argc 和 argv 将参数传递给 main()。这对不是从命令行运行的嵌入式程序带来了特殊的挑战。通常，argc 和 argv 通过 .args 段提供给程序。有多种方法可以填充此段以供程序使用。

要使链接器分配大小适当的 .args 段，请使用 --arg_size=*size* 链接器选项。此选项通知链接器分配一个名为 .args 的未初始化段，这样，加载器可以使用该段从加载器的命令行向程序传递参数。*size* 是要分配的字节数。当使用 --arg_size 选项时，链接器定义 __c_args__ 符号以包含 .args 段的地址。

加载器负责填充 .args 段。加载器和目标启动代码可以使用 .args 段和 __c_args__ 符号来确定是否以及如何将参数从主机传递到目标程序。参数的格式是指向目标上 char 类型的指针数组。由于加载器的变化，因此没有规定加载器如何确定将哪些参数传递给目标。

如果使用 Code Composer Studio 运行应用程序，则可以使用 Scripting Console 工具来填充 .args 段。要打开此工具，请从 CCS 菜单中选择 **View > Scripting Console**。可以使用 loadProg 命令将目标文件及其关联的符号表加载到存储器中，并将参数数组传递给 main()。这些参数会自动写入到分配的 .args 段。

loadProg 语法如下，其中 *file* 是可执行文件，*args* 是参数对象数组。使用此命令之前，请使用 JavaScript 声明参数数组。

```
loadProg(file, args)
```

对于不基本 SYS/BIOS 的可执行文件，.args 段加载下述数据，其中，argv[] 数组中的每个元素都包含与该参数对应的字符串：

```
Int argc;
Char * argv[0];
Char * argv[1];
...
Char * argv[n];
```

对于基于 SYS/BIOS 的可执行文件，.args 段中的元素如下：

```
Int argc;
Char ** argv;    /* points to argv[0] */
Char * envp;     /* ignored by loadProg command */
Char * argv[0];
Char * argv[1];
...
Char * argv[n];
```

有关更多详细信息，请参阅“Scripting Console”页面。

2.7 了解诊断消息

编译器和链接器的主要功能之一是报告源代码程序的诊断消息。诊断消息指示程序可能出了问题。当编译器或链接器检测到可疑情况时，它会采用以下格式显示一条消息：

" *file.c* **", line** *n* **:** *diagnostic severity* **:** *diagnostic message*

" *file.c* **"**	所涉及的文件的名称
line *n* **:**	诊断适用的行号
diagnostic severity	诊断消息的严重性（严重性类别说明如下）
diagnostic message	描述问题的文本

诊断消息的严重性如下：

- **致命错误**表示问题严重到无法继续编译。此类问题的示例包括命令行错误、内部错误和缺少包含文件。如果正在编译多个源文件，则不会编译当前源文件之后的任何其他源文件。
- **错误**表示违反了 C/C++ 语言的语法或语义规则。编译可以继续，但不会生成目标代码。
- **警告**表示可能有问题但不能证明是错误。例如，编译器会针对未使用的变量发出警告。未使用的变量不会影响程序执行，但它的存在表明您可能有意使用它。编译会继续并生成目标代码（如果没有检测到错误）。
- **备注**不如警告那么严重。它可以表示在极少数情况下存在潜在问题，或者该备注可能只是为了提供参考信息。编译会继续并生成目标代码（如果没有检测到错误）。默认情况下不会发出备注。使用 --issue_remarks 编译器选项可启用备注。

诊断消息以类似于以下示例的形式写入标准错误：

```
"test.c", line 5: error: a break statement may only be used within a loop or switch
    break;
    ^
```

默认情况下不会打印源代码行。使用 --verbose_diagnostics 编译器选项来显示源代码行和错误位置。上面的示例使用了此选项。

消息会标识诊断中所涉及的文件和行，并且源行本身（位置由 ^ 字符表示）跟在消息之后。如果几条诊断消息适用于一个源行，则每条诊断消息都具有所示的形式：源代码行的文本会显示几次，每次都显示在一个适合的位置。

必要时，长消息会换行到其他行。

可以使用 --display_error_number 命令行选项来请求将诊断的数字标识符包含在诊断消息中。如果显示了诊断标识符，诊断标识符还指示是否可以在命令行上覆盖诊断的严重性。如果可以覆盖严重性，则诊断标识符包括后缀 -D（*酌情处理*）；否则，不存在后缀。例如：

```
"Test_name.c", line 7: error #64-D: declaration does not declare anything
    struct {};
      ^
"Test_name.c", line 9: error #77: this declaration has no storage class or type specifier
    xxxxx;
    ^
```

由于错误是根据特定上下文中的严重性确定的，因此错误在某些情况下可以是酌情处理的，而在其他情况下则不是。所有警告和备注都是酌情处理的。

对于某些消息，实体（函数、局部变量、源文件等）列表很有用；实体在初始错误消息之后列出：

```
"test.c", line 4: error: more than one instance of overloaded function "f"
            matches the argument list:
            function "f(int)"
            function "f(float)"
            argument types are: (double)
   f(1.5);
   ^
```

在某些情况下，还会提供附加的上下文信息。特别是，如果前端在执行模板实例化时或在生成构造函数、析构函数或赋值运算符函数时发出诊断消息，上下文信息很有用。例如：

```
"test.c", line 7: error: "A::A()" is inaccessible
   B x;
   ^
          detected during implicit generation of "B::B()" at line 7
```

没有上下文信息，就很难确定错误指的是什么。

2.7.1 控制诊断消息

C/C++ 编译器提供诊断选项来控制编译器和链接器生成的诊断消息。必须在 --run_linker 选项之前指定诊断选项。

--diag_error=*num*
将由 *num* 标识的诊断分类为错误。若要确定诊断消息的数字标识符，请在单独的编译中首先使用 --display_error_number 选项。然后使用 --diag_error=*num* 将诊断重新归类为错误。您只能更改任意诊断消息的严重性。

--diag_remark=*num*
将由 *num* 标识的诊断分类为备注。若要确定诊断消息的数字标识符，请在单独的编译中首先使用 --display_error_number 选项。然后使用 --diag_remark=*num* 将诊断重新归类为备注。您只能更改任意诊断消息的严重性。

--diag_suppress=*num*
抑制由 *num* 标识的诊断。若要确定诊断消息的数字标识符，请在单独的编译中首先使用 --display_error_number 选项。然后使用 --diag_suppress=*num* 来抑制诊断。您只能抑制任意诊断消息。

--diag_warning=*num*
将由 *num* 标识的诊断分类为警告。若要确定诊断消息的数字标识符，请在单独的编译中首先使用 --display_error_number 选项。然后使用 --diag_warning=*num* 将诊断重新分类为警告。您只能更改任意诊断消息的严重性。

--display_error_number
显示诊断的数字标识符及其文本。使用此选项来确定需要向诊断抑制选项提供哪些参数（--diag_suppress、--diag_error、--diag_remark 和 --diag_warning）。此选项还指示诊断是否是任意的。任意诊断是指其严重性可以被忽略的诊断。任意诊断包括后缀 -D；否则，不存在后缀。请参阅节 2.7。

--emit_warnings_as_errors
将所有警告视为错误。此选项不能与 --no_warnings 选项一同使用。--diag_remark 选项优先于此选项。此选项优先于 --diag_warning 选项。

--issue_remarks
发出默认情况下被抑制的备注（非严重警告）。

--no_warnings
抑制诊断警告（仍会发出错误）。

--set_error_limit=*num*
将错误限制设置为 *num*，可以是任何十进制值。在出现此数量的错误后，编译器将放弃编译。（默认为 100。）

--verbose_diagnostics
提供详细的诊断消息，以换行方式显示原始源，并指示错误在源行中的位置。请注意，此命令行选项不能在 Code Composer Studio IDE 中使用。

--write_diagnostics_file	生成具有相同源文件名且扩展名为 *.err* 的诊断消息信息文件。（链接器不支持 --write_diagnostics_file 选项。）请注意，此命令行选项不能在 Code Composer Studio IDE 中使用。

2.7.2 如何使用诊断抑制选项

以下示例演示了如何控制编译器发出的诊断消息。可以使用类似的方式控制链接器诊断消息。

```
int one();
int I;
int main()
{
   switch (I){
   case  1;
      return one ();
      break;
   default:
      return 0;
   break;
   }
}
```

如果使用 --quiet 选项调用编译器，结果如下：

```
"err.c", line 9: warning: statement is unreachable
"err.c", line 12: warning: statement is unreachable
```

因为标准的编程做法是在每个 case 支臂的末尾包含 break 语句以避免导向条件，所以可以忽略这些警告。使用 --display_error_number 选项，可以找出这些警告的诊断标识符。结果如下：

```
[err.c]
"err.c", line 9: warning #111-D: statement is unreachable
"err.c", line 12: warning #111-D: statement is unreachable
```

接下来，可以使用诊断标识符 111 作为 --diag_remark 选项的参数，将此警告视为备注。此编译不产生诊断消息（因为默认情况下禁用备注）。

备注

可以抑制任何非致命错误，但务必确保仅抑制您理解的且已知不会影响程序正确性的诊断消息。

2.8 其他消息

其他与源代码无关的错误消息（例如错误的命令行语法或无法找到指定的文件）通常是致命的。这些错误消息由消息前的符号 >> 标识。

2.9 生成交叉参考列表信息（--gen_cross_reference_listing 选项）

--gen_cross_reference_listing 选项生成一个交叉参考列表文件，其中包含源文件中每个标识符的引用信息。列表文件描述了引用和定义每个符号的位置。

为每个源文件生成扩展名为 .crl 的交叉参考列表文件。这些文件与其对应的源文件具有相同的名称。（--gen_cross_reference_listing 选项与 --asm_cross_reference_listing 是分开的，后者是一个汇编器选项而不是编译器选项。）

交叉引用列表文件中的信息使用以下格式显示：

sym-id name X filename line number column number

sym-id	唯一分配给每个标识符的整数
name	标识符名称
X	以下值之一：
	D 定义

	d	声明（不是定义）
	M	修改
	A	已取地址
	U	已用
	C	已更改（已在单个操作中使用和修改）
	R	任何其他类型的引用
	E	错误；引用是不确定的
filename	源文件	
line number	源文件中的行号	
column number	源文件中的列号	

2.10 生成原始列表文件（--gen_preprocessor_listing 选项）

--gen_preprocessor_listing 选项生成一个原始列表文件，有助于了解编译器如何预处理源文件。预处理列表文件（使用 --preproc_only、--preproc_with_comment、--preproc_with_line 和 --preproc_dependency 预处理器选项生成）显示了源文件的预处理版本，而原始列表文件提供原始源代码行与预处理输出之间的比较情况。原始列表文件与扩展名为 *.rl* 的相应源文件具有相同的名称。

原始列表文件包含以下信息：

- 每个原始源代码行
- 转入和转出包含文件
- 诊断消息
- 如果执行了特殊处理，则预处理源代码行（删除注释微不足道；其他预处理则很特殊）

原始列表文件中的每个源代码行都以表 2-30 中列出的标识符之一开头。

表 2-30. 原始列表文件标识符

标识符	定义
N	正常的源代码行
X	扩展的源代码行。如果进行了特殊预处理，则会立即出现在正常的源代码行之后。
S	跳过的源代码行（false #if 子句）
L	源代码位置变化，格式如下： L *line number filename key* 其中，*line number* 是源文件中的行号。仅当包含文件的进入/退出而发生变化时，*key* 才存在。可能的 *key* 值为： 1 = 进入包含文件 2 = 从包含文件退出

--gen_preprocessor_listing 选项还包括表 2-31 中定义的诊断标识符。

表 2-31. 原始列表文件诊断标识符

诊断标识符	定义
E	错误
F	致命
R	注释
W	警告

诊断原始列表信息按以下格式显示：

S filename line number column number diagnostic

S	表 2-31 中的标识符之一指示诊断的严重程度
filename	源文件
line number	源文件中的行号
column number	源文件中的列号
diagnostic	用于诊断的消息文本

文件结尾后的诊断消息表示为文件的最后一行，列号为 0。当诊断消息文本需要多行时，后续的每一行都包含相同的文件、行和列信息，但使用小写版本的诊断标识符。有关诊断消息的更多信息，请参阅节 2.7。

2.11 使用内联函数扩展

当调用内联函数时，在调用点插入该函数的 C/C++ 源代码副本。这就是所谓的内联函数扩展，通常称为*函数内联*或简称*内联*。内联函数扩展可以通过消除函数调用开销来加快执行速度。这对于经常被调用的非常小的函数特别有用。函数内联涉及到在执行速度和代码大小之间进行权衡，因为代码在每个函数调用点都是重复的。在许多位置被调用的大型函数不适合内联。

备注

过多内联会降低性能：过多内联会使编译器显著变慢并降低所生成代码的性能。

以下情况会触发函数内联：

- 使用内置的内在函数运算。内在函数运算看起来像函数调用，即使不存在函数体，也会自动内联。
- 使用 内联 关键字或等效的 __内联 关键字。如果设置 --opt_level=0 或更大值，则使用内联关键字声明的函数可能会被编译器内联。内联关键字是程序员对编译器提出的建议。即使优化级别很高，内联对于编译器来说仍

然是可选的。编译器根据函数的长度、函数被调用的次数、--opt_for_speed 设置以及函数中任何不允许函数内联的内容来决定是否内联函数（请参阅节 2.11.2）。如果函数体在同一模块中可见，或者使用了 -pm 且函数在正在编译的模块之一中可见，则可以在 --opt_level=0 或更高级别内联函数。如果包含定义信息的文件和调用点都使用了 --opt_level=4 进行编译，则可以在链接时内联函数。同时定义为静态和内联的函数更有可能被内联。

- 当使用 --opt_level=3 或更高级别时，编译器可能会自动内联符合条件的函数，即使这些函数没有被声明为内联函数也是如此。此过程会用到使用内联关键字显式定义的函数对应列出的相同决策因素列表。有关自动函数内联的更多信息，请参阅节 3.5。
- 除非 --opt_level=off，否则 pragma FUNC_ALWAYS_INLINE（节 6.9.11）和等效的 `always_inline` 属性（节 6.15.2）会强制内联函数（这样做是合法的）。也就是说，即使函数未声明为内联且 --opt_level=0 或 --opt_level=1，pragma FUNC_ALWAYS_INLINE 也会强制函数内联。
- FORCEINLINE pragma（节 6.9.9）会强制函数内联到带注释的语句中。也就是说，它通常对这些函数没有影响，只对单个语句中的函数调用产生影响。FORCEINLINE_RECURSIVE pragma 不仅强制内联在语句中可见的调用，而且还强制内联该语句内联的调用体。
- --disable_inlining 选项阻止任何内联。pragma FUNC_CANNOT_INLINE 阻止函数被内联。NOINLINE pragma 阻止单个语句中的调用被内联。（NOINLINE 与 FORCEINLINE pragma 相反。）

备注

函数内联可以大大增加代码大小：函数内联会增加代码大小，尤其是内联在多个地方调用的函数。函数内联最适合仅从少数地方调用的函数以及小函数。

C 代码中的 `inline` 关键字的语义遵循 C99 标准。C++ 代码中的 `inline` 关键字的语义遵循 C++ 标准。

`inline` 关键字在所有 C++ 模式中、所有 C 标准的宽松 ANSI 模式中以及 C99 和 C11 的严格 ANSI 模式中都受支持。该关键字在 C89 的严格的 ANSI 模式中被禁用，因为它是一种可能与严格遵守标准的程序相冲突的语言扩展。如果要在严格 ANSI C89 模式下定义内联函数，请使用备用关键字 `__inline`。

影响内联的编译器选项有：--opt_level、--auto_inline、--remove_hooks_when_inlining、--opt_for_speed 和 --disable_inlining。

2.11.1 内联内在函数运算符

编译器具有大量内置函数式运算，称为内在函数。内在函数的实现由编译器处理：其用一系列指令代替函数调用。这类似于对内联函数的处理方式；然而，由于编译器知道内在函数的代码，因此可以进行更好的优化。

无论是否使用优化器，内在函数通常都是内联的。但是，如果 --opt_for_speed 选项设置为级别 0 或 1，编译器可能会选择不内联扩展为大量指令的内在函数。例如，--idiv_support=idiv0 选项启用的整数除法内在函数会扩展为比大多数内在函数更多的指令。

有关内在函数的详细信息以及内在函数列表，请参阅节 7.6。除了所列出的这些之外，abs 和 memcpy 也是作为内在函数实现的。

2.11.2 内联限制

编译器会根据节 2.11 中提到的因素决定内联哪些函数。此外，还有一些限制可以取消函数被自动内联或基于关键字内联的资格。

如果函数符合以下条件，编译器将保留调用：

- 具有与调用站点不同数量的参数
- 一个参数的类型与相应的调用站点参数不兼容
- 未声明为内联并返回 void 但需要其返回值

如果函数具有会给编译器带来困难情形的特性，编译器也不会内联调用：

- 具有可变长度的参数列表
- 从不返回
- 是超出深度限制的递归或非叶函数

- 未声明为内联且包含一条不是注释的 asm() 语句
- 是中断函数
- 是 main() 函数
- 未声明为内联，并且需要将过多的栈空间用于本地数组或结构体变量
- 包含易失性局部变量或参数
- 是包含 catch 的 C++ 函数
- 未在当前编译单元中定义且未使用 -O4 优化

无论其他指示如何（包括被调用函数上的 FUNC_ALWAYS_INLINE pragma 或 always_inline 属性），使用 NOINLINE pragma 注释的语句中的调用都不会被内联。

如果使用 FORCEINLINE pragma 注释的语句中的调用未因上述原因之一被取消资格，即使被调用函数具有 FUNC_CANNOT_INLINE pragma 或 cannot_inline 属性，则该调用都是被内联的。

换句话说，语句级 pragma 会覆盖函数级 pragma 或属性。如果 NOINLINE 和 FORCEINLINE 都适用于同一条语句，则首先出现的语句被使用，其余语句被忽略。

2.11.3 不受保护定义控制的内联

内联关键字导致函数在调用它的位置进行内联扩展，而不是使用标准调用过程。编译器对用内联关键字声明的函数执行内联扩展。

必须使用任何 --opt_level 选项来调用优化器以启用定义控制的内联。使用 --opt_level=3 时也会启用自动内联。

示例 2-1 使用内联关键字。函数调用将替换为被调用函数中的代码。

示例 2-1. 使用内联关键字

```
inline float volume_sphere(float r)
{
   return 4.0/3.0 * PI * r * r * r;
}
int foo(...)
{
   ...
   volume = volume_sphere(radius);
   ...
}
```

2.11.4 保护内联和 _INLINE 预处理器符号

将头文件中的函数声明为静态内联函数时，必须遵循额外的过程以避免在优化器未运行时出现潜在的代码大小增加问题。

为了防止头文件中的静态内联函数在关闭内联时导致代码大小增加，请执行以下程序。这允许在关闭内联时进行外部链接；这样一来，整个目标文件中只存在一个函数定义。

- 构建该函数的静态内联版本原型。然后，构建该函数的替代性非静态外部链接版本原型。使用 _INLINE 预处理器符号对这两个原型进行有条件的预处理，如示例 2-2 所示。
- 在 .c 或 .cpp 文件中创建相同版本的函数定义，如示例 2-3 所示。

在以下示例中，strlen 函数有两个定义。第一个定义（示例 2-2）位于头文件中，是内联定义。仅当 _INLINE 为真时使用优化器时会自动为您定义 _INLINE），该定义才启用，并且原型将声明为静态内联。

第二个定义（请参阅示例 2-3）用于库，确保在内联禁用时 strlen 的可调用版本存在。由于这不是内联函数，因此 _在包含 string.h 之前_INLINE 预处理器符号是未定义的(#undef)，（以生成 strlen 原型的非内联版本。

示例 2-2. 头文件 string.h

```
/****************************************************************************/
/* string.h  vx.xx                                                          */
/* Copyright (c) 1993-2006 Texas Instruments Incorporated                   */
/* Excerpted ...                                                            */
/****************************************************************************/
#ifdef _INLINE
#define _IDECL static inline
#else
#define _IDECL extern _CODE_ACCESS
#endif
_IDECL size_t  strlen(const char *_string);
#ifdef _INLINE
/****************************************************************************/
/*  strlen                                                                  */
/****************************************************************************/
static inline size_t strlen(const char *string)
{
   size_t      n = (size_t)-1;
   const char *s = string - 1;
   do n++; while (*++s);
   return n
}
#endif
```

示例 2-3. 库定义文件

```
/****************************************************************************/
/*  strlen                                                                  */
/****************************************************************************/
#undef _INLINE
#include <string>
_CODE_ACCESS size_t strlen(cont char * string)
{
   size_t      n = (size_t)-1;
   const char *s = string - 1;
   do n++; while (*++s);
   return n;
}
```

2.12 使用交叉列出功能

编译器工具包括将 C/C++ 源语句插入到编译器的汇编语言输出中的功能。交叉列出功能可用于检查为每个 C 语句生成的汇编代码。交叉列出的行为有所不同，具体取决于是否使用了优化器以及指定了哪些选项。

调用交叉列出功能的最简单方法是使用 --c_src_interlist 选项。要在名为 function.c 的程序上编译和运行交叉列出功能，请输入：

```
cl2000 --c_src_interlist function
```

--c_src_interlist 选项阻止编译器删除交叉列出的汇编语言输出文件。输出汇编文件 function.asm 被正常汇编。

在没有优化器的情况下调用交叉列出功能时，交叉列出将作为代码生成器与汇编器之间的单独通道运行。该功能读取汇编和 C/C++ 源文件，合并这些文件，然后将 C/C++ 语句作为注释写入汇编文件中。

有关将交叉列出功能与优化器一起使用的信息，请参阅节 3.10。使用 --c_src_interlist 选项会导致性能和/或代码大小下降。

以下示例显示了一个典型的交叉列出的汇编文件。

```
;*--------------------------------------------------------------------------
; 1 | int main()
;*--------------------------------------------------------------------------
;*************************************************************
;* FNAME: _main                         FR SIZE: 0           *
;*                                                           *
;* FUNCTION ENVIRONMENT                                      *
;*                                                           *
;* FUNCTION PROPERTIES                                       *
;*                           0 Parameter, 0 Auto, 0 SOE      *
;*************************************************************
_main:
;*--------------------------------------------------------------------------
; 3 | printf("Hello World\n");
;*--------------------------------------------------------------------------
        MOVL    XAR4,#SL1               ; |3|
        LCR     #_printf                ; |3|
        ; call occurs [#_printf] ; |3|
;*--------------------------------------------------------------------------
; 4 | return 0;
;*--------------------------------------------------------------------------
;*************************************************************
;* STRINGS *
;*************************************************************
        .sect   ".econst"
SL1:    .string "Hello World",10,0
;*************************************************************
;* UNDEFINED EXTERNAL REFERENCES *
;*************************************************************
.global _printf
```

2.13 关于应用程序二进制接口

应用程序二进制接口 (ABI) 定义了目标文件之间以及可执行文件与其执行环境之间的低级接口。ABI 允许将符合 ABI 的目标文件链接在一起，而不管其来源如何，并允许生成的可执行文件在支持 ABI 的任何系统上运行。

符合不同 ABI 的目标文件不能链接在一起。链接器会检测到这种情况并生成错误。

C28x 编译器支持这两种 ABI。ABI 是通过 --abi 选项选择的，如下所示：

- **COFF ABI** (`--abi=coffabi`) COFF ABI 是原始的 ABI 格式。这是默认值。
- **EABI** (`--abi=eabi`) 使用此选项可选择 C28x 嵌入式应用程序二进制接口 (EABI)。

应用程序中的所有代码都必须是针对相同的 ABI 构建的。在将 COFF ABI 应用程序迁移到 EABI 之前，请确保所有库都在 EABI 模式下可用。

更多有关 ABI 的详细信息，请参阅节 6.11。

2.14 启用入口挂钩和出口挂钩函数

入口挂钩是程序中每个函数进入时调用的例程。出口挂钩是每个函数退出时调用的例程。挂钩的应用包括调试、跟踪、分析和检查栈溢出。使用以下选项启用入口和出口挂钩：

--entry_hook[=*name*]	启用入口挂钩。若已指定，挂钩函数被称为 *name*。否则，默认的入口挂钩函数名称为 __entry_hook。
--entry_parm{=name|address|none}	指定挂钩函数的参数。name 参数指定调用函数的名称作为参数传递给挂钩函数。在这种情况下，挂钩函数的签名为：void hook(const char *name); address 参数指定调用函数的地址传递给挂钩函数。在这种情况下，挂钩函数的签名为：void hook(void (*addr)()); none 参数指定调用挂钩函数时不带参数。这是默认设置。在这种情况下，挂钩函数的签名为：void hook(void);
--exit_hook[=*name*]	启用出口挂钩。若已指定，挂钩函数被称为 *name*。否则，默认的出口挂钩函数名称为 __exit_hook。
--exit_parm{=name|address|none}	指定挂钩函数的参数。name 参数指定调用函数的名称作为参数传递给挂钩函数。在这种情况下，挂钩函数的签名为：void hook(const char *name); address 参数指定调用函数的地址传递给挂钩函数。在这种情况下，挂钩函数的签名为：void hook(void (*addr)()); none 参数指定调用挂钩函数时不带参数。这是默认设置。在这种情况下，挂钩函数的签名为：void hook(void);

挂钩选项的存在创建了带有给定签名的挂钩函数的隐式声明。如果挂钩函数的声明或定义出现在使用这些选项编译的编译单元中，则其必须与上面列出的签名一致。

在 C++ 中，挂钩声明为 extern "C"。因此，可以在 C（或汇编）中定义挂钩，而不必担心名称改编问题。

挂钩可以声明为内联，在这种情况下，编译器会尝试使用与其他内联函数相同的标准来内联这些挂钩。

入口挂钩和出口挂钩是相互独立的。可以启用一个但不启用另一个，或同时启用两个。同一个函数可以同时用作入口挂钩和作出口挂钩。

必须小心避免对挂钩函数进行递归调用。挂钩函数不应调用本身插入了挂钩调用的任何函数。为了防止这种情况，不会为内联函数或挂钩函数本身生成挂钩。

可以使用 --remove_hooks_when_inlining 选项删除优化器自动内联的函数的入口/出口挂钩。

有关 NO_HOOKS pragma 的信息，请参阅节 6.9.20。

2.15 实时固件更新 (LFU)

备注

仅在与 EABI 配合使用时支持该功能。在与 COFF ABI 配合使用时不支持该功能。

可能需要设计高可用性系统，以便可以在系统脱机的情况下升级固件。示例包括为数据中心、医院和军事应用的系统。在系统运行期间更新系统固件并开始使用新固件的功能称为实时固件更新 (LFU)，也称为“热”启动。有关创建和调用自定义入口点来执行热启动的信息，请参阅*采用 C2000 MCU 的实时固件更新参考设计* (TIDUEY4)。

C28x 和 CLA 编译器为使用 EABI 且基于 ELF 的固件映像提供 LFU 支持。该支持允许您可以切换到新的 LFU 映像，同时选择是保留、更新（重新初始化）还是添加从参考 ELF 二进制文件中读取的单个全局和静态符号。

为了支持 LFU，代码生成工具需要确保更新过程中不会发生系统复位，不会错过实时中断，并可以维持系统状态（全局和静态变量）。固件映像的程序和数据内存都可能需要更新。在热启动期间可以通过以下方式处理全局和静态变量：

- **保留：**在热启动之前保留存储在这些符号中的值。这些符号的地址与参考 ELF 映像中的地址相同。每个这样的符号都有一个 `.TI.bound` 段。如果 .TI.bound 段在存储器中是连续的，链接器可以将这些段合并到单个输出段中，从而减少初始化这些段所需的 CINIT 记录数量。（如果没有使用 --lfu_default 和任何属性来指定其他值，这将是默认值。但是，此默认值不适用于常量数据，常量数据需要显式使用 preserve 属性。）
- **更新：**在热启动期间重新初始化为这些符号存储的值。与参考 ELF 映像中的地址相比，这些符号的地址可能发生变化。此类符号由链接器收集到单个 `.TI.update` 输出段中。此段默认进行复制压缩（即，在热启动期间无需解压缩），这减少了 LFU 映像切换时间。
- **允许移动：**这些符号可以在热启动期间分配在任何内存地址上。变量不会重新初始化，因此它们的值是未规定的。只有当使用了`--lfu_default=none` 选项并且全局或静态变量既没有“preserve”属性也没有“update”属性时，才会发生这种行为。

使用 LFU 支持时存在以下限制：

- 必须使用 EABI。COFF 不支持 LFU。（所有 LFU 功能）
- 不支持手工编码的汇编。（仅限保留或更新）
- 调试信息必须保留在参考 ELF 映像中；默认情况下，此功能处于启用状态。参考映像中的符号表用于确定地址。（仅限保留）
- 在不同固件版本之间，C 文件名必须保持相同以支持保留。（仅限保留）

为支持 LFU 功能而提供的特征包括：

- `--lfu_reference_elf` 编译器选项，它指向先前的 ELF 可执行二进制文件，用作获取全局和静态符号的内存地址列表的参考。请参阅节 2.3。
- `--lfu_default` 编译器选项，可用于设置在热启动期间如何处理全局和静态符号的默认值。请参阅节 2.3。
- `preserve` 属性，可在 C 代码中用于指定应保留的单个符号的地址。请参阅节 6.15.4。
- `update` 属性，可在 C 代码中用于指定在热启动期间应重新初始化的单个符号。请参阅节 6.15.4。
- .TI.update 和 .TI.bound 段，链接器使用这些段来收集在热启动期间（分别）需要更新和保留的符号。请参阅节 7.1.1。
- __TI_auto_init_warm() RTS 例程，应在热启动期间从自定义入口点被调用。请参阅节 8.4。

章节 3

优化您的代码

编译器工具可以通过简化循环、重新排列语句和表达式以及将变量分配到寄存器中进行大量的优化，以加快执行速度并减小 C 和 C++ 程序的大小。

本章介绍如何调用不同级别的优化，并介绍在每个级别上哪些优化被执行。本章还介绍在执行优化时如何使用交叉列出功能，以及如何分析或调试优化的代码。

3.1 调用优化

C/C++ 编译器能够执行各种优化，这些优化由优化器和代码生成器执行：

优化器 在独立优化通道中执行高级别优化。使用更高的优化级别（例如 --opt_level=2 和 --opt_level=3）以获得最优代码。

代码生成器 执行多个额外的优化。这些是特定于目标的低级别优化。无论您是否调用优化器，代码生成器都会执行这些优化，并且这些优化会始终启用，不过在使用优化器时它们会更高效。

调用优化的最简单方法是使用编译器程序，在编译器命令行上指定 --opt_level=*n* 选项。您可以使用 -O*n* 作为 --opt_level 选项的别名。*n* 表示优化级别（0、1、2、3 和 4），其控制优化的类型和程度。

- **--opt_level=off** 或 **-Ooff**
  - 不执行优化
- **--opt_level=0** 或 **-O0**
  - 执行控制流图简化（节 3.16.3）
  - 将变量分配给寄存器（节 3.16.12）
  - 执行循环旋转（节 3.16.10）
  - 消除未使用的代码
  - 简化表达式和语句（节 3.16.5）
  - 扩展对声明的内联函数的调用（节 3.16.6）
- **--opt_level=1** 或 **-O1** 执行所有 --opt_level=0 (-O0) 优化，加上：
  - 执行本地复制/常量传播（节 3.16.4）
  - 删除未使用的赋值（节 3.16.4）
  - 消除局部公用表达式 (节 3.16.4)
- **--opt_level=2** 或 **-O2** 执行所有 --opt_level=1 (-O1) 优化，加上：
  - 执行循环优化
  - 消除全局公用子表达式（节 3.16.4）
  - 消除全局未使用的赋值（节 3.16.4）
  - 执行循环展开（节 6.9.24）
- **--opt_level=3** 或 **-O3** 执行所有 --opt_level=2 (-O2) 优化，加上：
  - 删除所有从未调用过的函数（节 3.4）
  - 简化返回值从未使用过的函数（节 3.4）
  - 内联函数对小函数的调用（节 2.11 和节 3.5）
  - 重新排序函数声明；当调用方被优化后，被调用函数的属性是已知的
  - 当所有调用在相同的参数位置传递相同的值时，将参数传播到函数体中
  - 识别文件级变量特征（节 3.4）
  - 执行其他优化（节 3.3 和节 3.4）
- **--opt_level=4** 或 **-O4**
  - 执行链接时优化。(节 3.6)

有关 --opt_level 和 --opt_for_speed 选项以及各种 pragma 如何影响内联的详细信息，请参阅节 2.11。

调试默认启用，并且优化级别不受调试信息生成的影响。

3.2 控制代码大小与速度

要在代码大小和速度之间实现平衡，请使用 --opt_for_speed 选项。优化级别 (0-5) 控制代码大小或代码速度优化的类型和程度：

- --opt_for_speed=0

  优化能够恶化或影响性能的风险*较为 高*的代码大小。
- --opt_for_speed=1

  优化能够恶化或影响性能的风险*中偏 中的代码大小*。
- --opt_for_speed=2

  优化能够恶化或影响性能的风险*较 低*的代码大小。
- --opt_for_speed=3

  优化能够恶化或影响代码大小的风险*较 低*的代码性能/速度。
- --opt_for_speed=4

  优化能够恶化或影响代码大小的风险*偏 中的代码性能/速度*。
- --opt_for_speed=5

  优化能够恶化或影响代码大小的风险*较为 高* 的代码性能/速度。

如果未使用参数指定 --opt_for_speed 选项，则默认设置为 --opt_for_speed=4。如果未指定 --opt_for_speed 选项，则默认设置为 2

用于控制代码空间的旧机制 --opt_for_space 选项与 --opt_for_speed 选项具有以下等效项：

--opt_for_space	--opt_for_speed
无	=4
=0	=3
=1	=2
=2	=1
=3	=0

使用 --opt_for_speed 选项时，默认情况下会生成快速分支 (BF) 指令。不使用 --opt_for_speed 时，编译器仅在条件代码为 NEQ、EQ、NTC 和 TC 之一时生成 BF 指令。原因是具有这些条件代码的 BF 可以优化为 SBF。当条件代码不是 NEQ、EQ、NTC 和 TC 之一时，使用 BF 指令会使代码大小有所损失。（还会为具有 ramfunc 函数属性的函数生成快速分支指令。）

--no_fast_branch 选项已被弃用，不起作用。

3.3 执行文件级优化（--opt_level=3 选项）

--opt_level=3 选项（别名为 -O3 选项）指示编译器执行文件级优化。可以单独使用 --opt_level=3 选项来执行一般的文件级优化，也可以将该选项与其他选项结合使用以执行更具体的优化。表 3-1 中列出的选项与 --opt_level=3 一起使用以执行指定的优化：

表 3-1. 可与 --opt_level=3 结合使用的选项

如果您...	使用此选项	请参阅
希望创建优化信息文件	--gen_opt_level=*n*	节 3.3.1
希望编译多个源文件	--program_level_compile	节 3.4

3.3.1 创建优化信息文件（--gen_opt_info 选项）

使用 --opt_level=3 选项调用编译器时，可以使用 --gen_opt_info 选项创建一个可以阅读的优化信息文件。选项后面的数字表示级别（0、1 或 2）。生成的文件具有 .nfo 扩展名。请根据表 3-2 选择相应的级别以附加到该选项。

表 3-2. 为 --gen_opt_info 选项选择一个级别

如果您...	使用此选项
不希望生成信息文件，但在命令文件或环境变量中使用了 --gen_opt_level=1 或 --gen_opt_level=2 选项。--gen_opt_level=0 选项恢复优化器的默认行为。	--gen_opt_info=0
希望生成优化信息文件	--gen_opt_info=1
希望生成详细的优化信息文件	--gen_opt_info=2

3.4 程序级优化（--program_level_compile 和 --opt_level=3 选项）

可以通过使用 --program_level_compile 选项和 --opt_level=3 选项（别名为 -O3）来指定程序级优化。（如果使用 --opt_level=4 (-O4)，则不能使用 --program_level_compile 选项，因为链接时优化提供了与程序级优化相同的优化机会。）

通过程序级优化，所有源文件都会编译成称为*模块*的中间文件。该模块会转入到编译器的优化和代码生成阶段。由于编译器可以看到整个程序，因此其会执行一些在文件级优化中很少应用的优化：

- 如果函数中的特定参数总是具有相同的值，则编译器将参数替换为该值，并传递该值而不是该参数。
- 如果函数中的返回值从未被使用，则编译器将删除函数中的返回代码。
- 如果 函数未被 main() 直接或间接调用，则编译器将删除该函数。

--program_level_compile 选项要求使用 --opt_level=3 或更高版本，以便执行这些优化。

要查看编译器正在应用哪些程序级优化，请使用 --gen_opt_level=2 选项来生成信息文件。有关更多 信息，请参阅节 3.3.1。

在 Code Composer Studio 中，当使用 --program_level_compile 选项时，具有相同选项的 C 和 C++ 文件将被一起编译。但是，如果任何文件具有未被选为项目范围选项的文件专用选项，则该文件将被单独编译。例如，如果项目中的每个 C 和 C++ 文件都有一组不同的文件专用选项，则即使已指定了程序级优化，也会单独编译每个文件。要将所有的 C 和 C++ 文件一起编译，请确保这些文件没有文件专用选项。请注意，如果先前使用了文件专用选项，则将 C 和 C++ 文件一起编译可能不安全。

备注

使用 --program_level_compile 和 --keep_asm 选项编译文件

如果使用--program_level_compile 和--keep_asm 选项编译所有文件，则编译器只会生成一个 .asm 文件，而不是为每个对应的源文件都生成一个。

3.4.1 控制程序级优化（--call_assumptions 选项）

可以使用--call_assumptions 选项控制由--program_level_compile --opt_level=3 调用的程序级优化。具体而言，--call_assumptions 选项表示其他模块中的函数是否可以调用模块的外部函数或修改模块的外部变量。--call_assumptions 后面的数字表示您为允许调用或修改的模块而设置的级别。--opt_level=3 选项将此信息与其自身的文件级分析相结合，以决定是否将该模块的外部函数和变量声明视为静态声明。使用表 3-3 选择合适的级别以附加到 --call_assumptions 选项。

表 3-3. 为 --call_assumptions 选项选择一个级别

如果模块...	使用此选项
具有从其他模块调用的函数以及在其他模块中修改的全局变量	--call_assumptions=0
不具有由其他模块调用的函数，但具有在其他模块中修改的全局变量	--call_assumptions=1
不具有由其他模块调用的函数，也不具有在其他模块中修改的全局变量	--call_assumptions=2
具有从其他模块调用的函数，但不具有在其他模块中修改的全局变量	--call_assumptions=3

在某些情况下，编译器恢复到与指定级别不同的 --call_assumptions 级别，或者可能完全禁用程序级优化。表 3-4 列出了 --call_assumptions 级别与导致编译器恢复到其他 --call_assumptions 级别的条件的组合。

表 3-4. 使用 --call_assumptions 选项时的特殊注意事项

如果 --call_assumptions 为...	在以下条件下...	则 --call_assumptions 级别...
未指定	指定了 --opt_level=3 优化级别	默认为 --call_assumptions=2
未指定	编译器在 --opt_level=3 优化级别下发现对外部函数的调用	恢复为 --call_assumptions=0
未指定	未定义 main	恢复为 --call_assumptions=0
--call_assumptions=1 或 --call_assumptions=2	没有将 main 定义为入口点的函数，也没有定义中断函数，也 没有由 FUNC_EXT_CALLED pragma 标识的函数	恢复为 --call_assumptions=0
--call_assumptions=1 或 --call_assumptions=2	定义了 main 函数，*或* 定义了中断函数，*或* 者用 FUNC_EXT_CALLED pragma 标识了函数	保留 --call_assumptions=1 或 --call_assumptions=2
--call_assumptions=3	任何条件下	保留 --call_assumptions=3

在某些情况下，使用 --program_level_compile 和 --opt_level=3 时，则*必须* 使用 --call_assumptions 选项或 FUNC_EXT_CALLED pragma。有关这些情况的信息，请参阅节 3.4.2。

3.4.2 混合 C/C++ 和汇编代码时的优化注意事项

如果程序中有任何汇编函数，则在使用 --program_level_compile 选项时，请谨慎操作。编译器只识别 C/C++ 源代码，而不识别任何可能存在的汇编代码。由于编译器无法识别对 C/C++ 函数的汇编代码调用和变量修改，因此 --program_level_compile 选项会优化这些 C/C++ 函数。要保留这些函数，请将 FUNC_EXT_CALLED pragma（请参阅节 6.9.13）放在对要保留的函数的任何声明或引用之前。

在程序中使用汇编函数时可以采用的另一种方法是将 --call_assumptions=*n* 选项与 --program_level_compile 及 --opt_level=3 选项结合使用。有关--call_assumptions=*n* 选项的信息，请参阅节 3.4.1。

通常，采用明智的方式将 FUNC_EXT_CALLED pragma 与 --program_level_compile --opt_level=3 及 --call_assumptions=1 或 --call_assumptions=2 结合使用，可以获得理想结果。

如果您的应用程序出现以下任一情况，请使用建议的解决方案：

- **情况：**您的应用程序由调用汇编函数的 C/C++ 源代码组成。这些汇编函数不调用任何 C/C++ 函数或修改任何 C/C++ 变量。

  解决方案：使用 --program_level_compile --opt_level=3 --call_assumptions=2 进行编译，通知编译器：外部函数不会调用 C/C++ 函数，也不会修改 C/C++ 变量。

  如果仅使用 --program_level_compile --opt_level=3 选项进行编译，编译器会从默认优化级别 (--call_assumptions=2) 恢复到 --call_assumptions=0。编译器使用 --call_assumptions=0 ，因为编译器假定调用在 C/C++ 中定义的汇编语言函数可能会调用其他 C/C++ 函数或修改 C/C++ 变量。
- **情况：**您的应用程序由调用汇编函数的 C/C++ 源代码组成。汇编语言函数不会调用 C/C++ 函数，但会修改 C/C++ 变量。

  解决方案：尝试以下两种解决方案，然后选择最适合您的代码的一种解决方案：
  - 使用 --program_level_compile --opt_level=3 --call_assumptions=1 进行编译。
  - 将 volatile 关键字添加到可能被汇编函数修改的变量中，并使用 --program_level_compile --opt_level=3 --call_assumptions=2 进行编译。
- **情况：**您的应用程序由 C/C++ 源代码和汇编源代码组成。汇编函数是调用 C/C++ 函数的中断服务例程；汇编函数调用的 C/C++ 函数永远不会从 C/C++ 调用。这些 C/C++ 函数的作用类似于 main：充当 C/C++ 的入口点。

  解决方案：将 volatile 关键字添加到可能被中断修改的 C/C++ 变量中。然后，可以通过以下方式之一优化代码：
  - 通过将 FUNC_EXT_CALLED pragma 应用于从汇编语言中断调用的所有入口点函数，然后使用 --program_level_compile --opt_level=3 --call_assumptions=2 进行编译，可以实现理想优化。*请确保将该 pragma 与所有入口点函数一起使用。*如果不这样做，编译器可能会删除前面没有 FUNC_EXT_CALLED pragma 标记的入口点函数。
  - 使用 --program_level_compile --opt_level=3 --call_assumptions=3 进行编译。由于不使用 FUNC_EXT_CALLED pragma，因此必须使用 --call_assumptions=3 选项，该选项不如 --call_assumptions=2 选项激进，因而优化可能没有那么高效。

  请记住，如果不使用附加选项而使用了 --program_level_compile --opt_level=3，编译器会删除汇编函数调用的 C 函数。请使用 FUNC_EXT_CALLED pragma 保留这些函数。

3.5 自动内联扩展（--auto_inline 选项）

当使用 --opt_level=3 选项（别名为 -O3）进行优化时，编译器自动内联小函数。命令行选项 --auto_inline=*size* 指定大小阈值。任何大于 *size* 阈值的函数都不会自动被内联。可以通过以下方式使用 --auto_inline=*size* 选项：

- 如果将 *size* 参数设置为 0 (--auto_inline=0)，则禁用自动内联扩展。
- 如果将 *size* 参数设置为非零整数，则编译器将使用此大小阈值限制其自动内联的函数的大小。编译器将函数内联的次数（如果函数在外部可见且其声明无法安全删除，则加 1）乘以函数的大小。

仅当结果小于 size 参数时，编译器才内联函数。编译器以任意单位测量函数的大小；但是，优化器信息文件（使用 --gen_opt_level=1 或 --gen_opt_level=2 选项创建）报告 --auto_inline 选项使用的相同单位中每个函数的大小。

--auto_inline=*size* 选项仅控制未明确声明为内联 的函数的内联。如果不使用 --auto_inline=*size* 选项，则编译器内联非常小的函数。

有关影响内联的命令行选项、pragma 和关键字之间的交互信息，请参阅节 2.11。

备注

优化级别 3 和内联：为了打开自动内联，必须使用 --opt_level=3 选项。如果需要 --opt_level=3 优化，但不想自动内联，请使用 --auto_inline=0 和 --opt_level=3 选项。

备注

内联和代码大小：内联扩展函数会增加代码大小，尤其是内联在多个地方被调用的函数。对于仅在少数地方被调用的函数以及小函数来说，函数内联是最优的。为了防止由于内联而增加代码大小，请使用 --auto_inline=0 选项。此选项使编译器仅内联内在函数。

3.6 链接时优化（--opt_level=4 选项）

链接时优化是一种优化模式，让编译器对整个程序具有可见性。优化发生在链接时，而不是像其他优化级别那样发生在编译时。

应使用 --opt_level=4 选项调用链接时优化。此选项必须放在命令行上的 --run_linker (-z) 选项*之前*，因为编译器和链接器都会参与链接时优化。在编译时，编译器将正在编译的文件的中间表示形式嵌入到生成的目标文件中。在链接时，从包含此表示形式的每个目标文件中提取此表示形式，并用于优化整个程序。

如果使用 --opt_level=4 (-O4)，则不能同时使用 --program_level_compile 选项，因为链接时优化提供了与程序级优化相同的优化机会（节 3.4）。链接时优化具有以下优点：

- 每个源文件都可以单独编译。程序级编译的一个问题是其要求所有源文件都要一次性传递给编译器。这通常需要对客户的构建过程进行重大修改。使用链接时优化，所有文件都可以单独编译。
- 自动处理对程序集的 C/C++ 符号的引用。在进行程序级编译时，编译器不知道符号是否被外部引用。当在最后一个链接中执行链接时优化时，链接器可以确定哪些符号被外部引用，并在优化过程中防止消除这些符号。
- 第三方目标文件可以参与优化。如果第三方供应商提供了使用 --opt_level=4 选项编译的目标文件，这些文件将与用户生成的文件一起参与优化。这包括作为 TI 运行时支持的一部分提供的目标文件。未使用 –opt_level=4 编译的目标文件仍可在执行链接时优化的链接中使用。未使用 –opt_level=4 进行编译的那些文件则不参与优化。
- 可以使用不同的选项集编译源文件。对于程序级编译，必须使用相同的选项集编译所有源文件。借助链接时优化，可以使用不同的选项来编译文件。如果编译器确定两个选项不兼容，就会发出错误。

3.6.1 选项处理

在执行链接时优化时，可以使用不同的选项来编译源文件。如果可能，编译期间使用的选项将在链接时优化期间使用。对于适用于程序级的选项，例如 --auto_inline，则使用用于编译 main 函数的选项。如果 main 未包含在链接时优化中，则使用命令行上指定的第一个目标文件所使用的选项集。一些选项，例如 --opt_for_speed，可以影响很大范围的优化。对于这些选项，程序级行为是从 main 派生出来的，而局部优化是从原始选项集得到的。

执行链接时优化时，有些选项是不兼容的。这些选项通常也会在命令行上产生冲突，但也可能是在链接时优化期间无法处理的选项。

3.6.2 不兼容的类型

在正常链接期间，链接器并不会检查以确保在不同的文件中使用相同的类型声明每个符号。这在正常链接期间不是必要的。但是，在执行链接时优化时，链接器必须确保在不同的源文件中使用兼容的类型声明所有的符号。如果发现具有不兼容类型的符号，则会发出错误。兼容类型的规则源自 C 和 C++ 标准。

3.7 使用反馈制导优化

反馈制导优化提供了一种方法，其使用基于编译器的检测在应用程序中查找频繁执行的路径。此信息反馈给编译器并用于执行优化。此信息还用于为您提供有关应用程序行为的信息。

3.7.1 反馈向导优化

反馈制导优化使用运行时反馈来识别和优化频繁执行的程序路径。反馈制导优化是一个两阶段的过程。

3.7.1.1 第 1 阶段 - 收集程序分析信息

此阶段使用选项 --gen_profile_info 调用编译器。该选项指示编译器添加检测代码以收集分析信息。编译器插入最少量的检测代码来确定控制流频率。分配内存以存储计数器信息。

使用代表性的输入数据集在目标上执行检测的应用程序。输入数据集应与程序在最终产品环境中的预期使用方式密切相关。程序完成后，运行时支持函数将收集的信息写入称为 PDAT 文件的分析数据文件中。可以使用不同的输入数据集多次执行程序，在这种情况下，运行时支持函数将收集到的信息附加到 PDAT 文件中。使用称为 Profile Data Decoder 或 pdd2000 的工具对生成的 PDAT 文件进行后处理。pdd2000 工具整合多个数据集，并将数据格式化为反馈文件（PRF 文件，请参阅节 3.7.2），供反馈制导优化的第 2 阶段使用。

3.7.1.2 第 2 阶段 - 使用应用程序分析信息进行优化

此阶段使用 --use_profile_info=*file.prf* 选项调用编译器，该选项读取在第 1 阶段中生成的指定 PRF 文件。第 2 阶段使用第 1 阶段生成的数据做出优化决策。使用分析反馈文件指导程序优化。编译器更积极地优化频繁执行的程序路径。

编译器使用分析反馈文件中的数据来指导对频繁执行的程序路径进行某些优化。

3.7.1.3 生成和使用配置文件信息

有两个选项可以控制反馈定向优化：

--gen_profile_info

告知编译器添加检测代码以收集配置文件信息。当程序执行 run-time-support exit() 函数时，配置文件数据会被写入 PDAT 文件。此选项适用于在命令行上编译的所有 C/C++ 源文件。

如果设置了主机上的环境变量 TI_PROFDATA，则将数据写入指定的文件中。否则，它使用默认文件名：pprofout.pdat。可以使用 TI_PROFDATA 主机环境变量指定 PDAT 文件的完整路径名（包括目录名）。

默认情况下，RTS 配置文件数据输出例程使用 C I/O 机制将数据写入 PDAT 文件。您可以为 PPHNDL 器件安装器件处理程序，以将配置文件数据重定向到自定义器件驱动程序例程。例如，这可用于将配置文件数据发送到不使用文件系统的器件。

反馈定向优化要求您在使用 --gen_profile_info 选项时至少打开一些调试信息。这使编译器能够输出调试信息，以允许 pdd2000 关联已编译的函数及其相关配置文件数据。

--use_profile_info

指定用于执行反馈定向优化的第 2 阶段的配置文件信息文件。可以在命令行上指定多个配置文件信息文件；编译器使用来自多个信息文件的所有输入数据。此选项的语法为：

--use_profile_info==*file1*[, *file2*, ..., *filen*]

如果未指定文件名，编译器将在调用编译器的目录中查找名为 pprofout.prf 的文件。

3.7.1.4 反馈制导优化的应用示例

这些步骤说明了反馈制导优化的创建和应用。

1. 生成分析信息。

```
cl2000 --opt_level=2 --gen_profile_info foo.c --run_linker --output_file=foo.out
     --library=lnk.cmd --library=rts2800_ml.lib
```

2. 执行应用程序。

   执行应用程序时会在当前（主机）目录中创建一个名为 pprofout.pdat 的 PDAT 文件。应用程序可以在连接到主机的目标硬件上运行。

3. 处理分析数据。

   在使用多个数据集运行应用程序后，在 PDAT 文件上运行 pdd2000 以创建与--use_profile_info 一起使用的分析信息（PRF）文件。

```
pdd2000 -e foo.out -o pprofout.prf pprofout.pdat
```

4. 使用分析反馈文件重新编译。

```
cl2000 --opt_level=2 --use_profile_info=pprofout.prf foo.c --run_linker
     --output_file=foo.out --library=lnk.cmd --library=rts2800_ml.lib
```

3.7.1.5 .ppdata 段

第 1 阶段收集的分析信息存储在 *.ppdata* 段中，而该段必须分配到目标存储器中。.ppdata 段包含使用 --gen_profile_info 进行编译的所有函数的分析器计数器。默认的 lnk.cmd 文件具有将 .ppdata 段放置在数据存储器中的指令。如果链接命令文件中没有用于分配 .ppdata 段的段指令，则链接步骤会将 .ppdata 段放置在可写存储器范围中。

必须以 32 字节的倍数为 .ppdata 段分配内存。请参阅分发中的链接器命令文件以了解示例用法。

3.7.1.6 反馈制导优化和代码大小调整

反馈制导优化与 Code Composer Studio (CCS) 中的代码大小调整 (Code Size Tune) 功能不同。代码大小调整功能使用 CCS 分析功能为每个函数选择特定的编译选项，以便在保持特定性能点的同时最小化代码大小。代码大小调整是粗粒度优化功能，因为它会为整个函数选择一个选项集。反馈制导优化功能沿函数内的特定区域选择不同的优化目标。

3.7.1.7 检测程序执行开销

在收集分析数据期间，应用程序的执行时间可能会延长。延长长度取决于应用程序的大小以及应用程序中为了分析而编译的文件数。

分析计数器会增加应用程序的代码和数据大小。在使用分析信息时，请考虑使用 选项来缓解代码大小增加的问题。这对正在收集的分析数据的准确性没有影响。由于分析仅计算执行频率而不计算周期数，因此代码大小优化标志不会影响分析器的测量。

3.7.1.8 无效的分析数据

使用 --use_profile_info 重新编译时，在以下情况中，分析信息无效：

- 源文件名在生成分析信息 (gen-profile) 与使用分析信息 (use-profile) 之间发生了变化。
- 自 gen-profile 以来修改了源代码。在这种情况下，分析信息对于修改后的函数无效。
- 与 gen-profile 搭配使用的某些编译器选项不同于与 use-profile 搭配使用的编译器选项。特别是，影响解析器行为的选项可能会在 use-profile 期间使分析数据无效。一般来说，在 use-profile 期间使用不同的优化选项应该不会影响分析数据的有效性。

3.7.2 分析数据解码器

代码生成工具包括称为 分析数据解码器或 pdd2000 的工具，该工具用于对分析数据 (PDAT) 文件进行后处理。pdd2000 工具生成分析反馈 (PRF) 文件。有关分析流程的哪个部分适合使用 pdd2000 的讨论，请参阅节 3.7.1。使用以下语法调用 pdd2000 工具：

pdd2000 -e *exec.out* **-o** *application.prf filename* **.pdat**

-a	计算数据集内数据值的平均值而不是累加数据值
-e *exec.out*	指定 *exec.out* 是应用程序可执行文件的名称。
-o *application.prf*	指定 *application.prf* 是格式化的分析反馈文件，在重新编译期间用作 --use_profile_info 的参数。如果未指定输出文件，则默认输出文件名为 pprofout.prf。
filename **.pdat**	是由运行时支持函数生成的分析数据文件的名称。这是默认名称，可以使用主机环境变量 TI_PROFDATA 将其覆盖。

运行时支持函数和 pdd2000 附加到各自的输出文件中，并且不会覆盖它们。这样就可以从应用程序的多次运行中收集数据集。

备注

Profile Data Decoder 要求：至少使用 DWARF 调试支持对应用程序进行编译，才能启用反馈定向优化。在针对反馈定向优化进行编译时，pdd2000 工具依赖于有关每个函数的基本调试信息来生成格式化的 .prf 文件。

运行时支持生成的 pprofout.pdat 文件是格式固定的原始数据文件，只有 pdd2000 才能理解这种格式。不应以任何方式修改此文件。

3.7.3 反馈制导优化 API

分析器机制有两个用户界面。可以使用以下运行时支持调用在应用程序中启动和停止分析。

- **_TI_start_pprof_collection()：**此接口通知运行时支持，即您希望从此时起开始进行分析收集，并使运行时支持清除应用程序中的所有分析计数器（即丢弃旧的计数器值）。
- **_TI_stop_pprof_collection()：**此接办指定运行时支持停止分析收集并将分析数据输出到输出文件（输出到默认文件或由 TI_PROFDATA 主机环境变量指定的文件）。除非您再次调用 _TI_start_pprof_collection()，否则运行时支持还会在 exit() 期间禁止将分析数据进一步输出到输出文件中。

3.7.4 反馈制导优化总结

选项

--gen_profile_info	将检测添加到已编译的代码中。执行该代码的结果是将分析数据发送到 PDAT 文件。
--use_profile_info=*file.prf*	使用分析信息进行优化和/或生成代码覆盖率信息。
--analyze=codecov	生成代码覆盖率信息文件并继续基于分析进行编译。必须与 --use_profile_info 一起使用。
--analyze_only	仅生成代码覆盖率信息文件。必须与 --use_profile_info 一起使用。同时指定 --analyze=codecov 和 --analyze_only 才能对检测的应用程序进行代码覆盖率分析。

主机环境变量

TI_PROFDATA	将分析数据写入指定的文件中
TI_COVDIR	在指定的目录中创建代码覆盖率文件
TI_COVDATA	将代码覆盖率数据写入指定的文件中

API

_TI_start_pprof_collection()	清除要归档的分析计数器
_TI_stop_pprof_collection()	将所有分析计数器写入文件中
PPHDNL	设备驱动程序句柄，用于从目标程序中写出分析数据的低级别 C I/O 驱动程序。

创建的文件

*.pdat	分析数据文件，通过执行检测的程序创建的，并作为分析数据解码器的输入
*.prf	分析反馈文件，由分析数据解码器创建的，并作为重新编译步骤的输入

3.8 使用配置文件信息分析代码覆盖率

可以使用来自 Profile Data Decoder 的分析信息来分析代码覆盖率。

3.8.1 代码覆盖

反馈定向优化期间收集的信息可用于生成代码覆盖率报告。与反馈定向优化一样，程序必须使用 --gen_profile_info 选项进行编译。代码覆盖率使用评测期间收集的数据，传递正在编译的文件中每行源代码的执行计数。

3.8.1.1 第 1 阶段 - 收集程序分析信息

此阶段使用 --gen_profile_info 调用编译器，该选项指示编译器添加检测代码以收集分析信息。编译器插入最少量的检测代码来确定控制流频率。分配内存以存储计数器信息。

使用代表性的输入数据集在目标上执行检测的应用程序。输入数据集应与程序在最终产品环境中的预期使用方式密切相关。程序完成后，运行时支持函数将收集的信息写入称为 PDAT 文件的分析数据文件中。可以使用不同的输入数据集多次执行程序，在这种情况下，运行时支持函数会将收集到的信息附加到 PDAT 文件中。使用称为 Profile Data Decoder 或 pdd2000 的工具对生成的 PDAT 文件进行后处理。pdd2000 工具整合多个数据集，并将数据格式化为反馈文件（PRF 文件，请参阅节 3.7.2），供反馈制导优化的第 2 阶段使用。

3.8.1.2 第 2 阶段 -- 生成代码覆盖信息报告

此阶段使用 --use_profile_info=*file.prf* 选项调用编译器。该选项指示编译器应读取在第 1 阶段中生成的指定 PRF 文件。应用还必须使用 --codecov 或 --onlycodecov 选项进行编译；编译器生成代码覆盖信息文件。--codecov 选项指示编译器在生成代码覆盖信息后继续编译，而 --onlycodecov 选项在生成代码覆盖数据后停止编译器。例如：

```
cl2000 --opt_level=2 --use_profile_info=pprofout.prf --onlycodecov foo.c
```

可以指定两个环境变量来控制代码覆盖信息文件的目标。

- TI_COVDIR 环境变量指定应生成代码覆盖文件的目录。默认是调用编译器的目录。
- TI_COVDATA 环境变量指定编译器生成的代码覆盖数据文件的名称。默认为 *filename*.csv，其中 filename 是正在编译的文件的基址名。例如，如果正在编译 foo.c，则默认的代码覆盖数据文件名是 foo.csv。

如果代码覆盖数据文件已存在，编译器会在文件末尾附加新数据集。

代码覆盖率数据是以逗号分隔的数据项列表，可以方便地由数据处理工具和脚本语言进行处理。代码覆盖数据的格式如下：

"*filename-with-full-path*","*funcname*",*line#*,*column#*,*exec-frequency*,"*comments*"

"*filename-with-full-path*"	条目对应的文件的完整路径名
"*funcname*"	函数的名称
line#	频率数据对应的源代码行行号
column#	源代码行的列号
exec-frequency	行的执行频率
"*comments*"	解析器生成的源代码的中间表示

完整的文件名、函数名和注释用引号 (") 引起来。例如：

```
"/some_dir/zlib/c2000/deflate.c","_deflateInit2_",216,5,1,"( strm->zalloc )"
```

可使用其他工具（例如电子表格程序）来格式化和查看代码覆盖数据。

3.8.2 相关的特征和功能

代码生成工具提供了一些可与代码覆盖率分析结合使用的特征和功能。这些特征和功能综述如下：

3.8.2.1 路径分析器

代码生成工具包括路径分析实用程序 pprof2000，该程序是从编译器 cl2000 运行的。从编译器命令行使用 --gen_profile 或 --use_profile 命令时，编译器会调用 pprof2000 实用程序：

cl2000 --gen_profile ... *file.c*
cl2000 --use_profile ... *file.c*

有关基于分析优化的更多信息以及分析基础架构的更多详细说明，请参阅节 3.7。

3.8.2.2 分析选项

路径分析实用程序 pprof2000 将代码覆盖率信息附加到包含同类型分析信息的现有 CSV（逗号分隔值）文件中。

该实用程序检查以确保现有 CSV 文件包含与要求生成的分析信息类型一致的分析信息。如果尝试在同一输出 CSV 文件中混合代码覆盖率和其他分析信息的行为被检测到，则 pprof2000 将发出致命错误并中止。

--analyze=codecov 指示编译器生成代码覆盖率分析信息。此选项取代了先前的 --codecov 选项。

--analyze_only 在分析信息生成完成后停止编译。

3.8.2.3 环境变量

为了协助管理输出 CSV 分析文件，pprof2000 支持以下环境变量：

TI_ANALYSIS_DIR 指定输出分析文件将在其内生成的目录。

3.9 使用优化时的特殊注意事项

编译器旨在改进符合 ANSI/ISO 标准的 C 和 C++ 程序，同时保持其正确性。但是，在编写用于优化的代码时，应注意以下各节中讨论的特殊注意事项，以确保程序按预期执行。

3.9.1 在优化代码中谨慎使用 asm 语句

在优化代码中使用 asm（内联汇编）语句时必须非常小心。编译器会重新排列代码段，自由使用寄存器，并可以彻底删除变量或表达式。尽管编译器从不会优化 asm 语句（除非无法访问），但插入了汇编代码的周围环境可能与原始 C/C++ 源代码会有很大的不同。

使用 asm 语句来操作硬件控制（例如中断屏蔽）通常是安全的做法，但是试图与 C/C++ 环境进行交互或访问 C/C++ 变量的 asm 语句可能会产生意想不到的结果。编译后，检查汇编输出以确保 asm 语句正确并保持程序的完整性。

3.9.2 使用易失性关键字进行必要的内存访问

编译器分析数据流以尽可能地避免访问内存。如果代码完全依赖于 C/C++ 代码中编写的内存访问，则必须使用易失性关键字来标识这些访问。编译器不会优化任何对易失性变量的引用。

在以下示例中，循环等待一个位置被读取为 0xFF：

```
unsigned int *ctrl;
while (*ctrl !=0xFF);
```

在此示例中，*ctrl 是循环不变表达式，因此该循环会优化为单个内存读取。要更正此问题，请将 ctrl 声明为：

```
volatile unsigned int *ctrl;
```

3.9.2.1 访问别名变量时的注意事项

当可以通过多种方式访问单个对象时，例如当两个指针指向同一个对象或一个指针指向一个命名对象时，便会出现别名。别名会破坏优化，这是因为任何间接引用都可以引用另一个对象。编译器分析代码以确定哪里可以出现别名，哪里不可以出现别名，然后在保持程序正确性的同时尽可能地进行优化。编译器谨慎作为。

编译器假定，如果将局部变量的地址传递给函数，则该函数可能通过指针写入来更改局部变量，但返回后不会将局部变量的地址提供给其他地方使用。例如，被调用函数不能将局部变量的地址分配给全局变量或返回该地址。如果此假定无效，请使用 -ma 编译器选项迫使编译器假定最坏情况的别名。在最坏情况的别名中，任何间接引用（即，使用指针）都可以引用这样的变量。

3.9.2.2 使用 --aliased_variables 选项来指示采用了以下技术

通过优化调用时，编译器会假定当任何变量的地址作为参数传递到函数时，所调用函数中设置的别名都不会对这些变量进行后续修改。示例包括：

- 从函数返回地址
- 将地址分配给一个全局变量

如果您在代码中使用类似这样的别名，则必须在优化代码时使用 --aliased_variables 选项。例如，如果您的代码与下面的类似，请使用 --aliased_variables 选项。

```
int *glob_ptr;
g()
{
    int  x = 1;
    int *p = f(&x);

    *p        = 5;   /* p aliases x        */
    *glob_ptr = 10;  /* glob_ptr aliases x */
    h(x);
}

int *f(int *arg)
{
    glob_ptr = arg;
    return arg;
}
```

3.9.2.3 仅在 FPU 目标上：使用 restrict 关键字指示指针没有别名

在 FPU 目标上，使用 --opt_level=2，优化器进行依赖性分析。为了帮助编译器确定内存器依赖关系，可以使用 restrict 关键字限定指针、引用或数组。restrict 关键字是一种类型限定符，可以应用于指针、引用和数组。关键字的使用代表程序员的保证：在指针声明的范围内，所指向的对象只能由该指针访问。任何违反此保证的行为都会导致程序未定义。这种做法可以帮助编译器优化某些代码段，因为这样可以更加轻松地确定别名信息。由于更多的 FPU 运算可以被并行化，因此可以提高性能和增加代码大小。

如示例 3-1 和示例 3-2 所示，可以使用 restrict 关键字通知编译器：a 和 b 永远不会指向 foo 中的同一个对象。此外，还会向编译器保证 a 和 b 所指向的对象在内存中不会重叠。

示例 3-1. restrict 类型限定符与指针搭配使用

```
void foo(float * restrict a, float * restrict b)
{
   /* foo's code here */
}
```

示例 3-2. restrict 类型限定符与指针搭配使用

```
void foo(float c[restrict], float d[restrict])
{
   /* foo's code here */
}
```

3.10 通过优化使用交叉列出特性

使用 --optimizer_interlist 和 --c_src_interlist 选项进行优化（ --opt_level=*n* 或 -O*n* 选项）编译时，可以控制交叉列出特性的输出。

- --optimizer_interlist 选项将编译器注释与汇编源语句交叉列出。

- --c_src_interlist 和 --optimizer_interlist 选项一起将编译器注释和原始 C/C++ 源代码与汇编代码交叉列出。

当 --optimizer_interlist 选项与优化一起使用时，交叉列出功能*不会*单独运行。相反，编译器会在代码中插入注释，指示编译器已如何重新排列和优化代码。这些注释在汇编语言文件中以 ;** 开头显示。除非也使用了 --c_src_interlist 选项，否则不会交叉列出 C/C++ 源代码。

交叉列出功能会影响优化代码，因为其可能会阻止某些优化跨越 C/C++ 语句边界。优化使正常的源代码交叉列出变得不切实际，因为编译器会大幅度重新排列程序。因此，使用 --optimizer_interlist 选项时，编译器会编写重构的 C/C++ 语句。

备注

对性能和代码大小的影响：--c_src_interlist 选项可能会对性能和代码大小产生负面影响。

当 --c_src_interlist 和 --optimizer_interlist 选项与优化一起使用时，编译器会插入其注释，并且交叉列出功能在汇编器之前运行，从而将原始 C/C++ 源代码合并到汇编文件中。例如，假设下述 C 代码是使用优化 (--opt_level=2) 和 --optimizer_interlist 选项编译的：

```
int copy (char *str, const char *s, int n)
{
   int i;
   for (i = 0; i < n; i ++)
      *str++ = *s++;
}
```

汇编文件包含与汇编代码交叉列出的编译器注释。

```
;****************************************************************
;* FNAME: _copy                         FR SIZE: 0              *
;* FUNCTION ENVIRONMENT                                         *
;* FUNCTION PROPERTIES                                          *
;*                            0 Parameter, 0 Auto, 0 SOE        *
;****************************************************************
_copy:
;*** 6  -----------------------    if ( n <= 0 ) goto g4;
        CMPB      AL,#0                 ; |6|
        B         L2,LEQ                ; |6|
        ; branch occurs ; |6|
;***    -----------------------    #pragma MUST_ITERATE(1, 4294967295, 1)
:***    -----------------------    L$1 = n-1;
        ADDB      AL,#-1
        MOVZ      AR6,AL
L1:
;***    -----------------------g3:
;*** 7  -----------------------    *str++ = *s++;
;*** 7  -----------------------    if ( (--L$1) != (-1) ) goto g3;
        MOV       AL,*XAR5++            ; |7|
        MOV       *XAR4++,AL            ; |7|
        BANZ      L1,AR6--
        ; branch occurs ; |7|
;***    -----------------------g4:
;***    -----------------------    return;
L2:
        LRETR
        ; return occurs
```

如果添加 --c_src_interlist 选项（使用 --opt_level=2、--c_src_interlist 和 --optimizer_interlist 进行编译），则汇编文件会包含与汇编代码交叉列出的编译器注释和 C 源代码。

```
;------------------------------------------------------------------------
;   2 | int copy (char *str, const char *s, int n)
;------------------------------------------------------------------------
;****************************************************************
;* FNAME: _copy                         FR SIZE:   0           *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;* FUNCTION PROPERTIES                                          *
;*                            0 Parameter,  0 Auto,  0 SOE      *
;****************************************************************
_copy
;* AR4   assigned to _str
;* AR5   assigned to _s
;* AL    assigned to _n
;* AL    assigned to _n
;* AR5   assigned to _s
;* AR4   assigned to _str
;* AR6   assigned to L$1
;*** 6  ----------------------    if ( n <= 0 ) goto g4;
;------------------------------------------------------------------------
;   4 | int i;
;------------------------------------------------------------------------
;------------------------------------------------------------------------
;   6 | for (i = 0; i < n; i++)
;------------------------------------------------------------------------
        CMPB      AL,#0                 ; |6|
        B         L2,LEQ                ; |6|
        ; branch occurs ; |6|
;***    ----------------------    #pragma MUST_ITERATE(1, 4294967295, 1)
:***    ----------------------    L$1 = n-1;
        ADDB      AL,#-1
        MOVZ      AR6,AL
        NOP
L1:
;*** 7  ----------------------    *str++ = *s++;
;*** 7  ----------------------    if ( (--L$1) != (-1) ) goto g3;
;------------------------------------------------------------------------
;   7 | *str++ = *s++;
;------------------------------------------------------------------------
        MOV       AL,*XAR5++            ; |7|
        MOV       *XAR4++,AL            ; |7|
        BANZ      L1,AR6--
        ; branch occurs ; |7|
;***    ----------------------g4:
;***    ----------------------    return;
L2:
        LRETR
        ; return occurs
```

3.11 数据页 (DP) 指针加载优化

当按名称访问全局变量时，编译器使用直接寻址。C28x 支持通过数据页指针寄存器 DP 进行直接寻址。DP 寄存器指向长度为 64 字的页的开头。

为了避免每次直接访问都加载 DP，编译器会对某些数据和段进行“分块”。分块技术可确保对象要么完全包含在单个页中，要么与 64 字页的边界对齐。这种数据页分块技术允许编译器更频繁地使用直接寻址模式。因此，在访问已知存储在单个数据页上的全局变量时，该功能最大程度地减少了对 DP 加载指令的需求。

对于 COFF ABI，所有非常量数据均被分块。

对于 EABI，默认的分块规则是：

- 数组及其各段不分块。
- 具有外部链接（C 中的 extern）的结构需分块。
- 具有内部链接（C 中的 static）的结构不分块，但这些结构的各段需分块。

然而，由于数据与页边界对齐，数据页分块会导致内存中出现对齐孔。因此，应用程序对代码大小和速度优化的需求与对数据大小优化的需求之间需要进行权衡。可以使用 **blocked** 和 **noblocked** 数据属性来控制对具体变量的分块。有关详细信息，请参阅节 6.15.4。

另一种用于管理如何在内存中组织全局变量的技术包括对应该存储在结构中一起的全局变量进行分组。也可以使用 DATA_SECTION（节 6.9.6）和 SET_DATA_SECTION（节 6.9.23）pragma 来管理数据页。

有关 C28x 数据页分块的示例，请参阅 E2E 社区上的 C2000 MCU 编译器中的数据分块功能主题。

3.12 调试和分析优化代码

默认情况下，编译器会在所有优化级别上生成符号调试信息。生成调试信息不会影响编译器优化和生成的代码。然而，更高级别的优化会由于所完成的代码转换而对调试体验产生负面影响。为获得最佳调试体验，请使用 --opt_level=off。

默认的优化级别为 off。

调试信息会增加目标文件的大小，但不会影响目标上的代码或数据的大小。如果目标文件大小是需一个问题并且不需要调试，请使用 --symdebug:none 禁用调试信息的生成。

3.12.1 分析优化的代码

要分析优化的代码，请使用优化（ --opt_level=0 到 --opt_level=3 ）。

3.13 提高代码大小优化级别（ --opt_for_space 选项 ）

--opt_for_space 选项提高了编译器执行的代码大小优化的级别。这些优化以牺牲性能为代价的。优化包括过程抽象，其中公共代码块替换为函数调用。例如，prolog 和 epilog 代码、某些内在函数和其他常见代码序列可以替换为运行时库中定义的函数的调用。使用 --opt_for_space 选项时，必须与提供的运行时库链接。没有必要使用优化来调用 --opt_for_space 选项。

为了说明 --opt_for_space 选项的工作原理，下面将介绍如何替换 prolog 和 epilog 代码。根据 SOE 寄存器的数量、帧的大小以及是否使用帧指针，将此代码更改为函数调用。使用 --opt_for_space 选项在每个文件中定义这些函数，如下所示：

```
_prolog_c28x_1
_prolog_c28x_2
_prolog_c28x_3
_epilog_c28x_1
_epilog_c28x_2
```

假设以下 C 代码是使用 --opt_for_space 选项编译的。

```
extern int x, y, *ptr;
extern int foo();
int main(int a, int b, int c)
{
   ptr[50] = foo();
   y = ptr[50] + x + y + a +b + c;
}
```

生成的输出如下所示：

```
FP      .set    XAR2
        .global _prolog_c28x_1
        .global _prolog_c28x_2
        .global _prolog_c28x_3
        .global _epilog_c28x_1
        .global _epilog_c28x_2
        .sect   ".text"
        .global _main
;****************************************************************
;* FNAME: _main                          FR SIZE:   6           *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;*                            0 Parameter,  0 Auto,  6 SOE      *
;****************************************************************
_main:
        FFC       XAR7,_prolog_c28x_1
        MOVZ      AR3,AR4               ; |5|
        MOVZ      AR2,AH                ; |5|
        MOVZ      AR1,AL                ; |5|
        LCR       #_foo                 ; |6|
        ; call occurs [#_foo] ; |6|
        MOVW      DP,#_ptr
        MOVL      XAR6,@_ptr            ; |6|
        MOVB      XAR0,#50              ; |6|
        MOVW      DP,#_y
        MOV       *+XAR6[AR0],AL        ; |6|
        MOV       AH,@_y                ; |7|
        MOVW      DP,#_x
        ADD       AH,AL                 ; |7|
        ADD       AH,@_x                ; |7|
        ADD       AH,AR3                ; |7|
        ADD       AH,AR1                ; |7|
        ADD       AH,AR2                ; |7|
        MOVB      AL,#0
        MOVW      DP,#_y
        MOV       @_y,AH                ; |7|
        FFC       XAR7,_epilog_c28x_1
        LRETR
        ; return occurs
```

3.14 编译器支持重入 VCU 代码

当服务例程中断时，--isr_save_vcu_regs 编译器选项生成指令以便使用堆栈来保存和恢复 VCU 寄存器，因此允许 VCU 代码可重入。使用该选项后，就算 ISR 中断 VCU 的计算，结果也不会受影响。

只有设置了 --vcu_support 选项才能使用该选项。

- 如果 --vcu_support 设置为 **vcu0** 或 **vcu2**，则在所有 ISR 的开头/结尾处添加下述说明：

```
VMOV32    *SP++,VCRC
VMOV32    *SP++,VSTATUS
    <ISR code here>
VMOV32    VSTATUS,*--SP
VMOV32    VCRC,*--SP
```

- 如果 --vcu_support 设置为 **vcrc**，则在所有 ISR 的开头/结尾处添加下述说明：

```
VMOV32    *SP++,VCRC
VMOV32    *SP++,VSTATUS
VMOV32    *SP++,VCRCPOLY
VMOV32    *SP++,VCRCSIZE
    <ISR code here>
VMOV32    VCRCSIZE,*--SP
VMOV32    VCRCPOLY,*--SP
VMOV32    VSTATUS,*--SP
VMOV32    VCRC,*--SP
```

3.15 编译器支持生成 DMAC 指令

C28x 编译器支持 DMAC 指令；该指令同时对两个相邻的有符号整数执行乘法累加运算，并可选择性地对乘积移位。乘法累加运算将两个数相乘，并将该乘积添加到累加器中。也就是，计算出 a = a + (b x c)。有三个级别的 DMAC 支持，需要不同级别的 C 源代码修改：

- 从 C 代码自动生成 DMAC 指令（请参阅节 3.15.1）。
- 使用数据地址对齐断言来启用 DMAC 指令生成（请参阅节 3.15.2）。
- 使用 __dmac 内在函数（请参阅节 3.15.3）。

3.15.1 自动生成 DMAC 指令

如果编译器将 C 语言语句识别为 DMAC 机会，并且编译器可以验证所操作的数据地址是 32 位对齐的，则编译器会自动生成 DMAC 指令。这是最好的方案，因为除了数据对齐 pragma 之外，不需要修改任何代码。下面是一个示例：

```
int src1[N], src2[N];
#pragma DATA_ALIGN(src1,2);          // int arrays must be 32-bit aligned
#pragma DATA_ALIGN(src2,2);

{...}
int i;
long res = 0;
for (i = 0; i < N; i++)              // N must be a known even constant
    res += (long)src1[i] * src2[i];  // Arrays must be accessed via array indices
```

在优化级别 >= -O2 时，如果 N 是已知偶数常数，则编译器会为上述示例代码生成一条 RPT || DMAC 指令。

DMAC 指令还可以在累加之前将乘积向左移动 1 或向右移动 1 到 6。例如：

```
for (i = 0; i < N; i++)
    res += (long)src1[i] * src2[i] >> 1;   // product shifted right by 1
```

3.15.2 指定数据地址对齐的断言

在某些情况下，编译器可能会识别 C 语言语句中的 DMAC 机会，但无法验证传递给计算的数据地址是否均为 32 位对齐。在这种情况下，放入代码中的断言可以使编译器生成 DMAC 指令。下面是一个示例：

```
int *src1, *src2;    // src1 and src2 are pointers to int arrays of at least size N
                     // You must ensure that both are 32-bit aligned addresses
{...}
int i;
long res = 0;
_nassert((long)src1 % 2 == 0);
_nassert((long)src2 % 2 == 0);
for (i = 0; i < N; i++)             // N must be a known even constant
    res += (long)src1[i] * src2[i]; // src1 and src2 must be accessed via array indices
```

在优化级别 >= -O2 时，如果 N 是已知偶数常数，则编译器会为上述示例代码生成一条 RPT || DMAC 指令。

_nassert 内在函数不会生成任何代码，因此不是表 7-6 中列出的编译器内在函数。相反，该函数告诉优化器使用 assert 函数声明的表达式成立。这可以用来提示优化器哪些优化可能有效的。在本示例中，_nassert 用于断言由 src1 和 src2 指针表示的数据地址是 32 位对齐的。请务必确保只有 32 位对齐的数据地址通过这些指针传递。如果断言的条件不成立，代码将导致运行时失败。

DMAC 指令还可以在累加之前将乘积向左移动 1 或向右移动 1 到 6。例如：

```
for (i = 0; i < N; i++)
    res += (long)src1[i] * src2[i] >> 1;    // product shifted right by 1
```

3.15.3 __dmac 内在函数

可以使用 __dmac 内在函数迫使编译器生成 DMAC 指令。在这种情况下，务必确保数据地址为 32 位对齐。

```
void __dmac(long *src1, long *src2, long &accum1, long &accum2, int shift);
```

- src1 和 src2 必须是指向 int 数组的 32 位对齐地址。
- Accum1 和 accum2 是两个临时累加的引用传递长度。必须在内在函数之后将它们加在一起计算总和。
- shift 指定每次累加前的乘积移位距离。有效的 shift 值分别为 -1（左移 1）、0（不移位）和 1-6（右移 1-6）。请注意，这个参数是必需的，所以如果您不想移位，则必须传递 0。

有关 __dmac 内在函数的详细信息，请参阅表 7-6。

示例 1：

```
int src1[2N], src2[2N]; // src1 and src2 are int arrays of at least size 2N
                        // You must ensure that both start on 32-bit
                        // aligned boundaries.
{...}
int i;
long res = 0;
long temp = 0;
for (i=0; i < N; i++)            // N does not have to be a known constant
     __dmac(((long *)src1)[i], ((long *)src2)[i], res, temp, 0);
res += temp;
```

示例 2：

```
int *src1, *src2;     // src1 and src2 are pointers to int arrays of at
                      // least size 2N.User must ensure that both are
                      // 32-bit aligned addresses.
{...}
int i;
long res = 0;
long temp = 0;
long *ls1 = (long *)src1;
long *ls2 = (long *)src2;
for (i=0; i < N; i++)     // N does not have to be a known constant
    __dmac(*ls1++, *ls2++, res, temp, 0);
res += temp;
```

在这些示例中，res 保存了 长度为 2*N 的 int 数组的乘法累加运算的最终和，同时执行两次计算。 。

此外， 必须使用优化级别 >= -O2 来生成有效的代码。此外，如果 这些示例的循环体中没有其他内容，编译器会生成 RPT || DMAC 指令，从而进一步提高性能。

3.16 正在执行什么类型的优化？

TMS320C28x C/C++ 编译器使用各种优化技术来提高 C/C++ 程序的执行速度并减小其大小。以下是编译器执行的一些优化：

优化	请参阅
基于成本的寄存器分配	节 3.16.1
别名消歧	节 3.16.2
分支优化和控制流简化	节 3.16.3
数据流优化 • 复制传播 • 通用子表达式消除 • 冗余分配消除	节 3.16.4
表达式简化	节 3.16.5
函数的内联扩展	节 3.16.6
函数符号别名	节 3.16.7
归纳变量和强度降低	节 3.16.8
循环不变量代码运动	节 3.16.9
循环旋转	节 3.16.10
指令调度	节 3.16.11

C28x 专用优化	请参阅
寄存器变量	节 3.16.12
寄存器跟踪/定位	节 3.16.13
尾部合并	节 3.16.14
自动增量寻址	节 3.16.15
将比较值删除为 0	节 3.16.16
RPTB 生成 (仅限 FPU 目标)	节 3.16.17

3.16.1 基于成本的寄存器分配

启用优化后，编译器会根据类型、用途和频率为用户变量和编译器临时值分配寄存器。循环中使用的变量经过加权后优先于其他变量，而那些使用不重叠的变量可以分配到同一个寄存器。

归纳变量消除和循环测试替换功能允许编译器将循环识别为简单的计数循环并展开或消除循环。强度降低功能将数组引用转换为具有自动增量的高效指针引用。

3.16.2 别名消歧

C 和 C++ 程序通常使用许多指针变量。通常，编译器无法确定两个或多个 l 值 (小写 L：符号、指针引用或结构引用) 是否指向同一内存位置。内存位置的这种别名通常会阻止编译器在寄存器中保留值，因为无法确保寄存器和内存是否会随着时间的推移继续保持相同的值。

别名消歧是确定两个指针表达式何时不能指向同一位置的技术，允许编译器可以自由地优化此类表达式。

3.16.3 分支优化和控制流简化

编译器分析程序的分支行为并重新排列操作的线性序列（基本块），以去除分支或冗余条件。不可达代码被删除，分支到分支被绕过，无条件分支上的条件分支被简化为单个条件分支。

当在编译期间确定条件的值时（通过复制传播或其他数据流分析），编译器可以删除条件分支。切换实例列表的分析方式与条件分支相同，有时会完全消除此类列表。一些简单的控制流结构被简化为条件指令，完全消除了对分支的需求。

这种类型的优化通过 --opt_level=0 和更高的优化设置来启用。

3.16.4 数据流优化

总的来说，以下数据流优化会将表达式替换为成本较低的表达式，检测并删除不必要的赋值，并避免对已计算过的值进行运算。启用优化的编译器在局部（在基本块内）和全局（跨整个函数）执行这些数据流优化。

- **复制传播。**在对变量赋值之后，编译器用变量值替换对变量的引用。该值可以是另一个变量、常量或通用子表达式。因此导致更多的机会使常量折叠、通用子表达式消除甚至变量完全消除。这种类型的优化通过 --opt_level=1 和更高的优化设置来启用。
- **通用子表达式消除。**当两个或多个表达式产生相同的值时，编译器一次计算该值，保存并重复使用该值。这种类型的优化通过 --opt_level=2 和更高的优化设置来启用。
- **冗余赋值消除。**通常，复制传播和通用子表达式消除优化会导致对变量进行不必要的赋值（在另一个赋值之前或函数结束之前无后续引用的变量）。编译器会删除这些无效的赋值。此类优化可通过 --opt_level=1（对于局部赋值）和 --opt_level=2（对于全局赋值）来启用。

3.16.5 表达式简化

为了优化评估，编译器将表达式简化为需要更少指令或寄存器的等效形式。常量之间的运算被折叠成单个常量。例如，a = (b + 4) - (c + 1) 变为 a = b - c + 3。

这种类型的优化通过 --opt_level=0 和更高的优化设置来启用。

3.16.6 函数的内联扩展

编译器用内联代码替换对小函数的调用，从而节省与函数调用相关的开销，并提供了更多应用其他优化的机会。有关影响内联的命令行选项、pragma 和关键字之间的交互信息，请参阅节 2.11。

这种类型的优化通过 --opt_level=0 和更高的优化设置来启用。

3.16.7 函数符号别名

编译器识别其定义仅包含对另一个函数的调用的函数。如果这两个函数具有相同的签名（相同的返回值以及相同数量、相同类型且顺序相同的参数），则编译器可以使调用函数成为被调用函数的别名。

例如，考虑以下情况：

```
int bbb(int arg1, char *arg2);
int aaa(int n, char *str)
{
   return bbb(n, str);
}
```

在本示例中，编译器使 aaa 成为 bbb 的别名，因此在链接时，对函数 aaa 的所有调用都应重定向到 bbb。如果链接器可以成功地将所有引用重定向到 aaa，则可以删除函数 aaa 的主体，并将符号 aaa 定义在与 bbb 相同的地址处。

有关使用 GCC 函数属性语法来声明函数别名的信息，请参阅节 6.15.2。

3.16.8 归纳变量和强度降低

归纳变量是指其在循环中的值与循环的执行次数直接相关的变量。循环的数组索引和控制变量通常是归纳变量。

强度降低是指用更高效的表达式替换涉及归纳变量的低效表达式的技术。例如，索引数组元素序列的代码用通过数组递增指针的代码替换。

归纳变量分析和强度降低功能相结合通常一起删除对环路控制变量的所有引用，从而消除该变量。

这种类型的优化通过 --opt_level=2 和更高的优化设置来启用。

3.16.9 循环不变量代码运动

此优化识别循环中始终计算为相同值的表达式。计算被移到循环的前面，且循环中每次出现的表达式都被替换为对预计算值的引用。

3.16.10 循环旋转

编译器在循环底部评估循环条件，从而减少循环外的额外分支。在许多情况下，初始入口条件检查和分支都被优化出来。

这种类型的优化通过 --opt_level=0 和更高的优化设置来启用。

3.16.11 指令排程

编译器会执行指令排程，即以提高性能的方式重新排列机器指令，同时保持原始顺序的语义。指令排程用于提高指令并行性并隐藏延迟。它还可用于缩减代码大小。

3.16.12 寄存器变量

编译器有助于最大程度地使用寄存器来存储局部变量、参数和临时值。访问存储在寄存器中的变量比访问内存中的变量更高效。寄存器变量对指针特别有效。

这种类型的优化通过 --opt_level=0 和更高的优化设置来启用。

3.16.13 寄存器跟踪/定位

编译器跟踪寄存器的内容，以避免在很快再次使用它们时重新加载值。通过直线代码跟踪变量、常量和结构引用（例如 (a.b)）。寄存器定向在需要时直接将表达式计算到特定寄存器中，比例寄存器变量分配或从函数中返回值。

3.16.14 尾部合并

如果正在优化代码大小，尾部合并对于某些函数可能非常有效。尾部合并可以找到以相同指令序列结尾并具有共同目标的基本块。如果找到这样的一组块，则会将该相同指令序列放入自己的块中。随后将这些指令从这组块中删除，并用新创建的块的分支替换。这样，指令序列就只有一个副本，而不是这组块中的每个块都有一个副本。

3.16.15 自动增量寻址

对于 *p++ 形式的指针表达式，编译器使用高效的 C28x 自动增量寻址模式。在许多情况下，当代码在循环中遍历数组时（如下所示），循环优化过程通过自动递增的寄存器变量指针将数组引用转换为间接引用。

```
for (I = 0; I <N; ++I) a(I)...
```

3.16.16 删除与零的比较

大多数 32 位指令和一些 16 位指令在其运算结果为 0 时可以修改状态寄存器，因此可能不需要与 0 进行显式比较。如果可以修改前一条指令以适当设置状态寄存器，则 C28x C/C++ 编译器会删除与 0 的比较。

3.16.17 RPTB 生成（仅适用于 FPU 目标）

当目标具有硬件浮点支持时，某些循环可以转换为称为重复块 (RPTB) 的硬件循环。通常，循环如下所示：

```
Label:
   ...loop body...
   SUB loop_count
   CMP
   B    Label
```

同样的循环在转换为 RPTB 循环后如下所示：

```
   RPTB   end_label, loop_count
   ...loop body...
end_label:
```

重复块循环被加载到硬件缓冲区中，并被执行指定的迭代次数。这种循环的分支开销最小或为零，并且可以提高性能。循环计数存储在特殊的寄存器 RB（重复块寄存器）中，并由硬件无缝地递减计数，而无需指定任何显式减法。因此，不会由于减法、比较和分支而产生开销。唯一的开销是在循环之前执行一次的 RPTB 指令产生的。如果迭代次数为常数，则 RPTB 指令需要一个周期，否则需要 4 个周期。此开销在每个循环中产生一次。

由于缓冲区的存在，有资格成为重复块的循环在最小循环大小和最大循环大小上都受到限制。此外，循环不能包含任何内循环或函数调用。

章节 4

链接 C/C++ 代码

C/C++ 代码生成工具提供了两种链接程序的方法：

- 可以编译单个模块并将它们链接在一起。在有多个源文件时，这种方法特别有用。
- 可以一步完成编译和链接。在有单个源代码模块时，这种方法很有用。

本章介绍如何使用每种方法调用链接器。此外，还将讨论链接 C/C++ 代码，包括运行时支持库、指定初始化类型以及分配程序分配到内存中的特殊要求。有关链接器的完整说明，请参阅《*TMS320C28x 汇编语言工具用户指南*》。

4.1 通过编译器调用链接器（-z 选项）

本节介绍如何在编译和汇编程序后调用链接器：作为单独的步骤还是作为编译步骤的一部分。

4.1.1 单独调用链接器

将 C/C++ 程序作为单独步骤进行链接的一般语法如下：

cl2000 --run_linker {**--rom_model | --ram_model**} *filenames*
[*options*] [**--output_file=** *name.out*] **--library=** *library* [*lnk.cmd*]

cl2000 --run_linker	调用链接器的命令。
--rom_model | **--ram_model**	通知链接器使用 C/C++ 环境定义的特殊约定的选项。当使用 cl2000 --run_linker 而不列出要在命令行编译的任何 C/C++ 文件时，*必须* 在命令行上或链接器命令文件中使用 **--rom_model** 或 **--ram_model**。--rom_model 选项在运行时进行自动变量初始化；--ram_model 选项在加载时进行变量初始化。有关使用 --rom_model 和 --ram_model 选项的详细信息，请参阅节 4.3.5。如果未能指定 ROM 或 RAM 模型，您将看到一条链接器警告，内容为： `warning: no suitable entry-point found; setting to 0`
filenames	目标文件、链接器命令文件或存档库的名称。输入文件的默认扩展名为 .c.obj（用于 C 源文件）和 .cpp.obj（用于 C++ 源文件）。必须显式指定任何其他 扩展名。链接器可以确定输入文件是包含链接器命令的目标文件还是 ASCII 文件。除非使用 --output_file 选项，否则默认输出文件名为 ***a.out***。
options	影响链接器处理目标文件的方式的选项。链接器选项只能出现在命令行上的 **--run_linker** 选项之后，否则可以按任意顺序出现。（《*TMS320C28x 汇编语言工具用户指南*》中详细讨论了这些选项。）
--output_file= *name.out*	对输出文件命名。
--library= *library*	标识包含 C/C++ 运行时支持和浮点数学函数或链接器命令文件的合适的存档库。如果正在链接 C/C++ 代码，必须使用运行时支持库。可以使用编译器中包含的库，也可以创建您自己的运行时支持库。如果在链接器命令文件中指定了运行时支持库，则不需要此参数。--library 选项的缩写形式为 -l。
lnk.cmd	包含链接器的选项、文件名、指令或命令。

备注

编译器创建的目标文件的默认文件扩展名已更改。从 C 源文件生成的目标文件具有 .c.obj 扩展名。从 C++ 源文件生成的目标文件具有 .cpp.obj 扩展名。

当将库指定为链接器输入时，链接器仅包含和链接那些解析未定义引用的库成员。链接器使用默认分配算法将程序分配到内存中。可以使用链接器命令文件中的 MEMORY 和 SECTIONS 指令来自定义分配过程。有关信息，请参阅《*TMS320C28x 汇编语言工具用户指南*》。

可以使用以下命令将包含目标文件 prog1.c.obj、prog2.c.obj 和 prog3.cpp.obj 的 C/C++ 程序与名为 prog.out 的可执行目标文件进行链接：

```
cl2000 --run_linker --rom_model prog1 prog2 prog3
     --output_file=prog.out --library=rts2800_ml.lib
```

4.1.2 调用链接器作为编译步骤的一部分

在编译步骤中链接 C/C++ 程序的一般语法如下：

```
cl2000 filenames [options] --run_linker   [--rom_model | --ram_model] filenames
          [options] [--output_file= name.out] --library= library [lnk.cmd]
```

--run_linker 选项将命令行分为编译器选项（--run_linker 之前的选项）和链接器选项（--run_linker 之后的选项）。--run_linker 选项必须跟在命令行上的所有源文件和编译器选项之后。

命令行上 --run_linker 后面的所有参数都传递给链接器。这些参数可以是链接器命令文件、附加目标文件、链接器选项或库。这些参数与节 4.1.1 中所述的参数相同。

命令行上 --run_linker 之前的所有参数都是编译器参数。这些参数可以是 C/C++ 源文件、汇编文件或编译器选项。节 2.2 介绍了这些参数。

可以使用以下命令来编译包含目标文件 prog1.c、prog2.c 和 prog3.c 的 C/C++ 程序，并将该程序与名为 prog.out 的可执行目标文件进行链接：

```
cl2000 prog1.c prog2.c prog3.c --run_linker --rom_model --output_file=prog.out
     --library=rts2800_ml.lib
```

当列出要在同一命令行上编译的至少一个 C/C++文件之后使用 cl2000 --run_linker 时，默认情况下会在运行时使用 **--rom_model** 进行变量的自动初始化。有关使用 --rom_model 和 --ram_model 选项的详细信息，请参阅节 4.3.5。

备注

在链接器中处理参数的顺序：链接器处理参数的顺序很重要。编译器按以下顺序将参数传递给链接器：

1. 从命令行获取的目标文件名
2. 命令行上 --run_linker 选项后面的参数
3. C2000_C_OPTION 环境变量中 --run_linker 选项后面的参数

4.1.3 禁用链接器（--compile_only 编译器选项）

可以使用 --compile_only 编译器选项来覆盖 --run_linker 选项。-run_linker 选项的缩写形式为 -z，--compile_only 选项的缩写形式为 -c。

如果在 C2000_C_OPTION 环境变量中指定了 --run_linker 选项，并希望有选择地禁用命令行上的--compile_only 选项，那么 --compile_only 选项特别有用。

4.2 链接器代码优化

4.2.1 生成函数子段（--gen_func_subsections 编译器选项）

编译器将源模块转换为目标文件。它可以将所有函数放在单个代码段中，也可以创建多个代码段。多个代码段的好处是链接器可以忽略可执行文件中未使用的函数。

当链接器收集要放入可执行文件的代码时，它不能拆分代码段。如果编译器没有使用多个代码段，并且特定模块中任何函数都需要链接到可执行文件中，则该模块中的所有函数都会链接进来，即使函数没有被使用。

一个示例是包含有符号除法例程和无符号除法例程的库 *.c.obj 文件。如果应用程序只需要有符号除法，则链接只需要有符号除法例程。如果只使用了一个代码段，则有符号和无符号例程都会链接进来，因为它们存在于同一个 *.c.obj 文件中。

--gen_func_subsections 编译器选项通过将文件中的每个函数放在其自己的子段来解决这个问题。因此，只有在应用程序中引用的函数才会链接到最终的可执行文件中。这将导致整体代码大小减小。

如果未使用此选项，则 C28x 编译器的默认值为“off”，CLA 编译器的默认值为“on”。如果使用了此选项但既未指定“on”也未指定“off”，则默认为“on”。

4.2.2 生成聚合数据子段（--gen_data_subsections 编译器选项）

与上一节中描述的代码段类似，数据可以放在单个段中，也可以放在多个段中。多个数据段的好处是链接器可以从可执行文件中省略未使用的数据结构。此选项会将聚合数据（数组、结构体和联合体）放置在数据段的单独子段中。

如果未使用此选项，则 C28x 编译器和 CLA 编译器的默认值为“off”。如果使用了此选项但既未指定“on”也未指定“off”，则会提供错误消息。

如果使用了 SET_DATA_SECTION pragma，则忽略 --gen_data_subsections=on 选项。用户定义的段放置优先于子段的自动生成。

4.3 控制链接过程

无论选择哪种方法来调用链接器，在链接 C/C++ 程序时都有特殊要求。请务必：

- 包含编译器的运行时支持库
- 指定引导时初始化的类型
- 确定如何将程序分配到内存中

本节讨论如何控制这些因素并提供标准默认链接器命令文件的示例。更多有关如何操作链接器的信息，请参阅《*TMS320C28x 汇编语言工具用户指南*》中的链接器说明。

4.3.1 包含运行时支持库

必须将所有 C/C++ 程序与运行时支持库链接起来。该库包含标准 C/C++ 函数以及由编译器的用于管理 C/C++ 环境的函数。以下几节介绍了两种包含运行时支持库的方法。

4.3.1.1 自动选择运行时支持库

如果指定了 --rom_model 或 --ram_model 链接器选项，或者命令行中列出了至少一个要编译的 C/C++ 文件，则链接器会假设您正在使用 C 和 C++ 约定。有关使用 --rom_model 和 --ram_model 选项的详细信息，请参阅节 4.3.5。

如果链接器假设您正在使用 C 和 C++ 约定，并且程序的入口点（通常是 c_int00）没有被任何指定的目标文件或库解析，则链接器会试图自动为您的程序纳入兼容性最高的运行时支持库。编译器选择的运行时支持库将在命令行或链接器命令文件中使用 --library 选项指定任何其他库之后，再搜索。如果明确使用了 libc.a，则合适的运行时支持库将包含在指定了 libc.a 的搜索顺序中。

可以使用 --disable_auto_rts 选项禁用运行时支持库的自动选择。

如果链接期间在 --run_linker 选项之前指定了 --issue_remarks 选项，则会生成一条备注，指示链接到哪个运行时支持库。如果需要使用与 --issue_remarks 报告的库不同的运行时支持库，则必须使用 --library 选项指定所需的运行时支持库的名称，并在必要 时在链接器命令文件中指定。

示例 4-1. 使用 --issue_remarks 选项

```
cl2000 --issue_remarks main.c --run_linker --rom_model
<Linking>
remark: linking in "libc.a"
remark: linking in "rts2800_ml.lib" in place of "libc.a"
```

4.3.1.2 手动选择运行时支持库

通过显式指定要使用的所需运行时支持库，可以避开自动选择库。使用 --library 链接器选项指定库的名称。链接器将搜索由 --search_path 选项指定的路径，然后搜索命名库的 C2000_C_DIR 环境变量。可以在命令行上或命令文件中使用 --library 链接器选项。

cl2000 --run_linker {**--rom_model** | **--ram_model**} *filenames* **--library=** *libraryname*

4.3.1.3 用于搜索符号的库顺序

通常，应该在命令行上将运行时支持库指定为最后一个名称，因为链接器会按照在命令行上指定文件的顺序搜索库中未解析的引用。如果有任何目标文件紧随某个库，则不会解析这些目标文件对该库的引用。可以使用 --reread_libs 选项强制链接器重新读取所有库，直到引用被解析为止。每当库指定为链接器输入时，链接器仅包含和链接那些会解析未定义的引用的库成员。

默认情况下，如果一个库引入了一个未解析引用，并且多个库具有该应用的定义，则会使用这个引入了未解析引用的库中的定义。如果希望链接器使用包含该定义的命令行上的第一个库中的定义，请使用 --priority 选项。

4.3.2 运行时初始化

必须使用代码将所有 C/C++ 程序链接起来，以初始化并执行称为引导程序的程序。引导程序负责执行以下任务：

1. 设置状态寄存器和配置寄存器
2. 设置栈
3. 处理 .cinit 运行时初始化表以自动初始化全局变量（使用 --rom_model 选项时）
4. 调用所有全局对象构造函数（.pinit 或 .init_array，具体取决于 ABI）
5. 调用 main() 函数
6. 当 main() 返回时调用 exit()

备注

_c_int00 符号：如果使用 --ram_model 或 --rom_model 链接选项，_c_int00 会自动定义为程序的入口点。如果命令行未列出任何要编译的 C/C++ 文件，并且未指定 --ram_model 和 --rom_model 链接选项，则链接器不知道是否使用了 C/C++ 约定，并且您将收到链接器警告“警告：没有找到合适的程序入口：设置为”。有关使用 --rom_model 和 --ram_model 选项的详细信息，请参阅节 4.3.5。

4.3.3 通过中断向量进行初始化

如果 C/C++ 程序在 RESET 处开始运行，必须设置 RESET 向量以分支到合适的引导程序，例如 _c_int00。此外，还必须确保项目中包含中断向量，通常是使用 --undef_sym 链接器选项将中断向量开头的符号作为根链接器对象。

rts2800_ml.lib 中的 vectors.c.obj 中提供了一个示例中断向量。对于 C28x，该向量的前几行是：

```
        .def  _Reset
        .ref  _c_int00
_Reset: .vec _c_int00, USE_RETA
```

4.3.4 全局对象构造函数

具有构造函数和析构函数的全局 C++ 变量要求在程序初始化期间调用构造函数，并在程序终止期间调用析构函数。C++ 编译器生成在启动时待调用的构造函数表。

单个模块的全局对象的构造函数按源代码中声明的顺序被调用，但未指定不同目标文件的对象的相对顺序。

全局构造函数在其他全局变量初始化之后和 main() 函数调用之前调用。在退出运行时支持函数期间调用全局析构函数，类似于通过 atexit 注册的函数。

节 7.10.3.4 讨论了全局构造函数表的格式。

4.3.5 指定全局变量初始化类型

C/C++ 编译器生成用于初始化全局变量的数据表。节 7.10.3.1 讨论了这些初始化表的格式。按照以下方式之一使用初始化表：

- 在*运行时*初始化全局变量。使用 --rom_model 链接器选项（请参阅节 7.10.3.2）。
- 在*加载时*初始化全局变量。使用 --ram_model 链接器选项（请参阅节 7.10.3.3）。

如果在不编译任何 C/C++ 文件的情况下使用链接器命令行，必须使用 --rom_model 或 --ram_model 选项。这些选项告知链接器两个信息。首先，选项指示链接器应遵循 C/C++ 约定，在 c_int00 启动例程中使用 main() 定义进行链接。其次，选项告知链接器是在运行时还是在加载时选择初始化。如果命令行在需要时未能包含这些选项之一，则将看到“警告： 没有找到合适的入口点；设置为 0”。

如果使用单个命令行进行编译和链接，则 --rom_model 选项是默认选项。如果使用了 --rom_model 或 --ram_model 选项，该选项必须跟在 --run_linker 选项之后（请参阅节 4.1）。

有关 EABI 使用 --rom_model 和 --ram_model 的链接约定的信息，请分别参阅节 7.10.4.3 和节 7.10.4.5。

有关 COFF 使用--rom_model 和 --ram_model 的链接约定的信息，请分别参阅节 7.10.3.2 和节 7.10.3.3。以下列表概述了 COFF ABI 使用--rom_model 或 --ram_model 的链接约定：

- 符号 _c_int00 定义为程序入口点，用于标识 boot.c.obj 中 C/C++ 引导程序的开始。使用 --rom_model 或 --ram_model 时，会自动引用_c_int00，从而确保 boot.c.obj 从运行时支持库中自动链接进来。
- 初始化输出段中填充有终止记录，以便加载器（加载时初始化）或引导程序（运行时初始化）知道何时停止读取初始化表。
- 在加载时初始化（--ram_model 选项）时，会发生以下情况：
  - 链接器将初始化表符号设置为 -1。这表示初始化表不在内存中，因此在运行时不执行初始化。
  - STYP_COPY 标志设置在初始化表的段头中。STYP_COPY 是特殊属性，其告诉加载器直接执行自动初始化，而不是将初始化表加载到内存中。链接器不会在内存中为初始化表分配空间。
- 在运行时进行自动初始化（--rom_model 选项）时，链接器将初始化表符号定义为初始化表的起始地址。引导程序使用此符号作为自动初始化的起点。

4.3.6 指定在内存中分配段的位置

编译器生成可重定位的代码块和数据块。这些块，称为*段*，以各种方式分配在内存中，以符合各种系统配置。有关编译器如何使用这些段的完整说明，请参阅节 7.1.1。

编译器创建两种基本类型的段：初始化段和未初始化段。表 4-1 总结了初始化段。表 4-2 总结了未初始化段。

表 4-1. 初始化段

名称	内容	限制
.args	在引导程序调用 main() 函数之前保留用于复制命令行参数的空间。请参阅节 2.6。	
.binit	引导时间复制表（有关链接器命令文件中 BINIT 的信息，请参阅*汇编语言工具用户指南*）	
.c28xabi.exidx	用于异常处理的索引表；只读。（仅限 EABI）	程序
.c28xabi.extab	用于异常处理的展开指令；只读。（仅限 EABI）	程序
.cinit	用于显式初始化全局和静态变量的表。(1)	程序
.const	显式初始化且包含字符串文字的全局和静态常量变量。字符串文字放置在 .const:.string 子段中，以更大力度地控制链接时放置位置。（仅限 EABI）	数据
.data	显式初始化的全局和静态非常量变量。(1)	数据
.econst	显式初始化或包含字符串文字的全局常量变量。字符串文字放置在 .econst:.string 子段中，以更大力度地控制链接时放置位置。（仅限 COFF）	数据
.init_array	启动时要调用的构造函数表。（仅限 EABI）	程序
.ovly	复制除引导时间 (.binit) 复制表以外的表。	
.pinit	启动时要调用的构造函数表。（仅限 COFF）	程序
.ppdata	用于基于编译器的分析的数据表（请参阅 --gen_profile_info 选项）。	数据
.ppinfo	用于基于编译器的分析的相关性表（请参阅 --gen_profile_info 选项）。	数据
.switch	大型切换语句的跳转表。	不带 -unified_memory 选项的程序 任何位置都带有 -unified_memory 选项
.text	可执行代码和常量。	程序

(1) 对于 EABI，编译器创建初始化 .data 段。链接器创建 .cinit 段。

表 4-2. 未初始化段

名称	内容	限制
.bss	全局和静态变量（仅限 EABI）	数据中的任何位置
.ebss	全局和静态变量（仅限 COFF）	数据中的任何位置
.stack	栈	低 64K
.sysmem	malloc 函数的内存（堆）（仅限 EABI）	数据中的任何位置
.esysmem	malloc 函数的内存（堆）（仅限 COFF）	数据中的任何位置

链接程序时，必须指定内存中分配这些段的位置。通常，初始化段链接到 ROM 或 RAM 中，而未初始化段链接到 RAM 中。

链接器提供了 MEMORY 和 SECTIONS 指令用于分配段。有关将段分配到存储器中的更多信息，请参阅 *TMS320C28x 汇编语言工具用户指南*。

4.3.7 链接器命令文件示例

链接器命令文件（COFF 示例）显示了一个链接 C 程序的典型链接器命令文件。本示例中的命令文件名为 lnk.cmd 。该命令文件链接三个目标文件（x.c.obj、y.c.obj 和 z.c.obj），并创建一个程序 (prog.out) 和一个映射文件 (prog.map)。

要链接该程序，请输入以下命令：

```
cl2000 --run_linker lnk.cmd
```

MEMORY，可能还有 SECTIONS 指令可能需要修改才能在您的系统中工作。更多有关这些指令的信息，请参阅《*TMS320C28x 汇编语言工具用户指南*》。

链接器命令文件（COFF 示例）

```
x.c.obj y.c.obj z.c.obj          /* Input filenames */
--output_file=prog.out    /* Options */
--map_file=prog.map
--library=rts2800_ml.lib     /* Get run-time support */
MEMORY                    /* MEMORY directive */
{
   RAM:  origin = 100h     length = 0100h
   ROM:  origin = 01000h   length = 0100h
}
SECTIONS                  /* SECTIONS directive */
{
   .text:    > ROM
   .data:    > ROM
   .ebss:    > RAM
   .pinit:   > ROM
   .cinit:   > ROM
   .switch:  > ROM
   .econst:  > RAM
   .stack:   > RAM
   .esysmem: > RAM
```

如果使用的是 EABI 而不是 COFF，请根据需要更改以下段：

- .ebss 改为 .bss
- .esysmem 改为 .sysmem
- .econst 改为 .const
- .pinit 改为 .init_array

4.4 链接 C28x 和 C2XLP 代码

C28x 链接器中阻止链接 64 字页大小 (C28x) 和 128 字页大小 (C2XLP) 代码的错误已更改为警告。可以从 C28x C/C++ 代码中调用 C2XLP 汇编函数。一种可能的方法是用一个为 C2XLP 代码正确设置参数和调用栈的 veneer 函数来替换对 C2XLP 函数的调用。例如，要调用一个需要五个整数参数的 C2XLP 函数，请将 C28x 代码更改为：

```
extern void foo_veneer(int, int, int, int, int);
void bar()
{
   /* replace the C2XLP call with a veneer call */
   /* foo(1, 2, 3, 4, 5); */
   foo_veneer(1, 2, 3, 4, 5);
}
```

用于链接 C2xx 和 C2XLP 代码的 Veneer 函数 说明了 veneer 函数可以看起来：：

用于链接 C2xx 和 C2XLP 代码的 Veneer 函数

```
        .sect ".text"
        .global _foo_veneer
        .global _foo
_foo_veneer:
        ;save registers
        PUSH AR1:AR0
        PUSH AR3:AR2
        PUSH AR5:AR4
        ;set the size of the C2XLP frame (including args size)
        ADDB SP,#10
        ;push args onto the C2XLP frame
        MOV *-SP[10],AL ;copy arg 1
        MOV *-SP[9],AH  ;copy arg 2
        MOV *-SP[8],AR4 ;copy arg 3
        MOV *-SP[7],AR5 ;copy arg 4
        MOV AL,*-SP[19]
        MOV *-SP[6],AL  ;copy arg 5
        ;save the return address
        MOV *-SP[5],#_label
        ;set AR1,ARP
        MOV AL,SP
        SUBB AL,#3
        MOV AR1,AL
        NOP *ARP1
        ;jump to C2XLP function
        LB _foo
_label:
        ;restore register
        POP AR5:AR4
        POP AR3:AR2
        POP AR1:AR0LRETR
```

由于 veneer 函数框架将充当所有 C2XLP 调用的框架，因此有必要为第一个 C2XLP 函数的任何后续调用添加足够的框架大小。

对于为 COFF ABI 编译的 C28x C/C++ 代码，全局变量放置在.ebss 段中。与 C28x 代码链接时，不能保证 C2XLP .ebss 段从 128 字边界开始。为避免此问题，请定义一个新段，将 C2XLP 全局变量更改为新段，并更新链接器命令文件以确保此新段从 128 字边界开始。

章节 5

链接后优化器

TMS320C28x 链接后优化器删除或修改汇编语言指令以生成更好的代码。链接后优化器检查通过链接确定的符号的最终地址，并使用此信息更改代码。

链接后优化需要 -plink 编译器选项。-plink 编译器选项调用工具的附加过程，包括运行绝对列表器以及重新运行汇编器和链接器。必须在 --run_linker 链接器选项之后使用 -plink 选项。

5.1 链接后优化器在软件开发流程中的作用

链接后优化器不是正常开发流程的一部分。图 5-1 显示了包括链接后优化器的流程；只有在使用编译器和 -plink 选项时，才会发生此流程。

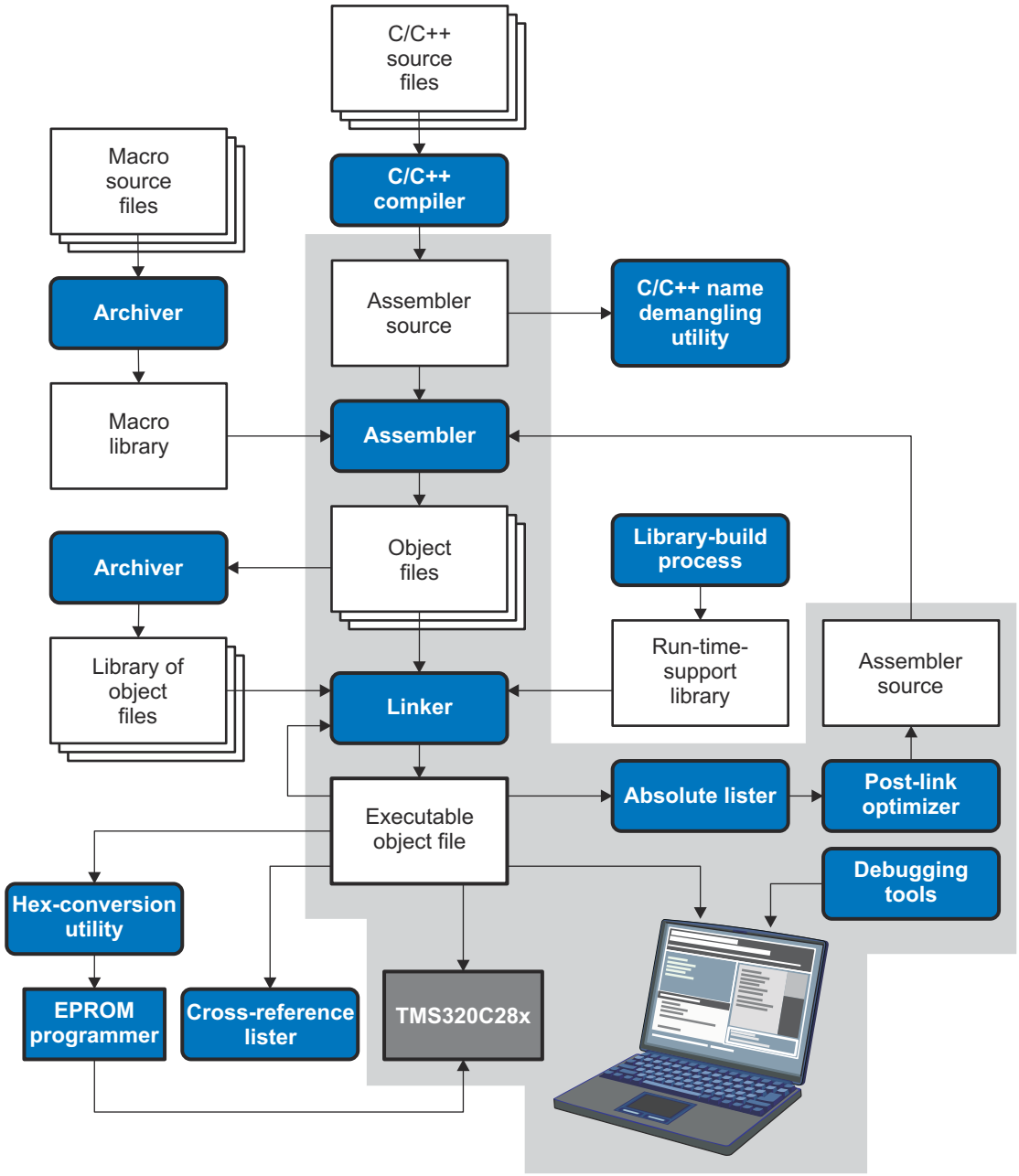


图 5-1. TMS320C28x 软件开发流程中的链接后优化器

如流程所示，绝对列表器 (abs2000) 也是链接后优化过程的一部分。绝对列表器输出所有全局定义的符号和 COFF 段的绝对地址。链接后优化器将 .abs 文件作为输入，并使用这些地址执行优化。输出是一个 .pl 文件，它是原始 .asm 文件的优化版本。然后，该流程重新运行汇编器和链接器，以生成最终的输出文件。

仅当使用编译器 (cl2000) 和 -plink 选项时，才支持所描述的流程。如果使用批处理文件单独调用每个工具，则必须调整流程以改用编译器。此外，当使用 -plink 选项时，必须使用 --output_file 选项指定输出文件名。有关更多详细信息,请参阅节 5.8。

例如，将下述各行：

```
cl2000 file1.asm file1.c.obj
cl2000 file2.asm file2.c.obj
cl2000 --run_linker file1.c.obj file2.c.obj lnk.cmd --output_file=prog.out
```

替换为此行：

```
cl2000 file1.asm file2.asm --run_linker lnk.cmd --output_file=prog.out -plink
```

advice_only 模式是为 FPU 和 VCU 目标提供的唯一的链接后优化支持。

5.2 删除冗余 DP 负载

链接后优化通过删除冗余的 DP 负载，降低管理数据页 (DP) 寄存器的难度。这是通过跟踪 DP 的当前值并确定 MOV DP,#address 指令中的地址是否位于 DP 当前指向的同一个 64 字页面上来实现的。如果可以使用当前 DP 值访问此地址，则该指令是冗余的，可以删除。例如，考虑以下代码段：

```
    MOVZ      DP,#name1
    ADD       @name1,#10
    MOVZ      DP,#name2
    ADD       @name2,#10
```

如果 name1 和 name2 链接到同一个页面，链接后优化器会认为不需要加载带有 name2 地址的 DP，并注释冗余负载。

```
    MOVZ      DP,#name1
    ADD       @name1,#10
     ; <<REDUNDANT>>          MOVZ      DP,#name2
    ADD       @name2,#10
```

这种优化也可以用于 C 文件上。尽管编译器对模块内定义的所有全局变量引用进行了 DP 管理，但对于外部定义的全局变量的引用，该编译器还是保守地发出了 DP 负载。在这些情况下，使用链接后优化器可以帮助减少 DP 负载的数量。

此外，--map_file 链接器选项可用于生成按数据页排序的符号列表，以帮助进行数据布局和减少 DP 负载。有关更多信息，请参阅 *TMS320C28x 汇编语言工具用户指南* 中的“链接器说明”一章 。

5.3 跟踪跨分支的 DP 值

为了跟踪跨分支的 DP 值，链接后优化器要求没有间接调用或分支，并且所有可能的分支目标都有标签。如果遇到间接分支或调用，链接后优化器将只跟踪基本块内的 DP 值。没有标签的分支目标可能会导致来自链接后优化器的输出错误。

如果链接后优化器遇到间接调用或分支，则会发出以下警告：

```
NO POST LINK OPTIMIZATION DONE ACROSS BRANCHES
Branch/Call must have labeled destination
```

发出此警告的目的是，如果文件是手写的汇编文件，则可以尝试将间接调用/分支更改为直接调用/分支，以从后链接器获得最佳优化。

5.4 跟踪跨函数调用的 DP 值

如果函数在同一文件范围内定义，则链接后优化器会在调用函数后优化 DP 负载。例如，考虑下述链接后优化代码：

```
_main:
    LCR    #_foo
    MOVB    AL, #0
;<<REDUNDANT>>    MOVZ    DP,#_g2
    MOV    @_g2, #20
    LRETR
    .global    _foo
_foo:
    MOVZ    DP, #g1
    MOV      @_g1, #10
    LRETR
```

由于变量 _g1 和 _g2 在同一页面中，且函数 _foo 已经设置了 DP，因此，链接后优化器会删除对 _foo 的函数调用后的 MOVZ DP。

为了让链接后优化器跨函数调用进行优化，函数应该只有一条 return 语句。如果对手写的汇编运行链接后优化器，并且对于每个函数有不止一条 return 语句，则链接后优化器的输出可能不正确。通过在 -plink 选项之后指定 -nf 选项，可以关闭跨函数调用的优化。

5.5 其他链接后优化

用作常量操作数的外部定义的符号会强制汇编器选择 16 位编码来保留直接值。由于链接后优化器可以访问外部定义的符号值，因此在可能的情况下，该优化器会将 16 位编码替换为 8 位编码。例如：

```
    .ref ext_sym ; externally defined to be 4
    :
    :
    ADD AL, #ext_sym ; assembly will encode ext_sym with 16 bits
```

由于 ext_sym 是在外部定义的，因此汇编器为 ext_sym 选择 16 位编码。链接后优化器将 ext_sym 的编码更改为 8 位编码：

```
    .ref ext_sym
    :
    :
    ; << ADD=>ADDB>> ADD AL,#ext_sym
    ADDB AL, #ext_sym
```

类似地，链接后优化器尝试将以下 2 字指令减少为 1 字指令：

2 字指令	1 字指令
ADD ACC, *#imm*	ADDB ACC, *#imm*
ADD AL, *#imm*	ADDB AL, *#imm*
AND AL, *#imm*	ANDB AL, *#imm*
CMP AL, *#imm*	CMPB AL, *#imm*
MOVL XARn, *#imm*	MOVB XARn, *#imm*
OR AL, *#imm*	ORB AL, *#imm*
SUB ACC, *#imm*	SUBB ACC, *#imm*
SUB AL, *#imm*	SUBB AL, *#imm*
XOR AL, *#imm*	XORB AL, *#imm*

5.6 控制链接后优化

有三种方法可以控制链接后优化：排除文件，在汇编文件中插入特定注释，以及手动编辑链接后优化文件。

5.6.1 排除文件（-ex 选项）

通过使用 -ex 选项，可以从链接后优化过程中排除特定文件。待排除的文件必须跟在 -ex 选项之后，并包括文件扩展名。-ex 选项必须用在 -plink 选项之后，且后面不跟任何其他选项。例如：

```
cl2000 file1.asm file2.asm file3.asm --keep_asm --run_linker lnk.cmd -plink -o=prog.out
-ex=file3.asm
```

file3.asm 将被排除在链接后优化过程之外。

5.6.2 控制汇编文件中的链接后优化

在汇编文件中，可以通过使用两个特殊格式的注释语句来禁用或启用链接后优化：

```
;//NOPLINK//
;//PLINK//
```

NOPLINK 注释后的汇编语句未被优化。可以使用 //PLINK// 注释重新启用链接后优化。

PLINK 和 NOPLINK 注释格式不区分大小写。分号和 PLINK 分隔符之间可以有空格。PLINK 和 NOPLINK 注释必须各自出现在单独的行中，并且必须从第 1 列开始。例如：

```
;     //PLINK//
```

5.6.3 保留链接后优化器输出（--keep_asm 选项）

借助 --keep_asm 选项，可以保留任何由 -plink 选项生成的链接后文件 (.pl) 和 .absolute 列表文件 (.abs)。使用 --keep_asm 选项可查看链接后优化器所做的任何更改。

.pl 文件包含以 <<REDUNDANT>> 显示的注释掉的语句，或对指令的任何改进（如 <<ADD=>ADDB>>）。将再次对 .pl 文件进行汇编和链接，以排除注释掉的行。

5.6.4 禁用跨函数调用的优化（-nf 选项）

-nf 选项禁用跨函数调用的链接后优化。链接后优化器通过 return 语句识别函数末尾，并假设每个函数只有一条 return 语句。在一些手写的汇编代码中，每个函数可能有不止一条 return 语句。在这种情况下，链接后优化的输出可能不正确。可以使用 -nf 选项关闭跨函数调用的优化。此选项会影响所有文件。

5.6.5 使用建议对汇编代码进行注释（--plink_advice_only 选项）

如果出于流水线方面的考虑而无法安全地进行更改（例如启用浮点支持或 VCU 支持时），可通过 --plink_advice_only 选项使用注释对汇编代码进行注释。

如果使用此选项，请注意，只有同时使用 --keep_asm 选项时，才会保留包含生成建议的链接后文件 (.pl)。

5.7 有关使用链接后优化器的限制

以下限制会影响链接后优化：

- advice_only 模式是为 FPU 和 VCU 目标提供的唯一链接后优化支持。
- 对未标记目标的分支或调用会使 DP 负载优化无效。所有分支目标都必须有标签。
- 如果数据段的位置取决于代码段的大小，则用于确定要删除哪些 DP 负载指令的数据页布局信息可能不再有效。

  例如，考虑下述链接命令文件：

```
SECTIONS
{
    .text    > MEM,
    .mydata  > MEM,}
```

  优化后 .text 段大小的变化会导致 .mydata 段发生移位。确保所有输出数据段都在 64 字边界上对齐，就可以消除这种移位问题。例如，考虑下述链接命令文件：

```
SECTIONS
{
    .text > MEM,
    .mydata align = 64 > MEM,}
```

5.8 命名输出文件（--output_file 选项）

使用 -plink 选项时，必须包含 --output_file 选项。如果在链接器命令文件中指定了输出文件名，则编译器无权访问该文件名而无法将其传递到链接后优化的其他阶段，该过程将失败。例如：

```
cl2000 file1.c file2.asm --run_linker --output_file=prog.out lnk.cmd -plink
```

因为链接后优化流程使用绝对列表器 abs2000，所以必须将该列表器包含在路径中。

章节 6

C/C++ 语言实现

受 C28x 支持的 C 语言由美国国家标准学会 (ANSI) 下属的一个委员会开发，随后被国际标准化组织 (ISO) 采用。受 C28x 支持的 C++ 语言由 ANSI/ISO/IEC 14882:2003 标准定义，但有一些例外。

6.1 TMS320C28x C 的特征

C 编译器支持 1989、1999 和 2011 版 C 语言：

- **C89。**使用 --c89 选项编译会使编译器符合 ISO/IEC 9899:1990 C 标准，该标准先前被批准为 ANSI X3.159-1989。“C89”和“C90”指的是同一种编程语言。本文档中使用了“C89”。
- **C99。**使用 --c99 选项编译会使编译器符合 ISO/IEC 9899:1999 C 标准。
- **C11。**使用 --c11 选项编译会使编译器符合 ISO/IEC 9899:2011 C 标准。

Kernighan 和 Ritchie 的 *C 程序设计语言* (K&R) 第二版中也介绍了 C 语言。编译器还可以在 GNU C 编译器中接受许多语言扩展（请参阅节 6.15）。

在支持 C89 的默认宽松 ANSI 模式下，编译器支持 C99 和 C11 的某些功能。它支持 C99 模式下 C99 的所有语言功能以及 C11 模式下 C11 的所有语言功能。请参阅节 6.14。

*不*支持 C11 标准中描述的原子操作。

ANSI/ISO 标准确定了可能受目标处理器特性、运行时环境或主机环境影响的 C 语言的某些功能。这组功能在标准编译器中会有所不同。

不受支持的 C 库功能包括：

- 运行时库对宽字符的支持很少。类型 wchar_t 实现为 16 位 int（对于 COFF）和 32 位 long（对于 EABI）。宽字符集相当于 char 类型的值集。该库包含头文件 <wchar.h> 和 <wctype.h>，但不包含标准中指定的所有函数。请参阅节 6.4，了解有关扩展字符集和多字节字符集的信息。
- 运行时库包含头文件 <locale.h>，但极少实现。唯一受支持的区域设置是 C 区域设置。也就是说，指定为随区域设置而变化的库行为被硬编码为 C 区域设置的行为，并且通过调用 setlocale() 来安装不同的区域设置的尝试将返回 NULL。
- 不支持 C99/C11 规范中的某些运行时函数和功能。请参阅节 6.14。

6.1.1 实现定义的行为

C 标准要求，符合规范的实现方案应提供有关编译器如何处理实现定义行为实例的文档。

TI 编译器正式支持独立的环境。C 标准不需要一个独立的环境来提供 C 语言的所有特性；特别是，库不需要是完整的。但是，TI 编译器力求提供适用于托管环境的大多数特性。

下述列表中的章节编号对应于 C99 标准附录 J 中的章节编号。每项末尾括号中的数字是 C99 标准中讨论该主题的章节编号。此列表中省略了 C99 标准附录 J 中列出的某些项。

J.3.1 转换

- 编译器和相关工具以几种不同的格式发出诊断消息。诊断消息被发送到 stderr；stderr 上的任何文本都可以被认为是诊断信息。如果存在任何错误，该工具将退出并显示指示失败（非零）的退出状态。（3.10，5.1.1.3）
- 保存非空的非空的空白字符序列，这些字符在转换阶段 3 中不会被单个空格字符替换。(5.1.1.2)

J.3.2 环境

- 编译器在标识符、字符串文字和字符常量中不支持多字节字符。没有从多字节字符到源字符集的映射。但是，编译器在注释中接受多字节字符。有关详细信息，请参阅节 6.4 (5.1.1.2)
- 程序启动时调用的函数的名称为“main”。(5.1.2.1)
- 程序终止不会影响环境；无法将退出代码返回给环境。正如我们所知，默认情况下，当执行到达特殊的 C$ $EXIT 标签时，程序就会停止。(5.1.2.1)
- 在宽松 ANSI 模式下，编译器接受“void main(void)”和“void main(int argc, char *argv[])”作为 main 的备用定义。在严格 ANSI 模式下，备用定义会被拒绝。(5.1.2.2.1)
- 如果在链接时使用 --args 选项为程序参数提供了空间，并且程序在可以填充 .args 段（例如 CCS）的系统下运行，则 argv[0] 将包含可执行文件的文件名，argv [1] 到 argv[argc-1] 将包含程序的命令行参数，而 argv[argc] 将为 NULL。在其他情况下，argv 和 argc 的值是未定义的。(5.1.2.2.1)
- 交互式设备包括 stdin、stdout 和 stderr（当连接到接受 CIO 请求的系统时）。交互式设备不限于这些输出位置；程序可以访问与外部状态交互的硬件外设。(5.1.2.3)
- 信号不受支持。函数信号不受支持。（7.14、7.14.1.1）
- 库函数 getenv 是通过 CIO 接口实现的。如果程序在支持 CIO 的系统下运行，系统会在主机系统上执行 getenv 调用并将结果传回程序。否则 getenv 的操作是未定义的。没有提供从目标程序内部改变环境的方法。(7.20.4.5)
- 系统函数不受支持。(7.20.4.6)

J.3.3.标识符

- 编译器在标识符中不支持多字节字符。有关详细信息，请参阅节 6.4。(6.4.2)
- 标识符中有效初始字符的数量无限制。（5.2.4.1，6.4.2）

J.3.4 字符

- 字节中的位数 (CHAR_BIT) 是 16。有关数据类型详细信息，请参阅节 6.3。(3.6)
- 执行字符集与基本执行字符集相同，为纯 ASCII。(5.2.1)
- 为标准字母转义序列生成的值如下。(5.2.2)：

转义序列	ASCII 含义	整数值
\a	BEL（响铃）	7
\b	BS（退格）	8
\f	FF（换页）	12
\n	LF（换行）	10
\r	CR（回车）	13
\t	HT（水平制表符）	9
\v	VT（垂直制表符）	11

- char 对象（其中存储了除基本执行字符集成员之外的任何其他字符）的值是该字符的 ASCII 值。(6.2.5)
- Plain char 等同于 signed char。（6.2.5，6.3.1.1）
- 源字符集和执行字符集都是纯 ASCII，因此它们之间的映射是一一对应的。编译器在注释中接受多字节字符。有关详细信息，请参阅节 6.4。（6.4.4.4，5.1.1.2）
- 编译器目前仅支持一个区域设置“C”。(6.4.4.4)
- 编译器目前仅支持一个区域设置“C”。(6.4.5)

J.3.5 整数

- 未提供扩展整数类型。(6.2.5)
- 有符号整数类型的负值用 2 的补码表示，没有陷阱表示。(6.2.6.2)
- 未提供扩展整数类型，因此整数秩没有变化。(6.3.1.1)

- 当整数转换为不能代表该值的有符号整数类型时，通过丢弃不能存储在目标类型中的位，该值会被截断（不发出信号）；最低位不会被修改。(6.3.1.3)
- 有符号整数值的右移执行算术（有符号）移位。除右移以外的按位操作对位的操作方式与对无符号值的操作方式完全相同。即，在通常的算术转换之后，执行按位运算而不考虑整数类型的格式，尤其是符号位。(6.5)

J.3.6 浮点

- 浮点运算 (+ - * /) 的精度是精确到位的。未指定返回浮点结果的库函数的准确性。(5.2.4.2.2)
- 编译器不会为 FLT_ROUNDS 提供非标准值。(5.2.4.2.2)
- 编译器不会为 FLT_EVAL_METHOD 提供非标准负值。(5.2.4.2.2)
- 整数转换为浮点数时的舍入方向是 IEEE-754“舍入到最近”。(6.3.1.4)
- 浮点数转换为更窄浮点数时的舍入方向是 IEEE-754“舍入到偶数”。(6.3.1.5)
- 对于不能准确表示的浮点常量，实现方案使用最接近的可表示值。(6.4.4.2)
- 编译器不会收缩浮点表达式。(6.5)
- FENV_ACCESS pragma 的默认状态为“关闭”。(7.6.1)
- TI 编译器不会定义任何额外的浮点异常。（7.6，7.12）
- FP_CONTRACT pragma 的默认状态为“关闭”。(7.12.2)
- 如果舍入结果等于数学结果，则不会产生“不精确”的浮点异常。(F.9)
- 如果结果很小但不是不精确，则不会产生“下溢”和“不精确”的浮点异常。(F.9)

J.3.7 数组和指针

- 在将指针转换为整数或将整数转换为指针时，该指针被视为是具有相同大小的无符号整数，并且适用普通整数转换规则。有些指针的大小与任何整数类型的大小都不同，但转换过程就像这种类型确实存在一样，并遵循普通整数所转换隐含的规则。
- 在将指针转换为整数或将长整型转换为指针时，如果目标的按位表示可以保存源命令的按位表示中的所有位，则这些位被精确复制。(6.3.2.3)
- 两个指向同一数组元素的指针相减的结果大小就是 ptrdiff_t 的大小，如节 6.3 中所定义。(6.5.6)

J.3.8 提示

- 使用优化器时，将忽略寄存器存储类说明符。不使用优化器时，编译器会尽可能优先将寄存器存储类对象放入寄存器中。编译器有权利将任何寄存器存储类对象放置在寄存器以外的位置。(6.7.1)
- 除非使用优化器，否则内联函数说明符将被忽略。有关内联的其他限制，请参阅节 2.11.2。(6.7.4)

J.3.9 结构体、联合体、枚举和位字段

- “plain”整数位字段会被视为有符号整数位字段。（6.7.2，6.7.2.1）
- 除了 _Bool、signed int 和 unsigned int，编译器还允许将 char、signed char、unsigned char、signed short、unsigned short、signed long、unsigned long、signed long long、unsigned long long 和枚举类型作为位字段类型。(6.7.2.1)
- 位字段不能跨越存储单元边界。请参阅节 7.1.7。(6.7.2.1)
- 位字段在一个单元内按字节顺序分配。(6.7.2.1)
- 结构体的非位字段成员按照节 7.1.7 中指定的方式对齐。(6.7.2.1)
- 每个枚举类型下的整数类型如节 6.3.1 中所述。(6.7.2.2)

J.3.10 限定符

- TI 编译器不会缩小或增加易失性访问。用户有责任确保访问大小适合仅允许访问特定宽度的器件。除非有必要，否则 TI 编译器不会更改对易失性变量的访问次数。这对于诸如 += 之类的读-改-写表达式很重要；对于没有相应的读-改-写指令的构架，编译器将被迫使用两种访问，一种用于读取，一种用于写入。即使对于具有此类指令的构架，也不能保证编译器能够将此类表达式映射到具有单个内存操作数的指令。不能保证内存系统会在指令执行期间锁定该内存位置。在多核系统中，其他一些内核可能会在 RMW 指令读取后但在写入结果之前写入该位置。TI 编译器不会对两个易失性访问重新排序，但可能会对一个易失性和一个非易失性访问重新排序，因此易失性不能用于创建临界区。如果您需要创建临界区，请使用某种锁。(6.7.3)

J.3.11 预处理指令

- Include 指令可能为以下两种形式之一，" " 或 < >。对于这两种形式，编译器将使用头文件搜索路径通过该名称查找磁盘上的真实文件。请参阅节 2.5.2。(6.4.7)
- 控制条件包含的常量表达式中的字符常量的值与执行字符集中的同一字符常量的值相匹配（两者都是 ASCII）。(6.10.1)
- 编译器使用文件搜索路径来搜索包含的 < > 分隔的头文件。请参阅节 2.5.2。(6.10.2)
- 编译器使用文件搜索路径来搜索包含的 " " 分隔的头文件。请参阅节 2.5.2。(6.10.2)
- #include 处理没有任意的嵌套限制。(6.10.2)
- 有关公认的非标准 pragma 的说明，请参阅节 6.9。(6.10.6)
- 转换日期和时间始终可从主机获得。(6.10.8)

J.3.12 库函数

- 托管实现所需的几乎所有库函数都由 TI 库提供，但节 6.14.1 中的情况例外。(5.1.2.1)
- 断言宏命令输出的诊断信息的格式为“断言失败，（*断言宏参数*），文件 *file*，行 *line*”。(7.2.1.1)
- 除了“C”和 "" 之外的任何其他字符串都不能作为第二个参数传递给 setlocale 函数。(7.11.1.1)
- 不支持信号处理。(7.14.1.1)
- +INF、-INF、+inf、-inf、NAN 和 nan 样式可用于输出无穷大或 NaN。（7.19.6.1、7.24.2.1）
- 在 fprintf 或 fwprintf 函数中，%p 转换的输出与适当大小的 %x 相同。（7.19.6.1、7.24.2.1）
- 由 abort、exit 或 _Exit 函数返回给主机环境的终止状态不会返回主机环境。（7.20.4.1，7.20.4.3，7.20.4.4）
- 系统函数不受支持。(7.20.4.6)

J.3.13 架构

- 分配给标头 float.h、limits.h 和 stdint.h 中指定宏命令的值或表达式与整数类型的大小和格式都在节 6.3 中进行了介绍。（5.2.4.2，7.18.2，7.18.3）
- 中介绍了任何对象中字节的数量、顺序和编码。(6.2.6.1)
- sizeof 运算符的结果值是每种类型的存储大小，以字节为单位。请参阅节 6.3。(6.5.3.4)

6.2 TMS320C28x C++ 的特征

根据 ANSI/ISO/IEC 14882:2003 标准 (C++03) 中的定义，C28x 编译器支持 C++，包括以下特性：

- 支持完整的 C++ 标准库，但具有以下例外情况。
- 模板
- 异常，通过 --exceptions 选项启用；请参阅节 6.6。
- 运行时类型信息 (RTTI)，可通过 --rtti 编译器选项启用。

编译器支持 ISO 标准化的 2003 年标准 C++。但是，以下特性*未*实现或完全受支持：

- 编译器不支持嵌入式 C++ 运行时支持库。
- 此库支持宽字符 (wchar_t)，因为为字符定义的模板函数和类也适用于 wchar_t。例如，实现了宽字符流类 wios、wiostream、wstreambuf 等（对应于字符类 ios、iostream、streambuf）。但是，没有用于宽字符的低级别文件 I/O。此外，C 库接口对宽字符的支持（通过 C++ 标头 <cwchar> 和 <cwctype>）是有限的，如上面 C 库中所述。
- 如果内联函数的定义包含静态变量，并且它出现在多个编译单元中（通常是因为它是头文件中所定义的类的成员函数），编译器将生成静态变量的多个副本，而不是将它们解析为单个定义。在这种情况下，编译器会发出警告 (#1369)。
- 未实现 export 关键字。

6.3 数据类型

表 6-1 列出了 C28x 编译器的每种标量数据类型的大小、表示形式和范围。许多范围值在头文件 limits.h 中作为标准宏命令提供。

大小为 16 位的类型在 16 位边界上对齐。大小为 32 位或更大的类型在 32 位（2 个字）边界上对齐。有关 EABI 字段对齐的详细信息，请参阅《*C28x 嵌入式应用二进制接口 (EABI)》参考指南*(SPRAC71)。 节 7.1.7 中详细介绍了数据类型的存储和对齐。

表 6-1. TMS320C28x C/C++ COFF 和 EABI 数据类型

			范围	
类型	**大小**	**表示**	**最小值**	**最大值**
char、signed char	16 位	ASCII	-32 768	32 767
unsigned char	16 位	ASCII	0	65 535
_Bool	16 位	二进制	0 (false)	1 (true)
short	16 位	二进制	-32 768	32 767
unsigned short	16 位	二进制	0	65 535
int、signed int	16 位	二进制	-32 768	32 767
unsigned int	16 位	二进制	0	65 535
long、signed long	32 位	二进制	-2 147 483 648	2 147 483 647
unsigned long	32 位	二进制	0	4 294 967 295
long long、signed long long	64 位	二进制	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 位	二进制	0	18 446 744 073 709 551 615
enum (1)	不尽相同	二进制	不尽相同	不尽相同
float (2)	32 位	IEEE 32 位	1.19 209 290e-38(3)	3.40 282 35e+38
double (COFF)	32 位	IEEE 32 位	1.19 209 290e-38(3)	3.40 282 35e+38 (COFF)
double (EABI)	64 位	IEEE 64 位	2.22 507 385e-308(3)	1.79 769 313e+308
long double	64 位	IEEE 64 位	2.22 507 385e-308(3)	1.79 769 313e+308
指针 (4)	32 位	二进制	0	0xFFFFFFFF

(1) 有关枚举类型大小的详细信息，请参阅节 6.3.1。
(2) 建议将 COFF 的 32 位浮点值声明为 `float`，而不是 `double`。
(3) 数字是最低精度。
(4) 即使指针是 32 位的，编译器也假定全局变量和函数的地址在 22 位以内。

wchar_t 类型对于 COFF 实现为 int（16 位），对于 EABI 实现为 long（32 位）。

有符号类型的负值用 2 的补码表示。

备注

TMS320C28x 字节是 16 位

根据 ANSI/ISO C 的定义，sizeof 运算符生成存储对象所需的字节数。ANSI/ISO 进一步规定，当 sizeof 应用于 char 时，结果为 1。由于 TMS320C28x char 是 16 位（使其可单独寻址），因此一个字节也是 16 位。这可能会产生无法预料的结果；例如，size of (int) = = 1（不是 2）。TMS320C28x 字节和字是等效的（16 位）。若要以 8 位为增量访问数据，请使用节 7.6 中所述的 __byte() 和 __mov_byte() 内在函数。

6.3.1 枚举类型大小

在下述声明中，enum e 是一个*枚举类型*。每一个 a 和 b 均为*枚举常量*。

```
enum e { a, b=N };
```

每个枚举类型均会被分配一个可保存所有枚举常量的整型。这个整型是“基础类型”。每个枚举常量的类型也是整型，并且在 C 语言中可能并不是相同的类型。请务必注意*枚举类型的基础类型*与*枚举常量类型*之间的区别。

为枚举类型和每个枚举常量选择的大小和符号取决于枚举常量的值以及编译的对象 C 还是 C++ 。C++11 允许为枚举类型指定特定类型；如果提供了此种类型，则会使用该类型，并且此段的其余部分不适用。

在 C++ 模式中，编译器允许枚举常量最高为最大整型（64 位）。C 标准规定所有严格符合 C 代码的枚举常量 (C89/C99/C11) 均必须具有适合“int”类型的值；不过，作为扩展，即使在 C 模式下，也可以使用大于“整数”的枚举常量。

对于枚举类型，编译器选择下述列表中第一个足够大且符号正确的类型来表示所有枚举常量值。 不使用“char”和“short”类型，因为它们与“int”类型大小相同。

- unsigned int
- igned int
- unsigned long
- signed long
- unsigned long long
- signed long long

例如，下述枚举类型将会以“unsigned int”作为其基础类型：

```
enum ui { a, b, c };
```

但下述类型将会以“signed int”作为其基础类型：

```
enum si { a, b, c, d = -1 };
```

而下述类型将会以“signed long”作为其基础类型：

```
enum sl { a, b, c, d = -1, e = UINT_MAX };
```

对于 C++，枚举常量全都具有与枚举类型相同的类型。

对于 C，则会根据它们的值来为枚举常量分配类型。所有值可以放入“int”的枚举常量都会被指定“int”类型，即使枚举类型的基础类型小于“int”也是如此。所有不能放入“int”的枚举常量都会被指定与枚举类型的基础类型相同的类型。这意味着，一些枚举常量可能与枚举类型具有不同的大小和符号。

6.3.2 支持 64 位整数

TMS320C28x 编译器支持 long long 数据类型和 unsigned long long 数据类型。范围值在头文件 limits.h 中作为标准宏命令提供。

long long 数据类型存储在寄存器对中。在存储器中，它们作为 64 位对象存储在双字 (32 位) 对齐的地址中。

long long 整数常量可以有一个 ll 或 LL 后缀。如果没有后缀，常量的值将决定常量的类型。

C I/O 中 long long 的格式规则要求格式字符串中包含 ll。例如：

```
printf("%lld", 0x0011223344556677);
printf("%llx", 0x0011223344556677);
```

添加了运行时支持库函数 llabs()、strtoll() 和 strtoull()。

6.3.3 C28x double 和 long double 浮点类型

为 TMS320C28x 编译 C/C++ 代码时，`long double` 浮点类型使用 IEEE 64 位双精度格式。

`double` 类型根据您是为 COFF 编译还是为 EABI 编译而不同。为 COFF 编译时，它使用 IEEE 32 位单精度类型。为 EABI 编译时，它使用 IEEE 64 位双精度类型。

C28x 浮点类型包括：

类型	Format
float	IEEE 32 位单精度
double (COFF)	IEEE 32 位单精度
double (EABI)	IEEE 64 位双精度
long double	IEEE 64 位双精度

建议将 32 位浮点值声明为 `float` ，而不是 `double`。

将 long double 初始化为常量时，必须使用 l 或 L 后缀。该常量被视为不带后缀的 double 类型，并为初始化调用运行时支持 double 到 long 转换例程。这会导致精度下降。例如：

```
long double a = 12.34L; /* correctly initializes to double precision */
long double b = 56.78;  /* converts single precision value to double precision */
```

C I/O 中 long double 的格式规则要求格式字符串中使用大写字母“L”。例如：

```
printf("%Lg", 1.23L);
printf("%Le", 3.45L);
```

对于 EABI 模式，请参阅《*C28x 嵌入式应用二进制接口*》应用报告 (SPRAC71)，了解有关 64 位类型调用约定的信息。

对于 COFF，long double 类型的调用约定如下：

所有 long double 参数都通过引用传递。通过引用返回 long double 返回值。前两个 long double 参数将在 XAR4 和 XAR5 中传递它们的地址。所有其他 long double 参数的地址都将在栈上传递。

如果函数返回一个 long double，那么进行该调用的函数会将返回地址放在 XAR6 中。例如：

```
long double foo(long double a, long double b, long double c)
{
   long double d = a + b + c;
   return d;
}
long double a = 1.2L;
long double b = 2.2L;
long double c = 3.2L;
long double d;
void bar()
{
   d = foo(a, b, c);
}
```

在函数 bar() 中，在调用 foo() 时， 寄存器值为：

寄存器	等于
XAR4	a 的地址
XAR5	b 的地址
*–SP[2]	c 的地址
XAR6	d 的地址

运行时支持库中包含必要的 long double 算术运算和转换函数。

6.4 文件编码和字符集

编译器接受具有两种不同编码之一的源文件：

- **具有字节顺序标记 (BOM) 的 UTF-8。**这些文件可能在 C/C++ 注释中包含扩展（多字节）字符。在所有其他上下文（包括字符串常量、标识符、汇编文件和链接器命令文件）中，都只支持 7 位 ASCII 字符。
- **纯 ASCII 文件。**此类文件只能包含 7 位 ASCII 字符。

若要在 Code Composer Studio 中选择 UTF-8 编码，请打开“Preferences”对话框，选择 **General > Workspace**，然后将 **Text File Encoding** 设为 UTF-8。

如果您使用没有“纯 ASCII”编码模式的编辑器，则可以使用 Windows-1252（也称为 CP-1252）或 ISO-8859-1（也称为 Latin 1），两者都接受所有 7 位 ASCII 字符。但是，编译器可能无法接受这些编码中的扩展字符，因此您不应使用扩展字符，即使在注释中也是如此。

编译器支持宽字符 (wchar_t) 类型和操作。但是，宽字符串不能包含超过 7 位 ASCII 的字符。宽字符的编码是 7 位 ASCII，0 扩展到 wchar_t 类型的宽度。

6.5 关键字

C28x C/C++ 编译器支持所有标准 C89 关键字，包括 const、volatile 和 register。它支持所有标准的 C99 关键字，包括 inline 和 restrict。它支持所有标准的 C11 关键字。它还支持 TI 扩展关键字 __interrupt、__cregister、和 __asm。 编译器支持 FPU 目标的 restrict 关键字；对于其他目标，restrict 会被忽略。某些关键字在严格 ANSI 模式下不可用。

以下关键字可能出现在其他目标文档中，需要以与 interrupt 和 restrict 关键字相同的方式进行处理：

- 陷阱[trap]
- reentrant
- cregister

6.5.1 const 关键字

C/C++ 编译器在所有模式下都支持 ANSI/ISO 标准关键字 *const* 除外。此关键字使您能够更好地优化和控制某些数据对象的分配。您可以将 const 限定符应用于任何变量或数组的定义，以确保其值不被更改。

限定为 常量的全局对象放置在 .econst 或 .const 段中。链接器从 ROM 或闪存中分配 .econst or .const 段，它们通常比 RAM 更丰富。const 数据存储分配规则有以下例外情况：

- 如果对象具有自动存储功能（函数作用域）。
- 如果对象是具有“可变”成员的 C++ 对象。
- 如果使用编译时未知的值（例如另一个变量的值）来初始化对象。

在这些情况下，对象的存储与未使用常量关键字时相同。

const 关键字的位置很重要。例如，下面的第一条语句定义了指向可修改 int 的常量指针 p。第二条语句定义了指向常量 int 的可修改指针 q：

```
int * const p = &x;
const int * q = &x;
```

使用常量关键字，您可以定义大型常量表并将它们分配到系统 ROM 中。例如，若要分配 ROM 表，可使用以下定义：

```
const int digits[] = {0,1,2,3,4,5,6,7,8,9};
```

6.5.2 __cregister 关键字

编译器通过添加 cregister 关键字来扩展 C/C++ 语言，从而使用高级别语言访问控制寄存器。此关键字在正常模式下可用，但在严格 ANSI/ISO 模式下不可用（使用 --strict_ansi 编译器选项）。备用关键字 __cregister 提供相同的功能，但在严格 ANSI/ISO 模式或正常模式下可用。

将 cregister 关键字用于一个对象时，编译器会将对象的名称与 C28x 的标准控制寄存器列表进行比较（请参阅表 6-2）。如果名称匹配，编译器将生成引用控制寄存器的代码。如果名称不匹配，编译器将发出错误。

表 6-2. 有效控制寄存器

寄存器	说明
IER	中断启用寄存器
IFR	中断标志寄存器

cregister 关键字只能在文件作用域内使用。函数边界内的任何声明都不允许使用 cregister 关键字，其只能用于整数或指针类型的对象。任何浮点类型的对象或任何结构体或联合体对象都不得使用 cregister 关键字。

cregister 关键字并不意味着对象是易失的。如果引用的控制寄存器是易失的（即，可以由某些外部控件修改），还必须使用 volatile 关键字声明该对象。

若要使用表 6-2 中的控制寄存器，必须按如下方式声明每个寄存器。c28x.h 头文件通过以下语法定义所有控制寄存器：

extern cregister volatile unsigned int *register* **;**

完成对寄存器的声明后，就可以直接使用寄存器名称，尽管使用方式有限。IFR 为只读，只能通过使用带立即数的 | (OR) 运算来置位。IFR 只能通过使用带立即数的 & (AND) 运算来清除。例如：

```
IFR |= 0x4;
IFR &= 0x0800
```

IER 寄存器进行分配时，可不使用 OR 和 AND 运算。因为 C28x 架构用于操作这些寄存器的指令有限，如发现寄存器的使用不合规，编译器将终止以下消息：

```
>>> Illegal use of control register
```

有关声明和使用控制寄存器的示例，请参阅定义和使用控制寄存器。

定义和使用控制寄存器

```
extern cregister volatile unsigned int IFR;
extern cregister volatile unsigned int IER;
extern int x;
main()
{
   IER = x;
   IER |= 0x100;
   printf("IER = %x\n", IER);
   IFR &= 0x100;
   IFR |= 0x100;
```

6.5.3 __interrupt 关键字

编译器通过添加 __interrupt 关键字来扩展 C/C++ 语言，该关键字指定函数被视为中断函数。此关键字是一个 IRQ 中断。除了在严格 ANSI C 或 C++ 模式中，还可以使用备用关键字“interrupt”。

请注意，节 6.9.15 中描述的中断函数属性是声明中断函数的推荐语法。

处理中断的函数遵循特殊的寄存器保存规则和特殊的返回序列。该实现方案强调安全性。中断例程不假定各种 CPU 寄存器和状态位的 C 运行时惯例有效；相反，它会重新建立运行时环境假定的任何值。当 C/C++ 代码被中断时，中断例程必须保留例程或例程所调用任何函数使用的所有机器寄存器的内容。在函数定义中使用 __interrupt 关键字时，编译器会根据中断函数的规则和中断的特殊返回序列生成寄存器保存。

您只能将 __interrupt 关键字与定义为返回 void 且没有参数的函数一同使用。中断函数的主体可以有局部变量，并且可以自由地使用栈或全局变量。例如：

```
__interrupt void int_handler()
{
    unsigned int flags;
    ...
}
```

名称 c_int00 是 c/c++ 的入口点。此名称是为系统复位中断而保留的。这个特殊的中断例程可初始化系统并调用 main() 函数。因为它没有调用方，所以 c_int00 不保存任何寄存器。

备注

Hwi 对象和 __interrupt 关键字：将 SYS/BIOS Hwi 对象与 C 函数一同使用时，不得使用 __interrupt 关键字。Hwi_enter/Hwi_exit 宏命令和 Hwi 调度程序已经包含此功能；使用 C 修饰符可能导致出现不希望的冲突。

6.5.4 restrict 关键字

为了帮助编译器确定内存依赖关系，可以使用 restrict 关键字来限定指针、引用或数组。restrict 关键字是一个类型限定符，可以应用于指针、引用和数组。使用它表示用户保证，在指针声明的范围内，指向的对象只能由该指针访问。任何违反此保证的行为都会导致程序未定义。这种做法可以帮助编译器优化代码的某些部分，因为这样可以更加轻松地确定别名信息。

“restrict”关键字是一个 C99 关键字，在严格的 ANSI C89 模式下不被接受。如果必须使用严格的 ANSI C89 模式，请使用“__restrict”关键字。请参阅节 6.14。

以下示例使用 restrict 关键字来告诉编译器，永远不会使用指向存储器中重叠对象的指针 a 和 b 来调用函数 func1。您承诺通过 a 和 b 进行访问永远不会发生冲突；因此，通过一个指针进行的写入操作不能影响通过任何其他指针进行的读取操作。ANSI/ISO C 标准的 1999 版本中描述了 restrict 关键字的精确语义。

```
void func1(int * restrict a, int * restrict b)
{
  /* func1's code here */
}
```

以下示例在将数组传递给函数时使用 restrict 关键字。在这里，数组 c 和 d 不得重叠，c 和 d 也不得指向同一数组。

```
void func2(int c[restrict], int d[restrict])
{
  int i;
  for(i = 0; i < 64; i++)
  {
    c[i] += d[i];
    d[i] += 1;
  }
}
```

目前，restrict 关键字仅对 FPU 目标有用。对于非 FPU 目标，则会忽略 restrict 关键字。

6.5.5 volatile 关键字

C/C++ 编译器在所有模式下都支持 *volatile* 关键字，但 。此外，在 C89、C99、C11 和 C++ 的宽松 ANSI 模式下支持 __volatile 关键字。

volatile 关键字指示编译器如何访问变量，这要求编译器不得投机取巧地优化涉及该变量的表达式。例如，外部程序、中断、另一个线程或外围设备也以访问该变量。

编译器会使用数据流分析来确定访问是否合法，从而尽可能消除冗余的存储器访问。不过，一些存储器访问可能在编译器未看到的方面比较特殊，在这类情况下，您应当使用 volatile 关键字来防止编译器优化掉某些重要内容。对于已声明为 volatile 的变量，编译器不会优化掉对该变量的任何访问。volatile 读取和写入的次数将与 C/C++ 代码中的完全相同，不多不少而且顺序也完全相同。

任何可能由明显程序控制流程外部的事物（例如中断服务例程）进行修改的变量必须声明为 volatile。这会告诉编译器，中断函数可能会随时修改该值，因此编译器不应执行会更改该变量的编号或访问顺序的优化。这就是 volatile 关键字的主要用途。在下述示例中，循环旨在等待位置被读取为 0xFF ：

```
   unsigned int *ctrl;
   while (*ctrl !=0xFF);
```

不过，在此示例中，*ctrl 是循环不变量表达式，因此循环会优化为单个存储器读取。若要获取所需结果，应将 ctrl 定义为：

```
   volatile unsigned int *ctrl;
```

其中，*ctrl 指针旨在引用一个硬件位置，例如中断标志。

访问表示存储器映射外围设备的存储器位置时，也必须使用 volatile 关键字。此类存储器位置可能会以编译器无法预测的方式更改值。这些位置可能会在被访问时、或者当其他存储器位置被访问时或者出现某些信号时发生改变。

在调用 setjmp 的函数中，如果局部变量的值需要在发生 longjmp 时保持有效，则 volatile 也必须用于局部变量。

```
#include <stdlib.h>
jmp_buf context;
void function()
{
    volatile int x = 3;
    switch(setjmp(context))
    {
        case 0: setup(); break;
        default:
        {
           /* We only reach here if longjmp occurs.因为 x 的生命周期在 setjmp 之前开始并持续至 longjmp，C
标准要求将 x 声明为 "volatile"。*/
           printf("x == %d\n", x);
           break;
        }
    }
}
```

如果存储器映射配置为单个统一空间，则可以使用 --unified_memory 选项；此选项允许编译器可以为大多数 memcpy 调用和结构赋值生成更高效的指令。即使在统一存储器时，一些外设的存储器以及一些与这些外设关联的 RAM 都只在数据存储器中分配。 如果启用了 –unified_memory，您可以通过将这些符号声明为 volatile，阻止程序通过存储器地址访问特定的符号。

6.6 C++ 异常处理

编译器支持根据 ANSI/ISO 14882 C++ 标准定义的 C++ 异常处理功能。请参阅由 Bjarne Stroustrup 编写的 *《C++ 编程语言》第三版*。编译器的 --exceptions 选项启用异常处理功能。编译器的默认设置是不支持异常处理。

若要在异常下正常工作，应用程序中的所有 C++ 文件都必须使用 --exceptions 选项进行编译，而不管该文件中是否存在异常。混合使用启用了异常和禁用了异常的目标文件和库可能导致未定义的行为。

异常处理需要在运行时支持库中得到支持，该库以启用异常和禁用异常的形式提供；您必须使用正确的表单链接。使用自动选择库（默认）选项时，链接器会自动选择正确的库，请参阅节 4.3.1.1。如果手动选择库，并且启用异常，则必须使用名称中包含 _eh 的运行时支持库。

使用 --exceptions 选项会导致编译器插入异常处理代码。这段代码会增加程序的大小。也会略微增加。COFF ABI 尤其如此。

有关运行时库的详细信息，请参阅节 8.1。

6.7 寄存器变量和参数

C/C++ 编译器对寄存器变量（用 register 关键字定义的变量）的处理方式不同，具体取决于您是否使用 --opt_level (-O) 选项。

- **进行优化的编译**

  编译器会忽略任何寄存器定义，并使用能够充分利用寄存器的算法将寄存器分配给变量和临时值。
- **不进行优化的编译**

  如果您使用 register 关键字，则可以建议将变量作为分配到寄存器的候选对象。编译器使用与分配寄存器变量时所用的同一组寄存器来分配临时表达式结果。

编译器尝试遵守所有寄存器定义。如果编译器将合适的寄存器耗尽，它会通过将寄存器内容移动到存储器来释放寄存器。如果您将太多对象定义为寄存器变量，则会限制编译器具有的用于临时表达式结果的寄存器数量。此限制会导致寄存器内容过多地移动到存储器中。

任何具有标量类型（整数、浮点或指针）的对象都可以被定义为寄存器变量。对于其他类型的对象（例如数组），将忽略寄存器指示符。

寄存器存储类对参数和局部变量都有意义。通常，在函数中，一些参数会被复制到堆栈上的某个位置，并在函数体内的这个位置被引用。编译器将寄存器参数复制到寄存器而不是堆栈中，从而加快对函数内参数的访问。

有关寄存器惯例的更多信息，请参阅节 7.2。

6.8 __asm 语句

C/C++ 编译器可以将汇编语言指令或指示直接嵌入到编译器的汇编语言输出中。此功能是对 C/C++ 语言的扩展，通过 *__asm* 关键字来实现。__asm 关键字提供了对 C/C++ 无法提供的硬件功能的访问。

除了在严格 ANSI C 模式中，也可以使用备用关键字“asm”。该关键字可以在宽松 C 和 C++ 模式下使用。

使用 __asm 在语法上是调用名为 __asm 的函数，其带有一个字符串常量参数：

```
__asm(" assembler text ");
```

编译器将参数字符串直接复制到输出文件中。汇编器文本必须用双引号括起来。所有常用的字符串转义码都保留其定义。例如，可插入包含引号的 .byte 指令，如下所示：

```
__asm("STR: .byte \"abc\"");
```

naked 函数属性可用于识别使用 __asm 语句写为嵌入式汇编函数的函数。请参阅节 6.15.2。

插入的代码必须是合法的汇编语言语句。与所有汇编语言语句一样，引号内的代码行必须以标签、空格、制表符或注释（星号或分号）开头。编译器不检查字符串；如果有错误，汇编器会检测到错误。有关汇编语言语句的更多信息，请参阅 *TMS320C28x 汇编语言工具用户指南*。

__asm 语句不遵循普通 C/C++ 语句的语法限制。每个语句都可以显示为语句或声明，甚至在块之外。这对于在编译模块的最开始插入指令非常有用。

__asm 语句不提供任何引用局部变量的方法。如果汇编代码需要引用局部变量，则需要在汇编代码中编写整个函数。

如需更多信息，请参阅节 7.5.5。

备注

避免使用 asm 语句破坏 C/C++ 环境

注意请勿使用 __asm 语句破坏 C/C++ 环境。编译器不会检查插入的指令。在 C/C++ 代码中插入跳转指令和标签可能会导致在插入的代码中或其周围操作的变量产生不可预测的结果。更改段或以其他方式影响汇编环境的指令也可能会造成很多麻烦。

在使用 __asm 语句进行优化时需谨慎。尽管编译器无法删除 __asm 语句，但会大规模重新排列它们附近的代码，并导致不期望的结果。

备注

对 RPT 指令使用单条 asm 语句

添加 C28x RPT 指令时，请勿对 RPT 和重复指令使用单独的 asm 语句。编译器可以在 asm 指令之间插入调试指令，而汇编器不允许在 RPT 和重复指令之间插入任何指令。例如，如需插入 RPT MAC 指令，请使用以下命令：

```
asm("\tRPT #10\n\t||MAC P, *XAR4++, *XAR7++");
```

6.9 pragma 指令

以下 pragma 指令告知编译器如何处理某个函数、对象或代码段。

- CALLS (请参阅节 6.9.1)
- CLINK (请参阅节 6.9.2)
- CODE_ALIGN (请参阅节 6.9.3)
- CODE_SECTION (请参阅节 6.9.4)
- DATA_ALIGN (请参阅节 6.9.5)
- DATA_SECTION (请参阅节 6.9.6)
- diag_suppress、diag_remark、diag_warning、diag_error、diag_default、diag_push、diag_pop (请参阅节 6.9.7)
- FAST_FUNC_CALL (请参阅节 6.9.8)
- FORCEINLINE (请参阅节 6.9.9)
- FORCEINLINE_RECURSIVE (请参阅节 6.9.10)
- FUNC_ALWAYS_INLINE (请参阅节 6.9.11)
- FUNC_CANNOT_INLINE (请参阅节 6.9.12)
- FUNC_EXT_CALLED (请参阅节 6.9.13)
- FUNCTION_OPTIONS (请参阅节 6.9.14)
- INTERRUPT (请参阅节 6.9.15)
- LOCATION (请参阅节 6.9.16)
- MUST_ITERATE (请参阅节 6.9.17)
- NOINIT (请参阅节 6.9.18)
- NOINLINE (请参阅节 6.9.19)
- NO_HOOKS (请参阅节 6.9.20)
- once (请参阅节 6.9.21)
- PERSISTENT (请参阅节 6.9.18)
- RETAIN (请参阅节 6.9.22)
- SET_CODE_SECTION (请参阅节 6.9.23)
- SET_DATA_SECTION (请参阅节 6.9.23)
- UNROLL (请参阅节 6.9.24)
- WEAK (请参阅节 6.9.25)

不能在函数主体内定义或声明参数 *func* 和 *symbol*。必须在函数主体之外指定 pragma，并且 pragma 规范必须出现在对 func 或 symbol 参数的任何声明、定义或引用之前。如果不遵守这些规则，编译器会发出警告并可能忽略 pragma。

对于应用于函数或符号的 pragma，C 和 C++ 的语法不同。

- 在 C 语言中，必须提供要将 pragma 作为第一个参数应用的对象或函数的名称。操作的实体是指定的，因此 C 语言中的 pragma 可能与该实体的定义相距一段距离。
- 在 C++ 中，pragma 具有定向性。它们不会将操作的实体命名为参数,而是作用于在 pragma 之后定义的下一个实体。

6.9.1 CALLS Pragma

CALLS pragma 指定一组可以从指定调用函数间接调用的函数。

编译器使用 CALLS pragma 在目标文件中嵌入有关间接调用的调试信息。对进行间接调用的函数使用 CALLS pragma，可以在计算此类函数的 inclusive 栈大小时包括此类间接调用。更多有关生成函数栈使用信息的信息，请参阅《*TMS320C28x 汇编语言工具用户指南*》中的“调用目标文件显示实用程序”部分。

CALLS pragma 可以位于调用函数的定义或声明之前。在 C 语言中，pragma 必须至少有 2 个参数。第一个参数是调用函数，后面至少有一个将从调用函数间接调用的函数。在 C++ 语言中，pragma 应用于所声明或定义的下一个函数，并且 pragma 必须至少有一个参数。

C 语言中 CALLS pragma 的语法如下所示。这表明 calling_function 可以通过 function_n 间接调用 function_1。

#pragma CALLS (*calling_function*, *function_1*, *function_2*, ..., *function_n* **)**

C++ 中 CALLS pragma 的语法是：

#pragma CALLS (*function_1_mangled_name*, ..., *function_n_mangled_name* **)**

注意，在 C++ 语言中，CALLS pragma 的参数必须是可从调用函数间接调用的函数的全名。

GCC 样式的“调用”函数属性（请参阅节 6.15.2）具有与 CALLS pragma 相同的效果，语法如下：

```
__attribute__((calls("function_1","function_2",..., "function_n")))
```

6.9.2 CLINK Pragma

备注

CLINK pragma 用于 COFF 模式。在 EABI 模式下使用时没有效果，因为默认情况下启用了条件链接。

CLINK pragma 可以应用于代码符号或数据符号。它会导致在包含符号定义的段中生成 .clink 指令。.clink 指令告诉链接器在条件链接过程中段有资格被删除。因此，如果段未被正在编译和链接的应用中的任何其他段引用，它将不会包含在生成的输出文件中。

C 中 pragma 的语法为：

#pragma CLINK (*symbol* **)**

C++ 中 pragma 的语法为：

#pragma CLINK

RETAIN pragma 与 CLINK pragma 的效果相反。请参阅节 6.9.22 了解更多详细信息。

6.9.3 CODE_ALIGN Pragma

CODE_ALIGN pragma 沿指定的对齐方式对齐 ***func***。对齐 ***constant*** 必须是 2 的幂。如果要在某个边界上启动函数，CODE_ALIGN pragma 会非常有用。

CODE_ALIGN pragma 与使用 GCC 样式的 `aligned` 函数属性的效果相同。请参阅节 6.15.2。

C 中 pragma 的语法为：

#pragma CODE_ALIGN (*func* **,** *constant* **)**

C++ 中 pragma 的语法为：

#pragma CODE_ALIGN (*constant* **)**

6.9.4 CODE_SECTION Pragma

CODE_SECTION pragma 在一个名为 *section name* 的段中，为 C 中的 *symbol* 或在 C++ 中声明的下一个 symbol 分配空间。如果要将代码对象链接到与 .text 段分开的区域，CODE_SECTION pragma 会非常有用。CODE_SECTION pragma 具有与使用 GCC 样式 `section` 函数属性相同的效果。请参阅节 6.15.2。

C 中 pragma 的语法为：

#pragma CODE_SECTION (*symbol* **, "** *section name* **")**

C++ 中 pragma 的语法为：

#pragma CODE_SECTION (" *section name* **")**

以下示例演示了 CODE_SECTION pragma 的用法。

在 C 中使用 CODE_SECTION Pragma

```
char bufferA[80];
char bufferB[80];
#pragma CODE_SECTION(funcA, "codeA")
char funcA(int i);
char funcB(int i);
void main()
{
   char c;
   c = funcA(1);
   c = funcB(2);
}
char funcA (int i)
{
   return bufferA[i];
}
char funcB (int j)
{
   return bufferB[j];
}
```

以下示例 C 代码将生成下列汇编代码：

```
        .sect   ".text"
        .global _main;
*****************************************************************
;* FNAME: _main                          FR SIZE:   2           *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;*                            0 Parameter,  1 Auto,  0 SOE      *
;*****************************************************************
:_main:
        ADDB      SP,#2
        MOVB      AL,#1                 ; |12|
        LCR       #_funcA               ; |12|
               ; call occurs [#_funcA] ; |12|
        MOV       *-SP[1],AL            ; |12|
        MOVB      AL,#1                 ; |13|
        LCR       #_funcB               ; |13|
                ; call occurs [#_funcB] ; |13|
        MOV       *-SP[1],AL            ; |13|
        SUBB      SP,#2
        LRETR
        ; return occurs
        .sect   "codeA"
        .global _funcA
;*****************************************************************
;* FNAME: _funcA                         FR SIZE:   1           *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;*                            0 Parameter,  1 Auto,  0 SOE      *
;*****************************************************************
_funcA:
        ADDB      SP,#1
        MOV       *-SP[1],AL            ; |17|
        MOVZ      AR6,*-SP[1]           ; |18|
        ADD       AR6,#_bufferA         ; |18|
        SUBB      SP,#1                 ; |18|
        MOV       AL,*+XAR6[0]          ; |18|
        LRETR
        ;return occurs
         .sect   ".text"
         .global _funcB;
*****************************************************************
;* FNAME: _funcB                         FR SIZE:   1           *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;*                            0 Parameter,  1 Auto,  0 SOE      *
;*****************************************************************
_funcB:
        ADDB      SP,#1
        MOV       *-SP[1],AL            ; |22|
        MOVZ      AR6,*-SP[1]           ; |23|
        ADD       AR6,#_bufferB         ; |23|
        SUBB      SP,#1                 ; |23|
        MOV       AL,*+XAR6[0]          ; |23|
        LRETR
        ;return occurs
```

6.9.5 DATA_ALIGN Pragma

DATA_ALIGN pragma 将 C 中的 *symbol* 或在 C++ 中声明的下一个 symbol 对齐到对齐边界。对齐边界是 symbol 的默认对齐值或 *constant* 值中的最大值（以字节为单位）。常数必须是 2 的幂。最大对齐为 32768。

DATA_ALIGN pragma 不能用于减少对象的自然对齐。

使用 DATA_ALIGN pragma 与使用 GCC 样式 `aligned` 变量属性具有相同的效果。请参阅节 6.15.4。

C 中 pragma 的语法为：

#pragma DATA_ALIGN (*symbol* **,** *constant* **)**

C++ 中 pragma 的语法为：

#pragma DATA_ALIGN (*constant* **)**

6.9.6 DATA_SECTION Pragma

DATA_SECTION pragma 在一个名为 *section name* 的段中，为 C 中的 *symbol* 或在 C++ 中声明的下一个 symbol 分配空间。如要将数据对象链接至一个独立于 .ebss or .bss 段的区域，此 pragma 很有用。

使用 DATA_SECTION pragma 与使用 GCC 样式的 `section` 变量属性的效果相同。请参阅节 6.15.4。

C 中 pragma 的语法为：

#pragma DATA_SECTION (*symbol* **, "** *section name* **")**

C++ 中 pragma 的语法为：

#pragma DATA_SECTION (" *section name* **")**

示例 6-1 到示例 6-3 演示了 DATA_SECTION pragma 的用法。

示例 6-1. 使用 DATA_SECTION Pragma C 源文件

```
#pragma DATA_SECTION(bufferB, "my_sect")
char bufferA[512];
char bufferB[512];
```

示例 6-2. 使用 DATA_SECTION Pragma C++ 源文件

```
char bufferA[512];
#pragma DATA_SECTION("my_sect")
char bufferB[512];
```

示例 6-3. 使用 DATA_SECTION Pragma 汇编源文件

```
            .global _bufferA
            .ebss    _bufferA,512,4
            .global _bufferB
_bufferB:   .usect  "my_sect",512,4
```

6.9.7 诊断消息 Pragma

下述 pragma 可用于控制诊断消息，其方法与相应的命令行选项相同：

Pragma	选项	说明
diag_suppress *num*	-pds=*num*[, num_2, num_3...]	抑制诊断 *num*
diag_remark *num*	-pdsr=*num*[, num_2, num_3...]	将诊断 *num* 视为备注
diag_warning *num*	-pdsw=*num*[, num_2, num_3...]	将诊断 *num* 视为警告
diag_error *num*	-pdse=*num*[, num_2, num_3...]	将诊断 *num* 视为错误
diag_default *num*	不适用	使用诊断的默认严重性
diag_push	不适用	推送当前诊断严重性状态以将其存储起来以备后用。
diag_pop	不适用	弹出与 #pragma diag_push 一同存储的最新诊断严重性状态作为当前设置。

在 C 语言中，diag_suppress、diag_remark、diag_warning 和 diag_error pragmas 的语法为：

#pragma diag_ *xxx* [=]*num*[, num_2, num_3...]

请注意，这些 pragma 的名称是小写的。

使用错误号或错误标记名称指定受影响的诊断 (*num*)。等号 (=) 是可选的。任何诊断都可以被覆盖为错误，但只有严重性为任意错误或以下的诊断消息才能将其严重性降低为警告或以下，或被抑制。diag_default pragma 用于将诊断的严重性返回到发出任何 pragma 之前有效的诊断（即，由任何命令行选项修改的消息的正常严重性）。

使用 -pden 命令行选项时，诊断标识符编号将与消息一同输出。

6.9.8 FAST_FUNC_CALL Pragma

当应用于函数时，FAST_FUNC_CALL pragma 会生成一条 TMS320C28x FFC 指令来调用函数，而不是 CALL 指令。有关 FFC 指令的更多详细信息，请参阅《*TMS320C28x DSP CPU 和指令集用户指南*》。

C 中 pragma 的语法为：

#pragma FAST_FUNC_CALL (*func* **)**

C++ 中 pragma 的语法为：

#pragma FAST_FUNC_CALL (*func* **)**

FAST_FUNC_CALL pragma 应仅应用于对随 LB *XAR7 指令返回的汇编函数的调用。有关组合 C/C++ 和汇编代码的信息，请参阅节 7.5.1。

此 pragma 应仅应用于汇编函数，因此如果编译器在文件作用域中找到 *func* 的定义，它将发出警告并忽略该 pragma。

以下示例演示了 FAST_FUNC_CALL pragma 的用法。

使用 FAST_FUNC_CALL Pragma 汇编函数

```
_add_long:
        ADD ACC, *-SP[2]
        LB *XAR7
```

使用 FAST_FUNC_CALL Pragma C 源文件

```
#pragma FAST_FUNC_CALL (add_long)
long add_long(long, long);
void foo()
{
   long x, y;
   x = 0xffff;
   y = 0xff;
```

```
    y = add_long(x, y);
}
```

生成的汇编文件

```
;***************************************************************
;* FNAME: _foo                          FR SIZE:   6           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                           2 Parameter,  4 Auto,  0 SOE      *
;***************************************************************
_foo:
        ADDB      SP,#6
        MOVB      ACC,#255
        MOVL      XAR6,#65535           ; |8|
        MOVL      *-SP[6],ACC
        MOVL      *-SP[2],ACC           ; |10|
        MOVL      *-SP[4],XAR6          ; |8|
        MOVL      ACC,*-SP[4]           ; |10|
        FFC       XAR7,#_add_long       ; |10|
        ; call occurs [#_add_long]      ; |10|
        MOVL      *-SP[6],ACC           ; |10|
        SUBB      SP,#6
        LRETR
        ; return occurs
```

6.9.9 FORCEINLINE Pragma

FORCEINLINE pragma 可以放置在语句前，以强制将该语句中的任何函数调用内联。它对相同函数的其他调用没有影响。

编译器仅在合法内联函数的情况下内联函数。如果使用 --opt_level=off 选项调用编译器，则函数不会内联。即使函数未使用 inline 关键字进行声明，也可以内联函数。即使未使用任何 --opt_level 命令行选项来调用编译器，函数也可以内联。

此 pragma 在 C/C++ 中的语法为：

```
#pragma FORCEINLINE
```

例如，在下面的示例中，mytest() 和 getname() 函数是内联函数，而 error() 函数不是内联函数。

```
#pragma FORCEINLINE
if (!mytest(getname(myvar))) {
    error();
}
```

在调用 error() 之前放置 FORCEINLINE pragma 将内联该函数，但不会内联其他函数。

如需了解影响内联的命令行选项、pragma 和关键字之间的交互作用，请参阅节 2.11。

请注意，FORCEINLINE、FORCEINLINE_RECURSIVE 和 NOINLINE pragma 只影响 pragma 后面的 C/C++ 语句。FUNC_ALWAYS_INLINE 和 FUNC_CANNOT_INLINE pragma 影响整个函数。

6.9.10 FORCEINLINE_RECURSIVE Pragma

FORCEINLINE_RECURSIVE 可以放置在语句前，以强制在该语句中进行的任何函数调用与从这些函数进行的任何调用一起内联。也就是说，在语句中不可见但作为语句结果被调用的调用将被内联。

此 pragma 在 C/C++ 中的语法为：

```
#pragma FORCEINLINE_RECURSIVE
```

有关影响内联的命令行选项、pragma 和关键字之间的交互信息，请参阅节 2.11。

6.9.11 FUNC_ALWAYS_INLINE Pragma

FUNC_ALWAYS_INLINE pragma 指示编译器始终内联命名函数。

编译器仅在内联函数是合法的情况下内联函数。如果使用 --opt_level=off 选项来调用编译器，则绝不会内联函数。即使函数未使用 inline 关键字进行声明，也可以内联函数。即使未使用任何 --opt_level 命令行选项来调用编译器，也可以内联函数。有关各种类型的内联之间交互的详细信息，请参阅节 2.11。

此 pragma 必须出现在针对要内联的函数进行的任何声明或引用之前。在 C 语言中，参数 *func* 是将被内联的函数名称。在 C++ 中，pragma 适用于下一个声明的函数。

FUNC_ALWAYS_INLINE pragma 与使用 GCC 样式 `always_inline` 的函数的效果相同。请参阅节 6.15.2。

C 中 pragma 的语法为：

#pragma FUNC_ALWAYS_INLINE (*func*)

C++ 中 pragma 的语法为：

#pragma FUNC_ALWAYS_INLINE

下述示例使用此 pragma：

```
#pragma FUNC_ALWAYS_INLINE(functionThatMustGetInlined)
static inline void functionThatMustGetInlined(void) {
    P1OUT |= 0x01;
    P1OUT &= ~0x01;
}
```

备注

使用 FUNC_ALWAYS_INLINE Pragma 时要小心：FUNC_ALWAYS_INLINE pragma 会覆盖编译器的内联决策。过度使用此 pragma 会导致编译时间或内存使用量增加，可能会耗尽所有可用存储器，并导致编译工具失效。

6.9.12 FUNC_CANNOT_INLINE Pragma

FUNC_CANNOT_INLINE pragma 指示编译器命名函数不能内联展开。使用此 pragma 命名的任何函数都会覆盖您以任何其他方式指定的任何内联，例如使用内联关键字。自动内联也会被此 pragma 覆盖；请参阅节 2.11。

此 pragma 必须出现在要保留的函数的任何声明或引用之前。在 C 语言中，参数 *func* 是不能内联的函数名。在 C++ 中，pragma 应用于所声明的下一个函数。

FUNC_CANNOT_INLINE pragma 具有与使用 GCC 样式 `noinline` 函数属性相同的效果。请参阅节 6.15.2。

C 中 pragma 的语法为：

#pragma FUNC_CANNOT_INLINE (*func*)

C++ 中 pragma 的语法为：

#pragma FUNC_CANNOT_INLINE

6.9.13 FUNC_EXT_CALLED Pragma

使用 --program_level_compile 选项时，编译器使用程序级优化。使用这种类型的优化时，编译器将删除 main() 未直接或间接调用的任何函数。您的 C/C++ 函数可能而不是通过 main() 调用。

FUNC_EXT_CALLED pragma 指定优化器应保留这些 C 函数或这些 C/C++ 函数调用的任何函数。这些函数充当 C/C++ 的入口点。此 pragma 必须出现在针对要保留的函数进行的任何声明或引用之前。在 C 语言中，参数 *func* 是要保留的函数名。在 C++ 中，pragma 适用于下一个声明的函数。

C 中 pragma 的语法为：

```
#pragma FUNC_EXT_CALLED ( func )
```

C++ 中 pragma 的语法为：

```
#pragma FUNC_EXT_CALLED
```

除了为 C/C++ 程序系统复位中断预留的名称 _c_int00 之外，中断的名称（*func* 参数）无需遵循命名惯例。

使用程序级优化时，可能需要使用 FUNC_EXT_CALLED pragma 和某些选项。请参阅节 3.4.2。

6.9.14 FUNCTION_OPTIONS Pragma

FUNCTION_OPTIONS pragma 允许使用附加的命令行编译器选项在 C 或 C++ 文件中编译特定的函数。受影响的函数将被编译，就像指定的选项列表出现在所有其他编译器选项之后的命令行上一样。在 C 语言中，pragma 应用于指定的函数。在 C++ 语言中，pragma 应用于下一个函数。

C 中 pragma 的语法为：

```
#pragma FUNCTION_OPTIONS ( func , " additional options " )
```

C++ 中 pragma 的语法为：

```
#pragma FUNCTION_OPTIONS( " additional options " )
```

此 pragma 支持的选项包括 --opt_level、--auto_inline、--code_state、--opt_for_space 和 --opt_for_speed。

为了将 --opt_level 和 --auto_inline 与 FUNCTION_OPTIONS pragma 一同使用，必须在某种优化级别（即至少 --opt_level=0）调用编译器。如果 --opt_level=off，则忽略 FUNCTION_OPTIONS pragma。
FUNCTION_OPTIONS pragma 不能用于完全禁用编译函数的优化器；可以指定的最低优化级别是 --opt_level=0。

6.9.15 INTERRUPT Pragma

借助 INTERRUPT pragma，您可以使用 C 代码来直接处理中断。

在 C 语言中，参数 *func* 是函数的名称。C 中 pragma 的语法为：

```
#pragma INTERRUPT ( func )
```

在 C++ 中，pragma 适用于下一个声明的函数。C++ 中 pragma 的语法为：

```
#pragma INTERRUPT
void func ( void )
```

GCC 中断属性具有与 INTERRUPT pragma 相同的效果，语法如下所示。请注意，该中断属性可以放在函数的定义或其声明之前。

```
__attribute__((interrupt )) void func ( void )
```

FPU 上有两种类型的中断：高优先级中断 (HPI) 和低优先级中断 (LPI)。高优先级中断执行快速上下文保存操作并无法进行嵌套。低优先级中断与正常的 C28x 中断相似并可以进行嵌套。使用 pragma 的可选第二个参数指定中断类型。C 中 pragma 的语法为：

#pragma INTERRUPT (*func* **, {HPI|LPI})**

C++ 中 pragma 的语法为：

#pragma INTERRUPT ({HPI|LPI})

GCC 中断属性具有与 INTERRUPT pragma 相同的效果，语法如下所示：

__attribute__((interrupt("HPI"|"LPI"))) void *func* **(** void **)** { ...}

在 FPU 上，如果未指定中断优先级，则假定为 LPI。中断关键字中指定的中断也默认为 LPI。

CLA 中断和 CLA2 后台任务可以通过中断属性或 INTERRUPT pragma 来创建。例如，下述两个都可以与 CLA 中断搭配使用：

```
__attribute__((interrupt))
void interrupt_name(void) {...}
#pragma INTERRUPT(interrupt _name);
void interrupt _name(void) {...}
```

以下两个示例都会创建一个 CLA2 后台任务：

```
__attribute__((interrupt("BACKGROUND")))
 void task_name(void) {...}
#pragma INTERRUPT(task_name, "BACKGROUND");
void task_name(void) {...}
```

要在使用后台任务时启用非后台中断的寄存器保存和恢复，请指定 --cla_background_task=on 选项。有关详细信息，请参阅节 10.1.1。

备注

Hwi 对象和 INTERRUPT Pragma：当将 SYS/BIOS Hwi 对象与 C 函数一同使用时，不得使用 INTERRUPT pragma。Hwi_enter/Hwi_exit 宏命令和 Hwi 调度程序都包含此功能，并且使用 C 修饰符会导致出现负面结果。

6.9.16 LOCATION Pragma

该编译器支持在源代码级别上指定变量的运行时地址，可通过使用 LOCATION pragma 或 GCC 样式的位置属性来实现。LOCATION pragma 与使用 GCC 样式的 `location` 函数属性的效果相同。请参阅节 6.15.2。

备注

此 pragma 仅在与 EABI 搭配使用时受支持。不支持与 COFF ABI 搭配使用。

C 中 pragma 的语法为：

#pragma LOCATION(*x* **,** *address* **)**
int *x*

这两个 pragma 在 C++ 语言中的语法为：

#pragma LOCATION(*address* **)**

```
int x
```

GCC 样式属性（请参阅节 6.15.4）的语法为：

```
int x __attribute__((location( address )))
```

NOINIT pragma 可与 LOCATION pragma 结合使用以将变量映射到特定的存储器位置；请参阅节 6.9.18。

6.9.17 MUST_ITERATE Pragma

MUST_ITERATE pragma 为编译器指定循环的某些属性。使用此 pragma，即表示向编译器保证循环会执行特定的次数或在指定范围内执行多次。

只要向循环应用 UNROLL pragma，则应向同一循环应用 MUST_ITERATE。对于循环，MUST_ITERATE pragma 的第三个参数 multiple 最为重要，并且始终应该指定。

另外，应当尽可能多地向任何其他循环应用 MUST_ITERATE pragma。这是因为通过该 pragma 提供的信息（尤其是最小迭代次数）能够帮助编译器选择最优循环和循环变换（即嵌套循环变换）。此外，该 pragma 还可帮助编译器缩减代码大小。

MUST_ITERATE pragma 与其适用的 for、while 或 do-while 循环之间不能包含任何语句。不过，MUST_ITERATE pragma 与相应循环之间可以存在 UNROLL 等其他 pragma。

6.9.17.1 MUST_ITERATE Pragma 语法

该 pragma 的 C 和 C++ 语法为：

```
#pragma MUST_ITERATE ( min, max, multiple )
```

对应属性的 C++ 语法如下所示。没有可用的 C 属性语法。

```
[[TI::must_iterate( min, max, multiple )]]
```

参数 *min* 和 *max* 是由程序员保证的最小和最大行程计数。行程计数是指循环迭代的次数。循环的行程计数必须能够被 *multiple* 整除。所有参数都是可选的。例如，如果行程计数可以为 5 或更大值，那么您可以按如下所示指定参数列表：

```
#pragma MUST_ITERATE(5)
```

不过，如果行程计数可以为 5 的任何非零倍数，该 pragma 会类似如下：

```
#pragma MUST_ITERATE(5, , 5) /* Note the blank field for max */
```

有时为了让编译器展开，需要提供 min 和 multiple。当编译器无法轻松地确定循环要执行的迭代次数（即循环具有复杂的退出条件）时，尤其如此。

通过 MUST_ITERATE pragma 指定 multiple 时，如果行程计数不能被 multiple 整除，程序的结果会为 undefined。另外，如果行程计数小于指定的最小值或大于指定的最大值，程序的结果也会是 undefined。

如果未指定 min，则会使用 0。如果未指定 max，则会使用可能的最大值。如果为同一循环指定了多个 MUST_ITERATE pragma，则会使用最小的 max 和最大的 min。

下述示例使用 must_iterate C++ 属性语法：

```
void myFunc (int *a, int *b, int * restrict c, int n)
{
    ...
    [[TI::must_iterate(32, 1024, 16)]]
    for (int i = 0; i < n; i++)
```

```
    {
        c[i] = a[i] + b[i];
    }
    ...
}
```

6.9.17.2 使用 MUST_ITERATE 扩展编译器对循环的了解

通过使用 MUST_ITERATE pragma，可以保证循环执行一定的次数。下述示例会告知编译器，循环保证可以正好运行 10 次：

```
#pragma MUST_ITERATE(10,10)
for(i = 0; i < trip_count; i++)  { ...
```

在此示例中，即使没有 pragma，编译器也尝试生成循环。但如果没有为这样的循环指定 MUST_ITERATE，编译器会生成代码绕过循环，以解决可能出现的 0 次迭代。利用 pragma 规范，编译器知道循环至少会迭代一次，可以消除循环绕过代码。

MUST_ITERATE 可用于指定循环计数的范围以及循环计数的系数。下述示例会告知编译器，循环执行 8 次到 48 次之间，trip_count 变量是 8 的倍数（8、16、24、32、40、48）。倍数参数支持编译器展开循环。

```
#pragma MUST_ITERATE(8, 48, 8)
for(i = 0; i < trip_count; i++)  { ...
```

对于具有复杂边界的循环，应考虑使用 MUST_ITERATE。在下述示例中，编译器不得不生成一个除法函数调用，以便在运行时确定所执行的迭代次数。

```
for(i2 = ipos[2]; i2 < 40; i2 += 5)  { ...
```

编译器不会执行上述操作。在这种情况下，使用 MUST_ITERATE 指定循环始终执行八次，编译器将尝试生成循环：

```
#pragma MUST_ITERATE(8, 8)
for(i2 = ipos[2]; i2 < 40; i2 += 5)  { ...
```

6.9.18 NOINIT 和 PERSISTENT Pragma

默认情况下，全局和静态变量均会初始化为 0。不过，在使用非易失性存储器的应用中，可能最好不要包含已被初始化的变量。Noinit 变量是在启动或复位时不会初始化为 0 的全局或静态变量。

备注

这些 pragma 仅在与 EABI 搭配使用时受支持。它们不支持与 COFF ABI 搭配使用。

可以使用 pragma 或变量属性将变量声明为 noinit 或 persistent。有关在声明中使用变量属性的信息，请参阅节 6.15.4。

除是否在加载时进行初始化之外，Noinit 和 persistent 变量的作用完全相同。

- NOINIT pragma 只能与未初始化的变量搭配使用。其防止在复位时将此类变量设置为 0。其可以与 LOCATION pragma 结合使用来将变量映射到特定的存储器位置，例如存储器映射寄存器，从而免受意外写入。
- PERSISTENT pragma 只能与静态初始化的变量搭配使用。其防止在复位时初始化此类变量。Persistent 变量禁用启动初始化功能；当加载代码时,这些变量被赋予一个初始值，但不会再次被初始化。

默认情况下，noinit 或 persistent 变量将分别置于名为 `.TI.noinit` 和 `.TI.persistent` 的字段中。这些字段的位置由链接器命令文件控制。通常对于支持 FRAM 的器件，.TI.persistent 段置于 FRAM 中，.TI.noinit 段置于 RAM 中。

备注

在非易失性 FRAM 存储器中使用这些 pragma 时，可以通过器件的存储器保护单元来保护存储器区域免受意外写入。有些器件会默认启用存储器保护功能。有关存储器保护的信息，请参阅器件数据表。如果启用了存储器保护单元，那么在修改变量前需要先禁用该功能。

如果您使用的是非易失性 RAM，则可以定义 persistent 变量,将其初始值 0 载入 RAM 中。该程序可以让该变量随时间推移而递增来用作计数器，并且该计数不会因为器件断电和重新启动而消失，因为该存储器为非易失性存储器并且引导例程不会将其初始化为 0。例如：

```
#pragma PERSISTENT(x)
#pragma location = 0xC200   // memory address in RAM
int x = 0;
void main() {
    run_init();
    while (1) {
        run_actions(x);
        __delay_cycles(1000000);
        x++;
    }
}
```

这两个 pragma 在 C 语言中的语法为：

```
#pragma NOINIT ( x )
int x ;
```

```
#pragma PERSISTENT ( x )
int x =10;
```

这两个 pragma 在 C++ 语言中的语法为：

```
#pragma NOINIT
int x ;
```

```
#pragma PERSISTENT
int x =10;
```

GCC 属性的语法为：

```
int x __attribute__((noinit));
```

```
int x __attribute__((persistent)) = 0;
```

6.9.19 NOINLINE Pragma

NOINLINE pragma 可以放置在一条语句之前，用于防止该语句中所做的任何函数调用发生内联。该 pragma 对相同函数的其他调用则没有影响。

此 pragma 在 C/C++ 中的语法为：

```
#pragma NOINLINE
```

有关影响内联的命令行选项、pragma 和关键字之间的交互使用的信息，请参阅节 2.11。

6.9.20 NO_HOOKS Pragma

NO_HOOKS pragma 用于防止为一个函数而生成入口和出口钩子程序调用。

C 中 pragma 的语法为：

```
#pragma NO_HOOKS ( func )
```

C++ 中 pragma 的语法为：

```
#pragma NO_HOOKS
```

有关入口和出口钩子程序的详细信息，请参阅节 2.14。

6.9.21 once Pragma

once pragma 指示如果已包含该头文件，则 C 预处理程序要忽略 #include 指令。例如，如果头文件包含结构定义等定义，并且这些定义执行超过一次时会导致编译错误，则可以使用此 pragma。

此 pragma 应该用在只应包含一次的头文件的开头部分。例如：

```
// hdr.h
#pragma once
#warn You will only see this message one time
struct foo
{
    int member;
};
```

此 pragma 不是 C 或 C++ 标准的一部分，但它在预处理指令中广泛受到支持。请注意，此 pragma 不能防止包含已复制到其他目录且包含相同内容的头文件。

6.9.22 RETAIN Pragma

RETAIN pragma 可以应用于代码或数据符号。

EABI 模式假定所有段都*符合*通过条件链接移除的条件，在该模式下，此 pragma 会导致包含该符号定义的段中生成 .retain 指令。.retain 指令向链接器指示该段不符合在条件链接期间进行移除的条件。因此，不管正在编译和链接的应用程序中的其他段是否引用了该段，该段都会包含在链接的输出文件结果中。

在 COFF 模式下,假定所有段都*不符合*移除的条件，RETAIN pragma 会阻止编译器为符号发出 .clink 指令。

RETAIN pragma 与使用 `retain` 函数或变量属性的效果相同。请分别参阅节 6.15.2 和节 6.15.4。

C 中 pragma 的语法为：

#pragma RETAIN (*symbol* **)**

C++ 中 pragma 的语法为：

#pragma RETAIN

CLINK pragma 的效果与 RETAIN pragma 相反,并且只在 COFF 模式下使用。有关更多详细信息,请参阅节 6.9.2 。

6.9.23 SET_CODE_SECTION 和 SET_DATA_SECTION Pragma

这些 pragma 可用于为 pragma 下方的所有声明设置段。

这些 pragma 在 C/C++ 中的语法为：

#pragma SET_CODE_SECTION (" *section name* ")

#pragma SET_DATA_SECTION (" *section name* ")

在通过 SET_DATA_SECTION Pragma 设置段示例中，x 和 y 被置于 mydata 段中。若要将当前段复位为编译器使用的默认段，则应该向该 pragma 传递空白参数。简单来说，该 pragma 就像是会为其下方的所有符号应用 CODE_SECTION 或 DATA_SECTION pragma。

通过 SET_DATA_SECTION Pragma 设置段

```
#pragma SET_DATA_SECTION("mydata")
int x;
int y;
#pragma SET_DATA_SECTION()
```

这些 pragma 会应用到声明和定义。如果应用到声明而不应用到定义，该 pragma 会在声明中处于活动状态，用于为该符号设置对应的段。下面我们举例说明：

通过 SET_CODE_SECTION Pragma 设置段

```
#pragma SET_CODE_SECTION("func1")
extern void func1();
#pragma SET_CODE_SECTION()
...
void func1() { ...}
```

在通过 SET_CODE_SECTION Pragma 设置段 示例中，func1 被置于 func1 段中。如果声明和定义中指定了相互冲突的段，则会发出诊断。

当前的 CODE_SECTION 和 DATA_SECTION pragma 以及 GCC 属性可用于覆盖 SET_CODE_SECTION 和 SET_DATA_SECTION pragma。例如：

覆盖 SET_DATA_SECTION 设置

```
#pragma DATA_SECTION(x, "x_data")
#pragma SET_DATA_SECTION("mydata")
int x;
int y;
#pragma SET_DATA_SECTION()
```

在覆盖 SET_DATA_SECTION 设置 示例中，x 被置于 x_data 中，而 y 被置于 mydata 中。这种情况下不会发出诊断。

这些 pragma 适用于 C 和 C++。在 C++ 中，会针对模板和隐式创建的对象（例如隐式构造函数和虚拟函数表格）忽略这些 pragma。

如果使用 SET_DATA_SECTION pragma，其优先级会高于 --gen_data_subsections=on 选项。

6.9.24 UNROLL Pragma

UNROLL pragma 向编译器指定了循环应该展开的次数。必须调用优化器（使用 --opt_level=[1|2|3] 或者 -O1、-O2 或 -O3），该 pragma 指定的循环才会展开。编译器具有忽略此 pragma 的选项。

UNROLL pragma 与其适用的 for、while 或 do-while 循环之间不能包含任何语句。不过，UNROLL pragma 与相应循环之间可以存在 MUST_ITERATE 等其他 pragma。

C 和 C++ 中该 pragma 语法为：

```
#pragma UNROLL( n )
```

对应属性的 C++ 语法如下所示。不存在 C 属性语法。有关使用类似语法的示例，请参阅节 6.9.17.1。

```
[[TI::unroll( n )]]
```

如果可以，编译器会展开该循环，使得原始循环存在 *n* 个副本。编译器仅在可以确定按 *n* 的倍数展开是安全的情况下才会展开。为了增加循环展开的几率，编译器需要知道一些属性：

- 循环的迭代次数必须为 *n* 的倍数。此信息可以通过 MUST_ITERATE pragma 中的多个参数来向编译器指定。
- 循环的最小迭代次数
- 循环的最大迭代次数

编译器有时可以通过分析代码来自行获取此信息。不过，编译器有时可能对其假定过于保守，因此生成的代码会多于展开时所必需的代码。这也可能会导致完全不会展开。另外，如果用于确定循环何时应该退出的机制比较复杂，编译器可能无法确定循环的这些属性。在这些情况下，您必须通过使用 MUST_ITERATE pragma 告知编译器循环的属性。

指定 #pragma UNROLL(1) 会让循环不展开。在这种情况下，也不会执行自动循环展开。

如果为同一循环指定了多个 UNROLL pragma，则具体使用哪个 pragma（若有）为未定义。

6.9.25 WEAK Pragma

WEAK pragma 用于针对符号提供弱绑定。

备注

此 pragma 仅在与 EABI 搭配使用时受支持。不支持与 COFF ABI 搭配使用。

C 中 pragma 的语法为：

```
#pragma WEAK ( symbol )
```

C++ 中 pragma 的语法为：

```
#pragma WEAK
```

如果 *symbol* 为引用，WEAK pragma 会使它成为弱引用；如果为定义，则会使它成为弱定义。符号可以是数据或函数变量。实际上，未解析的弱 *引用* 不会导致链接器错误，在运行时也没有任何效果。以下适用于弱引用：

- 不会搜索库来解析弱引用。弱引用保持未解析状态并不是错误。
- 在链接期间，未定义弱引用的值为：
  - 零（如果重定位类型为绝对地址）
  - 位置地址（如果重定位类型为 PC 相对地址）
  - 标称基地址的地址（如果重定位类型为基址相对地址）

弱 *定义* 不会更改从库中选择目标文件所遵循的规则。不过，如果链接集同时包含弱定义和非弱定义，则始终会使用非弱定义。

WEAK pragma 具有与使用 `weak` 函数或变量属性相同的效果。请分别参阅节 6.15.2 和节 6.15.4。

6.10 _Pragma 运算符

C28x C/C++ 编译器支持 C99 预处理器 _Pragma() 运算符。此预处理器运算符类似于 #pragma 指令。但是，_Pragma 可用于预处理宏命令 (#defines)。

运算符的语法为：

_Pragma (" *string_literal* ");

参数 *string_literal* 的解释方式与 #pragma 指令之后的标记的处理方式相同。string_literal 必须用引号括起来。作为 string_literal 的一部分的引号前面必须有反斜杠。

可以使用 _Pragma 运算符在宏命令中表示 #pragma 指令。例如，DATA_SECTION 语法：

#pragma DATA_SECTION(*func* ," *section* ")

由 _Pragma() 运算符语法表示：

_Pragma ("DATA_SECTION(*func* ,\" *section* \")")

以下代码演示了如何使用 _Pragma 在宏命令中指定 DATA_SECTION pragma：

```
...
#define EMIT_PRAGMA(x) _Pragma(#x)
#define COLLECT_DATA(var) EMIT_PRAGMA(DATA_SECTION(var,"mysection"))
COLLECT_DATA(x)
int x;
...
```

需要使用 EMIT_PRAGMA 宏命令将段参数周围所需的引号正确展开为 DATA_SECTION pragma。

6.11 应用程序二进制接口

应用程序二进制接口 (ABI) 定义了单独编译或汇编（可能使用不同供应商的编译器进行编译）的函数如何协同工作。这涉及到数据类型表示、寄存器惯例、函数结构和调用惯例的标准化。ABI 定义了从 C 符号名称生成的链接名称。它定义了目标文件格式和调试格式。它定义了系统初始化的方式。如果是 C++，它则定义了对 C++ 名称的处理和异常处理支持。

TI C28x 代码生成工具支持 COFF ABI 和 EABI ABI。默认情况下，使用 COFF 生成目标文件。选择 ABI 由 --abi 命令行选项控制，并在节 2.13 进行了讨论。如果为 EABI 编译此代码，则定义 __TI_EABI__ 预定义符号并将其设置为 1。

若要生成与 EABI 兼容的目标文件，必须使用 C28x 编译器版本 18.8.0.STS 或更高版本。COFF ABI 是旧版编译器支持的唯一 ABI。

EABI ABI 需要 ELF 目标文件格式。此格式支持现代语言功能，如早期模板实例化和导出内联函数。有关 C28x EABI 的低层面细节，请参阅《*C28x 嵌入式应用程序二进制接口*》应用报告 (SPRAC71)。

链接到创建应用程序的所有目标文件必须使用相同的 ABI。也就是说，他们必须全部使用 COFF ABI 或全部使用 EABI；这些 ABI 彼此不兼容。

6.12 目标文件符号命名规则（链接名）

每个外部可见的标识符都会分配一个用于目标文件的唯一符号名，即所谓的*链接名*。该名称由编译器根据一种算法分配，该算法取决于符号的名称、类型和源语言。该算法可能会向标识符添加前缀（通常是下划线），并且可能会*改编*名称。

对于 EABI，用户定义的符号（使用 C 代码和汇编代码）存储在同一个命名空间中，这意味着需要确保 C 标识符不与汇编代码标识符相冲突。标识符可能与汇编关键字（例如寄存器名称）相冲突；在这种情况下，编译器会自动使用转义序列来防止冲突。编译器使用双平行线对标识符进行转义，这指示汇编器不要将标识符视为关键字。需要确保 C 标识符不与用户定义的汇编代码标识符相冲突。

对于 COFF，编译器会在 C 标识符链接名的开头放置一个下划线，确保在汇编代码中放心使用不以下划线开头的标识符。

名称改编会对函数链接名中函数参数的类型进行编码，仅发生在未声明为 extern "C" 的 C++ 函数中。改编会实现函数重载、运算符重载和类型安全链接。请注意，函数的返回值未在改编后的名称中编码，因为无法根据返回值重载 C++ 函数。

所使用的 COFF 改编算法与“参考手册注解”(ARM) 中描述的算法非常相似。

例如，名为 func 的函数的 C++ 链接名的一般形式如下：

_func__F *parmcodes*

其中，parmcodes 是对 func 参数类型进行编码的字母序列。

对于这个简单的 C++ 源文件：

```
int foo(int i){ }   //global C++ function
```

生成的汇编代码如下：

```
_foo__Fi
```

foo 的链接名是 _foo__Fi，表示 foo 是一个仅接受整数类型参数的函数。为了帮助检查和调试，提供了一个名称还原实用程序，可将名称还原为 C++ 源代码中的名称。请参阅章节 9，了解详情。

6.13 在 COFF ABI 模式下初始化静态和全局变量

ANSI/ISO C 标准规定，在程序开始运行之前，没有进行显式初始化的全局 (extern) 和静态变量必须被初始化为 0。此任务通常在加载程序时完成。加载过程在很大程度上依赖于目标应用系统的特定环境，因此在 COFF ABI 模式下，编译器本身没有针对在运行时将未初始化静态存储类变量初始化为 0 的任何规定。您的应用应满足此要求。

备注

初始化全局对象

您应该显式初始化所有您希望编译器默认设置为零的全局对象。

在 EABI 模式下，未初始化的变量被自动初始化为零。

6.13.1 使用链接器初始化静态和全局变量

如果您的加载程序没有预初始化变量，您可以使用链接器将目标文件中的变量预初始化为 0。例如，在链接器命令文件中，在 .ebss 段中使用填充值 0：

```
SECTIONS
   {
      ...
      .ebss: {} = 0x00;
      ...
   }
```

链接器会将归零 .ebss 段的完整加载映像写入输出 COFF 文件，因此该方法带来的不利影响是，它可能会显著增大输出文件（而非程序）的大小。

如果您将应用刻录到 ROM 中，则应明确初始化需要初始化的变量。上述方法仅在加载时（而不是在系统复位或上电时）将 .ebss 初始化为 0。若要在运行时使这些变量为 0，请在代码中明确定义它们。

有关链接器命令文件和 SECTIONS 指令的更多信息，请参阅《*TMS320C28x 汇编语言工具用户指南*》中的链接器说明信息。

6.13.2 使用常量类型限定符初始化静态和全局变量

没有显式初始化的 *const* 类型的静态和全局变量与其他静态和全局变量类似，因为它们可能不会被预初始化为 0（原因在节 6.13 中讨论过）。例如：

```
const int zero;      /*  might not be initialized to 0  */
```

但是，const 全局变量的初始化是不同的，因为这些变量是在名为 .econst 或 .const（取决于 ABI）的段中进行声明和初始化的。例如：

```
const int zero = 0   /*  guaranteed to be 0  */
```

对于 COFF，这对应于 .econst 段中的一个条目：

```
   .sect   .econst
_zero
   .word   0
```

对于 EABI，这对应于 .const 段中的一个条目：

```
   .sect   .const
zero
   .word   0
```

这个特性对于声明一个大型常量表特别有用，因为在系统启动时既不会浪费时间也不会浪费空间来初始化表。另外，链接器可用于在 ROM 中放置 .econst 或 .const 段。

您可以使用 DATA_SECTION pragma 将变量放置在 .econst 或 .const 之外的段中。例如，以下 C 代码：

```
#pragma DATA_SECTION (var, ".mysect")
   const int zero=0;
```

is compiled into this assembly code (in COFF mode):

```
   .sect   .mysect
_zero
   .word   0
```

6.14 更改 ANSI/ISO C/C++ 语言模式

语言模式命令行选项决定了编译器如何解释源代码。您可以指定一个选项来标识代码遵循的语言标准。您还可以指定一个单独的选项，以指定编译器期望代码符合标准的严格程度。

指定以下语言选项之一，以控制编译器希望源代码遵循的语言标准。选项：

- ANSI/ISO C89（--c89，C 文件的默认值）
- ANSI/ISO C99（--c99，请参阅节 6.14.1。）
- ANSI/ISO C11（--c11，请参阅节 6.14.2）
- ISO C++03（--c++03 ，用于所有 C++ 文件 ，请参阅节 6.2。）

使用以下选项之一指定代码符合标准的严格程度：

- 宽松 ANSI/ISO（--relaxed_ansi 或 -pr）这是默认设置。
- 严格 ANSI/ISO（--strict_ansi 或 -ps）

默认为宽松 ANSI/ISO 模式。在宽松 ANSI/ISO 模式下，编译器接受可能与 ANSI/ISO C/C++ 相冲突的语言扩展。在严格 ANSI 模式下，这些语言扩展遭到抑制，因此编译器将接受所有严格遵循规范的程序。（请参阅节 6.14.3。）

6.14.1 C99 支持 (--c99)

编译器支持 ISO 标准化的 1999 年标准 C。但是，以下运行时函数和功能列表*未*实现或完全受支持：

- inttypes.h
  - wcstoimax() / wcstoumax()
- math.h: 编译器在 C99 模式下使用的数学库提供了完整的 C99 数学支持，包括 long double（64 位）和浮点（32 位）版本的浮点数学例程。请参阅标准 math.h C99 例程列表。如需了解由 --fp_mode=relaxed 命令行选项启用的浮点算术优化，请参阅节 2.3.3。
- stdio.h
  - 当标准预期为“0”时，%e 指定符可能产生“-0”
  - 在写入宽字符数组时，snprintf() 不能正确填充空格
- stdlib.h
  - 浮点匹配失败时 vfscanf()/vscanf()/vsscanf() 返回值不正确
- wchar.h
  - getws()/fputws()
  - mbrlen()
  - mbsrtowcs()
  - wcscat()
  - wcschr()
  - wcscmp()/wcsncmp()
  - wcscpy()/wcsncpy()
  - wcsftime()
  - wcsrtombs()
  - wcsstr()
  - wcstok()
  - wcsxfrm()
  - 宽字符打印/扫描函数
  - 宽字符转换函数

6.14.2 C11 支持 (--c11)

编译器支持 ISO 标准化的 2011 年标准 C。但是，除了节 6.14.1 中的列表外，以下运行时函数和功能在 C11 模式下*未*实现或完全受支持：

- threads.h
- 原子操作

6.14.3 严格 ANSI 模式和宽松 ANSI 模式 (--strict_ansi 和 --relaxed_ansi)

在宽松 ANSI/ISO 模式 (默认模式) 下，编译器接受可能与严格遵循 ANSI/ISO C/C++ 的程序相冲突的语言扩展。在严格 ANSI 模式下，这些语言扩展遭到抑制，因此编译器将接受所有严格遵循规范的程序。

当您知道您的程序是一个遵循规范的程序，并且不会在宽松模式下编译时，请使用 --strict_ansi 选项。在此模式下，与 ANSI/ISO C/C++ 相冲突的语言扩展将被禁用，编译器将在标准要求时发出错误消息。本标准视为酌情处理的违规行为可作为警告发出。

示例：

以下是严格遵循规范的 C 代码，但在默认宽松模式下将不被编译器接受。若要使编译器接受这种代码，请使用严格 ANSI 模式。编译器将抑制 interrupt 关键字语言异常，然后，interrupt 可用作代码中的标识符。

```
int main()
{
    int interrupt = 0;
    return 0;
}
```

以下是未严格遵循规范的代码。编译器将不接受这种严格 ANSI 模式下的代码。若要使编译器接受这种代码，请使用宽松 ANSI 模式。 编译器将提供 interrupt 关键字扩展并接受此代码。

```
interrupt void isr(void);
int main()
{
    return 0;
}
```

以下代码在所有模式下均被接受。__interrupt 关键字与 ANSI/ISO C 标准不冲突，因此始终可以作为一种语言扩展。

```
__interrupt void isr(void);
int main()
{
    return 0;
}
```

默认模式为宽松 ANSI。 可以通过 --relaxed_ansi (或 -pr) 选项来选择此模式。宽松 ANSI 模式接受种类最多的程序，以及所有 TI 语言扩展，即使是那些与 ANSI/ISO 相冲突的扩展，也会忽略一些编译器能够合理处理的 ANSI/ISO 冲突。节 6.15 中描述的一些 GCC 语言扩展可能与严格 ANSI/ISO 标准相冲突，但许多 GCC 语言扩展可能不与这些标准相冲突。

6.15 GNU 和 Clang 语言扩展

GNU 编译器集合 (GCC) 定义了许多在 ANSI/ISO C 和 C++ 标准中没有的语言特性。这些扩展的定义和示例（针对 GCC 4.7 版）可以在以下 GNU 网站上找到：http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/C-Extensions.html#C-Extensions。其中大多数扩展也可用于 C++ 源代码。

编译器还支持以下 Clang 宏命令扩展，这些扩展在 Clang 6 文档中进行了描述：

- __has_feature（直到为 Clang 3.5 描述的测试）
- __has_extension（直到为 Clang 3.5 描述的测试）
- __has_include
- __has_include_next
- __has_builtin（请参阅节 6.15.6）
- __has_attribute

6.15.1 扩展

在宽松 ANSI 模式 (--relaxed_ansi) 下进行编译时，大多数 GCC 语言扩展都可在 TI 编译器中使用。

表 6-3 中列出了 TI 编译器支持的扩展，其基于 GNU 网站上的扩展列表。阴影行描述了不受支持的扩展。

表 6-3. GCC 语言扩展

扩展	说明
语句表达式	将语句和声明放在表达式中（用于创建智能的“安全”宏命令）
局部标签	语句表达式的局部标签
标签作为值	指向标签和计算得到的 goto 的指针
嵌套函数	就像在 Algol 和 Pascal 中一样，函数的词法范围
构造调用	分派对另一个函数的调用
命名类型(1)	为表达式类型指定名称
typeof 运算符	typeof 指的是表达式类型
广义左值	在左值中使用问号 (?)、逗号 (,) 和 cast
条件语句	省略 ?: 表达式的中间操作数
long long	Double long 字整数和 long long int 类型
十六进制浮点值	十六进制浮点常量
复数	复数的数据类型
零长度	零长度数组
可变参数宏命令	具有可变数量参数的宏命令
可变长度	在运行时计算长度的数组
空结构	无成员的结构
加下标	任何数组都可以加下标，即使它不是左值。
转义换行符	转义换行符的规则稍微宽松一些
多行字符串(1)	带有嵌入换行符的字符串文字
指针算术	空指针和函数指针的算术
初始化程序	非常量初始化程序
复合字面量	复合字面量将结构体、联合体或数组作为值
指定的初始化程序	初始化程序的标签元素
强制转换为 union	从 union 的任何成员强制转换为 union 类型
Case（强制转换）范围	“Case 1 ...9”等
混合声明	混合声明和代码

表 6-3. GCC 语言扩展（续）

扩展	说明
函数属性	声明函数没有任何副作用，或者其永远不会返回
属性语法	属性的正式语法
函数原型	原型声明和旧式定义
C++ 注释	系统会识别 C++ 注释。
美元符号	标识符中允许使用美元符号。
字符转义	字符 ESC 表示为 \e
变量属性	指定变量的属性
类型属性	指定类型的属性
对齐	查询类型或变量的对齐情况
内联	定义内联函数（和宏命令一样快）
汇编标签	指定要用于 C 符号的汇编器名称
扩展的 asm	带有 C 操作数的汇编器指令
约束条件	asm 操作数的约束条件
包装器头文件	包装器头文件可以使用 #include_next 包含另一个版本的头文件
替代关键字	头文件可以使用 __const__、__asm__ 等
显式寄存器变量	定义驻留在指定寄存器中的变量
不完整的枚举类型	定义枚举标签而不指定其可能的值
函数名称	作为当前函数名称的可打印字符串
返回地址	获取函数的返回地址或帧地址（有限支持）
其他内置	其他内置函数（请参见节 6.15.6）
矢量扩展	通过内置函数使用矢量指令
目标内置	专用于特定目标的内置函数
Pragma	GCC 接受的 pragma
未命名字段	结构体/联合体中的未命名结构体/联合体字段
线程本地	每线程变量
二进制常量	使用“0b”前缀的二进制常量。

(1) 为 GCC 3.0 定义的功能；请访问 http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/C-Extensions.html#C-Extensions 查看定义和示例

6.15.2 函数属性

支持以下 GCC 函数属性：

- alias
- aligned
- always_inline
- calls
- const
- constructor
- deprecated
- format
- format_arg
- interrupt
- malloc
- naked
- noinline
- noreturn
- pure

- section
- unused
- used
- warn_unused_result
- weak (仅限 EABI)

支持以下其他 TI 特定函数属性：

- retain
- ramfunc

例如，此函数声明使用 **alias** 属性使“my_alias”成为“myFunc”函数的别名：

```
void my_alias() __attribute__((alias("myFunc")));
```

aligned 函数属性会使用指定的对齐方式来对齐函数。该对齐必须为 2 的幂。此属性与 CODE_ALIGN pragma 具有相同的效果；请参阅节 6.9.3。

always_inline 函数属性与 FUNC_ALWAYS_INLINE pragma 具有相同的效果。请参阅 节 6.9.11

calls 属性与 CALLS pragma 具有相同的效果，相关描述请参阅 节 6.9.1。

format 属性应用于 stdio.h 中 printf、fprintf、sprintf、snprintf、vprintf、vfprintf、vsprintf、vsnprintf、scanf、fscanf、vfscanf、vscanf、vsscanf 和 sscanf 的声明。因此，当启用 GCC 扩展时，系统会根据格式字符串参数中的格式说明符对这些函数的数据参数进行类型检查，并在不匹配时发出警告。如果不需要这些警告，可以通过常见方式抑制这些警告。

有关如何使用 **interrupt** 函数属性的更多信息，请参阅节 6.9.15。

malloc 属性应用于 stdlib.h 中 malloc、calloc、realloc 和 memalign 的声明。

naked 属性标识了使用 `__asm` 语句编写为嵌入式汇编函数的函数。编译器不会为此类函数生成序言和结语序列。请参阅节 6.8。

noinline 函数属性与 FUNC_CANNOT_INLINE pragma 具有相同的效果。请参阅 节 6.9.12

ramfunc 属性指定一个函数将被放置在 RAM 中并从中执行。ramfunc 属性允许编译器优化 RAM 执行的函数，以及自动将函数复制到基于闪存的器件上的 RAM 上。例如：

```
__attribute__((ramfunc))
void f(void) {
   ...
}
```

--ramfunc=on 选项指定使用此选项编译的所有函数都放置在 RAM 中并从中执行，即使未使用此函数属性也是如此。

较新的 TI 链接器命令文件通过将具有此属性的函数放置在 .TI.ramfunc 段中来自动支持 ramfunc 属性。如果您的链接器命令文件不包含 .TI.ramfunc 段的段规格，您可以修改链接器命令文件以将此段放在 RAM 中。有关段放置的详细信息，请参阅 *TMS320C28x 汇编语言工具用户指南*。

为 ramfunc 函数生成快速分支指令。为所有其他函数生成常规分支指令。

CLA 编译器会忽略 ramfunc 属性。

retain 属性与 RETAIN pragma (节 6.9.22) 具有相同的效果。也就是说，即使在应用的其他地方没有引用包含该函数的段，也不会从条件链接输出中省略该段。

当 **section** 属性在函数上使用时，具有与 CODE_SECTION pragma 相同的效果。请参阅 节 6.9.4

weak 属性与 WEAK pragma (节 6.9.25) 具有相同的效果。

6.15.3 For 循环属性

如果您使用的是 C++，则有几个特定于 TI 的属性可应用于循环。C 中没有相应的语法。以下 TI 特定属性与其相应程序具有相同的功能：

- TI::must_iterate
- TI::unroll

有关使用 for 循环属性的示例，请参阅节 6.9.17.1。

6.15.4 变量属性

支持下述变量属性：

- aligned
- blocked 和 noblocked
- deprecated
- location
- mode
- noinit (仅限 EABI)
- persistent (仅限 EABI)
- preserve
- retain
- section
- transparent_union
- unused
- update
- used
- weak (仅限 EABI)

在变量上使用的 **aligned** 属性与 DATA_ALIGN pragma 的效果相同。请参阅 节 6.9.5

blocked 和 **noblocked** 属性可用于控制阻断特定变量 (包括数组和结构体)。有关阻断变量或解除阻断变量的原因，请参阅 节 3.11。这些属性必须在变量的定义和声明中都能使用。建议在声明变量的头文件中使用这些属性。例如：

```
__attribute__((blocked))
extern int my_array[];
__attribute__((noblocked))
extern struct_type my_struct;
```

noblocked 属性还可用于引用在汇编或未阻断的 CLA 转换单元中定义的数据。

如果在未阻断的数据上尝试将变量视为阻断的数据访问，链接器则会提供诊断消息。

location 属性与 LOCATION pragma 的效果相同。请参阅节 6.9.16。例如：

```
__attribute__((location(0x100))) extern struct PERIPH peripheral;
```

如果利用 location 属性指定的地址位于存储器的低 16 位，将使用具有较少 DP 负载的更高效指令。

或者，在取消引用文本时使用指定地址的语法使编译器选择更有效的指令以处理内存的低 16 位上的地址。例如，可以使用下述任一方式：

```
#define peripheral (*((struct PERIPH)(0x100)))

struct EPWM_REGS volatile* const epwm1 = (struct EPWM_REGS *)(0x4000);
```

noinit 和 **persistent** 属性（仅限 EABI）适用于 ROM 初始化模式，并允许应用程序在重置期间避免初始化特定全局变量。备选的 RAM 初始化模式只在加载映像时初始化变量；重置时不会初始化变量。请参阅《*TMS320C28x 汇编语言工具用户指南*》中的“RAM 模型与 ROM 模型”章节及其小节。

noinit 属性可用在未初始化的变量上；可防止这些变量在重置期间被设置为 0。**persistent** 属性可用在初始化的变量上；可防止这些变量在重置期间被初始化。默认情况下，标记为 noinit 或 persistent 的变量将分别置于 `.TI.noinit` 和 `.TI.persistent` 段。这些段的位置由链接器命令文件控制。通常对于支持 FRAM 的器件，.TI.persistent 段置于 FRAM 中，.TI.noinit 段置于 RAM 中。也请参见 节 6.9.18。

preserve 和 **update** 属性适用于实时固件更新 (LFU) 功能，该功能支持执行“热启动”来升级固件，而无需使系统离线。这两个属性使用从参考 ELF 映像中获得的符号地址列表，该映像是在编译和链接可执行文件时指定的。有关仅支持 EABI 的 LFU 的更多信息，请参阅节 2.15。

- **preserve** 属性使全局或静态变量保留其在热启动之前的地址和值。此符号的地址与参考 ELF 映像中的地址保持相同。每个这样的符号都有一个 `.TI.bound` 段。如果 .TI.bound 段在存储器中是连续的，链接器可以将这些段合并到单个输出段中，从而减少初始化这些段所需的 CINIT 记录数量。 下述示例使用 preserve 属性：

```
int __attribute__((preserve))        gvar_bss_preserve;
int __attribute__((preserve))        gvar_0 = 0x30;
static int __attribute__((preserve)) svar_0 = 0x50;
```

- **update** 属性使全局或静态变量可以在热启动期间被重新初始化。与参考 ELF 映像中的地址相比，这些变量的地址可能会改变。此类符号将由链接器收集到单个 `.TI.update` 输出段中。此段默认为复制压缩（即，在热启动期间无需解压缩），因此可以减少 LFU 映像切换时间。下述示例使用 update 属性：

```
int __attribute__((update))        gvar_bss_update;
int __attribute__((update))        gvar_1 = 0x50;
static int __attribute__((update)) svar_1 = 0x70;
```

retain 属性与 RETAIN pragma (节 6.9.22) 的效果相同。也就是说，即使在应用程序的其他地方没有引用该变量，包含该变量的段也不会从条件链接的输出中省略。

变量上使用的 **section** 属性与 DATA_SECTION pragma 的效果相同。请参阅 节 6.9.6

used 属性在 GCC 4.2 中定义（请参阅 http://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/Variable-Attributes.html#Variable-Attributes）。

weak 属性与 WEAK pragma (节 6.9.25) 的效果相同。

6.15.5 类型属性

编译器支持以下类型属性：

- aligned
- byte_peripheral
- deprecated
- transparent_union
- unused

您无法将 **aligned** 类型属性与大于此类型大小的对齐一同使用来创建数组。这是因为 C 语言可以保证数组的大小等于每个元素的大小乘以元素数量。如果数组使用了 aligned 类型属性，则可能不会出现这种情况，因为可以向数组添加填充字节。

有关 **byte_peripheral** 类型属性的详细信息，请参阅节 6.15.7。

6.15.6 内置函数

支持以下内置函数：

- __builtin_abs()
- __builtin_constant_p()
- __builtin_expect()
- __builtin_fabs()
- __builtin_fabsf()
- __builtin_frame_address()
- __builtin_labs()
- __builtin_sqrt()
- __builtin_sqrtf()
- __builtin_memcpy()
- __builtin_return_address()

__builtin_frame_address() 函数始终返回 0，。

仅当启用了硬件浮点支持时，才支持 __builtin_sqrt() 和 __builtin_sqrtf() 函数。也就是说，只有在启用三角函数加速器 (TMU) 的情况下，才支持 __builtin_sqrt()。而且，只有在启用 TMU 或使用 CLA 编译器的情况下，才支持 __builtin_sqrtf()。

调用运行时可能不可用的内置函数时，请使用以下示例中所示的 Clang __has_builtin 宏命令，以确保该函数受支持：

```
#if __has_builtin(__builtin_sqrt)
double estimate = __builtin_sqrt(x);
#else
double estimate = fast_approximate_sqrt(x);
#endif
```

如果支持内置函数，且设备具有适当的硬件支持，则内置函数将调用硬件支持。

如果支持内置函数，但设备未启用相应的硬件，则内置函数通常会成为对 RTS 库函数的调用。例如，__builtin_sqrt() 将成为对库函数 sqrt() 的调用。

__builtin_return_address() 函数始终返回零。

6.15.7 使用字节外设类型属性

C2000 架构具有 16 位字。但一些外设为 8 位字节可寻址。字节外设桥可将地址视为字节地址，从而在 CPU 和字节外设之间转换地址。因此，只有一些 C2000 地址可以正确映射到字节外设。16 位数据的偶数和奇数地址均映射到字节外设的同一数据元件。

32 位数据的地址也是一样。16 位访问的地址必须与 32 位对齐，32 位访问的地址必须与 64 位对齐。

提供 C2000 驱动程序库和位域头文件，以访问外设。为了支持对字节外设数据的正常访问，编译器提供 __byte_peripheral_32 内在函数和 byte_peripheral 类型属性。

C2000 驱动程序库可在正确的起始地址访问字节外设数据。但在 C2000 上，32 位数据的运算经常会拆分为两个 16 位数据的运算，因为在此架构上这种方式更加高效。对 32 位字节外设数据的访问无法有规律地拆分为两个 16 位访问，因为第二个 16 位的起始偏移量将不正确。__byte_peripheral_32 内在函数可用于访问一个 32 位字节外设数据地址，因而无需拆分访问。该内在函数会返回对无符号长整型值的引用，并可用于读取和写入数据。有关此内在函数的信息，请参阅 节 7.6。

byte_peripheral 类型属性可按如下方式应用于无符号整型和无符号长整型的 typedefs（类型定义），以支持位域。

```
typedef unsigned int bp_16 __attribute__((byte_peripheral)) ;
typedef unsigned long bp_32 __attribute__((byte_peripheral));
```

typedef 名称并不重要。属性会自动应用可变关键字并处理对齐。字节外设结构中的所有结构成员（无论是否为位域）必须通过 typedef 应用这些属性，以确保结构访问的正确对齐。请注意，对齐方式不同将导致结构布局不同，因此位域无法始终通过常规结构中的容器类型进行访问。

例如，字节外设位域类型的位位置在以下示例中与常规位域的位置进行比较。16 位访问必须与 32 位对齐，因此不可能访问 16 位容器偏移 16-31 的位。若要访问这些位，必须使用 32 位容器。在示例 1 中，您可以将 field4 到 field6 的域类型更改为 bp_32，创建与常规情况相同的布局。在示例 2 中，您要将 field2 到 field4 的域类型更改为 bp_32。

```
struct example1 {       // regular bits position   //byte peripherals
    bp_16 field1:9;     // 0-8                     // 0-8
    bp_16 field2:6;     // 9-14                    // 9-14
    bp_32 field3:4;     // 15-18                   // 15-18
    bp_16 field4:1;     // 19                      // 32
    bp_16 field5:5;     // 20-24                   // 33-37
    bp_16 field6:7;     // 25-31                   // 38-44
};
struct example2{        // regular bits position   //byte peripherals
    bp_32 field1:29;    // 0-28                    // 0-28
    bp_16 field2:1;     // 29                      // 32
    bp_16 field3:1;     // 30                      // 33
    bp_16 field4:1;     // 31                      // 34
};
```

由于对齐会在声明的任何对象中生成边界填充，建议您将字节外设地址转换为字节外设结构类型，而不是声明这些结构类型的对象。

您无法使用字节外设类型属性创建数组。这是因为 C 语言可以保证数组的大小等于每个元素的大小乘以元素数量。但在字节外设数组中不会出现这种情况，因为数组中需要有边界填充。

6.16 编译器限制

由于 C/C++ 编译器支持的主机系统的多样性，以及其中一些系统的局限性，编译器可能无法成功编译过大或过于复杂的源文件。通常，超过此系统限制会阻止继续编译，因此编译器会在打印错误消息后立即中止。简化程序以避免超过系统限制。

某些系统不允许文件名长度超过 500 个字符。确保您的文件名短于 500 个字符。

编译器并非任意限制，但受主机系统上可用内存量的限制。在较小的主机系统（如 PC）上，优化器可能会耗尽内存。如果出现这种情况，优化器将终止，shell 将继续使用代码生成器编译文件。这会导致编译文件时没有进行优化。优化器一次编译一个函数，因此更可能的原因是源模块中的函数太大或非常复杂。若要更正此问题，您可以进行如下选择：

- 不要优化有问题的模块。
- 确定导致问题的函数，并将其分解为较小的函数。
- 从模块中提取函数，并将其放在一个单独的模块中，该模块可在不进行优化的情况下进行编译，以便对其余函数进行优化。

章节 7

运行时环境

本章介绍 TMS320C28x C/C++ 运行时环境。为确保 C/C++ 程序的成功执行，所有运行时代码维护这一环境是至关重要的。如果要编写与 C/C++ 代码交互的汇编语言函数，那么遵循本章中的指导原则也是很重要的。

7.1 存储器模型

C28x 编译器将内存视为程序内存和数据内存这两个线性块：

- 程序内存包含可执行代码、初始化记录和切换表。
- 数据内存包含外部变量、静态变量和系统堆栈。

由 C/C++ 程序生成的代码块或数据块放置在合适的存储器空间的连续块中。

备注

链接器定义内存映射：由链接器而不是编译器定义内存映射并将代码和数据分配到目标内存中。编译器不考虑可用内存的类型、不考虑代码或数据（漏洞）的任何不可用的位置，也不考虑为 I/O 或控制目的保留的任何位置。编译器生成可重定位代码，允许链接器将代码和数据分配到合适的内存空间中。例如，可以使用链接器将全局变量分配到片上 RAM 中或将可执行代码分配到外部 ROM 中。可以将每个代码块或数据块单独分配到内存中，但这不是通用做法（一个例外是内存映射 I/O ，尽管可以使用 C/C++ 指针类型访问物理存储器位置）。

7.1.1 段

编译器生成称为*段*的可重定位代码块和数据块，这些代码块以多种方式分配到内存中，以符合各种系统配置。有关各段及其分配的更多信息，请参阅 *TMS320C28x 汇编语言工具用户指南*中介绍的目标文件信息。。

段有两种基本的类型：

- **未初始化的段**会在存储器中保留空间（通常为 RAM）。程序可以在运行时使用此空间来创建和存储变量。编译器会创建以下未初始化的段：
  - **.bss 段**为未初始化的全局变量和静态变量保留空间。未初始化且也未使用的变量通常创建为通用符号，而不是放置在 .bss 中，以便将该变量从生成的应用中排除。（EABI 仅用于编译器；EABI 和 COFF 用于汇编器）
  - **.ebss 段**为所有静态变量（已初始化或未初始化；全局或局部）保留空间。在程序启动时，C/C++ 引导例程从 .cinit 段（可能在 ROM 中）复制数据，并将其用于初始化 .ebss 段中的变量。（仅限 COFF）
  - **.stack 段**为保留空间。C/C++ 软件栈。此存储器用于将参数传递给函数，并为局部变量分配空间。
  - **.esysmem 段**为动态存储器分配保留空间。此空间由动态存储器分配例程使用，如 malloc、calloc、realloc 或 new。如果 C/C++ 程序不使用这些函数，编译器则不会创建 .esysmem 段。（仅限 COFF）
  - **.sysmem 段**为动态存储器分配保留空间。保留的空间由动态存储器分配例程使用，如 malloc()、calloc()、realloc() 或 new()。如果 C/C++ 程序不使用这些函数，编译器则不会创建 .sysmem 段。（仅限 EABI）
- **初始化段**包含数据或可执行代码。初始化段通常是只读的；例外情况如下所示。C/C++ 编译器会创建以下初始化段：
  - **.args 段**包含用于命令行参数的空间。请参阅 --arg_size 选项。
  - **.binit 段**包含引导时复制表。有关 BINIT 的详细信息，请参阅 *TMS320C28x 汇编语言工具用户指南*。
  - **.cinit 段**包含用于初始化变量和常量的表。C28x .cinit 记录限制为 16 位。这将初始化对象限制为 64K。（对于 EABI，链接器会创建 .cinit 段。对于 COFF，编译器会创建 .cinit 段。）
  - **.ovly 段**包含联合的复制表，其中的不同段具有相同的运行地址。
  - **.init_array 段**包含全局构造函数表。（仅限 EABI）
  - **.pinit 段**包含全局构造函数表。（仅限 COFF）
  - **.c28xabi.exidx 段**包含用于异常处理的索引表。**.c28xabi.extab 段**包含用于异常处理的堆栈展开指令。有关详细信息，请参阅 --exceptions 选项。（仅限 EABI）
  - **.ppdata 段**包含用于基于编译器的分析的数据表。有关详细信息，请参阅 --gen_profile_info 选项。此段是可写的。
  - **.ppinfo 段**包含用于基于编译器的分析的相关性表。有关详细信息，请参阅 --gen_profile_info 选项。
  - **.const 段**包含字符串文字和使用 C/C++ 限定符 *const* 定义的变量（即使该常量未定义为 *volatile* 也是如此，但如果该常量为节 6.5.1 中描述的异常之一，则不是）。字符串文字放置在 .const:.string 子段中，以更大力度地控制链接时放置位置。（仅限 EABI）

- **.econst 段**包含字符串文字和使用 C/C++ 限定符 *const* 定义的全局变量（即使该常量未定义为 *volatile* 也是如此，但如果该常量为节 6.5.1 中描述的异常之一，则不是）。字符串文字放置在 .econst:.string 子段中，以更大力度地控制链接时放置位置。（仅限 COFF）
- **.data 段**为初始化的非常量静态变量保留空间，无论它们是全局变量还是局部变量。（对于 EABI，编译器会生成此段，用于初始化全局变量和静态变量。对于 COFF，此段可由汇编代码使用，但不能在其他情况下使用。）此段是可写的。
- **.switch** 段包含用于 switch 语句的表。默认情况下，此段放置在数据存储器中。如果使用 --unified_memory 选项，此段将放置在程序存储器中。
- **.text 段**包含所有可执行代码和由编译器生成的常量。此段通常是只读的。
- **.TI.crctab 段**包含 CRC 检查表。
- **.TI.bound 段**用于将符号与特定存储器地址关联。当使用 LOCATION pragma（请参阅节 6.9.16）或“location”或“preserve”变量属性（请参阅节 6.15.4）将符号与特定存储器地址关联时，将为符号创建一个 .TI.bound 段。如果是“preserve”符号，当 .TI.bound 段在存储器中是连续的之时，链接器可以将其合并成一个输出段，从而减少初始化所需的 CINIT 记录数。此段可以是可写的，也可以是只读的。
- **.TI.update 段**包含在热启动时需要重新初始化的符号。重新初始化由 __TI_auto_init_warm() RTS 函数执行。建议您在链接器命令文件中添加一个条目，将 .TI.update 段置于适当的存储器区域。此段是可写的。

下表展示了各种类型的初始化变量的放置：

	EABI		COFF	
	全局	**局部**	**全局**	**局部**
const	.const	*	.econst	*
静态	.data	.data	.ebss	.ebss
字符串常量、文字	.const:.string	.const:.string	.econst:.string	.econst:.string

* 初始化的 const（非静态）局部变量不放置在输出段中；它们根据需要在运行时在存储器（通常是 RAM）中进行初始化。

汇编器会创建默认段 .text、.ebss 或 .bss（取决于 ABI）和 .data。您可以指示编译器使用 CODE_SECTION 和 DATA_SECTION pragma 创建其他段（请参阅节 6.9.4 和节 6.9.6）。

链接器从不同的目标文件中获取各个段，并合并具有相同名称的段。表 7-1 中列出了生成的输出段，以及每个段在存储器中的适当位置。您可以根据需要将这些输出段放置在地址空间中的任何位置，以满足系统要求。

链接器还会创建一些编译器未引用的其他段。例如，**.common** 段包含链接器分配的通用块符号。

对于 EABI，建议您使用统一的存储器方案，将所有段放置在第 0 页上。但是，默认链接器命令文件为段指定了第 0 页或第 1 页，如下表所示。通常，未初始化的常量值段由链接器命令文件放置在第 1 页上；所有其他段通常放置在第 0 页上。

表 7-1. 段和存储器位置摘要

段	存储器类型	默认页面
.binit	预计在闪存/ROM 中	1
.bss（仅 EABI）	必须在 RAM 中	1
.ebss（仅 COFF）	必须在 RAM 中	1
.c28xabi.exidx（仅 EABI）	预计在闪存/ROM 中	1
.c28xabi.extab（仅 EABI）	预计在闪存/ROM 中	1
.cinit (1)	预计在闪存/ROM 中	0
.const（仅 EABI）	预计在闪存/ROM 中	1
.econst（仅 COFF）	预计在闪存/ROM 中	1
.data（主要由 EABI 使用）	必须在 RAM 中	0
.init_array（仅 EABI）	预计在闪存/ROM 中	0
.pinit（仅 COFF）	预计在闪存/ROM 中	0

表 7-1. 段和存储器位置摘要（续）

段	存储器类型	默认页面
.ppdata	必须在 RAM 中	1
.stack	必须在 RAM 中	1
.switch	取决于 --unified_memory 选项设置	0，1
.sysmem（仅 EABI）	必须在 RAM 中	1
.esysmem（仅 COFF）	必须在 RAM 中	1
.text	预计在闪存/ROM 中	0

(1) .cinit 段由 COFF 的编译器和 EABI 的链接器创建。

可以使用链接器命令文件中的 SECTIONS 指令来自定义段分配过程。有关将段分配到存储器中的更多信息，请参阅 *TMS320C28x 汇编语言工具用户指南* 中的链接器说明一章。

7.1.2 C/C++ 系统堆栈

C/C++ 编译器使用堆栈来：

- 分配局部变量
- 传递参数给函数
- 保存处理器状态
- 保存函数返回地址
- 保存临时结果

运行时堆栈从低位地址向上增长到高位地址。默认情况下，堆栈分配在 .stack 段中。（请参阅 run-time-support boot.asm 文件。）编译器使用硬件堆栈指针 (SP) 来管理此栈。

备注

链接 .stack 段：必须将 .stack 段链接到低 64K 的数据内存中。SP 是一个 16 位寄存器，不能访问超过 64K 的地址。

对于大小超过 63 个字（SP 偏移寻址模式的最大范围）的帧，编译器使用 XAR2 作为栈帧指针 (FP)。每次函数调用都会在堆栈顶部创建一个新的帧，从中分配局部和临时变量。FP 指向此帧的开头，以访问不能使用 SP 直接引用的内存位置。

堆栈大小由链接器设置。链接器创建全局符号 __STACK_SIZE（对于 COFF）或 __TI_STACK_SIZE（对于 EABI），并为其分配一个等于堆栈大小的值（以字节为单位）。默认的堆栈大小为 1K 字。可以在链接时使用 --stack_size 链接器选项来更改堆栈大小。

备注

堆栈溢出：编译器不提供编译期间或运行时检查栈溢出的方法。堆栈溢出会破坏运行时环境，导致程序失败。确保留出足够的空间让堆栈增长。可以使用 --entry_hook 选项在每个函数的开头添加代码以检查是否发生堆栈溢出；请参阅节 2.14。

7.1.3 将 .econst 分配给程序内存

备注

本节适用于使用 COFF ABI 的应用程序。

如果您的系统配置不支持将已初始化的段（如 .econst）分配给数据内存，则必须分配 .econst 段才能加载到程序内存中，并在数据内存中运行。在启动时，将 .econst 段从程序复制到数据内存。以下序列说明了如何执行此任务。

1. 从源库中提取 boot.asm：

```
ar2000 -x rts.src boot.asm
```

2. 编辑 boot.asm 并将 CONST_COPY 标志更改为 1：

```
CONST_COPY  .set 1
```

3. Assemble boot.asm:

```
cl2000 boot.asm
```

4. 将引导例程归档到对象库中：

```
ar2000 -r rts2800_ml.lib boot.c.obj
```

对于 .const 段，使用包含以下条目的链接器命令文件进行链接：

```
SECTIONS
{  ...
   .econst : load = PROG PAGE 1, run = DATA PAGE 1
             {
                /* GET RUN ADDRESS */
                __econst_run = .;
                /* MARK LOAD ADDRESS */
                *(.ec_mark)
                /* ALLOCATE .econst */
                *(.econst)
                /* COMPUTE LENGTH */
                __econst_length = - .__econst_run;
             }
}
```

在链接器命令文件中，您可以在第 0 页使用内存区域的名称替换名称 PROG，在第 1 页使用内存区域的名称替换 DATA。命令文件的其余部分必须使用上述名称。将 CONST_COPY 更改为 1 时启用的 boot.asm 中的代码取决于以这种方式使用这些名称的链接器命令文件。若要更改任何名称，必须编辑 boot.asm，并以相同的方式更改名称。

7.1.4 动态存储器分配

C28x 编译器随附的运行时支持库包含几个函数（例如 malloc、calloc 和 realloc），这些函数允许您在运行时为变量动态地分配存储器。

内存是从 .esysmem 或 .sysmem 段中定义的全局池（或堆）分配的。可以在链接器命令中使用 heap_size=*size* 选项来更改 .esysmem 或 .sysmem 段的大小。链接器还会创建一个全局符号 __SYSMEM_SIZE（对于 COFF）或 __TI_SYSMEM_SIZE（对于 EABI），并为其分配一个等于堆大小的值（以字为单位）。默认大小为 1K 字。有关 --heap_size 选项的更多信息，请参阅 *TMS320C28x 汇编语言工具用户指南* 中的链接器说明一章。

如果您使用任何 C I/O 函数，RTS 库会为您访问的每个文件分配一个 I/O 缓冲区。这个缓冲区将比 BUFSIZ 大一点，BUFSIZ 在 stdio.h 中定义，默认为 256）。确保为这些缓冲区分配了足够大的堆或使用 setvbuf 将缓冲区更改为静态分配的缓冲区。

动态分配的对象并非采用直接寻址方式（始终使用指针访问），并且存储器池位于单独的段（.esysmem 或 .sysmem）中。因此，动态存储器池的大小仅受系统中可用存储器大小的限制。为了节省 .ebss 或 .bss 段的空间，可以从堆中分配大型数组，而不是将它们定义为全局或静态数组。例如，不是定义如下：

```
struct big table[100];
```

而是改用指针并调用 malloc 函数：

```
struct big *table
table = (struct big *)malloc(100*sizeof(struct big));
```

当从堆进行分配时，请确保堆的大小足够满足分配要求。在分配可变长度数组时，这一点尤为重要。例如，分配一个可变长度数组至少需要 1500 字的堆存储器。

7.1.5 变量的初始化

C/C++ 编译器生成的代码适合在基于 ROM 的系统中作为固件使用。在这样的系统中，.cinit 段中的初始化表存储在 ROM 中。在系统初始化时，C/C++ 引导程序将数据从这些表（位于 ROM 中）复制到 .ebss 或 .bss (RAM) 中初始化的变量中。

当程序直接从目标文件加载到内存中并运行时，可以避免 .cinit 段占用内存空间。加载器可以直接从目标文件（而不是从 ROM）中读取初始化表，并在加载时直接执行初始化而不是在运行时执行初始化。可以使用 --ram_model 链接选项将其指定给链接器。有关更多信息，请参阅 节 7.10。

7.1.6 为静态变量和全局变量分配内存

为 C/C++ 程序中声明的所有静态变量分配一个唯一的空间。链接器确定空间的地址。编译器确保这些变量的空间以多个字的形式分配，以便每个变量在字边界上对齐。

编译器期望将全局变量分配到数据内存中。（对于 COFF，编译器会在 .ebss 中为此类变量保留空间。对于 EABI，则会在 .data 或以 DATA_SECTION pragma 命名的段中保留空间，相关信息请参阅节 6.9.6）。在同一模块中声明的变量会被分配到单个内存块中。该块可能是连续的，但不能保证这一点。

7.1.7 字段/结构对齐

当编译器确定结构的布局时，它会根据需要分配尽可能多的字来容纳所有成员，同时遵守每个成员的对齐约束。这意味着填充字节可以放置在成员之间和结构的末尾处。每个成员都按其类型要求对齐。

大小为 16 位的类型在 16 位边界上对齐。大小为 32 位或更大的类型在 32 位（2 个字）边界上对齐。有关 EABI 字段对齐的详细信息，请参阅《*C28x 嵌入式应用程序二进制接口 (EABI) 参考指南*》(SPRAC71)。

然而，如果位字段以前不是位字段，那么位字段可能不会像位字段的声明类型那样严格对齐。位字段按照源代码中显示的顺序打包。首先填充结构字的最低有效位。位字段仅按请求的位数分配。位字段会打包成一个字的相邻位。

对于 COFF，位字段不与以下重叠：

- 16 位类型的 16 位边界
- 32 位或更大类型的 32 位边界

如果某个位字段与下一个边界重叠，则会将整个位字段放入下一个边界中。下表提供了一些示例：

说明	示例	结构体的内容（以小端字节序顺序）
本示例中的位字段 A 和 B 适合 32 位边界。	`struct bf_tag {` `  uint32_t:8  A;` `  uint32_t:24 B;` `} bf;`	`bf  AAAA AAAA` `bf + 1  BBBB BBBB` `bf + 2  BBBB BBBB` `bf + 3  BBBB BBBB`
如果 A 增加到 9 位，则 B 的最高有效位将跨越 32 位边界。因此，B 会放置在下一个 32 位边界。	`struct bf_tag {` `  uint32_t:9  A;` `  uint32_t:24 B;` `} bf;`	`bf  AAAA AAAA` `bf + 1  .... ...A` `bf + 2  .... ....` `bf + 3  .... ....` `bf + 4  BBBB BBBB` `bf + 5  BBBB BBBB` `bf + 6  BBBB BBBB` `bf + 7  .... ....`
如果 B 增加到 25 位，则当直接放置在 8 位 A 之后时，它将跨越 32 位边界。因此，B 将放置在下一个 32 位边界。	`struct bf_tag {` `  uint32_t:8  A;` `  uint32_t:25 B;` `} bf;`	`bf  AAAA AAAA` `bf + 1  .... ....` `bf + 2  .... ....` `bf + 3  .... ....` `bf + 4  BBBB BBBB` `bf + 5  BBBB BBBB` `bf + 6  BBBB BBBB` `bf + 7  .... ...B`

7.1.8 字符串常量

在 C 语言中，字符串常量用于下述方式之一：

- 初始化字符数组。例如：

```
char s[] = "abc";
```

  当字符串用作初始化值时，其被简单地视为初始化数组；每个字符都是一个单独的初始化值。更多有关初始化的信息，请参阅节 7.10。
- 在表达式中。例如：

```
strcpy (s, "abc");
```

  在表达式中使用字符串时，字符串本身是在 .econst 或 .const（根据 ABI 而定）段中使用 .string 汇编器指令定义的，并带有指向该字符串的唯一标签；包括终止 0 字节。例如，以下行定义了字符串 abc 和终止 0 字节（标签 SL5 指向该字符串）：

```
      .sect   ".econst"
SL5: .string "abc",0
```

  字符串标签的形式为 SL*n*，其中 *n* 是编译器分配的数字，用于使标签唯一。该数字从 0 开始，每定义一个字符串就增加 1。源模块中使用的所有字符串都在编译后的汇编语言模块的末尾定义。

  标签 SL*n* 表示字符串常量的地址。编译器使用此标签引用字符串表达式。

  由于字符串存储在 econst 或 .const 段中（可能在 ROM 中）并被共享，因此对于程序来说修改字符串常量是一种不好的做法。以下代码是错误使用字符串的示例：

```
const char   *a = "abc"
a[1] = 'x';             /* Incorrect! undefined behavior */
```

7.2 寄存器惯例

严格的惯例将特定寄存器与 C/C++ 环境中的特定运算相关联。如计划在 C/C++ 程序中使用汇编语言例程，则必须理解并遵循这些寄存器惯例。

寄存器惯例规定了编译器如何使用寄存器以及如何在函数调用之间保留值。寄存器变量寄存器有两种类型：入口保存和调用保存。两者之间的区别在于其在函数调用中的保存方法。被调用函数负责保存入口保存寄存器，如果需要保存调用保存寄存器的值，则由调用函数负责保存。

7.2.1 TMS320C28x 寄存器的使用和保留

表 7-2 总结了编译器如何使用 TMS320C28x 寄存器并显示了哪些寄存器定义为在函数调用之间保留。FPU 使用所有 C28x 寄存器以及表 7-3 中所述的寄存器。

有关 C28x 寄存器的详细信息，请参阅《*TMS320C28x CPU 和指令集参考指南*》(SPRU430)。

表 7-2. 寄存器使用和保留惯例

寄存器	使用	入口保存	调用保存
AL	表达式、参数传递以及从函数返回 16 位结果	否	是
AH	表达式和参数传递	否	是
DP	数据页指针（用于访问全局变量）	否	否
PH	乘法表达式和临时变量	否	是
PL	乘法表达式和临时变量	否	是
SP	堆栈指针	(1)	(1)
T	乘法和移位表达式	否	是
TL	乘法和移位表达式	否	是
XAR0	指针和表达式	否	是
XAR1	指针和表达式	是	否
XAR2	指针、表达式和框架指针（需要时）	是	否
XAR3	指针和表达式	是	否
XAR4	指针、表达式、参数传递，以及从函数返回 32 位指针值	否	是
XAR5	指针、表达式和参数	否	是
XAR6	指针和表达式	否	是
XAR7	指针、表达式、间接调用和分支（用于实现指向函数和 switch 语句的指针）	否	是

(1) SP 根据惯例保留，即压入栈的所有内容都在返回之前弹出。

表 7-3. FPU 寄存器使用和保留惯例

FPU32 寄存器	FPU64 寄存器	使用	入口保存	调用保存
R0H	R0 = R0H:R0L	表达式、参数传递以及从函数返回 32 位浮点值	否	是
R1H	R1 = R1H:R1L	表达式和参数传递	否	是
R2H	R2 = R2H:R2L	表达式和参数传递	否	是
R3H	R3 = R3H:R3L	表达式和参数传递	否	是
R4H	R4 = R4H:R4L	表达式	是	否
R5H	R5 = R5H:R5L	表达式	是	否
R6H	R6 = R6H:R6L	表达式	是	否
R7H	R7 = R7H:R7L	表达式	是	否

7.2.2 状态寄存器

表 7-4 显示了编译器使用的所有状态字段。假定值是编译器在输入函数或从函数返回时在该字段中期望的值；此列中的破折号表示编译器不需要特定值。已修改列指示编译器生成的代码是否修改过此字段。有关状态寄存器的详细信息，请参阅《*TMS320C28x CPU 和指令集参考指南*》(SPRU430)。

表 7-4. 状态寄存器字段

字段	名称	假定值	已修改
ARP	辅助寄存器指针	-	是
C	进位	-	是
N	负标志	-	是
OVM	溢出模式	0(1)	是
PAGE0	直接/栈地址模式	0(1)	否
PM	乘积移位模式	1(1) (2)	是
SPA	栈指针对齐位	-	是（在中断中）
SXM	符号扩展模式	-	是
TC	测试/控制标志	-	是
V	溢出标志	-	是
Z	零标志	-	是

(1) 设置 C 运行时环境的初始化例程会将这些字段设置为假定值。
(2) 如果 PM=1，则乘法后的结果不会移位。如果 PM=0，则每次乘法后结果左移 1。

表 7-5 显示了编译器用于 FPU 目标的其他状态字段。有关这些寄存器的详细信息，请参阅《*TMS320C28x 扩展指令集技术参考手册*》(SPRUHS1)。

表 7-5. 仅用于 FPU 目标的浮点状态寄存器 (STF(1)) 字段

字段	名称	假定值	已修改
LVF(2) (3)	锁存的溢出浮点标志	-	是
LUF(2) (3)	锁存的下溢浮点标志	-	是
NF(2)	负浮点标志	-	是
ZF(2)	零浮点标志	-	是
NI(2)	负整数标志	-	是
ZI(2)	零整数标志位	-	是
TF(2)	测试标志位	-	是
RNDF32	舍入 F32 模式(4)	-	是
RNDF64	舍入 F64 模式(4)	-	是
SHDWS	影子模式状态	-	是

(1) 未使用的 STF 寄存器位读取 0 和写入被忽略。
(2) STF 寄存器中的指定标志可以通过 MOVST0 指令导出至 ST0 寄存器。
(3) LVF 和 LUF 标志信号可以连接到 PIE，以生成溢出和下溢中断。这是一个有用的调试工具。
(4) 如果 RNDF32 或 RNDF64 为 0，则模式舍入到零（截断），否则模式舍入到最近（偶数）。

不使用所有其他状态寄存器字段，也不影响编译器生成的代码。

7.3 函数结构和调用惯例

C/C++ 编译器对函数调用强加了一套严格的规则。除了特殊的运行时支持函数，任何调用或被 C/C++ 函数调用的函数都必须遵循这些规则。不遵循这些规则会破坏 C/C++ 环境并导致程序失败。

有关调用惯例的详细信息，请参阅 *C28x 嵌入式应用二进制接口 (EABI) 参考指南* (SPRAC71)。

图 7-1 演示了典型的函数调用。在此示例中，不能放在寄存器中的参数被传递给堆栈上的函数。该函数然后分配局部变量并调用另一个函数。此示例显示了为被调用函数分配的本地帧和参数块。没有局部变量且不需要参数块的函数不会分配本地帧。

参数块 一词是指本地帧的一部分，用于将参数传递给其他函数。采用将参数移至参数块而不是将其压入堆栈的方式来将这些参数传递给函数。本地帧和参数块同时被分配。

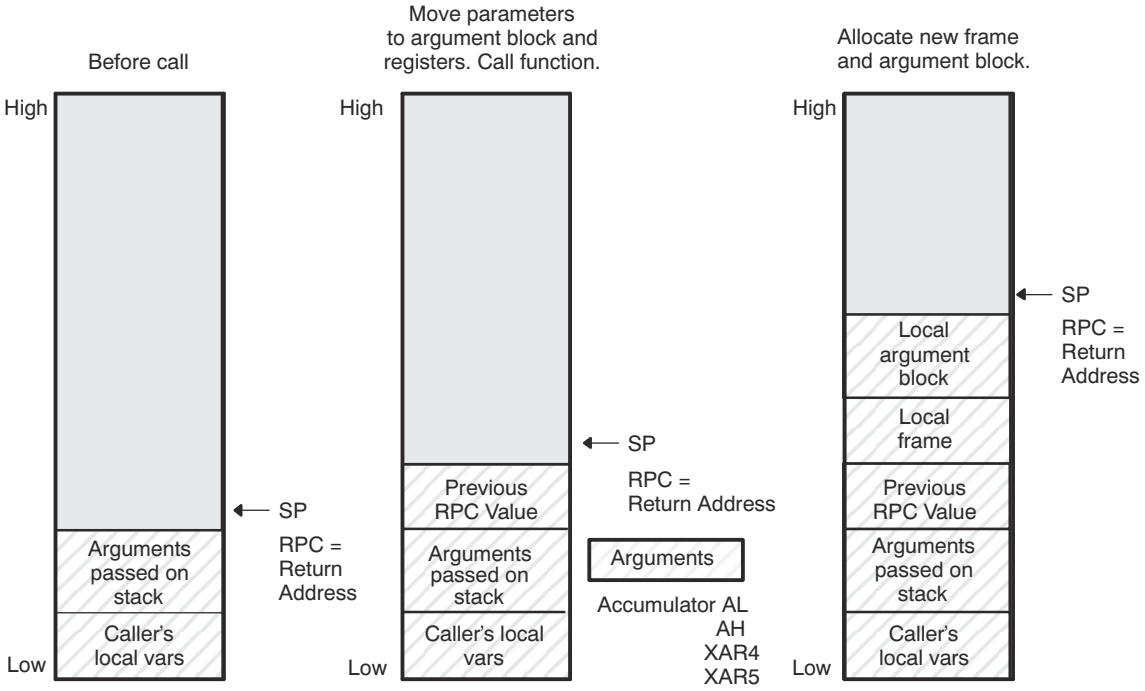


图 7-1. 在函数调用期间使用栈

7.3.1 函数如何进行调用

一个函数（父级函数）在调用另一个函数（子级函数）时会执行以下任务。

1. 寄存器的值未必由被调用函数保存（不是入口保存(SOE) 寄存器），但在函数返回后将需要保存在堆栈中。
2. 如果被调用函数返回一个结构体，则调用函数会为该结构体分配空间并将该空间的地址作为第一个参数传递给被调用函数。
3. 传递给被调用函数的参数放置在寄存器中，并在必要时放置在堆栈中。根据以下方案将参数放置在寄存器中：
   a. 如果目标是 FPU 并且存在任何 32 位浮点参数，则前四个浮点参数被放置在寄存器 R0H-R3H 中。
   b. 如果存在任何 64 位浮点参数 (long double)，它们将通过引用传递。
   c. 如果存在任何 64 位整数参数 (long long)，第一个参数将放置在 ACC 和 P 中（ACC 保存高 32 位，P 保存低 32 位）。所有其他 64 位整数参数以相反的顺序放置在堆栈中。

      如果 P 寄存器用于参数传递，则禁用该函数的 prolog/epilog 抽象。有关抽象的更多信息，请参阅节 3.13。

d. 如果有任何 32 位参数（long 或 float），则第一个参数放置在 32 位 ACC (AH/AL) 中。所有其他 32 位参数以相反的顺序放置在堆栈中。

```
func1(long a, long long b, int c, int* d);
      stack   ACC/P   XAR5,  XAR4
```

e. 指针参数放置在 XAR4 和 XAR5 中。所有其他指针都放置在堆栈中。

f. 其余的 16 位参数按 AL、AH、XAR4、XAR5（如果可用）的顺序放置。

4. 其余未放置在寄存器中的参数以相反的顺序推入堆栈中。也就是说，放在堆栈上最左边的参数最后被推入堆栈中。所有 32 位参数与堆栈中的偶数地址对齐。

   结构体参数作为结构体的地址传递。被调用函数必须创建本地副本。

   对于使用省略号声明的函数，表示该函数是用不同数量的参数调用的，相关惯例略有修改。最后一个显式声明的参数在堆栈上传递，因此其栈地址可以作为访问未声明参数的引用。

5. 在调用子级函数之前，栈指针 (SP) 必须由父级函数偶数对齐。如有必要，通过将栈指针增加 1 来完成。如果需要，编码器应该在调用之前递增 SP。这些函数调用示例展示了参数的放置位置：

```
func1 (int a, int b. long c)
       XAR4   XAR5    AH/AL
func1 (long a, int b, long c) ;
       AH/AL   XAR4   stack
vararg (int a, int b, int c, ...)
        AL     AH     stack
```

6. 调用方使用 LCR 指令来调用函数。RPC 寄存器值被压在堆栈上。然后将返回地址存储在 RPC 寄存器中。
7. 堆栈在函数边界对齐。

7.3.2 被调用函数如何响应

被调用函数（子函数）必须执行以下任务：

1. 如果被调用函数修改了 XAR1、XAR2 或 XAR3，则必须保存寄存器，因为调用函数假定这些寄存器的值在返回时被保留。如果目标是 FPU，那么除了 C28x 寄存器之外，被调用函数还必须保存寄存器 R4H、R5H、R6H 或 R7H，前提条件是被调用函数修改了其中任何一个寄存器。可以修改任何其他寄存器而无需保留。
2. 被调用函数在堆栈上为任何本地变量、临时存储区和该函数可能调用的函数的参数分配足够的空间。这种分配在函数开始时发生一次，通过向 SP 寄存器添加一个常量来实现。
3. 堆栈在函数边界对齐。
4. 如果被调用函数需要一个结构体参数，则会接收一个指向该结构体的指针。如果从被调用函数内部写入结构体，则必须在堆栈上为结构体的本地副本分配空间，并且必须从传递给结构体的指针复制本地结构体。如果未写入结构体，则可以通过指针参数在被调用函数中间接引用该结构体。

   无论是在调用函数时（以便结构体参数作为地址传递）还是在声明函数时（以便函数知道要将结构体复制到本地副本），都必须注意正确地声明接受结构体参数的函数。
5. 被调用函数会执行函数的代码。
6. 被调用函数将返回一个值。根据以下惯例将值放置在寄存器中：
   - 16 位整数值：AL
   - 32 位整数值：ACC
   - 64 位整数值：ACC/P
   - 32 位指针：XAR4
   - 结构体引用：XAR6

   如果目标是 FPU 并且返回一个 32 位浮点值，则被调用函数会将该值放置在 R0H 中。

   如果该函数返回一个结构体，则调用方为该结构体分配空间并将返回空间的地址传递给 XAR6 中的被调用函数。若要返回结构体，被调用函数会将该结构体复制到由额外参数指向的内存块。

通过这种方式，调用方可以用睿智的方式告知被调用函数从哪里返回结构体。例如，在语句 s = f(x) 中，其中 s 是一个结构体，f 是一个返回结构体的函数，调用方实际上可以像 f(&s, x) 那样进行调用。然后，函数 f 将返回结构体直接复制到 s 中，并自动执行赋值。

如果调用方不使用返回结构体值，则可以将地址值 0 作为第一个参数进行传递。这会指示被调用函数不要复制返回结构体。

无论是在调用函数时（以便传递额外参数）还是在声明函数时（以便函数知道复制结果），都必须注意正确地声明接受结构体参数的函数。64 位浮点值（长双精度型）的返回方式与结构体类似。

7. 被调用函数通过减去先前添加到 SP 的值来取消分配帧。
8. 被调用函数会恢复在步骤 1 中保存的所有寄存器的值。
9. 被调用函数使用 LRETR 指令返回。PC 被设置为 RPC 寄存器中的值。之前的 RPC 值从堆栈中弹出并存储在 RPC 寄存器中。

7.3.3 被调用函数的特殊情况（大帧）

如果需要在栈上分配的空间（上一节中的第 2 步）大于 63 个字，则需要额外的步骤和资源来确保可以访问所有本地非寄存器变量。大帧需要使用帧指针寄存器 (XAR2) 来引用帧内的本地非寄存器变量。在帧上分配空间之前，帧指针设置为指向栈上传递给被调用函数的第一个参数。如果没有将传入参数传递到栈，则帧指针将指向调用函数的返回地址，该地址在进入被调用函数时位于栈顶。

尽可能避免分配大量本地数据。例如，请勿在函数中声明大型数组。

7.3.4 访问参数和局部变量

函数通过 SP 或 FP（指定为 XAR2 的帧指针）间接访问其本地非寄存器变量及其栈参数。可以通过 SP 访问的所有本地数据和参数数据都使用 *−SP [offset] 寻址模式，因为 SP 始终指向栈顶 +1，并且栈会向更大的地址增长。

备注

必须复位 PAGE0 模式位： 编译器使用 *−SP [offset] 寻址模式，因此必须复位 PAGE0 模式位（设置为 0）。

使用 *−SP [offset] 时可用的最大偏移量为 63。如果对象距离 SP 太远，无法使用此访问模式，编译器将使用 FP (XAR2)。FP 指向帧的底部，因此使用 FP 进行的访问使用 *+FP [offset] 或 *+FP [AR0/AR1] 寻址模式。大帧需要利用 XAR2 并可能需要索引寄存器，因此，需要额外的代码和资源才能进行本地访问。

7.3.5 分配帧并访问内存中的 32 位值

一些 TMS320C28x 指令一次读取和写入 32 位内存（MOVL、ADDL 等）。这些指令要求在偶数边界上分配 32 位对象。为确保发生这种情况，编译器将执行以下步骤：

1. 它将 SP 初始化为偶数边界。
2. call 指令会向 SP 添加 2，因此它假定 SP 指向偶数地址。
3. 它确保在帧上分配的空间总数为偶数，以便 SP 指向偶数地址。
4. 它确保将 32 位对象分配给偶数地址，相对于 SP 中已知的偶数地址。
5. 中断无法假定 SP 为奇数或偶数，因此它会将 SP 与偶数地址对齐。

有关这些指令如何访问内存的更多信息，请参阅《*TMS320C28x 汇编语言工具用户指南*》。

7.4 访问 C 和 C++ 中的链接器符号

有关在 C/C++ 代码中引用链接器符号的信息，请参阅 *TMS320C28x 汇编语言工具用户指南* 中关于“链接器符号”的部分。

7.5 将 C 和 C++ 与汇编语言相连

以下是在 C/C++ 代码中使用汇编语言的方法：

- 使用汇编代码的单独模块并将它们与编译的 C/C++ 模块链接（请参阅节 7.5.1）。

- 在 C/C++ 源代码中使用汇编语言变量和常量（请参阅节 7.5.3）。
- 使用直接嵌入 C/C++ 源代码中的内联汇编语言（请参阅节 7.5.5）。
- 使用 C/C++ 源代码中的内在函数直接调用汇编语言语句（请参阅节 7.6）。

7.5.1 使用汇编语言模块与 C/C++ 代码

只要遵循 节 7.3 中定义的调用惯例，以及 节 7.2 中定义的寄存器惯例，即可同时使用 C/C++ 与汇编语言函数。C/C++ 代码可访问变量并调用使用汇编语言定义的函数，汇编代码也可访问 C/C++ 变量并调用 C/C++ 函数。

结合使用汇编语言和 C 时，请遵循以下指南：

- 所有函数，无论是以 C/C++ 还是以汇编语言编写，都必须遵循 节 7.2 中罗列的寄存器惯例。
- 必须保留由某函数修改的专用寄存器。专用寄存器包括：

XAR1	R4H（仅 FPU）
XAR2	R5H（仅 FPU）
XAR3	R6H（仅 FPU）
SP	R7H（仅 FPU）

  如果通常使用 SP，则无需显式保留。汇编函数可自由使用栈，只要在函数返回前将压入栈的全部内容弹出即可（因此要保留 SP）。

  非专用寄存器可自由使用，无需保留。
- 在调用子函数之前，栈指针 (SP) 必须由父函数偶数对齐。如有必要，这是通过将栈指针增加 1 来完成的。如果需要，编码器应在调用之前递增 SP。
- 栈在函数边界对齐。
- 中断例程必须保存所使用的*所有*寄存器。如需更多信息，请参阅 节 7.7。
- 如果通过汇编语言调用 C/C++ 函数，请加载指定的寄存器与参数，并将其余参数压入栈，如 节 7.3.1 中所述。

  在访问从 C/C++ 函数传入的参数时，适用相同的惯例。
- 长整型值和浮点值存储在存储器中，最低有效字位于低位地址。
- 结构也将返回，如 节 7.3.2 中所述。
- 任何汇编模块均不应出于任何目的使用 .cinit 段，除非进行全局变量的自动初始化。C/C++ 启动例程假设 .cinit 段包含*完整的* 初始化表。将其他信息放入 .cinit 会将表打乱，导致无法预测的结果。
- 编译器会在所有标识符的起始位置加一条下划线 (_)。在汇编语言模块中，对于要从 C/C++ 访问的所有对象，都必须使用前缀 _。例如，名为 x 的 C/C++ 对象在汇编语言中需调用 _x。对于只在一个或多个汇编语言模块中使用的标识符，未以下划线开头的任何名称均可安全使用，不会与 C/C++ 标识符产生冲突。
- 编译器会为所有外部对象指定链接名称。因此您在编写汇编语言代码时，使用的链接名称必须与编译器指定的相同。相关详细信息，请参阅节 6.12。
- 在汇编语言中声明、并从 C/C++ 访问或调用的任何对象或函数，必须在汇编语言修饰符中使用 .def 或 .global 指令进行声明。这样可将符号声明为外部引用，并允许链接器解析对它的引用。

  同理，若要从汇编语言中访问 C/C++ 函数或对象，应在汇编语言模块中使用 .ref 或 .global 指令来声明 C/C++ 对象。这样可创建未声明的外部引用，并由链接器解析。
- 由于编译代码运行时会复位 PAGE0 模式位，如果您在汇编语言函数中将 PAGE0 位设为 1，则在返回汇编代码前必须将其设回 0。
- 如果您在汇编语言中定义一个结构，并在 C 语言中使用 extern struct 来访问它，则该结构应被阻止。编译器假设会阻止结构定义对 DP 负载进行优化。因此定义应接受此假设。您可以在 .usect 指令中指定阻止标志，以阻止该结构。请参阅《*TMS320C28x 汇编语言工具用户指南*》，了解有关这些指令的更多信息。

7.5.2 从 C/C++ 访问汇编语言函数

对于将从汇编语言中调用的已在 C++ 中定义的函数，应在 C++ 文件中定义为 extern "C"。对于将从 C++ 中调用的已在汇编语言中定义的函数，必须在 C++ 中原型设计为 extern "C"。

示例 7-1 列举了一个名为 main 的 C++ 函数，此函数调用一个名为 asmfunc 的汇编语言函数，如示例 7-2 中所示。asmfunc 函数采用其单个参数，将其添加到名为 gvar 的 C++ 全局变量，并返回结果。

示例 7-1. 从 C/C++ 程序调用汇编语言函数

```
extern "C"{
extern int asmfunc(int a); /* 申明外部 asm 函数 */
int gvar = 0;              /* 定义全局变量       */
}
void main()
{
   int i = 5;
   i = asmfunc(i);        /* 正常调用函数       */
}
```

示例 7-2. 由 示例 7-1 调用的汇编语言程序

```
        .global _gvar
        .global _asmfunc
_asmfunc:
        MOVZ    DP,#_gvar
        ADDB    AL,#5
        MOV     @_gvar,AL
        LRETR
```

在示例 7-1 的 C++ 程序中，extern “C” 声明会告知编译器使用 C 命名规则（即，不进行名称改编）。当链接器解析 .global _asmfunc 引用时，程序集文件中的相应定义将匹配。

参数 i 在寄存器 AL 中传递。

7.5.3 从 C/C++ 访问汇编语言变量

有时，C/C++ 程序访问汇编语言中定义的变量或常量会很有用。根据项目定义的位置和方式，您可以使用以下方式完成此任务：在 .ebss 或 .bss 段中定义的变量、未在 .ebss 或 .bss 段中定义的变量，或者链接器符号。

7.5.3.1 访问汇编语言全局变量

从 .ebss 或 .bss 段或以 .usect 命名的段访问变量非常简单：

1. 使用 .bss 或 .usect 指令来定义变量。
2. 使用 .def 或 .global 指令将定义设置为外部引用。
3. 在汇编语言中使用适当的链接名。
4. 在 C/C++ 语言中，将变量声明为 *extern* 并正常访问它。

示例 7-4 和示例 7-3 说明了如何访问 .ebss 中定义的变量。

示例 7-3. 汇编语言变量程序

```
* Note the use of underscores in the following lines
        .ebss     _var,1   ; Define the variable
        .global   _var     ; Declare it as external
```

示例 7-4. C 程序从示例 7-3 中访问汇编语言

```
extern int var;        /* 外部变量 */
var = 1;               /* 使用该变量  */
```

7.5.3.2 访问汇编语言常量

您可以通过将 .set 指令与 .def 或 .global 指令结合使用，在汇编语言中定义全局常量，也可以使用链接器赋值语句在链接器命令文件中定义它们。这些常量只能通过使用特殊运算符从 C/C++ 中访问。

对于 C/C++ 或汇编语言中定义的**变量**，符号表包含变量所包含的*值的地址*。从 C/C++ 按名称访问程序集变量时，编译器使用符号表中的地址获取值。

但是，对于**汇编语言常量**，符号表包含常量的实际*值*。编译器无法分辨符号表中的哪些项是地址，哪些是值。如果按名称访问汇编语言（或链接器）常量，编译器将尝试使用符号表中的值作为地址来获取值。若要防止这种行为，必须使用 & (address of) 运算符来获取值 (_symval)。换而言之，如果 x 是汇编语言常量，那么它在 C/C++ 中的值是 &x。请参阅《*TMS320C28x 汇编语言工具用户指南*》中的“在 C/C++ 应用程序中使用链接器符号”，了解使用 _symval 的示例。

有关符号和符号表的更多信息，请参阅《*TMS320C28x 汇编语言工具用户指南*》中的“符号”部分。

您可以使用 cast 和 #define 来简化这些符号在程序中的使用，如以下示例中所示。

示例 7-5. 从 C 语言访问汇编语言常量

```
extern int table_size;                          /*external ref */
#define TABLE_SIZE ((int) (&table_size))        /* use cast to hide address-of */
    ...
for (I=0; i<TABLE_SIZE; ++I)                    /* use like normal symbol */
```

示例 7-6. 示例 7-5 的汇编语言程序

```
_table_size  .set10000      ; define the constant
             .global _table_size  ; make it global
```

由于您只引用存储在符号表中的符号值，符号的声明类型并不重要。在示例 7-5 中，使用了 int。您能够以类似的方式引用链接器定义的符号。

7.5.4 与汇编源代码共享 C/C++ 头文件

您可以使用 .cdecls 汇编器指令在 C 和汇编代码之间共享带有声明和原型的 C 头文件。任何合法的 C/C++都可以在 .cdecls 块中使用。C/C++ 声明将自动生成相应的汇编代码，以便您可以在汇编代码中引用 C/C++ 构造。更多相关信息，请参阅《*TMS320C28x 汇编语言工具用户指南*》中的 C/C++ 头文件章节。

7.5.5 使用内联汇编语言

在 C/C++ 程序中，您可以使用 asm 语句，在编译器创建的汇编语言文件中插入一行汇编语言。一系列 asm 语句可在编译器输出中放置多行连续的汇编语言，之间没有代码。有关更多信息，请参阅 节 6.8。

asm 语句可用于在编译器输出中插入注释。只需在汇编代码字符串前添加分号 (;)，如下所示：

```
asm(";*** this is an assembly language comment");
```

备注

使用 asm 语句：在使用 asm 语句时应注意以下要点：

- 要极为小心，不要干扰 C/C++ 环境。编译器不会检查或分析插入的指令。
- 避免在 C/C++ 代码中插入跳跃指令或标签，因为它们会迷惑代码生成器使用的寄存器跟踪算法，导致无法预测的结果。
- 使用 asm 语句时不要改变 C/C++ 变量的值。原因是编译器不会验证此类语句。它们会原样插入汇编代码中，如果您不确定它们的效果，可能会造成问题。
- 不要使用此语句插入可改变汇编环境的汇编器指令。
- 避免在 C 代码中创建汇编宏，并使用 --symdebug:dwarf (或 -g) 选项进行编译。C 环境的调试信息和汇编宏扩展不兼容。

7.6 使用内在函数访问汇编语言语句

C28x 编译器可识别多种内在函数运算符。一些汇编语句难以或无法用 C/C++ 表达，而内在函数能够表达这类语句的含义。内在函数的用法与函数类似；可以将 C/C++ 变量与这些内在函数结合使用，就像在任何普通函数中一样。

内在函数通过前导双下划线指定，可像用函数调用一样进行访问。例如：

```
long lvar;
int  ivar;
unsigned int uivar;
lvar = __mpyxu(ivar, uivar);
```

表 7-6 中列出的内在函数均可用。它们对应于所示的 TMS320C28x 汇编语言指令。更多信息，请参阅《*TMS320C28x CPU 和指令集参考指南*》。

表 7-6. TMS320C28x C/C++ 编译器内在函数

内在函数	汇编指令	说明
int **__abs16_sat(** int *src* **);**	**SETC OVM** **MOV AH,** *src* **ABS ACC** **MOV** *dst* **, AH** **CLRC OVM**	清除 OVM 状态位。将 *src* 载入 AH。取 ACC 的绝对值。将 AH 存储到 *dst*。清除 OVM 状态位。
void **__add(** int * *m* **,** int *b* **);**	**ADD** * *m* **,** *b*	将存储器位置 *m* 至 *b* 的内容相加并将结果以原子形式存储在 *m* 中。
long **__addcu(** long *src1* **,** unsigned int *src2* **);**	**ADDCU ACC,** {*mem* | *reg*}	将 *src2* 的内容和进位位的值加到 ACC 中。结果位于 ACC 中。
void **__addl(** long * *m* **,** long *b* **);**	**ADDL** * *m* **,** *b*	将存储器位置 *m* 至 *b* 的内容相加并将结果以原子形式存储在 *m* 中。
void **__and(** int * *m* **,** int *b* **);**	**AND** * *m* **,** *b*	对存储器位置 *m* 至 *b* 的内容进行与运算，并将结果以原子形式存储在 m 中。
int &**__byte(** int * *array* **,** unsigned int *byte_index* **);**	**MOVB** *array* **[** *byte_index* **].LSB,** *src* 或 **MOVB** *dst* **,** *array* **[** *byte_index* **].LSB**	C28x 中的最小可寻址单元为 16 位。因此，您通常无法访问存储器位置之外的 8 位实体。以下内在函数可帮助访问存储器位置之外的 8 位数，并可通过以下方式调用： `__byte(array,5) = 10;` `b = __byte(array,20);` 当用于读取存储器时，16 位结果的高 8 位始终为 0。
unsigned long &*y* **__byte_peripheral_32(**unsigned long **x* **);**		用于访问 32 位字节外设数据地址，而无需拆分访问。该内在函数会返回对无符号长整型值的引用，并可用于读取和写入数据。请参阅节 6.15.7。
void **__dec(** int * *m* **);**	**DEC** * *m*	以原子形式对存储器位置 *m* 的内容进行递减。
unsigned int **__disable_interrupts( );**	**PUSH ST1** **SETC INTM, DBGM** **POP** *reg16*	禁用中断并返回中断向量的旧值。
void **__dmac(** long **src1*, long **src2*, long *&accum1*, long *&accum2*, int *shift*);	**SPM** *n* ; the PM value required for *shift* **MOVL ACC,** *accum1* **MOVL P**, *accum2* **MOVL XARx,** *src1* **MOVL XAR7,** *src2* **DMAC ACC:P, *XARx++, *XAR7++**	设置移位所需的 PM 值。 将 accum1 和 accum2 移入 ACC 和 P。 将地址 src1 和 src2 移入 XARx 和 XAR7。 ACC = ACC + (src1[i+1] * src2[i+1]) << PM P = P + (src1[i] * src2[i]) << PM 更多信息请参阅节 3.15.3。
void **__eallow(** void **);**	**EALLOW**	允许 CPU 任意写入受保护的寄存器。
void **__edis(** void **);**	**EDIS**	在使用 EALLOW 后防止 CPU 任意写入受保护的寄存器。

表 7-6. TMS320C28x C/C++ 编译器内在函数（续）

内在函数	汇编指令	说明
unsigned int **__enable_interrupts( );**	**PUSH ST1** **CLRC INTM, DBGM** **POP** *reg16*	启用中断并返回中断向量的旧值。
uint32_t **__f32_bits_as_u32(** float *src*);	--	将浮点型中的位提取为 32 位寄存器。此内在函数不生成代码；该函数告诉编译器要改变解释位的方式。示例请参阅节 7.6.1。
uint64_t **__f64_bits_as_u64(** double *src*);	--	将双精度型中的位提取为 64 位寄存器。此内在函数不生成代码；该函数告诉编译器要改变解释位的方式。示例请参阅节 7.6.1。
int **__flip16(**int *src* **);**	**FLIP AX**	反转整型 *src* 中的位顺序。
long **__flip32(**long *src* **);**	**FLIP AX**	反转长整型 *src* 中的位顺序。
long long **__flip64(**long long *src* **);**	**FLIP AX**	反转超长整型 *src* 中的位顺序。
void **__inc(** int * *m* **);**	**INC** * *m*	以原子形式对存储器位置 *m* 的内容进行递增。
long**=__IQ(** long double *A* , int *N* **);**		将长双精度型 *A* 转换为正确的 IQN 值并作为长整型返回。如果两个参数都是常数，编译器会在编译期间将参数转换为 IQ 值。否则会调用 RTS 例程 __IQ。此内在函数无法用于将全局变量初始化为 .cinit 段。
long *dst* **=__IQmpy(** long *A* , long *B* , int *N* **);**		使用 C28 IQmath 库执行经优化的乘法。*dst* 成为 ACC 或 P，*A* 成为 XT：
如果 *N* == 0：	**IMPYL {ACC|P}, XT, B**	dst 为 ACC 或 P。如果 *dst* 为 ACC，该指令需要 2 个周期。如果 dst 为 P，该指令需要 1 个周期。
如果 0 < *N* < 16：	**IMPYL P, XT, B** **QMPYL ACC, XT, B** **ASR64 ACC:P, #** *N*	
如果 15 < *N* < 32：	**IMPYL P, XT, B** **QMPYL ACC, XT, B** **LSL64 ACC:P, #(32-** *N* **)**	
如果 *N* == 32：	**QMPYL {ACC|P}, XT, B**	
如果 *N* 为变量：	**IMPYL P, XT, B** **QMPYL ACC, XT, B** **MOV T,** *N* **LSR64 ACC:P, T**	
long *dst* **= __IQsat(** long *A* , long *max* , long *min* **);**		*dst* 成为 ACC。根据 *max* 和/或 *min* 的值，会生成不同的代码。使用 *max* < *min* 调用 __IQsat 会导致出现未定义的行为。
如果 *max* 和 *min* 为 22 位无符号常数：	**MOVL ACC, A** **MOVL XAR** *n* **, #** *22bits* **MINL ACC, XAR** *n* **MOVL XAR** *n* **, #** *22bits* **MAXL ACC, XAR** *n*	
如果 *max* 和 *min* 为其他常数：	**MOVL ACC, A** **MOV PL, #** *max lower 16 bits* **MOV PH, #** *max upper 16 bits* **MINL ACC, P** **MOV PL, #** *min lower 16 bits* **MOV PH, #** *min upper 16 bits* **MAXL ACC, P**	
如果 *max* 和/或 *min* 为变量：	**MOVL ACC, A** **MINL ACC,** *max* **MAXL ACC,** *min*	
long *dst* **= __IQxmpy(**long *A* , long *B* , int *N* **);**		使用 C28 IQmath 库来执行幂为 2 的经优化的乘法。*dst* 成为 ACC 或 P；*A* 成为 XT。代码会基于 *N* 的值生成。
如果 *N* == 0：	**IMPYL ACC/P, XT, B**	*dst* 位于 ACC 或 P 中。

表 7-6. TMS320C28x C/C++ 编译器内在函数（续）

内在函数	汇编指令	说明
如果 0 < *N* < 17：	**IMPYL P, XT, B** **QMPYL ACC, XT, B** **LSL64 ACC:P, #** *N*	*dst* 位于 ACC 中。
如果 0 > *N* > -17：	**QMPYL ACC, XT, B** **SETC SXM** **SFR ACC, #abs(** *N* **)**	*dst* 位于 ACC 中。
如果 16 < *N* < 32：	**IMPYL P, XT, B** **QMPYL ACC, XT, B** **ASR64 ACC:P, #** *N*	*dst* 位于 P 中。
如果 *N* == 32：	**IMPYL P, XT, B**	*dst* 位于 P 中。
如果 -16 > *N* > -33	**QMPYL ACC, XT, B** **SETC SXM SRF ACC, #16** **SRF ACC, #abs(** *N* **)−16**	*dst* 位于 ACC 中。
如果 32 < *N* < 49：	**IMPYL ACC, XT, B** **LSL ACC, #** *N* **-32**	*dst* 位于 ACC 中。
如果 -32 > *N* > -49：	**QMPYL ACC, XT, B** **SETC SXM SFR ACC, #16** **SFR ACC, #16**	*dst* 位于 ACC 中。
如果 48 < *N* < 65：	**IMPYL ACC, XT, B** **LSL64 ACC:P, #16** **LSL64 ACC:P, #** *N* **−48**	*dst* 位于 ACC 中。
如果 -48 > *N* > -65：	**QMPYL ACC, XT, B** **SETC SXM SFR ACC, #16** **SFR ACC, #16**	*dst* 位于 ACC 中。
long long **__llmax(**long long *dst* **,** long long *src* **);**	**MAXL ACC,** *src.hi32* **MAXCUL P,** *src.lo32*	如果 *src* > *dst*，则将 src 复制到 dst。
long long **__llmin(**long long *dst* **,** long long *src* **);**	**MINL ACC,** *src.hi32* **MINCUL P,** *src.lo32*	如果 *src* < *dst*，则将 src 复制到 dst
long **__lmax(**long *dst* **,** long *src* **);**	**MAXL ACC,** *src*	如果 *src* > *dst*，则将 src 复制到 dst。
long **__lmin(**long *dst* **,** long *src* **);**	**MINL ACC,** *src*	如果 *src* < *dst*，则将 src 复制到 dst
int **__max(**int *dst* **,** int *src* **);**	**MAX** *dst* **,** *src*	如果 *src* > *dst*，则将 src 复制到 dst
int **__min(**int *dst* **,** int *src* **);**	**MIN** *dst* **,** *src*	如果 *src* < *dst*，则将 src 复制到 dst
int **__mov_byte(** int *****src* **,** unsigned int *n* **);**	**MOVB AX.LSB,*+XARx[** *n* **]** 或 **MOVZ AR0/AR1, @** *n* **MOVB AX.LSB,*XARx[** {AR0|AR1} **]**	返回字节表中 *src* 指向的第 *n* 个 8 位元素。此内在函数用于确保向后兼容性。最好使用内在函数 __byte，因为它会返回引用值。使用 _byte() 无法完成的运算也无法使用 __mov_byte() 来完成。
long **__mpy(** int *src1* **,** int *src2* **);**	**MPY ACC,** *src1* **, #** *src2*	将 *src1* 移入 T 寄存器。将 16 位立即数 (*src2*) 乘以 T。结果位于 ACC 中。
long **__mpyb(** int *src1* **,** uint *src2* **);**	**MPYB** {ACC | P}**, T, #** *src2*	将 *src1*（T 寄存器）乘以无符号 8 位立即数 (*src2*)。结果位于 ACC 或 P 中。
long **__mpy_mov_t(** int *src1* **,** int *src2* **,** int ***** *dst2* **);**	**MPY ACC, T,** *src2* **MOV @** *dst2* **, T**	将 *src1*（T 寄存器）乘以 *src2*。结果位于 ACC 中。将 *src1* 移入 **dst2*。
unsigned long **__mpyu(**unit *src2* **,** unit *srt2* **);**	**MPYU** {ACC | P}**, T,** *src2*	将 *src1*（T 寄存器）乘以 *src2*。这两个操作数都会作为无符号 16 位数字处理。结果位于 ACC 或 P 中。
long **__mpyxu(** int *src1* **,** uint *src2* **);**	**MPYXU ACC, T,** {*mem*|*reg*}	T 寄存器会载入 *src1*。*src2* 由存储器引用或载入寄存器。结果位于 ACC 中。
long *dst* **= __norm32(**long *src* **,** int ***** *shift* **);**	**CSB ACC** **LSLL ACC, T** **MOV @** *shift* **, T**	将 *src* 归一化为 *dst* 并将 **shift* 更新为移位的位数。

表 7-6. TMS320C28x C/C++ 编译器内在函数（续）

内在函数	汇编指令	说明
long long *dst* = **__norm64(**long long *src* , int * *shift* **);**	**CSB ACC** **LSL64 ACC:P, T** **MOV @** *shift* **, T** **CSB ACC** **LSL64 ACC:P, T** **MOV TMP16, AH** **MOV AH, T** **ADD** *shift* **, AH** **MOV AH, TMP16**	将 64 位 *src* 归一化为 *dst* 并将 **shift* 更新为移位的位数。
void **__or(** int * *m* , int *b* **);**	**OR** * *m* , *b*	对存储器位置 *m* 至 *b* 的内容进行或运算，并将结果以原子形式存储在 m 中。
long **__qmpy32(** long *src32a* , long *src32b* , int *q* **);**	**CLRC OVM SPM − 1** **MOV T,** *src32a* **+ 1** **MPYXU P, T,** *src32b* **+ 0** **MOVP T,** *src32b* **+ 1** **MPYXU P, T,** *src32a* **+ 0** **MPYA P, T,** *src32a* **+ 1**	扩展的精密 DSP Q 数学。根据 *q* 的值会生成不同的代码。
如果 *q* = 31、30：	**SPM** *q* **− 30** **SFR ACC, #45 −** *q* **ADDL ACC, P**	
如果 *q* = 29：	**SFR ACC, #16** **ADDL ACC, P**	
如果 *q* = 28 至 24：	**SPM** *q* **- 30** **SFR ACC, #16** **SFR ACC, #29 -** *q* **ADDL ACC, P**	
如果 *q* = 23 至 13：	**SFR ACC, #16** **ADDL ACC, P** **SFR ACC, #29 −** *q*	
如果 *q* = 12 至 0：	**SFR ACC, #16** **ADDL ACC, P** **SFR ACC, #16** **SFR ACC, #13 −** *q*	
long **__qmpy32by16(**long *src32* , int *src16* , int *q* **);**	**CLRC OVM** **MOV T,** *src16* **+ 0** **MPYXU P, T,** *src32* **+ 0** **MPY P, T,** *src32* **+ 1**	扩展的精密 DSP Q 数学。根据 *q* 的值会生成不同的代码。
如果 *q* = 31、30：	**SPM** *q* **− 30** **SFR ACC, #46 −** *q* **ADDL ACC, P**	
如果 *q* = 29 至 14：	**SPM 0** **SFR ACC, #16** **ADDL ACC, P** **SFR ACC, #30 −** *q*	
如果 *q* = 13 至 0：	**SPM 0** **SFR ACC, #16** **ADDL ACC, P** **SFR ACC, #16** **SFR ACC, #14 −** *q*	
void **__restore_interrupts(**unsigned int *val* **);**	**PUSH** *val* **POP ST1**	恢复中断并将中断向量设置为值 *val*。
long **__rol(** long *src* **);**	**ROL ACC**	向左旋转 ACC。
long **__ror(** long *src* **);**	**ROR ACC**	向右旋转 ACC。

表 7-6. TMS320C28x C/C++ 编译器内在函数（续）

内在函数	汇编指令	说明
void * *result* = **__rpt_mov_imm(**void * *dst* , int *src* , int *count* **);**	**MOV** *result* , *dst* **MOV ARx,** *dst* **RPT #** *count* **|| MOV *XARx++, #** *src*	将 *dst* 寄存器移入 *result* 寄存器。将 *dst* 寄存器移入临时 (ARx) 寄存器。将立即数 *src* 复制到该临时寄存器 *count* + 1 次。 *src* 必须为 16 位立即数。*count* 可以是 0 至 255 的立即数或变量。
int **__rpt_norm_inc(** long *src* , int *dst* , int *count* **);**	**MOV ARx,** *dst* **RPT #** *count* **|| NORM ACC, ARx++**	重复归一化累加器值 *count* + 1 次。 *count* 可以是 0 至 255 的立即数或变量。
int **__rpt_norm_dec(**long *src* , int *dst* , int *count* **);**	**MOV ARx,** *dst* **RPT #** *count* **|| NORM ACC, ARx--**	重复归一化累加器值 *count* + 1 次。 *count* 可以是 0 至 255 的立即数或变量。
long **__rpt_rol(**long *src* , int *count* **);**	**RPT #** *count* **|| ROL ACC**	向左重复旋转累加器 *count* + 1 次。结果位于 ACC 中。 *count* 可以是 0 至 255 的立即数或变量。
long **__rpt_ror(**long *src* , int *count* **);**	**RPT #** *count* **|| ROR ACC**	向右重复旋转累加器 *count* + 1 次。结果位于 ACC 中。 *count* 可以是 0 至 255 的立即数或变量。
long **__rpt_subcu(**long *dst* , int *src* , int *count* **);**	**RPT** *count* **|| SUBCU ACC,** *src*	*src* 操作数从存储器引用，或者载入寄存器并用作 SUBCU 指令的操作数。结果位于 ACC 中。 *count* 可以是 0 至 255 的立即数或变量。该指令会重复 *count* + 1 次。
unsigned long **__rpt_subcul(**unsigned long *num*, unsigned long *den*, unsigned long *&remainder*, int *count* **);**	**RPT** *count* **|| SUBCUL ACC,** *den*	执行无符号模除法中常用的重复条件长整型减法。返回商。
long **__sat(** long *src* **);**	**SAT ACC**	在 ACC 中载入 32 位 *src*。结果位于 ACC 中。
long **__sat32(** long *src* , long *limit* **);**	**SETC OVM** **ADDL ACC,** {*mem*|**P**} **SUBL ACC,** {*mem*|**P**} **SUBL ACC,** {*mem*|**P**} **ADDL ACC,** {*mem*|**P**} **CLRC OVM**	使 32 位值饱和为 32 位掩码。在 ACC 中载入 *src*。Limit 值从存储器引用或者载入 P 寄存器。结果位于 ACC 中。
long **__sathigh16(**long *src* , int *limit* **);**	**SETC OVM** **ADDL ACC,** {*mem*|**P**}**<<16** **SUBL ACC,** {*mem*|**P**}**<<16** **SUBL ACC,** {*mem*|**P**}**<<16** **ADDL ACC,** {*mem*|**P**}**<<16** **CLRC OVM** **SFR ACC,** *rshift*	使 32 位值饱和至高 16 位。在 ACC 中载入 *src*。*limit* 值从存储器引用或者载入寄存器。结果位于 ACC 中。结果可以向右移位并存储为整型。例如： `ivar=__sathigh16(lvar, mask)>>6;`
long **__satlow16(** long *src* **);**	**SETC OVM** **MOV T, #0xFFFF** **CLR SXM** ; if necessary **ADD ACC, T <<15** **SUB ACC, T <<15** **SUB ACC, T <<15** **ADD ACC, T <<15** **CLRC OVM**	使 32 位值饱和至低 16 位。在 ACC 中载入 *src*。在 T 寄存器中载入 #0xFFFF。结果位于 ACC 中。
long **__sbbu(** long *src1* , uint *src2* **);**	**SBBU ACC,** *src2*	从 ACC (*src1*) 中减去 *src2* + C 的逻辑反向。结果位于 ACC 中。
void **__sub(** int * *m* , int *b* **);**	**SUB *** *m* **,** *b*	从存储器位置 *m* 的内容中减去 *b* ，并将结果以原子形式存储在 m 中。
long **__subcu(** long *src1* , int *src2* **);**	**SUBCU ACC,** *src2*	从 ACC (*src1*) 中减去向左移 15 位的 *src2*。结果位于 ACC 中。
unsigned long **__subcul(**unsigned long *num*, unsigned long *den*, unsigned long *&remainder* **);**	**SUBCUL ACC,** *den*	执行无符号模除法中常用的单次条件长整型减法。返回商。

表 7-6. TMS320C28x C/C++ 编译器内在函数（续）

内在函数	汇编指令	说明
void **__subl(** long * *m* , long *b* **);**	**SUBL** * *m* , *b*	从存储器位置 *m* 的内容中减去 *b*，并将结果以原子形式存储在 m 中。
void **__subr(** int * *m* , int *b* **);**	**SUBR** * *m* , *b*	从 *b* 中减去存储器位置 *m* 的内容，并将结果以原子形式存储在 m 中。
void **__subrl(** long * *m* , long *b* **);**	**SUBRL** * *m* , *b*	从 *b* 中减去存储器位置 *m* 的内容，并将结果以原子形式存储在 m 中。
if (__tbit(int *src* , int *bit* **));**	**TBIT** *src* , # *bit*	如果 *src* 的指定 *位* 为 1，则设置 TC 状态位。
float **__u32_bits_as_f32(** uint32_t *src*);	--	将 32 位寄存器打包为浮点型。此内在函数不生成代码；该函数告诉编译器要改变解释位的方式。示例请参阅节 7.6.1。
double **__u64_bits_as_f64(** uint64_t *src*);	--	将 64 位寄存器打包为双精度型。此内在函数不生成代码；该函数告诉编译器要改变解释位的方式。示例请参阅节 7.6.1。
void **__xor(** int * *m* , int *b* **);**	**XOR** * *m* , *b*	对存储器位置 *m* 至 *b* 的内容进行异或运算，并将结果以原子形式存储在 m 中。

7.6.1 浮点转换内在函数

下述内在函数提取浮点值中的位或将位打包成浮点值。不执行任何指令或数据移动；这些内在函数告诉编译器如何解释这些位。有关调用语法，请参阅节 7.6。

- __f32_bits_as_u32：将浮点型中的位提取为 32 位寄存器。
- __f64_bits_as_u64：将双精度型中的位提取为 64 位寄存器。
- __u32_bits_as_f32：将 32 位寄存器打包为浮点型。
- __u64_bits_as_f64：将 64 位寄存器打包为双精度型。

下述函数提取浮点数的符号位。其通过使用 __f32_bits_as_u32 内在函数将浮点数解释为 32 位寄存器来实现这一点。此内在函数不会移动存储在 *x* 中的位；它会告知编译器改变其解释位的方式。

```
bool sign_bit_set(float x) {
    return __f32_bits_as_u32(x) >> 31;
}
```

使用这些内在函数的另一种方法是将比所需精度更高的浮点常量转换为最低有效尾数位全为零的值。这使编译器能够使用一条指令而不是两条指令将 32 位常量加载到寄存器中。以下代码截断了一个常量：

```
__u32_bits_as_f32(__f32_bits_as_u32(3.14159f) & 0xffff0000)
```

7.6.2 浮点单元 (FPU) 内在函数

这些内在函数使用 32 位 (FPU32) 和 64 位 (FPU64) 硬件进行更快的浮点计算。如果 --float_support 编译器选项分别设为 fpu32 或 fpu64，则会启用这些内在函数。

如果您使用 COFF ABI，作为 float 型列出的参数和返回值也可以使用 double 型，因为两者都是 32 位类型。支持 FPU32 内在函数，但如果您使用的是 COFF ABI，则*不*支持 FPU64 内在函数。如果您在使用 COFF 时使用采用 long double 的 FPU64 内在函数，则会发生错误。

如果使用 EABI，则 float 类型为 32 位，double 类型为 64 位。如果启用了 FPU32，请仅使用 FPU32 版本。通常，如果启用了 FPU64，您可以同时使用 FPU32 和 FPU64 内在函数。但是，在 EABI 模式下使用 FPU64 时，没有交换两个 32 位 float 值的指令。在 FPU32 硬件上，SWAPF 汇编指令仅适用于 32 位 float 值，并且 __swapf 和 __swapff 内在函数是等效的。在 FPU64 硬件上，SWAPF 指令仅支持 64 位 double 值；支持 __swapf 内在函数，但 __swapff 内在函数会导致错误。

有关将 float 值和 double 值重新解释为寄存器（反之亦然）的内在函数的信息，请参阅节 7.6.1。

表 7-7. FPU 的 C/C++ 编译器内在函数

FPU 版本	内在函数	汇编指令	说明
FPU32	float **__einvf32(** float *x* **);**	**EINVF32** *x*	计算并返回 1/*x* (精度约为 8 位) 。
FPU64	double **__einvf64(** double *x* **);**	**EINVF64** *x*	计算并返回 1/*x* (精度约为 8 位) 。
FPU32	float **__eisqrtf32(** float *x* **);**	**EISQRTF32** *x*	求 1/*x* 的平方根 (精度约为 8 位) 。
FPU64	double **__eisqrtf64(** double *x* **);**	**EISQRTF64** *x*	求 1/*x* 的平方根 (精度约为 8 位) 。
FPU32	void **__f32_max_idx(** float &*dst* , float *src* , float &*idx_dst* , float *idx_src* **);**	**MAXF32** *dst* , *src* **|| MOV32** *idx_dst* , *idx_src*	If *src>dst*, copy src to dst, and copy *idx_src* to *idx_dst*.
FPU64	void **__f64_max_idx(** double &*dst* , double *src* , double &*idx_dst* , double *idx_src* **);**	**MAXF64** *dst* , *src* **|| MOV64** *idx_dst* , *idx_src*	If *src>dst*, copy src to dst, and copy *idx_src* to *idx_dst*.
FPU32	void **__f32_min_idx(** float &*dst* , float *src* , float &*idx_dst* , float *idx_src* **);**	**MINF32** *dst* , *src* **|| MOV32** *idx_dst* , *idx_src*	If *src<dst*, copy src to dst, and copy *idx_src* to *idx_dst*.
FPU64	void **__f64_min_idx(** double &*dst* , double *src* , double &*idx_dst* , double *idx_src* **);**	**MINF64** *dst* , *src* **|| MOV64** *idx_dst* , *idx_src*	If *src<dst*, copy src to dst, and copy *idx_src* to *idx_dst*.
FPU32	int **__f32toi16r(**float *src* **);**	**F32TOI16R** *dst* , *src*	将 float 转换为 int 并进行舍入。
FPU32	unsigned int **__f32toui16r(**float *src* **);**	**F32TOUI16R** *dst* , *src*	将 float 转换为无符号 int 并进行舍入。
FPU32	float **__fmax(** float *x* , float *y* **);** (1)	**MAXF32** *y* , *x*	如果 *x>y* , 将 *x* 复制到 *y*。
FPU32	float **__fmin(** float *x* , float *y* **);** (1)	**MINF32** *y* , *x*	如果 *x<y* , 将 *x* 复制到 *y*。
FPU32	float **__fracf32(**float *src* **);**	**FRACF32** *dst* , *src*	返回 *src* 的分数部分。
FPU64	double **__fracf64(**double *src* **);**	**FRACF64** *dst* , *src*	返回 *src* 的分数部分。
FPU32	float **__fsat(**float *val* , float *max* , float *min* **);**	**MAXF32** *dst* , *min* **MINF32** *dst* , *max*	如果 *min* < *val* < *max* , 则返回 val。如果 *val* < *min* , 则返回 *min*。如果 *val* > *max* , 则返回 *max*。在 *max* < *min* 的情况下调用 __fsat 会产生未定义的行为。
FPU32 (仅限该器件)	void **__swapff(** float &*a* , float &*b* **);**	**SWAPF** *a* , *b*	交换 *a* 和 *b* 的内容。
FPU32 (仅限该器件)	void **__swapf(** float &*a* , float &*b* **);**	**SWAPF** *a* , *b*	交换 *a* 和 *b* 的内容。
FPU64	void **__swapf(** double &*a* , double &*b* **);**	**SWAPF** *a* , *b*	交换 *a* 和 *b* 的内容。

(1) 当 C 代码使用比较运算符或三元运算符时，编译器会根据需要自动生成 MINF64/MAXF64 指令。

7.6.3 三角函数加速器 (TMU) 固有函数

以下固有函数使用三角函数加速器 (TMU) 来执行更快的三角函数计算。如果使用了 --tmu_support=tmu0 或 --tmu_support=tmu1 编译器选项，则会启用这些内在函数。阴影行列出了仅 --tmu_support=tmu1 时受支持以及仅针对 EABI 受支持的固有函数。

如果您使用 COFF ABI，作为浮点 (float) 型列出的参数和返回值也可以使用双精度 (double) 型，因为两者都是 32 位类型。如果使用 EABI，这些函数要求使用浮点型，因为双精度型为 64 位。

表 7-8. 适用于 TMU 的 C/C++ 编译器固有函数

内在函数	汇编指令	说明
float **__atan(** float *src* **);**	**ATANPUF32** *dst* , *src* **MPY2PIF32** *dst* , *src*	返回 *src* 弧度的反正切主值。
float **__atanpuf32(** float *src* **);**	**ATANPUF32** *dst* , *src*	返回 *src* 的反正切主值，作为标幺值提供。
float **__atan2(** float *y* , float *x* **);**	**QUADF32** *quadrant* , *ratio* , *y* , *x* **ATANPUF32** *atanpu* , *ratio* **ADDF32** *atan2pu* , *atanpu* **MPY2PIF32** *atan2* , *atan2pu*	返回 *x*、*y* 的反正切主值和象限。
float **__atan2puf32(** float *y*, float *x* **);**	**QUADF32** *quadrant* , *ratio* , *y* , *x* **ATANPUF32** *atanpu* , *ratio* **ADDF32** *dst* , *atanpu*	返回 *y*、*x* 的反正切主值和象限值。该值作为标幺值返回。
float **__cos(** float *src* **);**	**DIV2PIF32** *dst* , *src* **COSPUF32** *dst* , *src*	返回 *src* 的余弦，其中 *src* 以弧度形式提供。

表 7-8. 适用于 TMU 的 C/C++ 编译器固有函数（续）

内在函数	汇编指令	说明
float **__cospuf32(** float *src* **);**	**COSPUF32** *dst* **,** *src*	返回 *src* 的余弦，其中 *src* 以标幺值形式提供。
float **__divf32(** float *num* , float *denom* **);**	**DIVF32** *dst* **,** *num* **,** *denom*	返回使用 TMU 浮点除法硬件指令将 *num* 除以 *denom* 的值。
float **__div2pif32(** float *src* **);**	**DIV2PIF32** *dst* **,** *src*	返回将 *src* 乘以 1/2pi（实际上是除以 2pi）的结果。这会将弧度值转换为标幺值。
float **__fmodf(** float *x*, float *y* **);**	**DIVF32 R0H,R2H,R1H** **NOP; NOP; NOP; NOP;** **F32TOI32 R0H,R0H** **NOP** **I32TOF32 R0H,R0H** **NOP** **MPYF32 R0H,R1H,R0H** **NOP** **SUBF32 R0H,R2H,R0H**	返回除法运算 x/y 的浮点余数，该运算使用 x - ((int)(x/y)) * y 进行计算。（如果使用了 --fp_mode=relaxed，则可以使用 fmodf() RTS 函数来执行此内在函数。）
float **__iexp2(** float *x* **);**	**IEXP2F32** *result* **,** *x*	返回 2^^-|x| 的结果，其与 (1.0 / 2^^|x|) 相同。（仅限 tmu1 和 EABI）
float **__log2(** float *x* **);**	**LOG2F32** *logarithm* **,** *x*	返回二进制对数，也就是为获得值 x 而必须将数字 2 提升到的幂。（仅限 tmu1 和 EABI）
float **__mpy2pif32(** float *src* **);**	**MPY2PIF32** *dst* **,** *src*	返回 *src* 乘以 2pi 的结果。这会将标幺值转换为弧度值。标幺值通常用在控制应用中来表示标准化弧度。
float **__quadf32(** float *ratio*, float *y*, float *x* **);**	**QUADF32** *quadrant* **,** *ratio* **,** *y* **,** *x*	返回 *x* 和 *y* 的象限值（0.0、+/-0.25 或 +/-0.5）和比率，并且都以标幺值形式提供。
float **__sin(** float *src* **);**	**DIV2PIF32** *dst* **,** *src* **SINPUF32** *dst* **,** *src*	返回 *src* 的正弦，其中 *src* 以弧度形式提供。
float **__sinpuf32(** float *src* **);**	**SINPUF32** *dst* **,** *src*	返回 *src* 的正弦，其中 *src* 以标幺值形式提供。
float **__sqrt(** float *src* **);**	**SQRTF32** *dst* **,** *src*	返回 *src* 的平方根。

7.6.4 快速整数除法内在函数

表 7-9 中列出的内在函数可使用硬件快速整数除法支持功能更快速地执行除法运算。如果使用 --idiv_support=idiv0 编译器选项，则会启用这些内在函数。

若要使用这些内在函数，您的代码必须包含 `stdlib.h` 头文件，且 --float_support 选项必须设为 fpu32 或 fpu64。快速整数除法支持功能仅适用于 EABI。

这些内在函数遵循 ldiv 和 lldiv 标准库函数的格式。它们将 *dividend* 和 *divisor* 作为输入，并返回商和余数，分别显示在 *quot* 和 *rem* 字段中。__uldiv_t 和 __ulldiv_t 类型是由 `stdlib.h` 提供的 ldiv 和 lldiv 的无符号等效类型。此外，还额外提供了以下结构类型，以返回超长整型商和长整型余数：

```
typedef struct { long long quot; long rem; }                  __lldiv_t;
typedef struct { unsigned long long quot; unsigned long rem; } __ulldiv_t;
```

这些内在函数支持三种类型的整数除法：

- **传统除法。**也被称为截断除法或 C99 除法。这种类型的除法向零舍入。因此，余数符号与被除数的符号相同。
- **模数除法。**也被称为取整除法。这种类型的除法向负无穷大舍入。因此，余数符号与除数的符号相同。
- **欧几里德除法。**这种类型的除法向无穷级数舍入。因此，余数一直是非负数。

在三种除法运算中，*被除数* 都等于 *商* * *除数* + *余数*。如果余数为零或者被除数和除数为正数，则三种除法运算得出的商和余数都相同。

两个无符号值经过传统、欧几里得和模数除法计算得到的结果没有变化，因此只为无符号输入提供传统除法计算结果。

一般情况下，如果被除数和除数类型不同，除数会转换为被除数的类型，然后再进行计算。但是，如果被除数类型是有符号的，除数类型是无符号的，且被除数类型不大于除数类型，即对于 *_div_i32byu32 和 *_div_i64byu64 内在函数，则会使用比被除数和除数都更大的有符号类型执行除法运算，然后再转换为被除数的类型。

除以 0 的运算一直都是不明确的。

除了这些内在函数之外，当使用 --idiv_support=idiv0 编译器选项时，内置整数除法和模运算符（“/”和“%”）使用适当的更快速指令，如 节 7.8.2 中所述。无论是否包含 `stdlib.h` 头文件，都将使用这些内置运算符的更快速版本。

表 7-9. 快速整数除法的 C/C++ 编译器内在函数 (--idiv_support=idiv0)

内在函数	汇编指令	说明
16 位/16 位		
ldiv_t **__traditional_div_i16byi16(** int *dividend*, int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `I16TOF32 R1H,@Num` `MPYF32 R1H,R1H,#65536.0` `NOP` `F32TOI32 R1H,R1H` `ABSI32DIV32 R2H, R1H, R3H` `.loop #4 {SUBC4UI32 R2H, R1H, R3H}` `NEGI32DIV32 R1H, R2H`	返回 16 位/16 位传统除法的结果。
ldiv_t **__euclidean_div_i16byi16(** int *dividend*, int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `I16TOF32 R1H,@Num` `MPYF32 R1H,R1H,#65536.0` `NOP` `F32TOI32 R1H,R1H` `ABSI32DIV32 R2H, R1H, R3H` `.loop #4 {SUBC4UI32 R2H, R1H, R3H}` `ENEGI32DIV32 R1H, R2H, R3H`	返回 16 位/16 位欧几里得除法的结果。
ldiv_t **__modulo_div_i16byi16(** int *dividend*, int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `I16TOF32 R1H,@Num` `MPYF32 R1H,R1H,#65536.0` `NOP` `F32TOI32 R1H,R1H` `ABSI32DIV32 R2H, R1H, R3H` `.loop #4 {SUBC4UI32 R2H, R1H, R3H}` `MNEGI32DIV32 R1H, R2H, R3H`	返回 16 位/16 位模数除法的结果。
__uldiv_t **__traditional_div_u16byu16(** unsigned int *dividend*, unsigned int *divisor* **);**	`UI16TOF32 R3H,@Den` `F32TOUI32 R3H,R3H` `UI16TOF32 R1H,@Num` `MPYF32 R1H,R1H,#65536.0` `NOP` `F32TOUI32 R1H,R1H` `.loop #4{SUBC4UI32 R2H, R1H, R3H}`	当被除数和除数无符号时，返回 16 位/16 位传统除法的无符号结果。
32 位/32 位		
ldiv_t **__traditional_div_i32byi32(** long *dividend*, long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ABSI32DIV32 R2H, R1H, R3H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `NEGI32DIV32 R1H, R2H`	返回 32 位/32 位传统除法的结果。
ldiv_t **__euclidean_div_i32byi32(** long *dividend*, long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ABSI32DIV32 R2H, R1H, R3H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `ENEGI32DIV32 R1H, R2H, R3H`	返回 32 位/32 位欧几里得除法的结果。
ldiv_t **__modulo_div_i32byi32(** long *dividend*, long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ABSI32DIV32 R2H, R1H, R3H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `MNEGI32DIV32 R1H, R2H, R3H`	返回 32 位/32 位模数除法的结果。

表 7-9. 快速整数除法的 C/C++ 编译器内在函数 (--idiv_support=idiv0) （续）

内在函数	汇编指令	说明
ldiv_t **__traditional_div_i32byu32(** long *dividend*, unsigned long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ABSI32DIV32U R2H, R1H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `NEGI32DIV32 R1H, R2H`	当除数无符号时，返回 32 位/32 位传统除法的结果。
ldiv_t **__euclidean_div_i32byu32(** long *dividend*, unsigned long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ABSI32DIV32U R2H, R1H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `ENEGI32DIV32 R1H, R2H, R3H`	当除数无符号时，返回 32 位/32 位欧几里得除法的结果。

表 7-9. 快速整数除法的 C/C++ 编译器内在函数 (--idiv_support=idiv0) （续）

内在函数	汇编指令	说明
ldiv_t **__modulo_div_i32byu32(** long *dividend*, unsigned long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ABSI32DIV32U R2H, R1H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `MNEGI32DIV32 R1H, R2H, R3H`	当除数无符号时，返回 32 位/32 位模数除法的结果。
__uldiv_t **__traditional_div_u32byu32(** unsigned long *dividend*, unsigned long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ZERO R2` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}`	当被除数和除数无符号时，返回 32 位/32 位传统除法的无符号结果。
32 位/16 位		
ldiv_t **__traditional_div_i32byi16(** long *dividend*, int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `MOV R1H, @NUM` `.loop #3 {NOP}` `ABSI32DIV32 R2H, R1H, R3H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `NEGI32DIV32 R1H, R2H`	返回 32 位/16 位传统除法的结果。
ldiv_t **__euclidean_div_i32byi16(** long *dividend*, int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `MOV R1H, @NUM` `.loop #3 {NOP}` `ABSI32DIV32 R2H, R1H, R3H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `ENEGI32DIV32 R1H, R2H, R3H`	返回 32 位/16 位欧几里得除法的结果。
ldiv_t **__modulo_div_i32byi16(** long *dividend*, int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `MOV R1H, @NUM` `.loop #3 {NOP}` `ABSI32DIV32 R2H, R1H, R3H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `MNEGI32DIV32 R1H, R2H, R3H`	返回 32 位/16 位模数除法的结果。
__uldiv_t **__traditional_div_u32byu16(** unsigned long *dividend*, unsigned int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `MOV R1H, @NUM` `ZERO R2` `.loop #2 {NOP}` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}`	当被除数和除数无符号时，返回 32 位/16 位传统除法的无符号结果。
64 位/64 位		
lldiv_t **__traditional_div_i64byi64(** long long *dividend*, long long *divisor* **);**	`MOV32 R5H @DEN_L` `MOV32 R3H @DEN_H` `MOV32 R0H @NUM_L` `MOV32 R1H @NUM_H` `ABSI64DIV64 R2H:R4H, R1H:R0H, R3H:R5H` `.loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H,` `R3H:R5H}` `NEGI64DIV64 R1H:R0H, R2H:R4H`	返回 64 位/64 位传统除法的结果。
lldiv_t **__euclidean_div_i64byi64(** long long *dividend*, long long *divisor* **);**	`MOV32 R5H @DEN_L` `MOV32 R3H @DEN_H` `MOV32 R0H @NUM_L` `MOV32 R1H @NUM_H` `ABSI64DIV64 R2H:R4H, R1H:R0H, R3H:R5H` `.loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H,` `R3H:R5H}` `ENEGI64DIV64 R1H:R0H, R2H:R4H, R3H:R5H`	返回 64 位/64 位欧几里得除法的结果。

表 7-9. 快速整数除法的 C/C++ 编译器内在函数 (--idiv_support=idiv0) （续）

内在函数	汇编指令	说明
lldiv_t **__modulo_div_i64byi64(** long long *dividend*, long long *divisor* **);**	`MOV32 R5H @DEN_L` `MOV32 R3H @DEN_H` `MOV32 R0H @NUM_L` `MOV32 R1H @NUM_H` `ABSI64DIV64 R2H:R4H, R1H:R0H, R3H:R5H` `.loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H, R3H:R5H}` `MNEGI64DIV64 R1H:R0H, R2H:R4H, R3H:R5H`	返回 64 位/64 位模数除法的结果。
lldiv_t **__traditional_div_i64byu64(** long long *dividend*, unsigned long long *divisor* **);**	`MOV32 R5H @DEN_L` `MOV32 R3H @DEN_H` `MOV32 R0H @NUM_L` `MOV32 R1H @NUM_H` `ABSI64DIV64U R2H:R4H, R1H:R0H` `.loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H, R3H:R5H}` `NEGI64DIV64 R1H:R0H, R2H:R4H`	当除数无符号时，返回 64 位/64 位传统除法的结果。
lldiv_t **__euclidean_div_i64byu64(** long long *dividend*, unsigned long long *divisor* **);**	`MOV32 R5H @DEN_L` `MOV32 R3H @DEN_H` `MOV32 R0H @NUM_L` `MOV32 R1H @NUM_H` `ABSI64DIV64U R2H:R4H, R1H:R0H` `.loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H, R3H:R5H}` `ENEGI64DIV64 R1H:R0H, R2H:R4H, R3H:R5H`	当除数无符号时，返回 64 位/64 位欧几里得除法的结果。
lldiv_t **__modulo_div_i64byu64(** long long *dividend*, unsigned long long *divisor* **);**	`MOV32 R5H @DEN_L` `MOV32 R3H @DEN_H` `MOV32 R0H @NUM_L` `MOV32 R1H @NUM_H` `ABSI64DIV64U R2H:R4H, R1H:R0H` `.loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H, R3H:R5H}` `MNEGI64DIV64 R1H:R0H, R2H:R4H, R3H:R5H`	当除数无符号时，返回 64 位/64 位模数除法的结果。
__ulldiv_t **__traditional_div_u64byu64(** unsigned long long *dividend*, unsigned long long *divisor* **);**	`ZERO R2` `ZERO R4` `MOV32 R5H @DEN_L` `MOV32 R3H @DEN_H` `MOV32 R0H @NUM_L` `MOV32 R1H @NUM_H` `.loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H, R3H:R5H}`	当被除数和除数无符号时，返回 64 位/64 位传统除法的无符号结果。
64 位/32 位		
__lldiv_t **__traditional_div_i64byi32(** signed long long *dividend*, long *divisor* **);**	`ABSI64DIV32 R2H, R1H:R0H, R3H` `.loop 8` `SUBC4UI32 R2H, R1H, R3H` `.endloop` `SWAPF R1, R0` `.loop 8` `SUBC4UI32 R2H, R1H, R3H` `.endloop` `SWAPF R1, R0` `NEGI64DIV32 R1H:R0H, R2H` `LRETR`	返回 64 位/32 位传统除法的结果。
__lldiv_t **__euclidean_div_i64byi32(** signed long long *dividend*, long *divisor* **);**	`ABSI64DIV32 R2H, R1H:R0H, R3H` `.loop 8` `SUBC4UI32 R2H, R1H, R3H` `.endloop` `SWAPF R1, R0` `.loop 8` `SUBC4UI32 R2H, R1H, R3H` `.endloop` `SWAPF R1, R0` `ENEGI64DIV32 R1H:R0H, R2H, R3H` `LRETR`	返回 64 位/32 位欧几里得除法的结果。

表 7-9. 快速整数除法的 C/C++ 编译器内在函数 (--idiv_support=idiv0) （续）

内在函数	汇编指令	说明
__lldiv_t **__modulo_div_i64byi32(** signed long long *dividend*, long *divisor* **);**	ABSI64DIV32 R2H, R1H:R0H, R3H .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 MNEGI64DIV32 R1H:R0H, R2H, R3H LRETR	返回 64 位/32 位模数除法的结果。
__lldiv_t **__traditional_div_i64byu32(** signed long long *dividend*, unsigned long *divisor* **);**	ABSI64DIV32U R2H, R1H:R0H .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 ZERO R4 SWAPF R4, R2 NEGI64DIV64 R1H:R0H, R2H:R4H LRETR	当除数无符号时，返回 64 位/32 位传统除法的结果。
__lldiv_t **__euclidean_div_i64byu32(** signed long long *dividend*, unsigned long *divisor* **);**	ABSI64DIV32U R2H, R1H:R0H .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 ENEGI64DIV32 R1H:R0H, R2H, R3H LRETR	当除数无符号时，返回 64 位/32 位欧几里得除法的结果。
__lldiv_t **__modulo_div_i64byu32(** unsigned long long *dividend*, unsigned long *divisor* **);**	ABSI64DIV32U R2H, R1H:R0H .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 MNEGI64DIV32 R1H:R0H, R2H, R3H LRETR	当除数无符号时，返回 64 位/32 位模数除法的结果。
__ulldiv_t **__traditional_div_u64byu32(** unsigned long long *dividend*, unsigned long *divisor* **);**	ZERO R2 ZERO R4 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 LRETR	当被除数和除数无符号时，返回 64 位/32 位传统除法的无符号结果。

7.7 中断处理

只要您遵循本节中的准则，就可以中断并返回至 C/C++ 代码，而不会中断 C/C++ 环境。在 C/C++ 环境初始化完成后，启动例程不会启用或禁用中断。如果系统通过硬件复位进行初始化，则中断被禁用。如果您的系统使用中断，您必须处理任何所需的中断启用或屏蔽。此类操作对 C/C++ 环境没有影响，并且很容易与 asm 语句或调用汇编语言函数合并。

7.7.1 有关中断的要点

中断例程可以执行由任何其他函数执行的任何任务，包括访问全局变量、分配局部变量和调用其他函数。

编写中断例程时，请谨记以下要点：

- 中断处理例程不能有参数。如果声明了任何参数，则会被忽略。
- 中断处理例程可以被普通 C/C++ 代码调用，但是这样做效率很低，因为所有的寄存器都已保存。
- 中断处理例程可以处理单个中断或多个中断。编译器不会生成特定于某个中断的代码，c_int00 除外，它是系统复位中断。当您进入此例程时，您不能假设运行时栈已设置；因此，*您不能分配局部变量，也不能在运行时栈上保存任何信息*。
- 若要将中断例程与中断关联，中断函数的地址必须放置在适当的中断矢量中。通过使用 .sect 汇编器指令创建一个简单的中断地址表，您可以使用汇编器和链接器来完成此操作。
- 在汇编语言中，请记住在符号名称之前加上下划线。例如，将 c_int00 表示为 _c_int00。

7.7.2 使用 C/C++ 中断例程

如果 C/C++ 中断例程未调用任何其他函数，则只会保存和恢复中断处理程序使用的寄存器。但如果 C/C++ 中断例程调用了其他函数，这些函数可修改中断处理程序未使用的未知寄存器。因此，如果调用了任何其他函数，编译器会保存所有调用保存寄存器。

但是，如果使用了 --float_support=fpu32/fpu64 选项，则会保存和恢复 STF 寄存器。中断例程执行的任何 STF 标志修改都将在例程返回时撤消。

C/C++ 中断例程与任何其他 C/C++ 函数一样，可以有局部变量和寄存器变量；但是，其在声明时应该不带参数并且应该返回 void。不应直接调用中断处理函数。

可使用 INTERRUPT pragma 或 __interrupt 关键字，直接通过 C/C++ 函数处理中断。有关 INTERRUPT pragma 的信息，请参阅节 6.9.15。有关 __interrupt 关键字的信息，请参阅节 6.5.3。

7.8 整数表达式分析

本节介绍了在计算整数表达式时要记住的一些特殊注意事项。

7.8.1 使用运行时支持调用计算的运算

TMS320C28x 不直接支持某些 C/C++ 整数运算。这些运算的计算通过调用运行时支持例程来完成。这些例程用汇编语言硬编码。它们是工具集中的对象和源运行时支持库（例如 rts2800_ml.lib）的成员。

调用这些例程的惯例以标准 C/C++ 调用惯例为模型。

运算类型	使用运行时支持调用计算的运算
16 位整型	除法（有符号）
	模数
32 位长整型	除法（有符号）
	取模
64 位超长整型	乘法(1)
	除法
	按位 AND、OR 和 XOR
	比较

(1) 如果指定了 -mf=5，则内联 64 位超长整型乘法。

7.8.2 支持快速整数除法的除法运算

如果使用 --idiv_support=idiv0 命令行选项，编译器将在使用除法 ("/") 或模 ("%") 运算符，或者 div() 或 ldiv() 函数时生成更快的指令来执行整数除法。无论是否包含 `stdlib.h` 头文件，都将使用这些内置运算符的更快速版本。

下表显示了使用除法 ("/") 和模 ("%") 运算符执行整数运算时用到的内在函数。

操作的类型	等效的内在函数调用
int/int	__traditional_div_i16byi16(int, int).quot
int % int	__traditional_div_i16byi16(int, int).rem
unsigned int/unsigned int	__traditional_div_u16byu16(unsigned int, unsigned int).quot
unsigned int % unsigned int	__traditional_div_u16byu16(unsigned int, unsigned int).rem
long/long	__traditional_div_i32byi32(long, long).quot
long % long	__traditional_div_i32byi32(long, long).rem
unsigned long/unsigned long	__traditional_div_u32byu32(unsigned long, unsigned long).quot
unsigned long % unsigned long	__traditional_div_u32byu32(unsigned long, unsigned long).rem
long long/long long	__traditional_div_i64byi64(long long, long long).quot
long long % long long	__traditional_div_i64byi64(long long, long long).rem
unsigned long long/unsigned long long	__traditional_div_u64byu64(unsigned long long, unsigned long long).quot
unsigned long long % unsigned long long	__traditional_div_u64byu64(unsigned long long, unsigned long long).rem

节 7.6.4 中列出了使您能够更准确地指定要执行的运算的内在函数。

在 C 语言中，当整数除法或模数运算的操作数具有不同的类型时，编译器会自动执行“整型提升”（也被称为隐式类型转换）。也就是说，编译器会插入隐式转换以转换为公共类型，然后在该类型中执行运算。

根据所划分的类型，所执行的内在函数可能不同于预期的内在函数。例如：

```
long          dividend_i32, quotient_i32;
unsigned long  divisor_u32,  quotient_u32;
int           divisor_i16;
/* uses __traditional_div_u32byu32, not __traditional_div_i32byu32 */
quotient_u32 = dividend_i32 / divisor_u32;
```

```
/* uses __traditional_div_i32byi32, not __traditional_div_i32byi16 */
quotient_i32 = dividend_i32 / divisor_i16;
```

整型提升可能会造成混淆，因此最好通过确保操作数类型一致来避免此问题，可能需要使用强制转换运算符。有关整型提升的更多信息，请参阅*如何在 C 代码中正确编写乘法* (SPRA683)。

7.8.3 C/C++ 代码访问 16 位乘法的上 16 位

以下方法提供了对 C/C++ 语言中 16 位乘法的上 16 位的访问：

- 带符号结果方法：

```
    int m1, m2;
    int result;
    result = ((long) m1 * (long) m2) >> 16;
```

- 无符号结果方法：

```
    unsigned m1, m2;
    unsigned result;
    result = ((unsigned long) m1 * (unsigned long) m2) >> 16;
```

备注

复杂表达式的危险之处

编译器必须识别表达式的结构才能返回预期结果。避免使用复杂的表达式，如以下示例：

```
((long)((unsigned)((a*b)+c)<5)*(long)(z*sin(w)>6))>>16
```

7.9 浮点表达式分析

对于 COFF ABI，float 和 double 均使用 IEEE 单精度数（32 位）表示。Long double 值使用 IEEE 双精度数（64 位）表示。

对于 EABI，float 类型使用 IEEE 单精度数（32 位）表示。Double 和 long double 类型使用 IEEE 双精度数（64 位）表示。

运行时支持库 rts2800_ml.lib 包含一组浮点数学函数，支持：

- 加法、减法、乘法和除法
- 比较（>、<、>=、<=、==、!=）
- 从整数或长整数到浮点数以及浮点数到整数或长整数的转换，包括有符号和无符号
- 标准误差处理

调用这些例程的规则与用于调用整数运算例程的规则相同。转换是一元运算。

7.10 系统初始化

您必须先创建 C/C++ 运行时环境，才能运行 C/C++ 程序。C/C++ 启动例程使用被称为 c_int00 (or _c_int00) 的函数来执行此任务。运行时支持源码库 rts.src 在名为 boot.c（或 boot.asm）的模块中包含此例程的源码。

若要开始运行该系统，可以分支到或调用 c_int00 函数由复位硬件调用。您必须将 c_int00 函数与其他目标文件链接。当您使用 --rom_model or --ram_model 链接选项并将标准运行时支持库作为其中一个链接器输入文件时，此操作会自动发生。

链接 C/C++ 程序时，链接器会将可执行输出文件中的入口点值设置为符号 c_int00。

c_int00 函数会执行以下任务来对环境进行初始化：

1. 为系统堆栈定义一个被称为 .stack 的段并设置初始堆栈指针
2. 通过将数据从初始化表复制到 .ebss 或 .bss 段中为对应变量分配的存储空间，对全局变量进行初始化。如果您在加载时对变量进行初始化（--ram_model 选项），加载程序会在程序运行前执行此步骤（并非由启动例程执行）。如需更多信息，请参阅 节 7.10.3。
3. 执行在全局构造函数表中找到的全局构造函数。如需更多信息，请参阅 节 7.10.3.4。
4. 调用 main() 函数来运行 C/C++ 程序

您可以更换或修改启动例程以满足系统要求。不过，启动例程*必须*执行上面列出的操作来正确地初始化 C/C++ 环境。

7.10.1 用于系统预初始化的引导挂钩函数

引导挂钩是可将应用程序函数插入 C/C++ 引导进程的点。默认引导挂钩函数随运行时支持 (RTS) 库一同提供。但是，您可以实现这些引导挂钩函数的自定义版本，如果在运行时库之前链接了 RTS 库中的默认引导挂钩函数，则自定义版本将覆盖 RTS 库中的默认引导挂钩函数。在继续进行 C/C++ 环境设置之前，这些函数可以执行任何应用特定的初始化。

请注意，TI-RTOS 操作系统使用自定义版本的引导挂钩函数进行系统设置，因此，如果使用 TI-RTOS，则应小心覆盖这些函数。

以下引导挂钩函数可用：

_system_pre_init(): 此函数提供执行应用特定的初始化的位置。它在初始化栈指针之后，但在执行任何 C/C++ 环境设置之前被调用。默认情况下，_system_pre_init() 应返回非零值。如果 _system_pre_init() 返回 0，则会绕过默认的 C/C++ 环境设置。

_system_post_cinit(): 在 C/C++ 环境设置过程中，在 C/C++ 全局数据被初始化，但在调用任何 C++ 构造函数之前调用此函数。此函数不应返回值。

7.10.2 运行时栈

运行时栈是在单个连续存储器块中分配的，并从低位地址向下增长到高位地址。SP 指向栈顶。

代码不会检查运行时栈是否溢出。当栈增长超出为其分配的内存空间限值时，就会发生栈溢出。确保为栈分配足够的存储器空间。

通过在链接器命令行上使用 --stack_size 链接选项并在选项后直接将栈大小指定为常量，可以在链接时更改栈大小。

7.10.3 COFF 变量的自动初始化

备注

本节仅适用于使用 COFF ABI 的应用。

在 C/C++ 程序开始运行之前，必须为某些全局变量分配初始值。检索这些变量的数据并使用数据初始化变量的过程被称为自动初始化。

编译器在名为 .cinit 的特殊段中构建表，其中包含用于初始化全局变量和静态变量的数据。每个编译的模块都包含这些初始化表。链接器将它们合并到一个表中（一个 .cinit 段）。引导例程或加载程序使用此表初始化所有系统变量。

备注

初始化变量

在 ANSI/ISO C 中，未显式初始化的全局变量和静态变量必须在程序执行之前设置为 0。C/C++ 编译器不会对未初始化的变量执行任何预初始化。对初始值必须为 0 的任何变量进行显式初始化。

更简单的方法是让加载程序在程序开始运行之前清除 .ebss 或 .bss 段。另一种方法是在 .ebss 或 .bss 段的链接器控件映射中将填充值设置为零。

烧录到 ROM 中的代码无法使用这些方法。

全局变量在运行时或加载时自动初始化；请参阅节 7.10.3.2 和节 7.10.3.3。也请参见 节 6.13。

7.10.3.1 初始化表

.cinit 段中的表由可变大小的初始化记录组成。每个必须自动初始化的变量在 .cinit 段中都有一条记录。图 7-2 展示了 .cinit 段的格式和初始化记录。

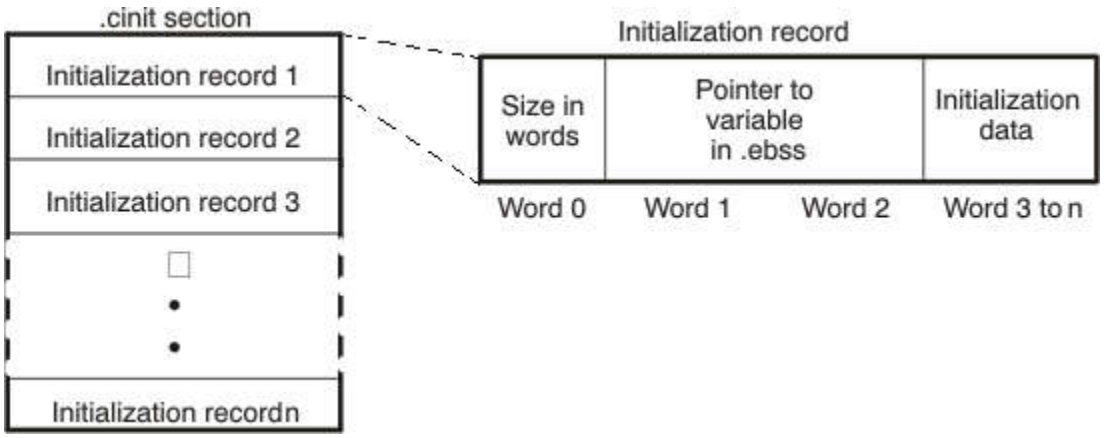


图 7-2. .cinit 段中初始化记录的格式 (COFF)

初始化记录的字段包含以下信息：

- 初始化记录的第一个字段包含初始化数据的大小（以字为单位）。也就是说，该字段包含第三个字段的大小。大小指定为负值；这是一种遗留行为，该字段的绝对值是数据的大小。
- 第二个字段包含 .ebss（或 .bss）段中初始化数据必须被复制的区域的起始地址。第二个字段需要两个字来保存地址。
- 第三个字段包含被复制到 .ebss（或 .bss）段以初始化变量的数据。该字段的宽度为变量。

每个必须自动初始化的变量在 .cinit 段中都有一条初始化记录。

以下示例展示了在 C 中定义的初始化全局变量。

```
int    i= 23;
int j[2] = { 1,2};
```

相应的初始化表如下所示：

```
        .global   _i
        .ebss     _i,1,1,0
        .global   _j
_j:     .usect    .ebss,2,1,0
        .sect     ".cinit"
        .align    1
        .field        1,16
        .field        _i+0,16
        .field        23,16          ; _i @ 0
        .sect     ".cinit"
        .align    1
        .field        -IR_1,16
        .field        _j+0,32
        .field        1,16           ; _j[0] @ 0
        .field        2,16           ; _j[1] @ 16
IR_1:   .set2
```

.cinit 段只能包含这种格式的初始化表。连接汇编语言模块时，请勿将 .cinit 段用于任何其他目的。

.pinit 或 .init_array 段（具体取决于 ABI）中的表仅包含待调用的构造函数的地址列表（请参见图 7-3）。构造函数显示在表中的 .cinit 初始化之后。

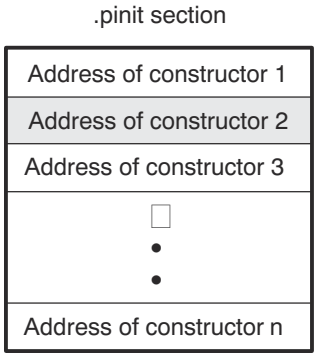


图 7-3. .pinit 或 .init_array 段中初始化记录的格式 (COFF)

当您使用 --rom_model 或 --ram_model 选项时，链接器会组合来自所有 C 模块的 .cinit 段，并在 .cinit 复合段的末尾附加一个空字。此终止记录显示为字段大小为 0 的记录，并标记初始化表的末尾。

同样，--rom_model 或 --ram_model 链接选项会使链接器组合来自所有 C/C++ 模块的所有 .pinit 或 .init_array 段，并在 .pinit 或 .init_array 复合段的末尾附加一个空字。启动例程在遇到空构造函数地址时知道全局构造函数表的末尾。

符合常量条件的变量的初始化方式有所不同；请参阅节 6.5.1。

7.10.3.2 在 COFF 运行时自动初始化变量

备注

本节仅适用于使用 COFF ABI 的应用。

在运行时自动初始化变量是自动初始化的默认方法。若要使用此方法，请使用 --rom_model 选项调用链接器。

使用此方法，.cinit 段与所有其他初始化段一同加载到内存中，全局变量在运行时初始化。链接器定义了一个名为 cinit 的特殊符号，该符号指向内存中初始化表的开头。当程序开始运行时，C/C++ 引导例程会将表中的数据（由 .cinit 指向）复制到 .ebss 段内的指定变量中。这允许初始化数据存储在 ROM 中，并在每次程序启动时复制到 RAM 中。

图 7-4 演示了运行时的自动初始化。可在任何系统中使用此方法，其中，您的应用程序会运行刻录到 ROM 中的代码。

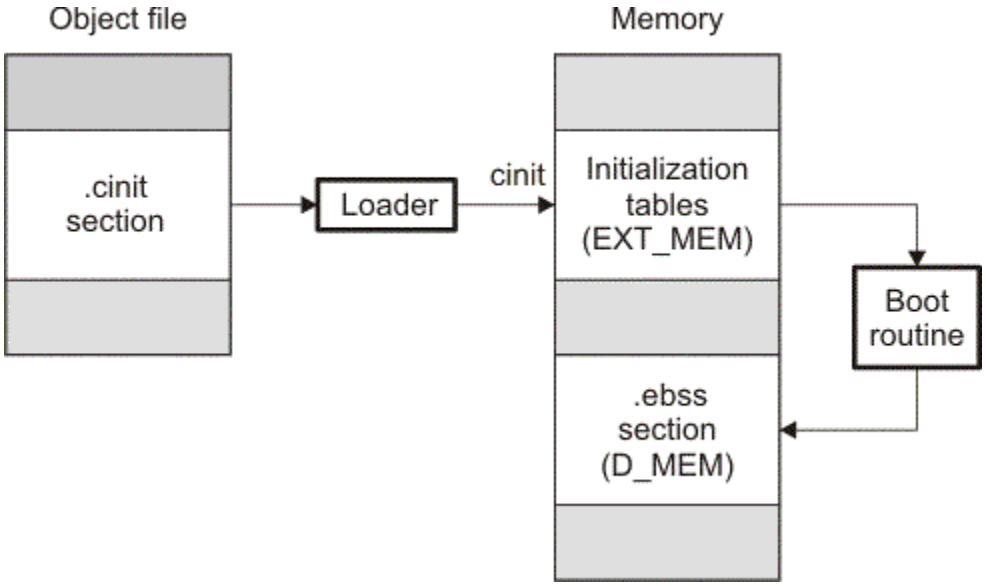


图 7-4. 运行时自动初始化 (COFF)

7.10.3.3 加载时初始化 COFF 格式的变量

备注

本节仅适用于使用 COFF ABI 的应用。

在加载时初始化变量可通过缩短引导时间和节省初始化表使用的内存来提高性能。若要使用此方法，请使用 --ram_model 选项调用链接器。

当您使用 --ram_model 链接选项时，链接器会在 .cinit 段的标头中设置 STYP_COPY 位。这会告诉加载程序不要将 .cinit 段加载至存储器。（.cinit 段不占用存储器映射中的任何空间。）链接器还会将 cinit 符号设置为 -1（通常，cinit 指向初始化表的开头）。这会向启动例程表明存储器中不存在初始化表；因此，在启动时不执行运行时初始化。

加载程序（并非编译器包的一部分）必须能够执行以下任务才能在加载时使用初始化：

- 检测目标文件中是否存在 .cinit 段
- 确定在 .cinit 段标头中设置了 STYP_COPY，这样它就知道不要将 .cinit 段复制到存储器中
- 了解初始化表的格式

图 7-5 演示了加载时变量的初始化。

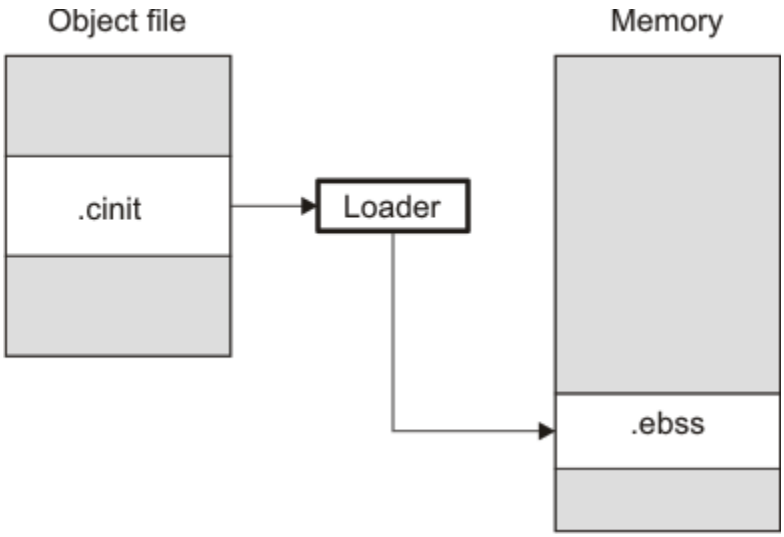


图 7-5. 加载时初始化 (COFF)

无论使用 --rom_model 还是 --ram_model 选项，始终在运行时加载和处理 .pinit 或 .init_array 段。

7.10.3.4 全局构造函数

备注

本节仅适用于使用 COFF ABI 的应用程序。

所有具有构造函数的全局 C++ 变量都必须在 main() 之前调用它们的构造函数。编译器会在全局构造函数地址的一个名为 .pinit 的段中构建表，该表必须在 main() 之前按顺序调用。链接器将每个输入文件的 .pinit 段组合起来，在 .pinit 段中形成一个表。启动例程使用此表来执行构造函数。

7.10.4 EABI 变量的自动初始化

备注

本节仅适用于使用 EABI 的应用。

在 C/C++ 程序开始运行之前，任何声明为预初始化的全局变量都必须为其分配初始值。检索这些变量的数据并使用数据初始化变量的过程被称为自动初始化。在内部，编译器和链接器会进行协调以生成压缩的初始化表。您的代码不应访问初始化表。

7.10.4.1 零初始化变量

备注

本节仅适用于使用 EABI 的应用。

在 ANSI C 中，未显式初始化的全局变量和静态变量必须在程序执行之前设置为 0。C/C++ 编译器默认支持对未初始化的变量执行预初始化。指定链接器选项 --zero_init=off 则可将此功能关闭。

只有使用 --rom_model 链接器选项（引发自动初始化）时，才发生零初始化。当您使用 --ram_model 选项进行连接时，链接器不会生成初始化记录，而加载程序必须处理数据和零初始化。

7.10.4.2 EABI 的直接初始化

编译器使用直接初始化来初始化全局变量。例如，考虑以下 C 代码：

```
int i    = 23;
int a[5] = { 1, 2, 3, 4, 5 };
```

编译器将变量“i”和“a[]”分配给 .data 段，并直接放置初始值。

```
        .global i
        .data
        .align  4
i:
        .field          23,32                   ; i @ 0
        .global a
        .data
        .align  4
a:
        .field          1,32                    ; a[0] @ 0
        .field          2,32                    ; a[1] @ 32
        .field          3,32                    ; a[2] @ 64
        .field          4,32                    ; a[3] @ 96
        .field          5,32                    ; a[4] @ 128
```

定义静态或全局变量的每个编译模块都包含这些 .data 段。链接器将 .data 段视为任何其他初始化段，并创建输出段。在加载时初始化模型中，段被加载到存储器中并由程序使用。请参阅节 7.10.4.5。

在运行时初始化模型中，链接器使用这些段中的数据创建初始化数据和附加的压缩初始化表。启动例程会处理初始化表，将数据从加载地址复制到运行地址。请参阅节 7.10.4.3。

7.10.4.3 EABI 运行时变量自动初始化

备注

本节仅适用于使用 EABI 的应用。

在运行时自动初始化变量是自动初始化的默认方法。若要使用此方法，请使用 --rom_model 选项调用链接器。

使用此方法时，链接器将从编译模块中直接被初始化的段中创建压缩初始化表和初始化数据。C/C++ 引导例程使用该表和数据以在 ROM 中初始化 RAM 中的变量。

图 7-6 演示了运行时的自动初始化。可在任何系统中使用此方法，其中，您的应用程序会运行刻录到 ROM 中的代码。

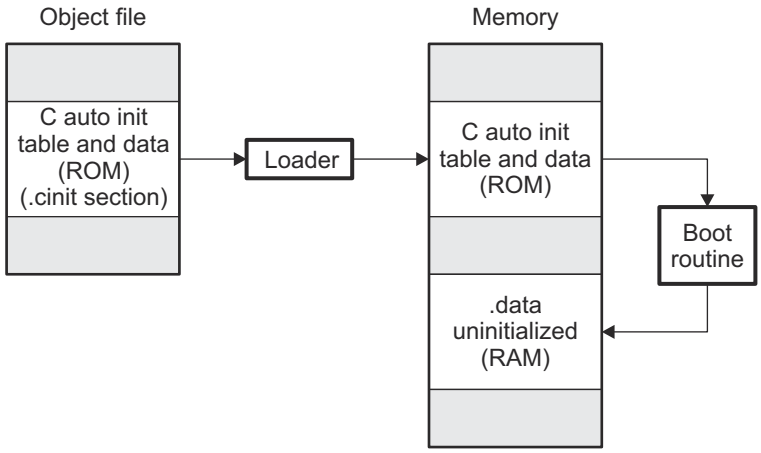


图 7-6. 运行时自动初始化 (EABI)

7.10.4.4 EABI 的自动初始化表

备注

本节仅适用于使用 EABI 的应用。

编译的目标文件没有初始化表。直接将变量初始化。当指定 --rom_model 选项时，链接器将创建 C 自动初始化表和初始化数据。链接器会在名为 .cinit 的输出段中创建表和初始化数据。

备注

从 COFF 迁移到 ELF 初始化

名称 .cinit 主要用于简化从 COFF 到 ELF 格式的迁移，链接器创建的 .cinit 段与 COFF cinit 记录没有任何共同之处（除名称外）。

自动初始化表的格式如下：

_TI_CINIT_Base:

32-bit load address	32-bit run address
⋮	⋮
32-bit load address	32-bit run address

_TI_CINIT_Limit:

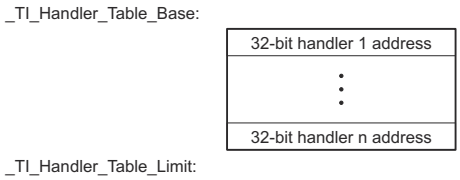
链接器定义的符号 __TI_CINIT_Base 和 __TI_CINIT_Limit 分别指向表的开头和结尾。此表中的每个条目对应一个需要初始化的输出段。可以使用不同的编码对每个输出段的初始化数据进行编码。

C 自动初始化记录中的加载地址指向以下格式的初始化数据：

8 位索引	编码数据

初始化数据的前 8 位是处理程序索引。它将索引到处理程序表中，以获取知道如何解码以下数据的处理程序函数的地址。

处理程序表是 32 位函数指针的列表。



8 位索引后面的*编码数据* 可以是以下格式类型之一。为清晰起见，还为每种格式介绍了 8 位索引。

7.10.4.4.1 数据格式遵循的长度

备注

本节仅适用于使用 EABI 的应用。

8 位索引	24 位边界填充	32 位长度 (N)	N 字节初始化数据 (未压缩)

编译器使用 24 位边界填充将长度字段对齐为 32 位边界。32 位长度字段以字节 (N) 为单位对初始化数据的长度进行编码。N 字节初始化数据不会进行压缩，按原样复制到运行地址。

运行时支持库有一个 __TI_zero_init() 函数，可处理此类型的初始化数据。此函数的第一个参数是指向位于 8 位索引后的字节的地址，第二个参数是 C 自动初始化记录的运行地址。

7.10.4.4.2 零初始化格式

备注

本节仅适用于使用 EABI 的应用。

8 位索引	24 位边界填充	32 位长度 (N)

编译器使用 24 位边界填充将长度字段对齐为 32 位边界。32 位长度字段将字节数编码为初始化的零。

运行时支持库有一个 __TI_zero_init() 函数，可处理零初始化。此函数的第一个参数是指向位于 8 位索引之后的字节的地址，第二个参数是 C 自动初始化记录的运行地址。

7.10.4.4.3 行程编码 (RLE) 格式

备注

本节仅适用于使用 EABI 的应用。

8 位索引	使用行程编码压缩的初始化数据

8 位索引之后的数据以行程编码 (RLE) 格式进行压缩。采用可以使用以下算法解压缩的简单行程编码：

1. 读取第一个字节，分隔符 (D)。
2. 读取下一个字节 (B)。
3. 如果 B != D，则将 B 复制到输出缓冲区并转到步骤 2。
4. 读取下一个字节 (L)。
   a. 如果 L == 0，则长度要么是 16 位，要么是 24 位值，或者我们已经到达数据的末尾，读取下一个字节 (L)。
      i. 如果 L == 0，则长度为 24 位值，或者已经到达数据的末尾，读取下一个字节 (L)。
         1. 如果 L == 0，则已经到达数据的末尾，转到步骤 7。
         2. 否则 L <<= 16，将接下来的两个字节读入 L 的低 16 位以完成 L 的 24 位值。
      ii. 否则 L <<= 8，将接下来的字节读入 L 的低 8 位以完成 L 的 16 位值。
   b. 否则，如果 L > 0 且 L < 4，则将 D 复制到输出缓冲区 L 次。转到步骤 2。
   c. 否则，长度为 8 位值 (L)。
5. 读取下一个字节 (C) ；C 是重复字符。
6. 将 C 写入输出缓冲区 L 次；转到步骤 2。
7. 处理结束。

运行时支持库有一个例程 __TI_decompress_rle24() 来解压缩使用 RLE 压缩的数据。此函数的第一个参数是指向位于 8 位索引之后的字节的地址，第二个参数是 C 自动初始化记录的运行地址。

备注

RLE 解压缩例程

先前的解压缩例程 __TI_decompress_rle() 包含在运行时支持库中，用于解压缩由旧版本链接器生成的 RLE 编码。

7.10.4.4.4 Lempel-Ziv-Storer-Szymanski 压缩 (LZSS) 格式

备注

本节仅适用于使用 EABI 的应用。

8 位索引	使用 LZSS 压缩的初始化数据

8 位索引之后的数据使用 LZSS 压缩进行压缩。运行时支持库具有例程 __TI_decompress_lzss() 来解压缩使用 LZSS 压缩的数据。此函数的第一个参数是指向位于 8 位索引后的字节的地址，第二个参数是 C 自动初始化记录的运行地址。

LZSS 的解压缩算法如下所示：

1. 读取 16 位，它们是标记下一个 LZSS 编码数据包开始的编码标志 (F)。
2. 对于 F 中的每个位 (B)，从最低有效位到最高有效位，执行以下操作：
   a. 如果 (B & 0x1)，则读取接下来的 16 位并将其写入输出缓冲区。然后前进到 F 中的下一位 (B) 并重复此步骤。
   b. 否则，将接下来的 16 位读入 temp (T)，长度 (L) = (T & 0xf) + 2，以及偏移量 (O) = (T >> 4)。
      i. 如果 L == 17，则读取接下来的 16 位 (L')；然后 L += L'。
      ii. 如果 O == LZSS end of data (LZSS_EOD)，我们已经到达数据的末尾，算法结束。
      iii. 在位置 (P) = 输出缓冲区 - 偏移量 (O) - 1 处，从位置 P 读取 L 个字节并将它们写入输出缓冲区。
      iv. 转到步骤 2a。

7.10.4.5 在加载时初始化变量

备注

本节仅适用于使用 EABI 的应用。

在加载时初始化变量可通过缩短引导时间和节省初始化表使用的内存来提高性能。若要使用此方法，请使用 --ram_model 选项调用链接器。

当您使用 --ram_model 链接选项时，链接器不会生成 C 自动初始化表和数据。编译后的目标文件中的直接初始化段 (.data) 根据链接器命令文件进行组合，以生成初始化输出段。加载程序会将初始化的输出部分加载到内存中。加载后，为变量指定初始值。

链接器不生成 C 自动初始化表，因此不执行引导时初始化。

图 7-7 演示了加载时变量的初始化。

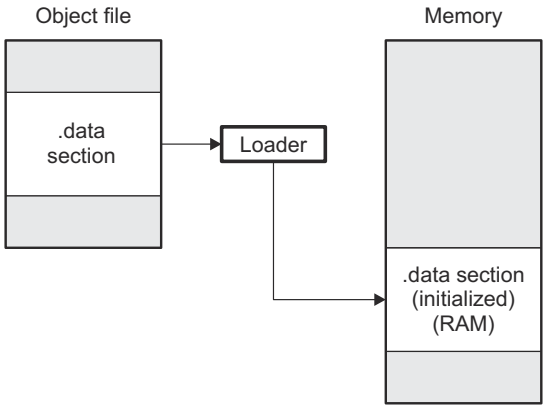


图 7-7. 加载时初始化 (EABI)

7.10.4.6 全局构造函数

备注

本节仅适用于使用 EABI 的应用。

所有具有构造函数的全局 C++ 变量都必须在 main() 之前调用它们的构造函数。编译器会构建全局构造函数地址表，必须在 main() 之前的名为 .init_array 的段中按顺序调用这些地址。链接器将每个输入文件的 .init_array 段组合起来，在 .init_array 段中形成一个表。启动例程使用此表来执行构造函数。链接器定义了两个符号来标识 .init_array 组合表，如下所示。该表不是由链接器终止的空值。

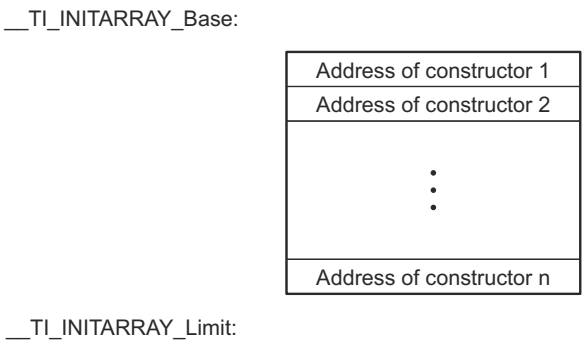


图 7-8. 构造函数表 (EABI)

章节 8

使用运行时支持函数并构建库

C/C++ 的一些功能（例如 I/O、动态内存分配、字符串操作和三角函数）作为 ANSI/ISO C/C++ 标准库提供，而不是作为编译器的一部分提供。TI 对此库的实现采用运行时支持库 (RTS)。C/C++ 编译器可实现 ISO 标准库，可处理例外情况、信号和区域问题（取决于当地语言、民族和文化的属性）的库功能除外。使用 ANSI/ISO 标准库可确保提供一组一致的函数，可移植性更高。

除了 ANSI/ISO 指定函数，运行时支持库还包括其他例程，提供处理器特定命令和 C 语言 I/O 直接请求。这些内容在节 8.1 和节 8.2 进行了详细介绍。

代码生成工具随附了库构建实用程序，可用于创建自定义运行时支持库。节 8.5 中对此过程进行了介绍。

8.1 C 和 C++ 运行时支持库

TMS320C28x 编译器版本包括可提供所有标准功能的预构建运行时支持 (RTS) 库。为 FPU 支持和 C++ 异常支持提供了单独的库。有关库命名规则的信息，请参阅节 8.1.8。

运行时支持库中包含以下内容：

- ANSI/ISO C/C++ 标准库
- C I/O 库
- 为主机操作系统提供 I/O 的低级别支持函数
- 基本算术例程
- 系统启动例程 _c_int00
- 编译器辅助函数（支持不能在 C/C++ 中直接高效表达的语言功能）

运行时支持库不包含涉及信号和区域设置问题的函数。

C++ 库支持宽字符，因为为字符定义的模板函数和类也适用于宽字符。例如，实现了宽字符流类 wios、wiostream、wstreambuf 等（对应于字符类 ios、iostream、streambuf）。但是，没有用于宽字符的低级别文件 I/O。此外，C 库接口对宽字符的支持（通过 C++ 头 <cwchar> 和 <cwctype>）是受限的，如节 6.1 中所述。

TI 不提供涵盖 C++ 库功能的文档。TI 建议参考以下任一资源：

- *The Standard C++ Library: A Tutorial and Reference*, Nicolai M. Josuttis, Addison-Wesley, ISBN 0-201-37926-0
- *The C++ Programming Language* (Third or Special Editions), Bjarne Stroustrup, Addison-Wesley, ISBN 0-201-88954-4 or 0-201-70073-5

8.1.1 将代码与对象库链接

链接程序时，必须将目标库指定为链接器输入文件之一，以便能够解析对 I/O 和运行时支持函数的引用。您可以指定库或让编译器为您选择一个。更多信息请参考 节 4.3.1。

链接库后，链接器仅包含解析未定义的引用所需的那些库成员。有关链接的更多信息，请参阅《*TMS320C28x 汇编语言工具用户指南*》。

C、C++ 以及混合的 C 和 C++ 程序可以使用相同的运行时支持库。可以从 C 和 C++ 调用和引用的运行时支持函数和变量将具有相同的链接。

8.1.2 头文件

使用 C/C++ 标准库中的函数时，必须使用编译器运行时支持随附的头文件。将 C2000_C_DIR 环境变量设为安装相关工具的 特定包含目录：“include\lib”。

8.1.3 修改库函数

您可以通过检查编译器安装 lib/src 子目录中的源代码来检查或修改库函数。例如，`C:\ti\ccsv7\tools\compiler\c2000_#.#.#\lib\src`。

找到相关的源代码后，更改特定的函数文件并重建库。

您可以使用此源码树重新构建 rts2800_ml.lib 库，或者构建新库。有关库命名的详细信息，请参阅节 8.1.8；有关构建的详细信息，请参阅节 8.5

8.1.4 支持字符串处理

RTS 库提供了标准 C 头文件 <string.h>，以及 POSIX 头文件 <strings.h>，后者提供了 C 标准不需要的附加功能。POSIX 头文件 <strings.h> 提供：

- bcmp()，等同于 memcmp()
- bcopy()，等同于 memmove()
- bzero()，等同于 memset(.., 0, ...);
- ffs()，它查找第一个位集并返回该位的索引
- index()，等同于 strchr()
- rindex()，等同于 strrchr()
- strcasecmp() 和 strncasecmp()，它们执行不区分大小写的字符串比较

此外，头文件 <string.h> 还提供了一个 C 标准不需要的附加函数。

- strdup()，它通过动态分配内存（就像使用 malloc 一样），并将字符串复制到此分配的内存来复制字符串

8.1.5 极少支持国际化

该库包含头文件 <locale.h>、<wchar.h> 和 <wctype.h>，它们提供了用以支持非 ASCII 字符集和惯例的 API。我们对这些 API 的实现在以下方面受到限制：

- 该库很少支持宽字符和多字节字符。类型 wchar_t is 实现为 int。宽字符集相当于 char 类型的值集。该库包含头文件 <wchar.h> 和 <wctype.h>，但不包含标准中指定的所有函数。请参阅节 6.4，了解有关扩展字符集的更多信息。
- C 库包含头文件 <locale.h>，但极少实现。唯一受支持的区域设置是 C 区域设置。也就是说，指定为随区域设置而变化的库行为被硬编码为 C 区域设置的行为，并且尝试通过调用 setlocale() 来安装不同的区域设置将返回 NULL。

8.1.6 时间和时钟函数支持

编译器 RTS 库在 `time.h` 中支持两个低级别时间相关标准 C 函数：

- `clock_t clock(void);`
- `time_t time(time_t *timer);`

time() 函数会返回挂钟时间。clock() 函数会返回自程序开始执行以来经过的时钟周期数；它与挂钟时间完全无关。

这些函数的默认实现要求程序在 CCS 或支持 CIO 系统调用协议的类似工具下运行。如果 CIO 不可用，而您需要使用这些函数中的其中一个，则您必须提供针对相应函数的自有定义。

clock() 函数会返回自程序开始执行以来经过的时钟周期数。这类信息可能存在于器件内的寄存器中，但位置会因平台而异。编译器的 RTS 库提供了采用 CIO 系统调用协议来与 CCS 进行通信的实现方案，这将确定如何为此器件计算正确的值。

如果 CCS 不可用，您必须提供一种有关 clock() 函数的实现方案来从器件中的相应位置收集时钟周期信息。

time() 函数会返回从 epoch 到现在的真实时间（以秒为单位）。

很多嵌入式系统中没有内部现实时钟，因此程序需要通过外部来源发现时间。编译器的 RTS 库提供了一种实现方案，利用 CIO 系统调用协议来与 CCS 进行通信，从而提供真实时间。

如果 CCS 不可用，您必须提供一种有关 time() 函数的实现方案来从一些其他来源查找时间。如果程序在操作系统中运行，该操作系统应该提供一种有关 time() 的实现方案。

time() 函数会返回从 *epoch* 到现在的秒数。在 POSIX 系统中，epoch 定义为自 1970 年 1 月 1 日 UTC 午夜零点到现在的秒数。不过，C 标准不需要任何特定的 epoch，并且 time() 的默认 TI 版本使用不同的 epoch：1900 年

1 月 1 日 UTC-6 (CST) 午夜零点。例外，默认的 TI time_t 类型为 32 位类型，而 POSIX 系统通常使用 64 位 time_t 类型。

RTS 库提供了一种有关 time() 函数的非默认实现方案，在该实现中，epoch 为 1970 年 1 月 1 日 UTC 午夜零点，time_t 类型为 64 位，也就是 __type64_t 的 typedef。

如果您的代码采用原始时间值，则您可以通过以下方式之一来处理 epoch 问题：

- 使用 epoch 为 1900 且 time_t 类型为 32 位的默认 time() 函数。本例中提供了一个单独的 __time64_t 类型。
- 定义宏命令 __TI_TIME_USES_64。time() 函数将使用 1970 epoch 和 64 位 time_t 类型，其中 time_t 为 __type64_t 的 typedef。

表 8-1. __time32_t 和 __time64_t 之间的区别

	__time32_t	__time64_t
Epoch (start)	1900 年 1 月 1 日 CST-0600	1970 年 1 月 1 日 UTC-0000
结束日期	2036 年 2 月 7 日 06:28:14	292277026596 年
符号	无符号，因此不能表示 epoch 之前的日期。	带符号，因此可以表示 epoch 之前的日期。

8.1.7 允许打开的文件数量

在 <stdio.h> 头文件中，宏命令 FOPEN_MAX 的值等于宏 _NFILE 的值，后者设置为 10。其影响就是一次只能同时打开 10 个文件（包括预定义的流 - stdin、stdout 和 stderr）。

C 标准要求 FOPEN_MAX 宏命令的最小值为 8。该宏命令决定了一次可以打开的最大文件数量。该宏命令在 stdio.h 头文件中定义，可通过更改 _NFILE 宏命令的值并重新编译库来进行修改。

8.1.8 库命名规则

默认情况下，链接器使用自动库选择功能来为您的应用程序选择正确的运行时支持库（请参阅节 4.3.1.1）。如果您手动选择库，则必须使用类似如下的命名方案来选择匹配的库：

```
rts2800_[ml|fpu32|fpu64][_eabi][_eh].lib
```

此命名规则的组成如下：

rts2800	指示该库是针对支持 C28x 构建的。
_ml	指示该库不包含对 FPU 的支持。
_fpu32	指示支持 32 位 FPU 目标。
_fpu64	指示支持 64 位 FPU 目标。（仅 EABI）
_eabi	指示该库为 EABI 格式。如果名称不包含“_eabi”，则该库为 COFF 库。
_eh	指示该库支持异常处理。

8.2 C I/O 函数

借助 C I/O 函数，能够访问主机的操作系统以执行 I/O。具备在主机上执行 I/O 的能力，您便可在调试和测试代码时拥有更多的选择。

I/O 函数在逻辑上分为多个层级：高级别、低级别和器件驱动程序级。

借助恰当编写的器件驱动程序，C 标准高级别 I/O 函数可用于在用户定义的自定义器件上执行 I/O 操作。这提供了一种在任意器件上使用高级别 I/O 函数的复杂缓冲技术的简易方法。

备注

默认主机所需的调试器：若要让默认主机器件正常工作，必须使用调试器来处理 C I/O 请求；默认的主机器件无法在嵌入式系统中自行工作。若要在嵌入式系统中工作，您需要为系统提供适当的驱动程序。

备注

C I/O 函数莫名失败：如果堆上没有足够的空间用于 C I/O 缓冲区，文件上的操作将会以静默方式失败。如果 printf() 调用莫名失败，那么原因可能就是这个。堆必须足够大，至少应足以分配执行 I/O 的每个文件所需的块大小 BUFSIZ（在 stdio.h 中定义），包括 stdout、stdin 和 stderr，以及用户代码执行的分配和分配记账开销。也可以声明一个大小字符数组 BUFSIZ，并将其传递给 setvbuf 来避免动态分配。要设置堆大小，请在链接时使用 --heap_size 选项（请参阅 *TMS320C28x 汇编语言工具用户指南* 中的*链接器说明*一章。

备注

Open 函数莫名失败：运行时支持会将打开的文件总数限制为相对于通用处理器的较小数字。如果您尝试打开的文件数量超过最大值，您可能会发现 open 函数将会莫名失败。您可以通过从 rts.src 提取源代码并编辑控制一些 C I/O 数据结构大小的常量，增加可打开文件的数量。宏命令 _NFILE 能控制一次可打开的 FILE (fopen) 对象数量（stdin、stdout 和 stderr 均计入此总数）。（另请参阅 FOPEN_MAX。）宏命令 _NSTREAM 能控制一次可打开的低级别文件描述符数量（stdin、stdout 和 stderr 下的低级别文件计入此总数）。宏命令 _NDEVICE 能控制一次可安装的器件驱动程序数量（主机器件计入此总数）。

8.2.1 高级别 I/O 函数

高级别函数是流 I/O 例程（printf、scanf、fopen、getchar 等等）的标准 C 库。这些函数会调用一个或多个低级别 I/O 函数来执行高级别 I/O 请求。高级别 I/O 例程采用 FILE 指针（也被称为*流*）运行。

便携式应用只应使用高级别 I/O 函数。

若要使用高级别 I/O 函数，请为引用 C I/O 函数的每个模块加上头文件 stdio.h（对于 C++ 代码，则为 cstdio）。例如，在名为 main.c 的文件中提供以下 C 程序：

```
#include <stdio.h>
void main()
{
   FILE *fid;
   fid = fopen("myfile","w");
   fprintf(fid,"Hello, world\n");
   fclose(fid);
   printf("Hello again, world\n");
}
```

通过发出以下编译器命令，从运行时支持库编译、链接和创建 main.out：

```
cl2000 main.c --run_linker --heap_size=400 --library=rts2800_ml.lib --output_file=main.out
```

执行 main.out 会得到

```
Hello, world
```

输出到文件以及

```
Hello again, world
```

输出到主机的 stdout 窗口。

8.2.1.1 格式化和格式转换缓冲区

C I/O 函数的内部例程，例如 printf()、vsnprintf() 和 snprintf()，会为格式转换缓冲区保留堆栈空间。缓冲区大小由宏命令 FORMAT_CONVERSION_BUFFER 设定，而该宏命令在 format.h 中定义。在减小此缓冲区的大小之前，请考虑以下问题：

- 默认缓冲区大小为 510 字节。如果定义了 MINIMAL，那么大小会设置为 32，这样便可以打印没有宽度说明符的整数值。
- 每个通过 %xxxx（%s 除外）指定的转换项目均必须适合 FORMAT_CONVERSION_BUFSIZE。这意味着，各个经过格式化并代表宽度和精度说明符的浮点或整数值需要能够放入该缓冲区。任何能表示出来的数字的实际值都应能够轻松放入该缓冲区，因此主要问题是确保宽度和/或精度大小满足约束条件。
- 使用 %s 转换的字符串的长度不受 FORMAT_CONVERSION_BUFSIZE 变化的影响。例如，您可以指定 `printf("%s value is %d", some_really_long_string, intval)`，这样不会有问题。
- 约束条件适用于要转换的每个项目。例如，`%d item1 %f item2 %e item3` 格式字符串不需要放入该缓冲区。而以 % 格式指定的每个转换项目都必须能够放入该缓冲区。
- 不存在缓冲区超限检查。

8.2.2 低级 I/O 实现概述

低级函数由以下七个基本的 I/O 函数组成：open、read、write、close、lseek、rename 和 unlink。这些低级例程提供了高级函数与器件级驱动程序之间的接口，其中器件级驱动程序用于在指定器件上实际执行 I/O 命令。

这些低级函数按适合所有 I/O 方法进行设计，甚至是那些实际上并非磁盘文件的方法。理论上，所有 I/O 通道都可以视为文件，尽管有些运算（例如 lseek）可能不合适。有关更多详细信息，请参阅节 8.2.3。

这些低级函数由名称相同的 POSIX 函数激发，但并不完全相同。

这些低级函数采用文件描述符工作。文件描述符是由 open 函数返回的整数，表示一个已打开的文件。多个文件描述符可能与一个文件关联；每个都有自己独立的文件位置指示符。

open

为 I/O 打开文件

语法

```
#include <file.h>
int open (const char * path , unsigned flags , int file_descriptor );
```

说明

open 函数用于打开 *path* 指定的文件并针对 I/O 进行准备。

- *path* 是要打开的文件的文件名，包括可选的目录路径和可选的器件指定符（请参阅节 8.2.5）。
- *flags* 是指定文件处理方式的属性。这些标志使用以下符号来指定：

```
O_RDONLY    (0x0000)   /* 打开以进行读取 */
O_WRONLY    (0x0001)   /* 打开以进行写入 */
O_RDWR      (0x0002)   /* 打开以进行读写 */
O_APPEND    (0x0008)   /* 在每次写入时添加 */
O_CREAT     (0x0200)   /* 打开并创建文件 */
O_TRUNC     (0x0400)   /* 打开并截断 */
O_BINARY    (0x8000)   /* 以二进制模式打开 */
```

  低级 I/O 例程会根据文件打开时所用的标志来允许或禁止某些操作。一些标志可能对一些器件没有意义，具体取决于器件实现对应文件的方式。
- *file_descriptor* 由 open 函数分配给一个已打开的文件。

  该函数会给每个新打开的文件分配下一个可用的文件描述符。

返回值

该函数将返回以下值之一：

非负文件描述符	成功时
-1	失败时

close

为 I/O 关闭文件

语法

#include <file.h>

int close (int *file_descriptor* **);**

说明

close 函数将关闭与 *file_descriptor* 关联的文件。

file_descriptor 是 open 函数分配给已打开文件的编号。

返回值

返回值为以下值之一：

0	成功时
-1	失败时

read

从文件读取字符

语法

#include <file.h>

int read (int *file_descriptor* **, char *** *buffer* **, unsigned** *count* **);**

说明

read 函数从与 *file_descriptor* 关联的文件读取 *count* 个字符并将其放入 *buffer*。

- *file_descriptor* 是 open 函数分配给已打开文件的编号。
- *buffer* 是读取字符的保存位置。
- *count* 是要从文件读取的字符数量。

返回值

该函数将返回以下值之一：

0	如果在读取任何字节前达到 EOF
#	读取的字符数量 (可能少于 *count*)
-1	失败时

write

向文件写入字符

语法

#include <file.h>

int write (int *file_descriptor* **, const char *** *buffer* **, unsigned** *count* **);**

说明

write 函数用于将 *count* 指定的字符数从 *buffer* 写入与 *file_descriptor* 关联的文件。

- *file_descriptor* 是 open 函数分配给已打开文件的编号。
- *buffer* 是为要写入的字符分配的保存位置。
- *count* 是要写入文件的字符数量。

返回值

该函数将返回以下值之一：

#	写入的字符数量（成功时，可能少于 *count*）
-1	失败时

lseek

设置文件位置指示符

C 语言的语法

#include <file.h>

off_t lseek (int *file_descriptor* **, off_t** *offset* **, int** *origin* **);**

说明

lseek 函数用于将给定文件的文件位置指示符设置为相对于指定来源的位置。文件位置指示符测量相对于文件开头的位置，以字符表示。

- *file_descriptor* 是 open 函数分配给已打开文件的编号。
- *offset* 指示相对于 *origin* 的偏移，以字符表示。
- *origin* 用于指示测量 *offset* 所用的基地址。*origin* 必须是以下宏命令之一：

  SEEK_SET (0x0000) 文件开头

  SEEK_CUR (0x0001) 文件位置指示符的当前值

  SEEK_END (0x0002) 文件结尾

返回值

返回值为以下值之一：

#	文件位置指示符的新值（如果成功）
(off_t)-1	失败时

unlink

删除文件

语法

#include <file.h>

int unlink (const char * *path*);

说明

unlink 函数用于删除 *path* 指定的文件。根据具体的器件，删除的文件可能仍会保留，直到为该文件打开的所有文件描述符均已关闭。请参阅节 8.2.3。

path 是这个文件的文件名，其中包括路径信息和可选的器件前缀。（请参阅节 8.2.5。）

返回值

该函数将返回以下值之一：

0	成功时
-1	失败时

rename

重命名文件

C 语言的语法

#include {<stdio.h> | <file.h>}

int rename (const char * *old_name* , const char * *new_name*);

C++ 语言的语法

#include {<cstdio> | <file.h>}

int std::rename (const char * *old_name* , const char * *new_name*);

说明

rename 函数用于更改文件的名称。

- *old_name* 是文件的当前名称。
- *new_name* 是文件的新名称。

备注

新名称中指定的可选器件必须与旧名称中的器件相匹配。如果这两个器件不匹配，则需要一个文件副本来执行重命名操作，并且 rename 函数无法执行此操作。

返回值

该函数将返回以下值之一：

0	成功时
-1	失败时

备注

尽管 rename 是低级函数，但是它由 C 标准定义并可供便携式应用程序使用。

8.2.3 器件驱动程序级别 I/O 函数

下一个级别是器件级别驱动程序。它们直接映射到低级 I/O 函数。默认器件驱动程序是主机器件驱动程序，它使用调试器来执行文件操作。主机器件驱动程序会自动用于默认的 C 流 stdin、stdout 和 stderr。

主机器件驱动程序与在主机系统上运行的调试器共享一个特殊的协议，因此主机可以执行程序所请求的 C I/O。程序要执行的 C I/O 操作指令会在 .cio 部分内名为 _CIOBUF_ 的特殊缓冲区中进行编码。调试器会在特殊断点 (C$$IO$$) 暂停程序，读取目标内存空间并进行解码，然后执行所请求的操作。结果会编码到 _CIOBUF_，程序会恢复运行，然后目标会对结果进行解码。

主机器件上实现了用于执行编码的七个函数，分别是 HOSTopen、HOSTclose、HOSTread、HOSTwrite、HOSTlseek、HOSTunlink 和 HOSTrename。每个函数均从具有相似名称的低级 I/O 函数调用。

器件驱动程序包含七个必需的函数。并非所有函数都需要对所有器件具有意义，但全部七个函数都必须进行定义。在这里，所有七个函数的名称都以 DEV 开头，但您可以选择使用 HOST 之外的任何名称。

DEV_open

为 I/O 打开文件

语法

int DEV_open (const char * *path* **, unsigned** *flags* **, int** *llv_fd* **);**

说明

此函数查找匹配 *path* 的文件并在 *flags* 请求时为 I/O 打开它。

- *path* 是要打开的文件的文件名。如果传递给 open 函数的文件的名称中带有器件前缀，器件前缀会被 open 去除，因此 DEV_open 不会看到它。（有关器件前缀的详细信息，请参阅节 8.2.5。）
- *flags* 是指定文件处理方式的属性。这些标志使用以下符号来指定：

```
O_RDONLY (0x0000) /* 打开以进行读取 */
O_WRONLY (0x0001) /* 打开以进行写入 */
O_RDWR (0x0002) /* 打开以进行读写 */
O_APPEND (0x0008) /* 在每次写入时添加 */
O_CREAT (0x0200) /* 打开并创建文件 */
O_TRUNC (0x0400) /* 打开并截断 */
O_BINARY (0x8000) /* 以二进制模式打开 */
```

  如需各个标志的进一步说明，请参阅 POSIX。
- *llv_fd* 被视为低级文件描述符。这是一个历史项目；新定义的器件驱动程序应该会忽略此参数。这与低级 I/O open 函数不同。

此函数必须安排要为每个文件描述符保存的信息，通常包括文件位置指示符以及任何重要标志。对于主机版本，所有记账工作都由在主机上运行的调试器负责处理。如果器件使用内部缓冲器，则可以在打开文件时创建缓冲器，或者在读取或写入期间创建缓冲器。

返回值

如果出于某些原因而无法打开文件，此函数必须返回 -1 以表示出错；例如，文件不存在、无法创建，或者打开了太多文件。可以选择设置 errno 的值来指示确切的错误（主机器件不会设置 errno）。一些器件可能具有特殊的故障条件；例如，如果器件为只读，则无法使用 O_WRONLY 来打开文件。

成功时，此函数必须返回一个非负的文件描述符，并且这个文件描述符必须在所有打开且由特定器件处理的文件中保持唯一。文件描述符不需要在不同器件上保持唯一。器件文件描述符仅由低级函数在调用器件驱动程序级函数时使用。低级函数 open 会为高级函数分配其自有的独特文件描述符，以便调用各个低级函数。仅使用高级 I/O 函数的代码不需要知道这些文件描述符。

DEV_close

为 I/O 关闭文件

语法

int DEV_close (int *dev_fd* **);**

说明

此函数关闭有效的 open 文件描述符。

在一些器件上，DEV_close 可能需要负责检查这是否是指向已取消链接的文件的最后一个文件描述符。如果是，它会负责确保该文件从对应器件上实际删除，并在适用时回收相应资源。

返回值

如果文件描述符在某种程度上无效，例如超出范围或已关闭，此函数应当返回 -1 以表示出错，但这不是必需的。用户不应使用无效文件描述符来调用 close()。

DEV_read

从文件读取字符

语法

int DEV_read (int *dev_fd* **, char *** *buf* **, unsigned** *count* **);**

说明

该读取函数从与 *dev_fd* 关联的输入文件读取 *count* 字节。

- *dev_fd* 是 open 函数分配给已打开文件的编号。
- *buf* 是读取字符的保存位置。
- *count* 是要从文件读取的字符数量。

返回值

如果出于某些原因而无法从文件读取任何字节，此函数必须返回 -1 以表示出错。原因可能是尝试从 O_WRONLY 文件读取，或是特定于器件的原因。

如果 count 为 0，则表示未读取任何字节，此函数会返回 0。

此函数返回读取的字节数量，范围为 0 到计数。0 表示在读取任何字节前达到 EOF。读取的字节数量小于计数字节并不表示出错；这种情况常见于文件中没有足够的字节，或者请求大于内部器件缓冲器大小。

DEV_write

向文件写入字符

语法

int DEV_write (int *dev_fd* **, const char *** *buf* **, unsigned** *count* **);**

说明

此函数会将 *count* 个字节写入输出文件。

- *dev_fd* 是 open 函数分配给已打开文件的编号。
- *buffer* 是写入字符的保存位置。
- *count* 是要写入文件的字符数量。

返回值

如果出于某些原因而无法将字节写入文件，此函数必须返回 -1 以表示出错。原因可能是尝试从 O_RDONLY 文件读取，或者是特定于器件的原因。

DEV_lseek

设置文件位置指示符

语法

off_t DEV_lseek (int *dev_fd* **, off_t** *offset* **, int** *origin* **);**

说明

此函数与 lseek 一样，用于为此文件描述符设置文件的位置指示符。

如果支持 lseek，则不应允许在文件开头之前使用查找，但应该在文件结尾之后支持查找。此类查找不会更改文件的大小，但如果后跟写入，文件大小会增加。

返回值

如果成功，此函数会返回文件位置指示符的新值。

如果出于某些原因而无法将字节写入文件，此函数必须返回 -1 以表示出错。对于许多设备，lseek 操作是没有意义的（例如计算机显示器）。

DEV_unlink

删除文件

语法

int DEV_unlink (const char * *path* **);**

说明

移除路径名与文件之间的关联。这意味着，不再能够使用此名称来打开该文件，但该文件不一定会被立即移除。

根据器件的不同，文件可能会被立即移除，但对于允许 open 文件描述符指向已取消链接的文件的器件，在最后一个文件描述符关闭之前，该文件实际上并不会被删除。请参阅节 8.2.3。

返回值

如果出于某些原因而无法取消对文件的链接（延迟移除并不算是取消链接失败），此函数必须返回 -1 以表示出错。

如果成功，此函数会返回 0。

DEV_rename

重命名文件

语法

int DEV_rename (const char * *old_name* **, const char *** *new_name* **);**

说明

此函数可更改与文件关联的名称。

- *old_name* 是文件的当前名称。
- *new_name* 是文件的新名称。

返回值

如果出于某些原因而无法重命名文件，此函数必须返回 -1 以表示出错，原因包括文件不存在或新名称已经存在等。

备注

文件位于不同的器件上，因此不宜重命名文件。通常，此操作需要用到整个文件副本，所需代价可能远超您的预期。

如果成功，此函数会返回 0。

8.2.4 为 C I/O 添加用户定义的器件驱动程序

通过 add_device 函数，您可以添加和使用器件。通过 add_device 注册器件后，高级 I/O 例程便可用于该器件上的 I/O。

您可以使用不同的协议来与任何所需器件进行通信，并使用 add_device 来安装该协议；不过，不应修改主机函数。默认流 stdin、stdout 和 stderr 可以按照示例 8-1 中所示使用 freopen() 来重新映射至用户定义的器件而非主机上的文件。如果以这种方式重新打开这些默认流，缓冲模式将更改为 _IOFBF（全缓冲）。若要恢复默认的缓冲行为，请在每个重新打开的文件中使用适当的值（对于 stdin 和 stdout，为 _IOLBF；对于 stderr，则为 _IONBF）来调用 setvbuf。

默认流 stdin、stdout 和 stderr 可以按照示例 8-1 中所示使用 freopen() 来映射至用户定义的器件而非主机上的文件。每个函数都必须根据需要设置和维护自身的数据结构。一些函数定义不执行任何操作并只应返回值。

备注

使用唯一的函数名称

函数名称 open、read、write、close、lseek、rename 和 unlink 供低级例程使用。对于由您编写的器件级别函数，请使用其他名称。

使用低级函数 add_device() 将器件添加至 device_table。器件表是一个静态定义并支持 *n* 个器件的数组，其中 *n* 由 stdio.h/cstdio 中的宏命令 _NDEVICE 定义。

器件表的第一个条目预定义为运行调试器的主机器件。低级例程 add_device() 会在器件表中查找第一个空位置，然后使用传递的参数对器件字段进行初始化。如需完整说明，请参阅 add_device 函数。

示例 8-1. 将默认流映射到器件

```
#include <stdio.h>
#include <file.h>
#include "mydevice.h"
void main()
{
    add_device("mydevice", _MSA,
               MYDEVICE_open, MYDEVICE_close,
               MYDEVICE_read, MYDEVICE_write,
               MYDEVICE_lseek, MYDEVICE_unlink, MYDEVICE_rename);
    /*--------------------------------------------------------------------*/
    /* Re-open stderr as a MYDEVICE file                                  */
    /*--------------------------------------------------------------------*/
    if (!freopen("mydevice:stderrfile", "w", stderr))
    {
        puts("Failed to freopen stderr");
        exit(EXIT_FAILURE);
    }
    /*--------------------------------------------------------------------*/
    /* stderr should not be fully buffered; we want errors to be seen as  */
    /* soon as possible.  Normally stderr is line-buffered, but this example */
    /* does not buffer stderr at all.This means that there will be one call */
    /* to write() for each character in the message.                      */
    /*--------------------------------------------------------------------*/
    if (setvbuf(stderr, NULL, _IONBF, 0))
    {
        puts("Failed to setvbuf stderr");
        exit(EXIT_FAILURE);
    }
    /*--------------------------------------------------------------------*/
    /* Try it out!                                                        */
    /*--------------------------------------------------------------------*/
    printf("This goes to stdout\n");
    fprintf(stderr, "This goes to stderr\n"); }
```

8.2.5 器件前缀

可以通过在路径名中使用器件前缀来为用户定义的器件驱动程序打开某个文件。器件前缀是调用中用来添加器件的器件名称，后跟冒号。例如：

```
FILE *fptr = fopen("mydevice:file1", "r");
int fd = open("mydevice:file2, O_RDONLY, 0);
```

如果不使用器件前缀，则将使用主机器件来打开对应文件。

add_device

向器件表添加器件

C 语言的语法

#include <file.h>

int add_device(char * *name*,
unsigned *flags* **,**
int (* *dopen* **)(const char *** *path* **, unsigned** *flags* **, int llv_fd),**
int (* *dclose* **)(int dev_fd),**
int (* *dread* **)(int** *dev_fd* **, char *** *buf* **, unsigned** *count* **),**
int (* *dwrite* **)(int** *dev_fd* **, const char *** *buf* **, unsigned** *count* **),**
off_t (* *dlseek* **)(int dev_fd, off_t** *ioffset* **, int** *origin* **),**
int (* *dunlink* **)(const char *** *path* **),**
int (* *drename* **)(const char *** *old_name* **, const char *** *new_name* **));**

定义位置

lowlev.c（在编译器安装程序的 lib/src 子目录中）

说明

add_device 函数将器件记录添加至器件表，以便在 C 语言中将该器件用于 I/O。器件表中的第一个条目预定义为运行调试器的主机器件。add_device() 函数会在器件表中查找第一个空位置，然后对表示器件的结构字段进行初始化。

若要在新添加的器件上打开一个流，请使用 fopen( ) 并以 *devicename* **:** *filename* 格式的字符串作为第一个参数。

- *name* 是表示器件名称的字符串，上限为 8 个字符。
- *flags* 是器件特性，具体如下：

  _SSA 表示器件一次仅支持一个开放流

  _MSA 表示器件支持多个开放流

  通过在 file.h 中进行定义，可以添加更多的标志。
- *dopen*、*dclose*、*dread*、*dwrite*、*dlseek*、*dunlink* 和 *drename* 说明符均为函数指针，指向器件驱动程序中的函数，这些函数由低级函数调用，用于在指定的器件上执行 I/O。您必须使用节 8.2.2 部分中指定的接口来声明这些函数。用于运行 TMS320C28x 调试器的主机所适用的器件驱动程序包含在 C I/O 库中。

返回值

该函数将返回以下值之一：

0	成功时
-1	失败时

示例

示例 8-2 将执行以下操作：

add_device（续）

向器件表添加器件

- 将器件 *mydevice* 添加至器件表
- 打开该器件上名为 *test* 的文件并将其与 FILE 指针 *fid* 关联
- 将字符串 *Hello, world* 写入该文件
- 关闭该文件

示例 8-2 显示了为 C I/O 添加和使用器件：

示例 8-2. 为 C I/O 器件编程

```
#include <file.h>
#include <stdio.h>
/*****************************************************************************/
/* 用户定义的器件驱动程序的声明                                              */
/*****************************************************************************/
extern int   MYDEVICE_open(const char *path, unsigned flags, int fno);
extern int   MYDEVICE_close(int fno);
extern int   MYDEVICE_read(int fno, char *buffer, unsigned count);
extern int   MYDEVICE_write(int fno, const char *buffer, unsigned count);
extern off_t MYDEVICE_lseek(int fno, off_t offset, int origin);
extern int   MYDEVICE_unlink(const char *path);
extern int   MYDEVICE_rename(const char *old_name, char *new_name);
main()
{
   FILE *fid;
   add_device("mydevice", _MSA, MYDEVICE_open, MYDEVICE_close, MYDEVICE_read,
              MYDEVICE_write, MYDEVICE_lseek, MYDEVICE_unlink, MYDEVICE_rename);
   fid = fopen("mydevice:test","w");
   fprintf(fid,"Hello, world\n");

   fclose(fid);
}
```

8.3 处理可重入性 (_register_lock() 和 _register_unlock() 函数)

C 标准假定只有一个执行线程，唯一的例外是为信号处理程序提供有限的支持。可重入性问题通过禁止在信号处理程序中执行任何操作来加以避免。不过，SYS/BIOS 应用程序具有多个线程，这些线程都需要修改相同的全局程序状态，例如 CIO 缓冲器，因此可重入性是个问题。

可重入性问题仍要由您自行负责解决，但运行时支持环境确实通过为临界区提供支持，从而对多线程的可重入性提供了基本的支持。这个实现方案并不能帮助您避免可重入性问题，例如从内部中断调用运行时支持函数；这仍然是您的责任。

运行时支持环境提供了钩子程序来安装临界区基元。默认情况下，假定使用单线程模型，并且不采用临界区基元。在 SYS/BIOS 等多线程系统中，内核会安排在这些钩子程序中安装信号量锁基元函数，然后在运行时支持输入需要由临界区加以保护的代码时调用这些函数。

在整个运行时支持环境中，当因访问全局状态而需要由临界区加以保护时，会调用函数 _lock()。此操作会调用提供的基元 (若已安装) 并获取信号量，然后再继续。在临界区完成后，会调用 _unlock() 来释放信号量。

通常，SYS/BIOS 负责创建和安装基元，因此您无需采取任何操作。不过，这种机制可以在不使用 SYS/BIOS 锁定机制的多线程应用程序中使用。

您不应直接定义 _lock() 和 _unlock() 函数；相反，通过调用安装函数来指示运行时支持环境使用以下基元：

```
void _register_lock  (void (  *lock)());
void _register_unlock(void (*unlock)());
```

_register_lock() 和 _register_unlock() 的参数应为无参数且不返回任何值的函数，此类函数会实现某种全局信号量锁定：

```
    extern volatile sig_atomic_t *sema = SHARED_SEMAPHORE_LOCATION;
    static int sema_depth = 0;
    static void my_lock(void)
    {
        while (ATOMIC_TEST_AND_SET(sema, MY_UNIQUE_ID) != MY_UNIQUE_ID);
        sema_depth++;
    }
    static void my_unlock(void)
    {
        if (!--sema_depth) ATOMIC_CLEAR(sema);
    }
```

运行时支持会对 _lock() 的调用进行嵌套，因此基元必须跟踪嵌套级别。

8.4 在热启动期间重新初始化变量

在系统运行期间更新系统固件并在更新完成后立即开始使用新固件，这种能力被称为实时固件更新 (LFU)。这也可以描述为“热启动”。实际的热启动由自定义入口点函数执行。

为了支持创建此类入口点，编译器提供了 __TI_auto_init_warm() RTS 函数。此函数会重新初始化具有更新属性的所有全局和静态变量，因此包含在 .TI.update 段中。请参阅节 6.15.4。调用此函数的语法如下所示：

```
void __TI_auto_init_warm();
```

如果没有全局或静态符号使用更新属性，则无需调用 __TI_auto_init_warm() 例程。自定义入口点函数负责设置堆栈指针 (SP)，然后调用 main()。有关此类函数的信息及示例，请参阅《*具有 C2000 MCU 的实时固件更新参考设计*》(TIDUEY4) 设计指南。

有关 EABI 仅支持的 LFU 的更多信息，请参阅节 2.15。

8.5 库构建流程

使用 C/C++ 编译器时，您可使用 大 量彼此不一定兼容的不同配置和选项来编译代码。由于囊括所有可能的运行时支持库变体并不实际，各个编译器版本只会预构建少量最常用的库， 例如 rts2800_ml.lib 。

为了尽可能提高灵活性，各个编译器版本中都提供了运行时支持源代码。您可根据需要构建缺少的库。链接器也可以自动构建缺少的库。这通过新库构建流程来完成的，其核心是从 CCS 5.1 开始提供的可执行的 mklib。

8.5.1 所需的非德州仪器 (TI) 软件

若要使用自包含运行时支持构建流程来使用自定义选项重建库，需要以下工具：

- sh (Bourne shell)
- gmake（GNU make 3.81 或更高版本）

  更多相关信息，请访问 GNU 网站 (http://www.gnu.org/software/make)。早期版本的 Code Composer Studio 中也提供了 GNU make (gmake)。一些适用于 Windows 的 UNIX 支持包中也包含 GNU make，例如 MKS Toolkit、Cygwin 和 Interix。从命令提示窗口执行以下命令时，Windows 平台上使用的 GNU make 应该会明确报告“此程序是为 Windows32 构建的”：

```
gmake -h
```

所有这三个程序都作为 CCS 5.1 的非可选功能提供。如果您使用的是早期版本的 CCS，它们也可以作为可选 XDC 工具功能的一部分获得。

mklib 程序会按照以下顺序查找这些可执行文件：

1. 在您的路径中
2. 在 getenv("CCS_UTILS_DIR")/cygwin 目录中
3. 在 getenv("CCS_UTILS_DIR")/bin 目录中
4. 在 getenv("XDCROOT") 目录中
5. 在 getenv("XDCROOT")/bin 目录中

如果您从命令行调用 mklib 程序，并且这些可执行文件不在您的路径中，则您必须对环境变量 CCS_UTILS_DIR 进行设置，以使 getenv("CCS_UTILS_DIR")/bin 包含正确的程序。

8.5.2 使用库构建流程

您通常应该让链接器根据需要自动重建库。如有必要，您可以直接调用 mklib 来填充库。请参阅节 8.5.2.2，以了解可能需要您这样做的情形。

8.5.2.1 通过链接器自动重建标准库

链接器主要通过 C2000_C_DIR 环境变量查找运行时支持库。通常， C2000_C_DIR 中的其中一个路径名为 *your install directory*/lib，其中包含所有预构建的库以及索引库 libc.a。链接器会搜索 C2000_C_DIR 来查找与应用程序的构建属性最为匹配的库。构建属性根据用于构建应用程序的命令行选项来间接设置。构建属性包含 CPU 版本等信息。如果明确指定了库名称（例如 -library=rts2800_ml.lib），运行时支持函数会精确地查找对应的库。如果没有指定库名称，链接器会使用索引库 libc.a 来挑选合适的库。如果通过路径指定了库（例如 –library=/foo/rts2800_ml.lib），则会假定对应库已经存在，而不会自动进行构建。

索引库描述了一组具有不同构建属性的库。链接器将会比较每个潜在库的构建属性与应用程序的构建属性，然后挑选最合适的库。有关索引库的详细信息，请参阅 *TMS320C28x 汇编语言工具用户指南* 中的归档器一章。

现在链接器已经决定了要使用的库，接下来它会检查 C2000_C_DIR 中是否存在运行时支持库。该库必须与索引库 libc.a 位于完全相同的目录中。如果该库不存在，链接器会调用 mklib 来构建它。当该库缺失时，不管是用户直接指定了该库的名称，还是允许链接器从索引库中挑选最合适的库，都会出现这种情况。

mklib 程序会构建所请求的库，并将其置于索引库所在同一目录中的 C2000_C_DIR 的“lib”目录部分中，以便用于后续编译。

要注意的事项：

- 链接器会调用 **mklib** 并等待其完成，然后再完成链接，因此您会在不常用库首次构建时遇到一次性延迟。已观察到的构建时间为 1-5 分钟。这取决于主机能力（CPU 数量等）。
- 在共享安装中，即编译器安装程序由多位用户共享时，两位用户可能会导致链接器同时重建相同的库。**mklib** 程序会尽可能减少竞争条件，但可能会出现一个构建损坏另一个构建的情形。在共享环境中，所有可能需要的库都应在安装时构建；有关直接调用 **mklib** 来避免此问题的说明，请参阅 节 8.5.2.2。
- 索引库必须存在，否则链接器无法自动重建各个库。
- 索引库必须位于用户可写入的目录中，否则不会构建该库。如果编译器必须安装为只读模式（共享安装时的一个好做法），则必须在安装时通过直接调用 **mklib** 来构建任何缺失的库。
- **mklib** 程序特定于特定库的特定版；您无法使用一个编译器版本的运行时支持 **mklib** 来构建另一个编译器版本的运行时支持库。

8.5.2.2 手动调用 mklib

在特殊情况下，您可能需要直接调用 **mklib**：

- 编译器安装目录为只读或共享。
- 您想要构建索引库 **libc.a** 中未预先配置或对 mklib 未知的运行时支持库变体。（例如，已打开源码级调试的变体。）

8.5.2.2.1 构建标准库

您可以直接调用 mklib 来构建在索引库 **libc.a** 中进行索引的任何库或所有库。这些库均会采用该库的标准选项来构建；对于 mklib，库名称和适当的标准选项集都是已知的。

实现此操作的最简单方法是将工作目录更改为编译器运行时支持库目录“lib”，并在该处调用 **mklib** 可执行文件：

```
mklib --pattern=rts2800_ml.lib
```

对于 C28x，下面是一些可以构建的库。有关完整的 RTS 库命名选项，请参阅节 8.1.8。

- rts2800_ml.lib（提供 COFF 输出的 C/C++ 运行时对象库）
- rts2800_fpu32.lib（适用于 32 位 FPU 目标且提供 COFF 输出的 C/C++ 运行时对象库）
- rts2800_fpu64_eabi.lib（适用于 64 位 FPU 目标且提供 EABI 输出的 C/C++ 运行时对象库）

8.5.2.2.2 共享或只读库目录

如果编译器工具要安装在共享或只读目录中，那么 mklib 无法在链接时构建标准库；必须在将库目录设置为共享或只读目录之前，构建对应的库。

安装时，安装用户必须构建任何其他用户将要使用的所有库。若要构建所有可能的库，请将工作目录更改为编译器 RTS 库目录“lib”并在此调用 mklib 可执行文件：

```
mklib --all
```

一些目标包含很多库，因此这一步可能需要很长时间。若要构建库的子集，请分别针对每个所需的库调用 mklib。

8.5.2.2.3 使用自定义选项构建库

您可以使用所需的任何额外自定义选项来构建库。在构建支持器件例外权变措施的库版本时，这会非常有用。生成的库不是标准库，也不得放入“lib”目录，而应当放在与项目对应的本地目录中。若要构建 rts2800_ml.lib 库的调试版本，请将工作目录更改为“lib”目录并运行以下命令：

```
mklib --pattern=rts2800_ml.lib --name=rts2800_debug.lib --install_to=$Project/Debug --
extra_options="-g"
```

8.5.2.2.4 mklib 程序选项摘要

运行以下命令来查看完整的选项列表，如表 8-2 中所述。

```
mklib --help
```

表 8-2. mklib 程序选项

选项	效果
--index= *filename*	此版本的索引库 (libc.a)。用于查找定制构建的模板库，以及查找源文件（位于编译器安装程序的 lib/src 子目录中）。必备选项。
--pattern= *filename*	用于构建库的模式。如果既未指定 --extra_options，也未指定 --options，那么该库将为具有对应标准选项的标准库。如果指定了 --extra_options 或 --options，那么该库为具有自定义选项的自定义库。除非使用了 --all，否则为必备选项。
--all	一次性构建所有标准库。
--install_to= *directory*	要将库写入的目录。对于标准库，这个默认为与索引库 (libc.a) 相同的目录。对于自定义库，这个选项为必备选项。
--compiler_bin_dir= *directory*	编译器可执行文件所在的目录。直接调用 mklib 时，可执行文件应位于路径中，但如果不在那里，则必须使用这个选项来告知 mklib 这些文件的位置。这个选项主要是在链接器调用 mklib 时使用。
--name= *filename*	库的文件名且没有目录部分。仅用于自定义库。
--options=' *str* **'**	构建库时使用的选项。默认选项（见下文）会由此字符串所*取代*。如果使用此选项，则库将为自定义库。
--extra_options=' *str* **'**	构建库时使用的选项。也会使用默认选项（见下文）。如果使用此选项，则库将为自定义库。
--list_libraries	列出此脚本能够构建的库并退出。普通系统特有目录。
--log= *filename*	将构建日志另存为 *filename*。
--tmpdir= *directory*	使用 *directory* 作为暂存空间，而不是普通系统特有目录。
--gmake= *filename*	要调用的兼容 Gmake 的程序，而不是“gmake”
--parallel= *N*	一次性编译 *N* 个文件 ("gmake -j N")。
--query= *filename*	此脚本是否知道如何构建 FILENAME？
--help 或 **--h**	显示此帮助。
--quiet 或 **--q**	以静默方式运行。
--verbose 或 **--v**	用于调试此可执行文件的额外信息。

示例：

构建所有标准库并将它们放入编译器的库目录：

```
mklib --all --index=$C_DIR/lib
```

构建一个标准库并将其放入编译器的库目录：

```
mklib --pattern=rts2800_ml.lib --index=$C_DIR/lib
```

构建类似 rts2800_ml.lib 的自定义库，但启用符号调试支持：

```
mklib --pattern=rts2800_ml.lib --extra_options="-g" --index=$C_DIR/lib --install_to=$Project/Debug
--name=rts2800_debug.lib
```

8.5.3 扩展 mklib

mklib API 是一种统一接口，让 Code Composer Studio 无需知道用于构建库的确切底层机制，就能构建库。每个库供应商（例如 TI 编译器）均会在库目录中提供一个可供调用的库专用“mklib”副本，该副本了解标准化选项集以及如何构建库。这样一来，只要供应商支持 mklib，链接器便能够自动构建任何供应商库的应用程序兼容版本，而无需事先注册对应的库。

8.5.3.1 底层机制

底层机制可以是供应商想要的任何内容。对于编译器运行时支持库，mklib 只是一个包装程序，此包装程序知道如何使用编译器安装目录中 lib/src 子目录内的文件和使用适当的选项调用 gmake 来构建每个库。如有必要，可以绕过 mklib 并直接使用 Makefile，但 TI 不支持这种运行模式，您需要自行对 Makefile 进行任何更改。Makefile 的格式以及 mklib 与 Makefile 之间的接口如有更改，恕不另行通知。mklib 程序是向前兼容的路径。

8.5.3.2 来自其他供应商的库

如果供应商想要分发可由链接器自动重建的库，则必须提供：

- 索引库（类似于“libc.a”，但具有不同的名称）
- 特定于该库的 mklib 副本
- 库源代码副本（任意方便使用的格式）

这些内容必须一起放在属于链接器库搜索路径（在 C2000_C_DIR 中或通过链接器 --search_path 选项指定）的同一目录中。

如果 mklib 需要无法作为命令行选项传递到编译器的额外信息，则供应商将需要提供一些其他的信息发现方式（例如由从内部 CCS 运行的向导写入的配置文件）。

供应商提供的 mklib 必须至少接受表 8-2 中列出的所有选项而不出现错误，即使这些选项不发挥任何作用也是如此。

章节 9

C++ 名称还原器

C++ 编译器通过在函数的链接级名称中对函数的原型和命名空间进行编码来实现函数重载、运算符重载和类型安全链接。将原型编码为链接名称的过程通常称为“名称改编”。当检查已改编的名称 (例如在汇编文件、反汇编器输出或者编译器或链接器诊断消息中) 时，很难将已改编的名称与其在 C++ 源代码中的相应名称关联起来。C++ 名称还原器是一种调试辅助工具，其将检测到的每个已改编的名称转换为其在 C++ 源代码中找到的原始名称。

这些主题将介绍如何调用和使用 C++ 名称还原器。C++ 名称还原器读取输入，查找已改编的名称。所有未改编的文本都将原封不动复制到输出中。在复制到输出之前，所有已改编的名称都会被还原。

9.1 调用 C++ 名称还原器

调用 C++ 名称还原器的语法如下：

dem2000 [*options*] [*filenames*]

dem2000	调用 C++ 名称还原器的命令。
options	影响名称还原器行为的选项。可以出现在命令行任何位置上的选项。
filenames	文本输入文件，例如编译器输出的汇编文件、汇编器列表文件、反汇编文件和链接器映射文件。如果命令行上没有指定文件名，则 dem2000 使用标准输入。

默认情况下，C++ 名称还原器输出到标准输出。如果要输出到文件，可以使用 -o 文件选项。

以下选项仅适用于 C++ 名称还原器：

--abi={eabi|coffabi}	默认情况下 EABI 标识符的还原是打开的。
--debug (--d)	打印调试消息。
--diag_wrap[=on,off]	将诊断消息设置为在 79 列换行（on，这是默认值）或不换行 (off)。
--help (-h)	打印帮助屏幕，该帮助屏幕提供 C++ 名称还原器选项的在线汇总。
--output= *file* (-o)	输出到指定的文件而不是标准输出。
--quiet (-q)	减少执行期间生成的消息数量。
-u	指定外部名称没有 C++ 前缀。（已弃用）

9.2 C++ 名称还原器的示例用法

本节中的示例说明名称还原过程。

该示例显示了示例 C++ 程序。在该示例中，所有函数的链接名称都已改编；也就是说，函数的签名信息已编码到函数的名称中。

```
int compute(int val, int *err);
int foo(int val, int *err)
{
   static int last_err = 0;
   int result = 0
   if (last_err == 0)
   result = compute(val, &last_err);
   *err = last_err;
   return result;
}
```

编译器输出的汇编代码结果如下。

```
;****************************************************************
;* FNAME: _foo_FiPi                     FR SIZE:   4            *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;*                            0 Parameter,  3 Auto,  0 SOE      *
;****************************************************************
_foo_FiPi:
        ADDB        SP,#4
        MOVZ        DP,#_last_err$1
        MOV         *-SP[1],AL
        MOV         AL,@_last_err$1
        MOV         *-SP[2],AR4
        MOV         *-SP[3],#0
        BF          L1,NEQ
        ; branch occurs
        MOVL        XAR4,#_last_err$1
        MOV         AL,*-SP[1]
        LCR         #_compute__FiPi
        ; call occurs [#_compute__FiPi]
        MOV         *-SP[3],AL
L1:
        MOVZ        AR6,*-SP[2]
        MOV         *+XAR6[0],*(0:_last_err$1)
        MOV         AL,*-SP[3]
        SUBB        SP,#4
        LRETR
        ; return occurs
```

执行 C++ 名称还原器将会还原其认为已改编的所有名称。输入：

```
dem2000 foo.asm
```

运行 C++ 名称还原器后的结果如下所示。foo( ) 和 compute( ) 中的链接名称已还原。

```
;***************************************************************
;* FNAME: foo(int, int *)                FR SIZE: 4            *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  3 Auto,  0 SOE     *
;***************************************************************
foo(int, int *):
       ADDB       SP,#4
       MOVZ       DP,#_last_err$1
       MOV        *-SP[1],AL
       MOV        AL,@_last_err$1
       MOV        *-SP[2],AR4
       MOV        *-SP[3],#0
       BF         L1,NEQ
       ; branch occurs
       MOVL       XAR4,#_last_err$1
       MOV        AL,*-SP[1]
       LCR        #compute(int, int *)
       ; call occurs [#compute(int, int *)]
       MOV        *-SP[3],AL
L1:
       MOVZ       AR6,*-SP[2]
       MOV        *+XAR6[0],*(0:_last_err$1)
       MOV        AL,*-SP[3]
       SUBB       SP,#4
       LRETR
       ; return occurs
```

章节 10

CLA 编译器

TMS320C28x 软件开发工具集还包括对编译控制律加速器 (CLA) C 代码的支持。由于 CLA 架构和编程环境的限制，C 语言支持存在一些限制，详见节 10.2.4。

与包含 CLA 汇编时一样，编译后的 CLA 代码与 C28x 代码链接在一起，以创建单个可执行文件。

10.1 如何调用 CLA 编译器

还可以使用 cl2000 命令调用控制律加速器 (CLA) 编译器。编译器将具有 .cla 扩展名的文件识别为 CLA C 文件。shell 调用独立的 CLA 版本的编译器通道，以生成 CLA 专用代码。还需要 --cla_support 选项来汇编 CLA 代码生成器的输出。

如果使用 --cla_default 选项，扩展名为 .c 的文件也会被编译为 CLA 文件。

支持类型 0、类型 1 和类型 2 CLA。

然后，编译所生成的目标文件可以与其他 C28x 目标文件链接，以创建一个组合的 C28x/CLA 程序。

如需调用 CLA 编译器，请输入：

cl2000 --cla_support=[cla0|cla1|cla2] [*other options*] *file* **.cla**

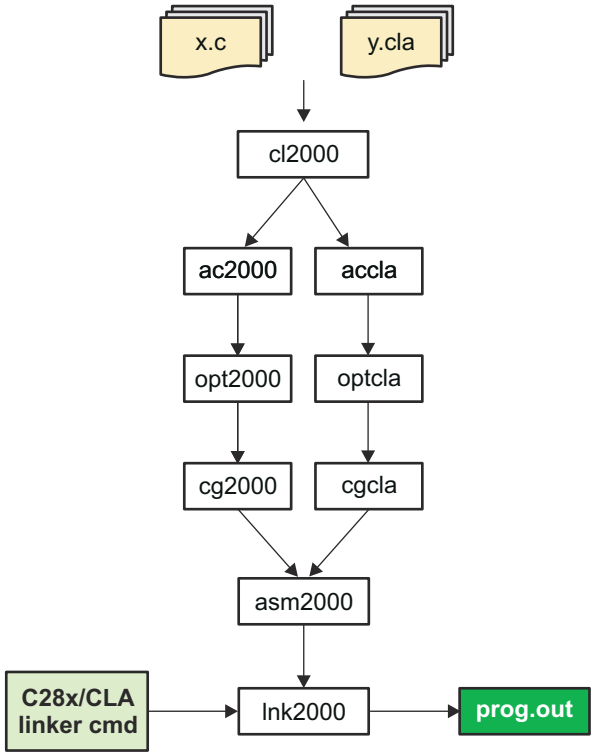


图 10-1. CLA 编译概述

关于 CLA C 文件的重要事实：

- 扩展名为 .cla 的文件使用不同的解析器、优化器和代码生成器进行编译。
- cl2000 不支持在单个调用中同时使用 C 和 CLA 编译器来编译文件。
- C28X/CLA 编译过程要求更改链接器命令文件，以容纳 CLA 编译器生成的各个段和暂存区。

10.1.1 CLA 特定的选项

除了 --cla_support 命令行选项外，以下命令行选项也专门适用于 CLA 编译器：

--cla_background_task={on|off}

启用或禁用对标识为后台任务的中断服务例程的支持。默认设置为关闭。当 --cla_background_task=on 时，系统会为非后台 ISR 保存和恢复 MR0、MR1、MR2、MAR0 和 MSTF 寄存器。有关创建后台任务的信息，请参阅节 6.9.15、节 10.2.2 和节 10.2.4。

如果后台任务被禁用（默认）并且：

- **任何 ISR 都标记为后台任务：**系统会提供 CLA 编译器错误。
- **没有 ISR 标记为后台任务：**不会向任何 ISR 添加寄存器保存/恢复。

如果启用了后台任务并且：

- **任何 ISR 都标记为后台任务：**为非后台任务的 ISR 添加了寄存器保存/恢复。
- **没有 ISR 标记为后台任务：**系统会为所有 ISR 添加寄存器保存/恢复。这增加了处理 ISR 的开销，通常没有必要。

由于只有 CLA2 支持后台任务，因此，如果 --cla_support 设置为非 cla2 值，那么在尝试使用 --cla_background_task=on 创建后台任务或启用后台任务时，系统会出现错误。

--cla_default

致使扩展名为 .c 的文件也被编译为 CLA 文件。

--cla_signed_compare_workaround={on|off}

允许对影响整数比较的 CLA 硬件缺陷自动使用权变措施。这是必要的，因为某些类型的整数比较可能会由于整数溢出而生成不正确的结果，例如，当比较的值具有相反的符号并且接近极值时。默认情况下，此选项处于禁用状态。

如果启用此选项，内部会使用浮点比较来检查正被比较的整数值的高位。如果所比较的值之间的差异过大，该比较会检测是否可能发生整数溢出。如果整数比较的结果不正确，则执行浮点比较。对于类似 `if (x < y)` 的比较，所执行的经修改的比较如下所示：

```
(float)x < (float)y || (float)x == (float)y && (x <= y)
```

备注

如果代码执行许多 32 位整数比较，启用此选项会增加代码大小和延长执行时间。

以下类型的整数比较始终是安全的，不会出现整数溢出。即使启用了此选项，也不会对以下类型的比较使用权变措施。

- 与零的比较
- 两个短整数之间的比较。

请注意，无符号整数之间的比较仍然会出现不正确的结果，因为比较是根据有符号整数在内部执行的。

如果您由于权变措施对代码大小和执行时间的影响而不想使用它，则可以使用以下任何手动编码替代方案：

- 使用表 10-2 中描述的 __mlt、__mleq、__mgt、__mgeq、__mltu、__mlequ、__mgtu 和 __mgequ 内在函数来确定已知的安全比较。这些内在函数向 CLA 编译器表明，正在比较的值不会导致溢出。例如，可以使用以下代码执行此类比较：

```
    if (__mlt(x, y))
```

- 如果您知道所比较的整数值始终为短整型，则将这些整数转换成短整型。例如：

```
if ((short)x < (short)y)
for (short i = 0; i < (short)y); i++)
```

- 如果您知道要比较的整数值采用浮点不会损失精度，则将这些整数转换为浮点。例如：

```
if ((float)x < (float)y)
for (float i = 0; i < (float)y); i++)
```

- 因为与零的比较不会溢出，所以可以将循环重写为“减计数器”。例如：

```
for (int i = y-1; i >= 0; i--)
```

10.2 CLA C 语言实现

CLA 实现只支持 C 语言。如果编译 C++ ，此编译器会生成错误。

CLA C 语言实现不支持 C 标准库。

CLA C 语言实现要求对标准 C 进行更改。下面的各小节将介绍相关内容。

10.2.1 变量和数据类型

备注

CLA 与 C28x CPU 之间共享的所有数据都必须在 C28x C 或 C++ 代码中定义，而不是在 CLA 代码中定义（即，不在 *.cla 文件中）。这是必需的，因为 CLA 代码中定义的共享变量可能未被正确阻断，这可能导致 C28x 上的变量访问无法正确地设置 DP。有关 COFF 和 EABI 的阻断和 DP 寄存器的信息，请参阅节 3.11。如果尝试在未阻断的数据上进行阻断的数据访问，链接器则提供诊断消息。

支持下述数据类型：

表 10-1. CLA 编译器数据类型

类型	EABI 大小（位）	COFF 大小（位）
char	16	16
short	16	16
int	32	32
long	32	32
long long	64	32
float	32	32
double	64	32
long double	64	32
指针	16	16

char/short 应限于加载/存储操作。

CLA 的指针大小始终为 16 位。这与具有 32 位指针的 C28x 不同。

备注

CLA 的 int 具有不同大小：CLA 的 int 的大小是 32 位，而不是 C28x 上的 16 位。为了避免在 CLA 和 C28x 之间共享数据时出现歧义，我们强烈建议您使用包含大小信息的 C99 类型声明（例如，int32_t 和 uint16_t）。

备注

使用 CLA 编译器和 COFF 时，不支持 64 位类型：使用 COFF ABI 时，CLA 编译器不支持 64 位类型。

10.2.2 Pragma、关键字和内在函数

CLA 接受除 FAST_FUNC_CALL 之外的 C28x pragma。

无法识别 far 和 ioport 关键字。

对 `'MMOV32 MSTF,mem32'` 和 `'MMOV32 mem32,MSTF'` 指令的访问是使用 cregister 关键字提供的。如需访问这些 MSTF 指令，请包含以下声明：

```
extern cregister volatile unsigned int MSTF;
```

支持表 10-2 中列出的内在函数。此外，运行时库函数 abs() 和 fabs() 是作为内在函数实现的。

表 10-2. 用于 CLA 的 C/C++ 编译器内在函数

内在函数	汇编指令	说明
uint32_t **__f32_bits_as_u32(** float *src* **);**	--	将浮点型中的位提取为 32 位寄存器。此内在函数不生成代码；该函数告诉编译器要改变解释位的方式。示例请参阅节 7.6.1。
uint64_t **__f64_bits_as_u64(** double *src* **);**	--	将双精度型中的位提取为 64 位寄存器。此内在函数不生成代码；该函数告诉编译器要改变解释位的方式。示例请参阅节 7.6.1。
__mdebugstop()	**MDEBUGSTOP**	调试停止
__meallow()	**MEALLOW**	启用对 EALLOW 寄存器的写访问
__medis()	**MEDIS**	禁用对 EALLOW 寄存器的写访问
float **__meinvf32(**float *x* **);**	**MEINVF32** *x*	估算 1/*x*（精度约为 8 位）。
float **__meisqrtf32(**float *x* **);**	**MEISQRTF32** *x*	估算 1/*x* 的平方根（精度约为 8 位）。牛顿-拉弗森法用于估算平方根。
short **__mf32toi16r(**float *src* **);**	**MF32TOI16R** *dst* , *src*	将 double 转换为 int 并进行舍入。
unsigned short **__mf32toui16r(**float *src* **);**	**MF32TOUI16R** *dst* , *src*	将 double 转换为无符号 int 并进行舍入。
float **__mfracf32(float** *src* **);**	**MFRACF32** *dst* , *src*	返回 *src* 的分数部分。
short **__mgeq(** signed int *x* , signed int *y* **);**	**MCMP32** *dst* , *src*	如果 *x* 大于或等于 *y*，则返回 1。否则，返回 0。
short **__mgequ(** unsigned int *x* , unsigned int *y* **);**	**MCMP32** *dst* , *src*	如果 *x* 大于或等于 *y*，则返回 1。否则，返回 0。
short **__mgt(** signed int *x* , signed int *y* **);**	**MCMP32** *dst* , *src*	如果 *x* 大于 *y*，则返回 1。否则，返回 0。
short **__mgtu(** unsigned int *x* , unsigned int *y* **);**	**MCMP32** *dst* , *src*	如果 *x* 大于 *y*，则返回 1。否则，返回 0。
short **__mleq(** signed int *x* , signed int *y* **);**	**MCMP32** *dst* , *src*	如果 *x* 小于或等于 *y*，则返回 1。否则，返回 0。
short **__mlequ(** unsigned int *x* , unsigned int *y* **);**	**MCMP32** *dst* , *src*	如果 *x* 小于或等于 *y*，则返回 1。否则，返回 0。
short **__mlt(** signed int *x* , signed int *y* **);**	**MCMP32** *dst* , *src*	如果 *x* 小于 *y*，则返回 1。否则，返回 0。
short **__mltu(** unsigned int *x* , unsigned int *y* **);**	**MCMP32** *dst* , *src*	如果 *x* 小于 *y*，则返回 1。否则，返回 0。
float **__mmaxf32(** float *x* , float *y* **);**	**MMAXF32** *dst* , *src*	返回两个 32 位浮点值的最大值。如果 *src* > *dst*，则将 src 复制到 dst。
float **__mminf32(** float *x* , float *y* **);**	**MMINF32** *dst* , *src*	返回两个 32 位浮点值的最小值。如果 *src* < *dst*，则将 src 复制到 dst。
__mnop()	**MNOP**	CLA 无操作

表 10-2. 用于 CLA 的 C/C++ 编译器内在函数（续）

内在函数	汇编指令	说明
__msetflg(unsigned short *flag* **,** unsigned short *value* **)**	**MSETFLG** *flag* **,** *value*	设置/清除 MSTF 寄存器中的标志。*flag* 是一个位掩码，用于指示要修改的位。*value* 提供要分配给这些位的值。有关 MSETFLG 指令和 MSET 寄存器的详细信息，请参阅 *CLA 参考指南* (SPRUGE6)。 本例将 RNDF32 标志（位 7）设置为 0，将 TF 标志（位 6）设置为 0，将 NF 标志（位 2）设置为 1。 0b 前缀是 GCC 语言扩展，表示这些是二进制数。 `__msetflg(0b11000100, 0b00000100);`
void **__mswapf(**float &*a* **,** float &*b* **);**	**MSWAPF** *a* **,** *b*	交换 *a* 和 *b* 的内容。
float **__sqrt(**float *x* **);**	**MEISQRTF32** *x*	估算 1/*x* 的平方根（精度约为 8 位）。牛顿-拉弗森法用于估算平方根。这是 __meisqrtf32 内在函数的别名。
float **__u32_bits_as_f32(** uint32_t *src*);	--	将 32 位寄存器打包为浮点型。此内在函数不生成代码；该函数告诉编译器要改变解释位的方式。示例请参阅节 7.6.1。
double **__u64_bits_as_f64(** uint64_t *src*);	--	将 64 位寄存器打包为双精度型。此内在函数不生成代码；该函数告诉编译器要改变解释位的方式。示例请参阅节 7.6.1。

为 CLA 提供了比较内在函数（如 _mlt 和 _mleq），以避免在执行有符号整数比较时可能出现的溢出。

下述内在函数仅在 CLA2 后台任务中可用。后台任务的优先级最低，因此当另一个中断可用时，此任务始终会被中断。这些内在函数可用于临时防止此类中断暂停后台任务。

- __disable_interrupts();
- __enable_interrupts();

10.2.3 使用 CLA 编译器进行优化

CLA 编译器支持 --opt_level 命令行选项。有关更多信息，请参阅节 3.1。

使用 CLA 编译器时，--gen_func_subsections 编译器选项默认处于开启状态。（对于常规 C28x 编译器，默认情况下它是关闭的。）

CLA 编译器可以生成具有 MMOV32 指令的并行 MMACF32。使用 --opt_level=2 或更高版本并循环展开为具有 MMOV32 指令的并行 MMACF32 创造更多机会。有关启用循环展开的详细信息，请参阅节 6.9.24。例如，将为以下函数中的循环生成并行指令：

```
// result, buff[], and coef[] are float
interrupt void Cla1Task1 ( void )  {
      result = 0.0f;
      int16_t i;
      #pragma UNROLL(20)
      for(i = 20; i > 0; i-)  {
          buff[i] = buff[i-1];
           result += coef[i] * buff[i];
      }
     result += coef[0] * buff[0];
 }
```

10.2.4 C 语言限制

CLA 的 C 语言具有几个附加限制。

- 不支持定义和初始化全局/静态数据。

  由于 CLA 代码是在中断驱动的环境中执行的，因此没有 C 系统引导序列。因此，必须在程序执行期间，通过 C28x 驱动程序代码或在 CLA 函数内执行全局/静态数据初始化。

  可以对定义为常量的变量进行全局初始化。编译器创建名为 .const_cla 的初始化数据段来保留这些变量。
- CLA 代码无法调用 C28x 函数。如果针对 C28 编译的代码调用针对 CLA 编译的代码，或者针对 CLA 编译的代码调用针对 C28 编译的代码，则链接器会提供诊断消息。
- 不支持递归函数调用。
- 不支持使用函数指针。

CLA 编译器支持大多数 GCC 扩展。默认情况下，C 和 CLA 编译器都启用了 GCC 扩展。有关更多信息，请参阅节 6.14.3 中的 --relaxed_ansi 选项和节 6.15 中的 GCC 语言扩展列表。

只要不在中断属性中包含任何参数，节 6.9.15 中介绍的中断函数属性就可以与 CLA 中断结合使用。CLA 编译器也支持 INTERRUPT pragma。例如，支持以下用法：

```
__attribute__((interrupt))
void interrupt_name(void) {...}
#pragma INTERRUPT(interrupt_name);
void interrupt _name(void) {...}
```

对于 CLA2 后台任务，“BACKGROUND”参数指定这是后台任务，而不是常规中断。支持以下用法。

```
__attribute__((interrupt("BACKGROUND")))
 void task_name(void) {...}
#pragma INTERRUPT(task_name, "BACKGROUND");
void task_name(void) {...}
```

CLA 编译器不允许在后台任务中进行函数调用。允许使用内联函数，因此可以改用此类函数。

10.2.5 存储器模型 - 相应的段

未初始化的全局数据放在 .bss_cla 段中。

初始化的常量数据放在 .const_cla 段中。

由于不支持 malloc()，因此 CLA 没有 C 系统堆。

局部变量和编译器临时变量放入暂存区存储器区域中，此区域充当 CLA C 软件栈。暂存区存储器区域应在应用程序的链接器命令文件中定义和管理。

每个函数都有一个生成的函数帧（它是 .scratchpad 段的一部分），而不使用堆栈。因此，需要放在链接器命令文件中的唯一的段是 .scratchpad 段。为每个函数指定一个暂存区帧，用于保存本地数据、函数参数和临时存储。链接器确定在放置时可以覆盖哪些函数帧以节省存储器。

CLA 函数帧位于 .scratchpad 段中，以“.scratchpad:*functionSectionName*”的形式命名。每个函数都有自己的子段，因而具有唯一的段名称。例如，如果源级函数名为“Cla1Task1”，那么 COFF 函数名将为“_Cla1Task1”，函数段名称将为“Cla1Prog:_Cla1Task1”，函数的暂存区帧将命名为“.scratchpad:Cla1Prog:_Cla1Task1”。函数的暂存区帧将使用基本符号“__cla_Cla1Task1_sp”。

CLA2 后台任务放在 .scratchpad 段中，并以“.scratchpad:background:*functionSectionName*”的形式命名。后台任务框架不能与任何其他函数框架重叠，因为后台任务很可能在遇到中断后返回。

请注意，如果汇编编写器对函数的数据空间使用不同的命名约定，则不能覆盖它，也不能将它放在 .scratchpad 段中。

不需要指定 .scratchpad 段的大小。

使用 6.4 之前的编译器版本编译的 CLA 目标文件与新生成的目标文件兼容，但前提是链接器命令文件同时支持这两种暂存区命名约定。但是，用于旧目标文件的暂存区段不能与新的 .scratchpad 段重叠，必须确保这两个段都有足够大小的存储器。

10.2.6 函数结构和调用惯例

CLA 编译器支持多个嵌套级别的函数调用。CLA 编译器还支持调用具有两个以上参数的函数。

指针参数在 MAR0 和 MAR1 中传递。32 位值在 MR0、MR1 和 MR2 中传递。16 位值在 MR0、MR1 和 MR2 中传递。任何其他参数都在函数帧（函数本地暂存区空间）上传递，从偏移量 0 开始。

除 MR3 之外的所有寄存器在调用时保存。MR3 在输入时被保存。

当与 CLA 汇编语言模块交互时，使用这些调用约定与已编译的 CLA 代码交互。

附录 A
术语表

TEXAS INSTRUMENTS

绝对列表器 一种调试工具，允许创建包含绝对地址的汇编器列表。

别名消歧 一种决定两个指针表达式何时不能指向同一位置的技术，从而允许编译器自由地优化此类表达式。

别名使用 以多种方式访问单个对象的能力，例如当两个指针指向单个对象时。其会破坏优化，这是因为任何间接引用都可能引用任何其它对象。

分配 链接器计算输出段最终存储器地址的过程。

ANSI 美国国家标准协会；一个建立行业自愿遵循的标准的组织。

应用程序二进制接口 (ABI) 一项指定两个目标模块之间接口的标准。ABI 规定了如何调用函数以及如何将信息从一个程序组件传递到另一个程序组件。

存档库 由归档器将单独文件组合成单个文件的集合。

归档器 将多个单独文件集合成一个单个文件（称为存档库）的软件程序。借助归档器，可以添加、删除、提取或替换存档库的成员。

汇编器 根据包含汇编语言指令、指示和宏定义的源文件创建机器语言程序的软件程序。汇编器将绝对操作码替换为符号操作码，并将绝对地址或可重定位地址替换为符号地址。

赋值语句 用值来初始化变量的语句。

自动初始化 在程序开始执行之前，初始化全局 C 变量（包含在 .cinit 段中）的过程。

运行时的自动初始化 链接器在链接 C 代码时使用的自动初始化方法。在使用 --rom_model 链接选项调用链接器时，链接器会使用此方法。链接器将数据表的 .cinit 段加载到内存中，并在运行时初始化变量。

大端 一种寻址协议，字中的字节从左至右进行编号。字中较高的有效字节存放在低地址处。字节序视硬件而定，并在复位时确定。另请参阅 *小端*

块 一组在大括号内组合在一起并被视为实体的语句。

字节 根据 ANSI/ISO C，可容纳一个字符的最小可寻址单元。对于 TMS320C28x，一个字节的大小为 16 位，这也是一个字的大小。

C/C++ 编译器 一种将 C 源语句转换成汇编语言源语句的软件程序。

代码生成器	一种编译器工具，采用解析器和优化器生成的文件并生成汇编语言源文件。
COFF	通用目标文件格式；根据 AT&T 开发的标准配置的目标文件系统。这些文件在内存空间中是可重定位的。
命令文件	包含链接器或十六进制转换实用程序的选项、文件名、指令或命令的文件。
注释	用于记录或提高源文件可读性的源语句（或源语句的一部分）。不对注释进行编译、汇编或链接；不会影响对象文件。
编译器程序	一种实用工具，可以一步完成编辑、汇编和选择性链接操作。通过编译器（包括解析器、优化器和代码生成器）、汇编器和链接器，编译器可以运行一个或多个源代码模块。
配置内存	链接器指定用于分配的存储器。
常量	其值不能改变的类型。
交叉引用列表	由汇编器创建的输出文件，其中列出了定义的符号、定义符号的行、引用符号的行以及符号的最终值。
.data 段[.data section]	默认的目标文件段之一。.data 段是包含初始化数据的初始化段。可以使用 .data 指令将代码汇编到 .data 段中。
直接调用	一种函数调用，其中一个函数使用函数名称调用另一函数。
指令	用于控制软件工具操作和功能的专用命令（与用于控制器件操作的汇编语言指令相反）。
消歧	请参阅*别名消歧*
动态内存分配	几个函数（如 malloc，calloc 和 realloc）在运行时为变量动态分配内存所使用的技术。这是通过定义较大的内存池（堆）并使用函数分配堆中的内存来实现。
ELF	可执行和可链接格式；根据系统 V 应用程序二进制接口规范配置的目标文件系统。
仿真器	复制 TMS320C28x 运行的硬件开发系统。
入口点	目标存储器中的执行起点。
环境变量	由用户定义并分配给字符串的系统符号。环境变量通常包含在 Windows 批处理文件或 UNIX shell 脚本（例如 .cshrc 或 .profile）中。
收尾程序	函数中恢复堆栈并返回的代码部分。
可执行目标文件	在目标系统上下载并执行的可执行链接目标文件。
表达式	一个常量、一个符号或由算术运算符分隔的一系列常量和符号。
外部符号	一种在当前程序模块中使用但在其他程序模块中定义或声明的符号。
文件级优化	一种优化级别，编译程序使用其具有的有关整个文件的信息来优化代码（与程序级优化相反，编译程序使用其具有的有关整个程序的信息来优化代码）。

函数内联	在调用点为函数插入代码的过程。这节省了函数调用的开销，并允许优化器在周围代码的上下文中优化函数。
全局符号	一种在当前模块中定义并在另一模块中访问或者在当前模块中访问但在另一模块中定义的符号。
高级别语言调试	编译程序保留符号和高级别语言信息（如类型和函数定义）的能力，这样调试工具就可以使用此类信息。
间接调用	一种函数调用，其中一个函数通过给出被调用函数的地址来调用另一个函数。
加载时初始化	链接 C/C++ 代码时由链接器使用的自动初始化方法。在使用 --ram_model 链接选项调用时，链接器会使用此方法。此方法在加载时而不是运行时初始化变量。
初始化段	从目标文件中链接到可执行目标文件中的段。
输入段	从目标文件中链接到可执行目标文件中的段。
集成预处理器	与解析器合并的 C/C++ 预处理器，以允许更快的编译。也可以使用独立的预处理或已预处理的列表。
交叠特征	一种将原始 C/C++ 源语句作为注释插入到汇编器的汇编语言输出中的特征。C/C++ 语句被会插入到等效汇编指令的旁边。
内联函数	像函数一样使用的运算符，可生成在 C 中无法表达或者需要更多时间和精力才能编写代码的汇编语言代码。
ISO	国际标准化组织；一个由国家标准机构组成的全球联合会，其制定了行业自愿遵循的国际标准。
K&R C	Kernighan 和 Ritchie C，在 *C 程序设计语言* (K&R) 第一版中定义的事实标准。大多数为早期非 ISO C 编译器编写的 K&R C 程序应该无需修改即可正确编译和运行。
标签	从汇编器源语句第 1 列开始并与该语句的地址相对应的符号。标签是唯一可以从第 1 列开始的汇编器语句。
链接器	一种将目标文件组合成可执行目标文件的软件程序，该文件可分配到系统内存中并由器件执行。
列表文件	由汇编器创建的输出文件，其中列出源语句、源语句的行号以及源语句对段程序计数器 (SPC) 的影响。
小端	一种寻址协议，字中的字节从右至左进行编号。字中较高的有效字节存放在高地址处。字节序视硬件而定，并在复位时确定。另请参阅*大端字节序*
加载器	一种将可执行目标文件放入系统内存的器件。
宏	可用作指令的用户定义例程。
宏调用	调用宏的过程。
宏定义	定义组成宏的名称和代码的源语句块。

宏扩展	在代码中插入源语句以代替宏调用的过程。
映射文件	由链接器创建的输出文件，其中显示内存配置、段组成、段分配、符号定义以及为程序定义符号的地址。
内存映射	被划分为功能块的目标系统内存空间的映射。
名称改编	编译器专用特征，其使用有关函数参数返回类型的信息对函数名称进行编码。
目标文件	包含机器语言目标代码的汇编或链接文件。
对象库	由单个目标文件组成的存档库。
操作数	汇编语言指令、汇编器指令或宏指令的参数，为由指令或指示执行的操作提供信息。
优化器	可提高执行速度并减小 C 程序大小的软件工具。
选项	允许您在调用软件工具时请求附加或特定函数的命令行参数。
输出段	可执行的已链接模块中的最终分配段。
覆盖页	一段物理存储器，其被映射到与另一段存储器相同的地址范围内。硬件开关决定哪个范围处于激活状态。
解析器	一种读取源文件、执行预处理函数、检查语法，以及生成中间文件以用作优化器或代码生成器的输入的软件工具。
分区	为每条指令分配数据路径的过程。
pop	从堆栈中检索数据对象的操作。
pragma	一种指示编译器如何处理特殊语句的预处理器指令。
预处理器	一种解释宏定义、扩展宏、解释头文件、解释有条件编译以及对预处理器指令起作用的软件工具。
程序级优化	一种将所有源文件编译成一个中间文件的积极的优化级别。由于编译器可以看到整个程序，因此在程序级优化中执行了一些很少在文件级优化中应用的优化。
序言	函数中设置堆栈的代码部分。
推入	将数据对象放在堆栈上以进行临时存储的操作。
无声运行	用于抑制正常横幅和进度信息的选项。
原始数据	输出段中的可执行代码或初始化数据。
重定位	一种当符号的地址改变时由链接器调整对符号的所有引用的过程。
运行时环境	程序必须在其中运行的运行时参数。这些参数由内存和寄存器约定、堆栈组织、函数调用约定及系统初始化定义。

运行时支持函数	标准的 ISO 函数，执行不属于 C 语言的任务（比如内存分配、字符串转换和字符串搜索等）。
运行时支持库	库文件 rts.src，其包含运行时支持函数的源代码。
段	一个可重定位的代码块或数据块，最终将与内存映射中的其他段接续。
符号扩展	用值的符号位来填充该值未使用的 MSB 的过程。
源文件	一种包含 C/C++ 代码或汇编语言代码的文件，该代码经编译或汇编后形成目标文件。
独立预处理器	一种将宏、#include 文件和条件编译扩展为独立程序的软件工具。其还执行集成预处理，包括解析指令。
静态变量	范围局限在一个函数或程序内的一种变量。当函数或程序退出时，静态变量的值不会被丢弃；当重新输入函数或程序时，将恢复其之前的值。
存储类	符号表中指示如何访问符号的条目。
字符串表	存储长度超过八个字符的符号名称的表（长度为八个字符或更长的符号名称不能存储在符号表中，而是存储在字符串表中）。符号入口点的名称部分指向字符串表中字符串的位置。
结构	一个或者多个变量组合在单个名称下的集合。
子段	一个可重定址的代码块或数据块，最终将占用存储器映射中的连续空间。子段为较大段中的小段。子段使用户能够更严格地控制存储器映射。
符号	表示地址或值的字母数字字符串。
符号调试	软件工具的能力，用于保留可供仿真器或模拟器等调试工具使用的符号信息。
目标系统	执行其上开发了目标代码的系统。
.text 段	默认的目标文件段之一。.text 段被初始化并包含可执行代码。可以使用 .text 指令将代码汇编到 .text 段中。
三字符序列	具有某种含义的 3 字符序列（由 ISO 646-1983 不变代码集定义）。这些字符不能在 C 字符集中表示，而是扩展为一个字符。例如，三个字符 ??' 扩展为 ^。
未配置的内存	未定义为存储器映射的一部分，且无法加载代码或数据的存储器。
未初始化段	在存储器映射中保留空间但没有实际内容的目标文件段。
无符号值	无论实际符号如何都会被当作非负数的值。
变量	表示可以假设一组值中的任何一个数的符号。
字	目标内存中的 16 位可寻址位置。

TEXAS INSTRUMENTS

Changes from JUNE 3, 2022 to OCTOBER 31, 2023 (from Revision Y (June 2022) to Revision Z (October 2023)) **Page**

- 添加了 --cla_background_task 选项的文档 21
- 添加了有关使用 --unified_memory 实现易失性和非易失性符号之间结构体赋值的信息 32
- 添加了有关使用实时固件更新 (LFU) 的限制的信息 54
- 更正了 .econst 段内容的说明 90
- 添加了有关使用 --cla_background_task 选项的信息。 124
- 删除了一条不正确的语句，该语句声明 .econst 段包含所有全局变量 148
- 添加了 COFF 位字段布局的详细信息和示例 153
- 阐明了 __byte 内在函数结果的高 8 位内容。 164
- 添加了 --cla_background_task 选项的文档。 221

Changes from DECEMBER 23, 2021 to JUNE 3, 2022 (from Revision X (December 2021) to Revision Y (June 2022)) **Page**

- --strict_compatibility 链接器选项不再起任何作用，已从文档中删除。 26
- 添加了在 --fp_mode=relaxed 时优化 fmodf() 的说明。 31
- 添加了 --fp_single_precision_constant 编译器选项。 31
- 更正了出现在其中的 --gen_cross_reference_listing 和 --asm_cross_reference_listing 选项的名称。 46
- 添加了 __fmodf 内在函数，以实现更高效的浮点余数计算。 170
- 添加了有关 CLA 编译器可能实现的优化的部分。 225

Changes from JUNE 16, 2021 to DECEMBER 22, 2021 (from Revision W (June 2021) to Revision X (December 2021)) **Page**

- 更正了有关 .switch 段限制的文档 90
- 删除了一条不正确的语句，该语句声明 .econst 段包含 switch 表。 148
- 删除了不受支持的 __fmin64 和 __fmax64 内在函数的文档。请注意，正确的 FPU64 指令是自动生成的。 169

Changes from DECEMBER 16, 2020 to JUNE 15, 2021 (from Revision V (December 2020) to Revision W (June 2021)) **Page**

- 阐述如何判断器件是否支持快速整数除法 32
- SET_DATA_SECTION pragma 优先于 --gen_data_subsections=on 选项。 86
- SET_DATA_SECTION pragma 优先于 --gen_data_subsections=on 选项。 131
- 更正了在 EABI 模式下初始化全局或静态变量的链接器语法。 136
- 记录了 __f32_bits_as_u32、__f64_bits_as_u64、__u32_bits_as_f32 和 __u64_bits_as_f64 内在函数。 164
- 记录调用 __IQsat 时 max 不得小于 min。 164
- 记录了 __f32_bits_as_u32、__f64_bits_as_u64、__u32_bits_as_f32 和 __u64_bits_as_f64 内在函数。 169
- 在 FPU 内在函数表的一行内使用一致的参数名称。 169
- 记录了调用 __fsat 时 max 不得小于 min。 169
- 更正了 __swapf() 和 __swapff() 内在函数的信息和说明 169

- 更正了 __atan2puf32 固有函数所示语法中的操作数顺序。 170
- 如果使用了 --float_support=fpu32/fpu64 选项，调用中断例程后将始终保存/恢复 STF 寄存器。 177
- 阐明了有关字符串处理函数的信息。 193
- 添加了关于时间和时钟 RTS 函数的信息。 193
- 记录了 __f32_bits_as_u32、__f64_bits_as_u64、__u32_bits_as_f32 和 __u64_bits_as_f64 内在函数。 .. 223

Changes from SEPTEMBER 1, 2020 to DECEMBER 15, 2020 (from Revision U (September 2020) to Revision V (December 2020)) **Page**

- 添加了 --lfu_reference_elf 和 --lfu_default 命令行选项 21
- 不再支持 MISRA-C 检查 21
- 更正了当 --fp_mode=relaxed 时如何转换双精度值的说明 31
- 不再支持 MISRA-C 检查 31
- 添加了实时固件更新 (LFU) 功能 54
- 阐明了 --opt_level=4 必须位于 --run_linker 选项之前 62
- 有文件证明不支持 C11 原子操作。 100
- 不再支持 MISRA-C 检查的 pragma 116
- 有文件证明不支持 C11 原子操作。 137
- 增加了符号的保留和更新属性，以便控制 LFU 行为。 142
- 新增了实时固件更新 (LFU) 使用的各段 148
- 更正了 PM 状态寄存器的假定值文档。 156
- 添加了关于 __TI_auto_init_warm() RTS 函数的信息。 210
- 记录了 EABI 的 CLA 编译器类型。 222

Changes from FEBRUARY 28, 2020 to AUGUST 31, 2020 (from Revision T (February 2020) to Revision U (August 2020)) **Page**

- 更新了整个文档中的表格、图和交叉参考的编号格式。 11
- 删除了整个文档中对处理器 Wiki 的引用。 11
- 添加了有关 --gen_func_subsections 选项的默认值的信息 86
- 更正了有关 --gen_data_subsections 选项的默认值的信息。 86
- 更新了有关枚举类型大小的信息。 106
- 阐明 --opt_level 和 FUNCTION_OPTIONS pragma 之间的交互。 124
- 为与 MUST_ITERATE pragma 对应的属性新增了 C++ 属性语法。 126
- 添加了与 UNROLL pragma 对应的各个属性的 C++ 属性语法。 132
- 增加了使用位置属性的示例，和基于内存位置进行优化的信息。 142

下表列出了更改文档编号格式前对此文档做出的改动。左列标识了本文档出现该特定改动的首个版本。

早期修订版本

添加内容的版本	章节	位置	添加/修改/删除
SPRU514T	使用编译器	节 2.3.4	通过使用带有寄存器操作数的 RPT，内联的 memcpy 调用现在支持 255 以上个字。因此，可支持内联最多 65535 个字的 memcpy。但是，使用 --rpt_threshold 指定的最大值仍然是 256。
SPRU514T	使用编译器，运行时环境	节 2.3.4 和节 7.6	阐明了 TMU1 支持仅适用于 EABI。
SPRU514T	使用编译器，CLA 编译器	节 2.3、节 10.1.1 和节 10.2.2	为 CLA 编译器添加了 --cla_signed_compare_workaround 选项。为 CLA 编译器添加了比较内在函数。
SPRU514T	链接	节 4.3.5	阐明了如果只有链接器在运行，则需要 --rom_model 或 --ram_model，但如果编译器在同一命令行中的 C/C++ 文件上运行，则 --rom_model 是默认选项。
SPRU514T	C/C++ 语言	节 6.9.21	`#pragma once` 现记录在头文件中使用。

早期修订版本（续）

添加内容的版本	章节	位置	添加/修改/删除
SPRU514T	C/C++ 语言	节 6.15.4、节 10.2.1 和节 10.2.4	链接器现在提供有关阻止 CLA 和 C28 代码之间访问和交互的某些问题的诊断信息。
SPRU514T	运行时环境	节 7.6.3	更正了与 __atan、__cos 和 __sin TMU 内在函数等效的汇编指令。
SPRU514T	运行时环境	节 7.6.4	阐明了快速整数除法需要 EABI 和 FPU32 或 FPU64。更改了将 64 位值除以 32 位值的快速整数除法内在函数返回的类型。此外，优化了由这些内在函数执行的汇编。
SPRU514T	运行时环境	节 7.10.4.1	阐明了只有在使用 --rom_model 链接器选项时，才发生零初始化，使用 --ram_model 选项时则不发生。
SPRU514S		-- 全文 --	更改了由编译器创建的目标文件的默认文件扩展名，以防止在 C 和 C++ 文件具有相同名称时发生冲突。从 C 源文件生成的目标文件具有 .c.obj 扩展名。从 C++ 源文件生成的目标文件具有 .cpp.obj 扩展名。
SPRU514S	使用编译器，优化	节 2.3 和节 3.14	添加了 --isr_save_vcu_regs 编译器选项。
SPRU514S	CLA 编译器	节 10.2.1 和节 10.2.4	添加了有关 CLA 和 C28x 代码共享变量的声明以及 C28x 和 CLA 代码之间的函数调用的信息。
SPRU514R.1	使用编译器，运行时环境	节 2.3.4、节 7.6.4 和节 7.8.2	添加了有关内置整数除法和模运算符 (/ 和 %) 以及内在函数的快速整数除法的信息。
SPRU514R.1	运行时环境	节 7.1.7	Clarify bit-field alignment rules.
SPRU514R.1	运行时环境	节 7.6.2	添加了特定于 FPU64 的内在函数以及有关 FPU64 的 COFF 与 EABI 使用的信息。
SPRU514R.1	CLA 编译器	节 10.2	阐明了对整数型使用的建议。
SPRU514R		-- 全文 --	添加了有关 EABI 支持的更多文档。其中包括标记仅支持 COFF 或 EABI 的功能。在必要时将示例标记为 COFF 特定示例。
SPRU514R	使用编译器	节 2.3	添加了 EABI 的命令行选项，包括 --cinit_compression、--copy_compression、--extern_c_can_throw、--retain、--unused_section_elimination 和 --zero_init。
SPRU514R	使用编译器	节 2.3.1	添加了 --emit_references:file 链接器选项。
SPRU514R	使用编译器	节 2.3.4	添加了 --silicon_errata_fpu1_workaround 选项。
SPRU514R	使用编译器	节 2.5.1	记录了支持 C 标准宏命令，例如 __STDC_VERSION__。
SPRU514R	使用编译器	节 2.11.1	添加了有关可能未内联内在函数的情况的信息。
SPRU514R	使用编译器，C/C++ 语言	节 2.13、节 6.11	添加了有关受支持的应用二进制接口 (ABI) 的信息。
SPRU514R	链接 C/C++ 代码，运行时环境	节 4.3.6、节 7.1.1	添加了 EABI 特定段，例如 .bss、.const、.sysmem 和 .init_array。还添加了 .args、.ppdata 和 .ppinfo 段。
SPRU514R	C/C++ 语言	节 6.3	添加了有关 double 和 wchar_t 数据类型的 EABI 特定信息。
SPRU514R	C/C++ 语言	节 6.9	添加了 LOCATION、NOINIT、PERSISTENT 和 WEAK pragma。
SPRU514R	C/C++ 语言	节 6.12	添加了有关命名空间的 EABI 特定信息。
SPRU514R	C/C++ 语言	节 6.14.1	更新了 C99 不受支持的运行时函数列表。
SPRU514R	C/C++ 语言	节 6.15.2	添加了 aligned、calls、naked 和 weak 函数属性的文档。
SPRU514R	C/C++ 语言	节 6.15.4	添加了 location、noinit、persistent 和 weak 变量属性的文档。
SPRU514R	运行时环境	节 7.2.1	添加了 FPU64 寄存器。
SPRU514R	运行时环境	节 7.6	更正了对由 TMU 内在函数 (例如 __sin() 和 __cos()) 返回的值的描述。
SPRU514R	运行时环境	节 7.10.4	添加了有关 EABI 变量自动初始化的信息。
SPRU514R	运行时支持函数	节 8.1.8	添加了特定于 EABI 的运行时库命名规则。
SPRU514R	运行时支持函数	DEV_lseek 主题	更正了 DEV_lseek 函数的语法记录。
SPRU514Q	简介，使用编译器，C/C++ 语言	节 1.3、节 2.3、节 6.1 和节 6.14.2	添加了对 C11 的支持。

早期修订版本（续）

添加内容的版本	章节	位置	添加/修改/删除
SPRU514Q	使用编译器	节 2.3.4	添加了对 EABI 的支持。COFF ABI 是默认值。
SPRU514Q	使用编译器	节 2.3.1	添加了 --ecc=on 链接器选项，支持生成 ECC。请注意，ECC 生成功能现在默认关闭。
SPRU514Q	使用编译器	节 2.3.4 和节 7.6	通过 --float_support=fpu64 添加了对 64 类 FPU 的支持。
SPRU514Q	使用编译器，运行时环境	节 2.3.4 和节 7.6	通过 --idiv_support 添加了对快速整数除法的支持。
SPRU514Q	使用编译器，C/C++ 语言	节 2.3.4 和节 7.6	通过 --tmu_support=tmu1 添加了对 TMU 的进一步支持。
SPRU514Q	使用编译器	节 2.3.4	通过 --vcu_support=vcrc 添加了对循环冗余校验的支持。
SPRU514Q	使用编译器	节 2.5.1	__TI_STRICT_ANSI_MODE__ 和 __TI_STRICT_FP_MODE__ 宏命令定义为在条件为 false 时为 0。
SPRU514Q	使用编译器，C/C++ 语言	节 2.11 和节 6.9	修订了有关内联函数扩展的段及子段，以包括新的 pragma 并更改了编译器关于内联哪些函数的决策。添加了 FORCEINLINE、FORCEINLINE_RECURSIVE 和 NOINLINE pragma。
SPRU514Q	优化，C/C++ 语言	节 3.11 和节 6.15.4	添加了 blocked 和 noblocked 属性以实现更好的数据页 (DP) 指针加载优化。不再推荐使用 --disable_dp_load_opt 选项。
SPRU514Q	C/C++ 语言	节 6.2	从例外列表中删除了几个 C++ 功能，因为有多个版本已支持这些功能。
SPRU514Q	C/C++ 语言	节 6.3.3 和节 7.6	添加了将 32 位浮点值声明为 float 而不是 double 的建议。（两者目前都是 32 位。）修改了内在函数语法描述以对 32 位值使用"float"。
SPRU514Q	C/C++ 语言	节 6.4	添加了有关字符集和文件编码的信息。
SPRU514Q	C/C++ 语言	节 6.15.2 和节 6.15.4	添加了 "retain" 作为函数属性和变量属性。
SPRU514Q	C/C++ 语言	节 6.15.6	阐明了 __builtin_sqrt() 和 __builtin_sqrtf() 函数的可用性。
SPRU514Q	CLA 编译器	节 10.2	更正了 __mswapf 内在函数的语法。
SPRU514P	C/C++ 语言	节 6.15	编译器现在支持多个 Clang __has_ 宏命令扩展。
SPRU514P	C/C++ 语言	节 6.15.1	现在支持包装器头文件 GCC 扩展 (#include_next)。
SPRU514O	C/C++ 语言	节 6.5.1	阐明了由常量关键字设置的常量数据存储的例外情况。
SPRU514N	优化	节 3.7.1.4	更正了处理配置文件数据的命令中的错误。
SPRU514M	使用编译器 C/C++ 语言和 CLA 编译器	节 2.3、节 2.3.4、节 2.5.1、节 6.9.15 和节 10.2	记录了对 CLA 版本 2 和 CLA v2 背景任务的支持。
SPRU514M	使用编译器，C/C++ 语言	节 2.3.3	修改为指明：即使使用 CHECK_MISRA pragma 也需要 --check_misra 选项。
SPRU514M	使用编译器	节 2.3.5	删除了不再受支持的 --symdebug:coff 选项。
SPRU514M	使用编译器	节 2.10	更正了文档以描述 ---gen_preprocessor_listing 选项。名称 --gen_parser_listing 不正确。
SPRU514M	优化	节 3.11	提供了有关数据页分块的信息。
SPRU514L	优化	节 3.7.3	更正了 _TI_start_pprof_collection() 和 _TI_stop_pprof_collection() 的函数名称。
SPRU514L	CLA 编译器	节 10.2	提供了有关 __msetflg 内在函数的附加信息和示例。
SPRU514K	使用编译器	--	几个编译器选项已被弃用、删除或重命名。编译器仍然接受一些已弃用的选项，但不建议使用它们。
SPRU514J	使用编译器	节 2.3 和节 4.2.2	添加了 --gen_data_subsections 选项。
SPRU514J	使用编译器	节 2.3.5	添加了 --symdebug:dwarf_version 编译器选项。此选项设置使用的 DWARF 调试格式版本。
SPRU514J	优化	节 3.7 和节 3.8	描述了反馈导向优化。该技术可用于代码覆盖分析。
SPRU514J	C/C++ 语言	节 6.9.1	添加了 CALLS pragma 以指定一组可从指定调用函数间接调用的函数。使用此 pragma 能够将此类间接调用包含在函数的 inclusive 栈大小的计算中。

早期修订版本（续）

添加内容的版本	章节	位置	添加/修改/删除
SPRU514J	C/C++ 语言	节 6.15.7	添加了一个 byte_peripheral 类型属性和一个内在函数来访问字节外围数据。
SPRU514J	运行时环境	节 7.6	添加了内在函数以执行无符号整数除法。新的内在函数是 __euclidean_div_i32byu32()、__rpt_subcul() 和 __subcul()。
SPRU514J	运行时环境	节 7.10.1	提供了额外的引导挂钩函数。这些可以定制以在系统初始化期间使用。
SPRU514I	使用编译器	表 2-7	添加了 --cla_default 选项。此选项会使扩展名为 .c 的文件作为 CLA 文件被处理。
SPRU514I	使用编译器	节 2.3.4	添加了 --ramfunc 选项。如果设置此选项，则将所有函数放在 RAM 中。
SPRU514I	使用编译器	--	弃用了 --no_fast_branch 选项。
SPRU514I	C/C++ 语言	节 6.14.1	现在支持 C99 数学函数，包括浮点数学函数的 float 和 long double 版本。
SPRU514I	C/C++ 语言	节 6.15.2	添加了 ramfunc 函数属性。它规定了一个函数应该放置在 RAM 中。
SPRU514I	运行时环境	节 7.3.2	在寄存器列表中添加了 XAR6 并更正了放置返回结构地址的位置。
SPRU514I	运行时环境	节 7.6	添加了 __eallow 和 __edis 内在函数。
SPRU514I	CLA 编译器	节 10.2.4	CLA 编译器现在支持大多数 GCC 扩展。
SPRU514H	引言	节 1.3	添加了对 C99 和 C++03 的支持。
SPRU514H	使用编译器	表 2-7	添加了对 C99 和 C++03 的支持。弃用了 -gcc 选项。--relaxed_ansi 现在是默认选项。
SPRU514H	使用编译器	节 2.3.3	添加了 --advice:performance 选项。
SPRU514H	使用编译器	节 2.3.4	不再支持 --silicon_version=27 选项。
SPRU514G	使用编译器	节 2.3.4	添加了 --tmu_support=tmu0 选项。此选项还会影响 --float_support 和 --fp_mode=relaxed 选项的行为。
SPRU514G	使用编译器	节 2.3.4	通过 --cla_support=cla1 添加了对 1 类 CLA 的支持。
SPRU514G	使用编译器	节 2.3.4	通过 --vcu_support=vcu2 添加了对 2 类 VCU 的支持。
SPRU514H	使用编译器	节 2.3.11	添加了有关 --flash_prefetch_warn 选项的信息。
SPRU514H	使用编译器	节 2.5.1	添加了几个未记录的预定义宏名称。
SPRU514H	使用编译器	节 2.5.3	记录了对 #warning 和 #warn 预处理器指令的支持。
SPRU514H	使用编译器	节 2.6	添加了有关向 main() 传递参数的技术的段。
SPRU514H	使用编译器	节 2.11	记录了 inline 关键字现在在除 C89 严格 ANSI 模式之外的所有模式中都启用。
SPRU514H	C/C++ 语言	节 6.3	C28x 上指针类型的大小现在是 32 位而不是 22 位。弃用了 near 和 far 关键字。不再支持小型存储器型号；唯一的存储器型号使用 32 位指针。不再使用 .bss、.const, 和 .sysmem 段；使用 .ebss、.econst 和 .esysmem 段。（出于性能原因，假设符号地址小于 22 位。）
SPRU514H	C/C++ 语言	节 6.1.1	添加了记录实现定义行为的段。
SPRU514H	C/C++ 语言	节 6.3.1	添加了有关枚举类型大小的文档。
SPRU514H	C/C++ 语言	节 6.9.15、节 6.9.22 和节 6.15.2	为 INTERRUPT 和 RETAIN pragma 添加了 C++ 语法。还从 #pragma 语法规范中删除了不必要的分号。现在还支持 GCC 中断和别名函数属性。
SPRU514H	C/C++ 语言	节 6.9.11 和节 6.9.12	添加了 FUNC_ALWAYS_INLINE 和 FUNC_CANNOT_INLINE pragma。
SPRU514H	C/C++ 语言	节 6.9.7	添加了 diag_push 和 diag_pop 诊断消息 pragma。
SPRU514H	C/C++ 语言	节 6.14、节 6.14.1 和节 6.14.3	添加了对 C99 和 C++03 的支持。--relaxed_ansi 现在是默认选项，--strict_ansi 是另一个选项；标准违反严格性的“正常模式”不再可用。
SPRU514H	运行时环境	节 7.4	添加了对*汇编语言工具用户指南*中有关在 C 和 C++ 语言中访问链接器符号一节的引用。
SPRU514G	运行时环境	表 7-6 和表 7-8	添加了用于 TMU 指令以及使用放置在高于通常的 22 位地址范围的数据的 32 位地址读取和写入存储器的内在函数。
SPRU514H	运行时支持函数	节 8.1.3	rtssrc.zip 文件中不再提供 RTS 源代码。相反，它位于编译器安装程序 lib/src 子目录内的单独文件中。

早期修订版本（续）

添加内容的版本	章节	位置	添加/修改/删除
SPRU514H	C++ 名称还原器	节 9.1	更正了有关名称还原器选项的信息。
SPRU514H	CLA 编译器	节 10.1	CLA 代码现在支持非递归函数调用和两个以上的函数参数。简化了 CLA 暂存区管理；不再需要在链接器命令文件中指定暂存区的大小。编译器现在支持 CLA 中断的中断属性和 INTERRUPT pragma。

This page intentionally left blank.

重要声明和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265