TMS320C6000 优化 C/C++ 编译器
v8.3.x

User’s Guide

Literature Number: ZHCUAQ1F
JULY 2015 – REVISED APRIL 2023

目录

插图清单

表格清单

关于本手册

TMS320C6000 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
- 汇编程序优化器
- 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 2014 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：

cl6x [*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：

cl6x --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*]。

- 本文档介绍了对 TMS320C6000™ 处理器系列中的 C64+、C6740 和 C6600 型号的支持。v8.0 和更高版本的 TI 代码生成工具不支持 C6200、C6400、C6700 和 C6700+ 版本。如果您使用其中某个旧器件，请使用 7.4 版本的代码生成工具，并参考 SPRU187 和 SPRU186 文档。

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 14882-2014, International Standard - Programming Languages - C++ (The 2014 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/)

OpenCL™ Specification version 1.2 (https://www.khronos.org/opencl/)

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：

SPRUI03 ***TMS320C6000 汇编语言工具用户指南。***介绍了用于 TMS320C6000 器件平台 C64+、C6740 和 C6600 版本的汇编语言工具（汇编器、链接器以及其他用于开发汇编语言代码的工具）、汇编器指

令、宏命令、通用目标文件格式和符号调试指令。使用旧版 C6200、C6400、C6700 和 C6700+ 器件时，请参阅 SPRU186。

SPRAB89 ***C6000 嵌入式应用二进制接口***。为德州仪器 (TI) 的 TMS320C6000 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。

SPRU190 ***TMS320C6000 DSP 外设概述参考指南。***提供了概述并简要介绍了 TMS320C6000 系列数字信号处理器 (DSP) 上可用的外设。

SPRU732 ***TMS320C64x/C64x+ DSP CPU 和指令集参考指南。***介绍了 TMS320C6000 DSP 系列 TMS320C64x 和 TMS320C64x+ 数字信号处理器 (DSP) 的 CPU 架构、流水线、指令集和中断。C64x/C64x+ DSP 代系包含 C6000 DSP 平台中的定点器件。C64x+ DSP 是 C64x DSP 的增强版，增加了功能并扩展了指令集。

SPRUGH7 ***TMS320C66x CPU 和指令集参考指南。***介绍了 TMS320C6000 DSP 平台 TMS320C66x 数字信号处理器 (DSP) 的 CPU 架构、流水线、指令集和中断。C66x DSP 代系包含 C6000 DSP 平台中的浮点器件。

SPRUFE8 ***TMS320C674x CPU 和指令集参考指南。***介绍了 TMS320C6000 DSP 平台 TMS320C674x 数字信号处理器 (DSP) 的 CPU 架构、流水线、指令集和中断。C674x 是一种浮点 DSP，它将 TMS320C67x+ DSP 和 TMS320C64x+ DSP 指令集架构组合到一个内核中。

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。

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

This page intentionally left blank.

章节 1

软件开发工具简介

TMS320C6000™ 由一套软件开发工具支持，其中包括优化 C/C++ 编译器、汇编优化器 、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 *TMS320C6000 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。

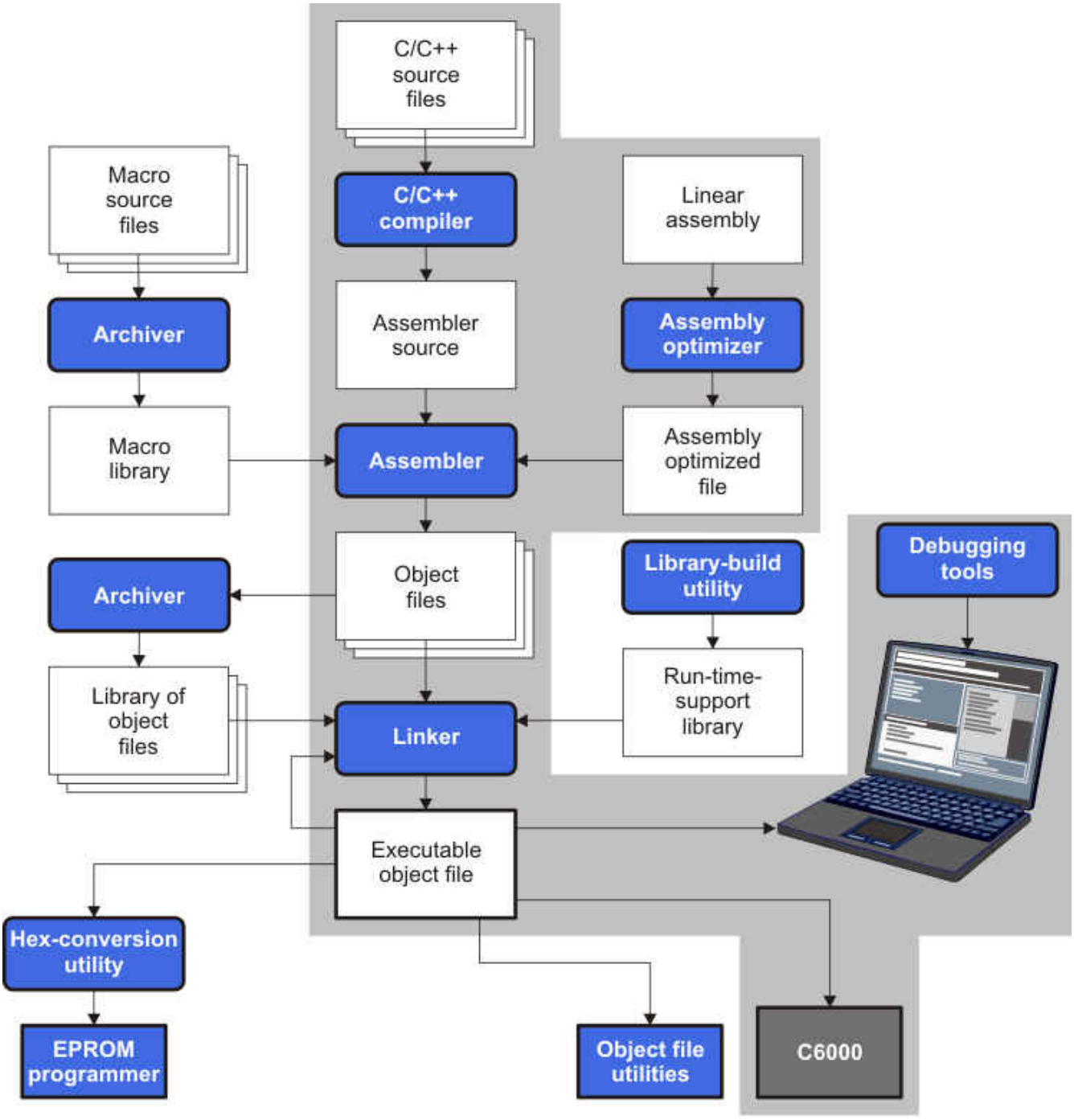


图 1-1. TMS320C6000 软件开发流程

以下列表描述了图 1-1 中显示的工具：

- **汇编优化器**允许在编写线性汇编代码时无需关注流水线结构或寄存器分配。它接受未进行寄存器分配和未调度的汇编代码。汇编优化器通过分配寄存器和循环优化将线性汇编转换为可利用软件流水线的高度并行汇编。请参阅章节 5。
- **编译器**接受 C/C++ 源代码，生成 C6000 汇编语言源代码。请参阅章节 3。
- **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 *TMS320C6000 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 6。请参阅 *TMS320C6000 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 *TMS320C6000 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 9。

  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 9.4。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 *TMS320C6000 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 10。
- **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 *TMS320C6000 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 **TMS320C6000** 器件上执行。在改进和更正代码时可使用 XDS 模拟器。

1.2 编译器接口

编译器是名为 cl6x 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 3.1。

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 8。

1.3 ANSI/ISO 标准

编译器支持 1989、1999 和 2011 版本的 C 语言以及 2014 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++ 标准的 C++14 版本。以前使用的是 C++03。请参阅 C++ 标准 ISO/IEC 14882:2014。有关*不受支持的* C++ 特性的说明，请参阅节 7.2。
- **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 9。

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 7.13 。

1.4 输出文件

以下类型的输出文件由编译器创建：

- **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 C6000 的 ELF 格式由 C6000 嵌入式应用程序二进制接口 (EABI) 扩展，相关信息请参阅 SPRAB89 文档。

v8.0 和更高版本的 TI 代码生成工具不支持 COFF 目标文件。如果希望生成 COFF 输出文件，请使用 v7.4 的代码生成工具，并参考 SPRU186 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 9.4。
- **C++ 名称还原器**

  C++ 名称还原器 (dem6x) 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 10。
- **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 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《*TMS320C6000 汇编语言工具用户指南*》所述。

章节 2

开始使用代码生成工具

本章概述了创建一个使用 C6000 代码生成工具的 Code Composer Studio 项目的过程。此外，它还介绍了编译器和链接器的命令行。

2.1 Code Composer Studio 项目如何使用编译器

如果将 Code Composer Studio (CCS) 用作开发环境，则在创建工程时会自动设置编译器和链接器选项。您所做的工程设置决定了使用哪些编译器和链接器命令行选项来构建工程。按照以下步骤在 CCS v6.0 中创建和编译工程。在 CCS 的其他版本中，确切的步骤可能有所不同。

表 2-1. 创建 CCS 工程的步骤

步进	使用编译器的影响
1.从菜单中选择 **File > New > CCS Project**。	
2.在“New CCS Project”向导中，首先选择 **Target**。您可以使用左侧的下拉菜单筛选右侧特定目标的列表。v8.x C6000 代码生成工具支持 C64x+、C6600 和 C6740 目标。	设置 --silicon_version (-mv) 编译器选项。请参阅节 3.3.5。此外，使用 --define 编译器选项定义与目标匹配的预处理器符号。请参阅节 3.3.2。
3.在 **Connection** 字段中，选择您将用来连接到器件的仿真器。	生成目标配置文件，以便在运行工程时使用。
4.在 **Project name** 字段中，键入工程的名称。	确定项目所在的文件夹。
5.展开 **Advanced settings** 区域。	
6.确保选择了要使用的 **Compiler version**。	将 --include_path 编译器选项设置为该版本代码生成工具的 include 目录。请参阅节 3.5.2.1。
7.默认情况下，C6000 应用程序被编译为小端字节序。在 **Device endianness** 字段中，您可以根据需要选择大端字节序。	如果未使用默认值，则设置 --big_endian 编译器选项。请参阅节 3.3.4。
8.链接器命令文件和运行时支持库是根据您在其他字段中的选择自动选择的。	
9.展开 **Project templates and examples** 区域。	
10.为工程选择模板。您可以选择的工程模板包括一个没有源文件的全空工程、一个仅包含 main.c 的工程、一个仅汇编工程和一个 Hello World 示例。“TI Resource Explorer”窗口中提供了使用您安装的插件软件元件的其他示例。	
11.点击“**Finish**”（完成）。	

创建 CCS 工程后，可以使用工程的“Properties”对话框查看编译器和链接器的使用方法，并修改编译和链接时使用的命令行选项。要打开此对话框，请在“Project Explorer”中选择工程，然后从菜单中选择 **Project > Properties**。展开类别树以选择 **Build > C6000 Compiler** 和 **Build > C6000 Linker**。如需详细了解您在此对话框中看到的所有命令行选项，请参阅章节 3。

2.2 从命令行编译

如果要在 Code Composer Studio 等 IDE 之外开发工程，则需要使用编译器和链接器的命令行界面。

编译器和链接器使用相同的可执行文件运行。此可执行文件是 **cl6x.exe** 文件，位于 TI 代码生成工具安装程序的 **bin** 子目录中。

您可以使用单个命令行来编译代码，以创建目标文件，并链接目标文件以创建可执行文件。出现在 --run_linker（或简写为 -z）选项之前的所有命令行选项都适用于编译器。出现在 --run_linker (-z) 选项之后的所有命令行选项都适用于链接器。在以下命令行中，-mv6740、--c99、--opt_level、--define 和 --include_path 选项是编译器选项。--library、--heap_size 和 --output_file 选项是链接器选项。

```
cl6x -mv6740 --c99 --opt_level=1 --define=c6748 --include_path="C:/ti/ti-cgt-c6000_8.3/include"
    hello.c objects.cpp algs.asm
    --run_linker --library=lnk.cmd --heap_size=0x800 --output_file=myprogram.out
```

章节 3

使用 C/C++ 编译器

编译器将您的源程序转换成 TMS320C6000 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。

3.1 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、汇编代码和线性汇编代码。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 3.3.10。
- **链接器**会组合目标文件以创建静态可执行文件。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 6。

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 6.1.1。

有关汇编器和链接器的完整说明，请参阅《*TMS320C6000 汇编语言工具用户指南*》。

3.2 调用 C/C++ 编译器

要调用编译器，请输入：

cl6x [*options*] [*filenames*] [**--run_linker** [*link_options*] *object files*]

cl6x	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 3-6 到表 3-28 列出了这些选项。
filenames	一个或多个 C/C++ 源文件、 汇编语言源文件 和线性汇编文件。
--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 6。
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：

```
cl6x symtab.c file.c seek.asm --run_linker --library=lnk.cmd
       --output_file=myprogram.out
```

3.3 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 **cl6x**。

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 C6X_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 3.4.1。

表 3-1 到表 3-28 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 3-1. 处理器选项

选项	别名	效果	段
--silicon_version=*id*	-mv	选择目标版本。默认为 6400+。其他支持的选项为 6600 和 6740。	节 3.3.5
--big_endian	-me	以大端格式生成目标代码。	节 3.3.4

表 3-2. 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 4.1
--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。	节 4.1、节 4.3
--opt_for_space=*n*	-ms	在四个级别（0、1、2 和 3）上控制代码大小。	节 4.9
--opt_for_speed[=*n*]	-mf	控制大小和速度之间的权衡（0-5 范围）。如果未指定此选项，或此选项未指定 *n*，则默认值为 4。	节 4.2

(1) **注意：**机器专用选项（参阅表 3-12）也会影响优化。

表 3-3. 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 4.5
--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 4.4.1
--disable_inlining		防止发生任何内联。	节 3.11
--fp_mode={relaxed|strict}		启用或禁用宽松浮点模式。	节 3.3.3
--fp_reassoc={on|off}		启用或禁用浮点算术的重新关联。	节 3.3.3

表 3-3. 高级优化选项(1) (continued)

选项	别名	效果	段
--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 3.3.3
--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 4.3.1
--optimizer_interlist	-os	交叉列出优化器注释与汇编语句。	节 4.16
--program_level_compile	-pm	组合源文件以执行程序级优化。	节 4.4
--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 3.3.3
--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 4.12.1

(1) **注意：**机器专用选项（参阅表 3-12）也会影响优化。

表 3-4. 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 3.3.6 节 4.17
--symdebug:dwarf_version=2|3		指定 DWARF 格式版本。	节 3.3.6
--symdebug:none		禁用所有符号调试。	节 3.3.6 节 4.17
--disable_push_pop		禁用调用 RTS 函数 _push_rts() 和 _pop_rts() 的代码大小优化。如果收到有关超出调用位置范围的 RTS 例程进行调用的警告，可能需要使用此选项。	--
--machine_regs		将寄存器操作数显示为汇编代码中的机器寄存器。	节 3.3.12

表 3-5. 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 3.5.2.1
--preinclude=*filename*		在编译开始时包含 *filename*。	节 3.3.3

表 3-6. 控制选项

选项	别名	效果	段
--compile_only	-c	禁用链接（否定 --run_linker）。	节 6.1.3
--help	-h	打印（在标准输出设备上）编译器理解的选项的说明。	节 3.3.2
--run_linker	-z	导致从编译器命令行调用链接器。	节 3.3.2
--skip_assembler	-n	编译 C/C++ 源文件或线性汇编源文件，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 3.3.2

表 3-7. 语言选项

选项	别名	效果	段
--c89		根据 ISO C89 标准处理 C 文件。	节 7.13
--c99		根据 ISO C99 标准处理 C 文件。	节 7.13
--c11		根据 ISO C11 标准处理 C 文件。	节 7.13
--c++14		根据 ISO C++14 标准处理 C++ 文件。 已弃用 --c++03 选项。	节 7.13
--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 3.3.8
--exceptions		启用 C++ 异常处理。	节 7.6
--extern_c_can_throw		允许外部 C 函数传播异常。	--
--float_operations_allowed ={none|all|32|64}		限制允许的浮点运算类型。	节 3.3.3

表 3-7. 语言选项 (continued)

选项	别名	效果	段
--gen_cross_reference_listing	-px	生成交叉引用列表文件 (.crl)。	节 3.9
--multithread		在编译器生成的目标文件中插入构建属性，这样使 TI 链接器在自动选择 RTS 库或解析对 libc.a 的引用时选择 RTS 库的线程安全版本。或者，可以使用同名的链接器选项 (--multithread) 来强制链接器在即使没有任何目标文件包含此构建属性的情况下也要选择 RTS 库的线程安全版本。如果使用 --openmp 选项，则会自动启用 --multithread 选项。	节 8.10.2
--openmp	--omp	支持 OpenMP。使用此选项会自动启用 --multithread 选项，这样使 TI 链接器在自动选择 RTS 库或解析对 libc.a 的引用时选择 RTS 库的线程安全版本。	节 8.10.1
--pending_instantiations=#		指定在任何给定时间可能正在进行的模板实例化的数量。使用 0 来指定无限数量。	节 3.3.4
--printf_support={nofloat|full|minimal}		支持小型且版本受限的 printf 函数系列（sprintf、fprintf 等）和 scanf 函数系列（sscanf、fscanf 等）运行时支持函数的支持。	节 3.3.3
--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 7.13.3
--rtti	-rtti	启用 C++ 运行时类型信息 (RTTI)。	--
--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 7.13.3
--vectypes={on|off}		启用对 TI 矢量数据类型的支持。	节 7.3.2
--wchar_t={32|16}		设置 C/C++ 类型 wchar_t 的大小。默认为 16 位。	节 3.3.4

表 3-8. 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 3.5.8
--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 3.5.9
--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 3.5.10
--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 3.5.4
--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 3.5.6
--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 3.5.5
--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 3.5.7

表 3-9. 预定义宏选项

选项	别名	效果	段
--define=*name*[=*def*]	-D	预定义 *name*。	节 3.3.2
--undefine=*name*	-U	未定义 *name*。	节 3.3.2

表 3-10. 诊断消息选项

选项	别名	效果	段
--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 3.7.1
--diag_remark=*num*	-pdsr	将 *num* 标识的诊断分类为备注。	节 3.7.1
--diag_suppress=*num*	-pds	抑制 *num* 标识的诊断。	节 3.7.1

表 3-10. 诊断消息选项 (continued)

选项	别名	效果	段
--diag_warning=*num*	-pdsw	将 *num* 标识的诊断分类为警告。	节 3.7.1
--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 3.7.1
--emit_warnings_as_errors	-pdew	将警告视为错误。	节 3.7.1
--issue_remarks	-pdr	发出备注（非严重警告）。	节 3.7.1
--no_warnings	-pdw	抑制诊断警告（仍会发出错误）。	节 3.7.1
--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 3.7.1
--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 3.7.1
--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 3.7.1

表 3-11. 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 3.10
--section_sizes={on|off}		生成段大小信息，包括含可执行代码和常量、常量或初始化数据（全局和静态变量）以及未初始化数据的段的大小。（如果此选项未包含在命令行中，则默认为 off。如果使用此选项但未指定值，则默认为 on。）	节 3.7.1

表 3-12. 运行时模型选项

选项	别名	效果	段
--assume_control_regs_read		假设读取 FP 和 SAT 位。	节 3.3.4
--common={on|off}		默认为 on。设置为 on 时，未初始化的文件范围变量作为通用符号发出。设置为 off 时，不会创建通用符号。	节 3.3.4
--debug_software_pipeline	-mw	生成详细的软件流水线报告。	节 4.6.2
--disable_software_pipeline	-mu	关闭软件流水线。	节 4.6.1
--fp_not_associative	-mc	阻止对关联浮点运算进行重新排序。	节 4.13
--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 6.2.3
--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 6.2.2
--interrupt_threshold[=*num*]	-mi	指定中断阈值。	节 3.12
--mem_model:const={far_aggregates|far|data}		允许常量对象远远独立于--mem_model:data 选项。	节 8.1.4.3
--mem_model:data={far_aggregates|near|far}		确定数据访问模型。	节 8.1.4.1
--no_bad_aliases	-mt	允许某些有关别名和循环的假设。	节 4.12.2
--no_compress		阻止压缩。	--
--no_reload_errors		关闭所有与重新加载相关的循环缓冲区错误消息。	--

表 3-12. 运行时模型选项 (continued)

选项	别名	效果	段
--profile:breakpt		启用基于断点的分析。	节 3.3.6 节 4.17.1
--speculate_loads=*n*	-mh	指定推测加载字节计数阈值。允许对具有分界地址范围的加载进行推测执行。	节 4.6.3.1
--speculate_unknown_loads		允许对具有无限地址的加载进行推测执行。	节 3.3.4
--use_const_for_alias_analysis	-ox	使用常量来消除指针的歧义。	节 3.3.4

表 3-13. 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 3.16
--entry_parm={none|name|address}		将函数的参数指定给 --entry_hook 选项。	节 3.16
--exit_hook[=*name*]		启用出口挂钩。	节 3.16
--exit_parm={none|name|address}		将函数的参数指定给 --exit_hook 选项。	节 3.16
--remove_hooks_when_inlining		删除自动内联函数的入口/出口挂钩。	节 3.16

表 3-14. 反馈选项

选项	别名	效果	段
--analyze={codecov|callgraph}		从配置文件数据生成分析信息。	节 4.11.4.2
--analyze_only		仅生成分析。	节 4.11.4.2
--gen_profile_info		生成检测代码以收集配置文件信息。	节 4.10.1.3
--use_profile_info=*file1*[, *file2*,...]		指定配置文件信息文件。	节 4.10.1.3

表 3-15. 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 3.3.12
--asm_listing	-al	生成汇编列表文件。	节 3.3.12
--c_src_interlist	-ss	交叉列出 C 源代码和汇编语句。	节 3.13 节 4.16
--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 3.3.2
--asm_cross_reference_listing	-ax	生成交叉引用文件。	节 3.3.12
--asm_define=*name*[=*def*]	-ad	设置 *name* 符号。	节 3.3.12
--asm_dependency	-apd	执行预处理；仅列出程序集依赖项。	节 3.3.12
--asm_includes	-api	执行预处理；仅列出包含的 #include 文件。	节 3.3.12
--asm_undefine=*name*	-au	不对预定义的常量 *name* 进行定义。	节 3.3.12
--include_file=*filename*	-ahi	包含汇编模块的指定文件。	节 3.3.12
--no_const_clink		停止为常量全局数组生成 .clink 指令。	节 3.3.3
--strip_coff_underscore		帮助将手工编码的汇编从 COFF 过渡到 EABI。	节 3.3.12

表 3-16. 文件类型说明符选项

选项	别名	效果	段
--ap_file=*filename*	-fl	不管其扩展名为何，都将 *filename* 标识为线性汇编源文件。默认情况下，编译器和汇编优化器将 .sa 文件视为线性汇编源文件。	节 3.3.8
--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 3.3.8
--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 3.3.8

表 3-16. 文件类型说明符选项 (continued)

选项	别名	效果	段
--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 3.3.8
--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 3.3.8

表 3-17. 目录说明符选项

选项	别名	效果	段
--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 3.3.11
--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 3.3.11
--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 3.3.11
--output_file=*filename*	-fe	指定编译输出文件名；可以覆盖 --obj_directory。	节 3.3.11
--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 3.3.11
--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 3.3.11

表 3-18. 默认文件扩展名选项

选项	别名	效果	段
--ap_extension=[.]*extension*	-el	设置线性汇编源文件的默认扩展名。	节 3.3.10
--asm_extension=[.]*extension*	-ea	设置汇编源文件的默认扩展名。	节 3.3.10
--c_extension=[.]*extension*	-ec	设置 C 源文件的默认扩展名。	节 3.3.10
--cpp_extension=[.]*extension*	-ep	设置 C++ 源文件的默认扩展名。	节 3.3.10
--listing_extension=[.]*extension*	-es	设置列表文件的默认扩展名。	节 3.3.10
--obj_extension=[.]*extension*	-eo	设置目标文件的默认扩展名。	节 3.3.10

表 3-19. 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 3.3.2

表 3-20. 性能顾问选项

选项	别名	效果	侧面
--advice:performance[={all|none}]		生成编译器优化建议。默认为 all。	节 3.14
--advice:performance_file={stdout|stderr|user_specified_filename}		指定将建议写入 stdout、stderr 或文件。	节 3.14
--advice:performance_dir={user_specified_directory_name}		指定在命名目录中创建建议文件。	节 3.14

3.3.1 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 6 以及《*TMS320C6000 汇编语言工具用户指南*》。

表 3-21. 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* 字节，并定义全局符号来指定栈大小。默认值 = 1K 字节。

表 3-21. 链接器基本选项 (continued)

选项	别名	说明
--heap_size=*size*	[-]-heap	将堆大小（对于 C 中的动态存储器分配）设为 *size* 字节，并定义全局符号来指定栈大小。默认值 = 1K 字节。

表 3-22. 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 6.3.1.1。
--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 3-23. 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 3-24. 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。
--warn_sections	-w	创建未定义的输出段时显示一条消息。

表 3-25. 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。
--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 3-26. 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_symtable	-s	从可执行目标文件中去除符号表信息和行号条目。
--retain={*symbol*|*section specification*}		指定要由链接器保存的符号或段。
--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。
--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 3-27. 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。默认为 rle。
--copy_compression[=type]		压缩由链接器复制表复制的数据。默认为 rle。
--fill_value=*value*	-f	为输出段中的空穴设置默认填充值
--multithread		使 TI 链接器在自动选择 RTS 库或解析对 libc.a 的引用时选择线程安全版 RTS 库，即使所有输入目标文件都不包含由 --multithread 编译器选项放置的 TI 构建属性，也是如此。如果您使用 --openmp 编译器选项创建任何目标文件，则会自动启用 --multithread 选项。
--ram_model	-cr	在加载时初始化变量。有关详细信息，请参阅节 6.3.4。
--rom_model	-c	在运行时自动初始化变量。有关详细信息，请参阅节 6.3.4。
--trampolines[=off|on]		生成 far call trampolines。默认为 on。

表 3-28. 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 on。
--linker_help	[-]-help	显示有关语法和可用选项的信息。
--minimize_trampolines[=off|postorder]		放置段以最大限度地减少所需的 far trampolines 数量。默认值为 postorder。
--preferred_order=*function*		为函数放置设定优先级。
--trampoline_min_spacing=*size*		当 trampoline 预留的间隔比指定的限值更近时，尝试使它们相邻。
--unused_section_elimination[=off|on]		消除可执行模块中不需要的段。默认为 on。
--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。

3.3.2 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 4.16。--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：

```
cl6x --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 C6X_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 6.1.3。

--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 3.5.2.1。

--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 6.1。

--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*
对预定义的常量 *name* 不定义。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。

3.3.3 其他有用的选项

以下是其他选项的详细说明：

--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：

- 如果双精度浮点表达式的结果被分配给单精度浮点或整数，或者立即在单精度上下文中使用，则表达式的计算将转换为单精度计算。如果表达式中的双精度常量可以正确地表示为单精度常量，那么它们会转换为单精度常量。
- 如果所有参数都是单精度的并且结果将在单精度上下文中使用，则对 math.h 中的双精度函数的调用将转换为对应的单精度函数。必须包含 math.h 头文件才能使此优化正常运行。
- 除以一个常数被转换为逆乘法。

在以下示例中，iN=整数变量，fN=浮点变量，dN=双精度变量：

```
i1 = f1 + f2 * 5.0 -> +, * are float, 5.0 is converted to 5.0f
i1 = d1 + d2 * d3 -> +, * are float
f1 = f2 + f3 * 1.1; -> +, * are float, 1.1 is converted to 1.1f
```

要启用宽松浮点模式，请使用 --fp_mode=relaxed，这也会设置 --fp_reassoc=on。要禁用宽松浮点模式，请使用 --fp_mode=strict，这也会设置 --fp_reassoc=off。

如果指定了 --strict_ansi，则会自动设置 --fp_mode=strict。可以通过在 --strict_ansi 之后指定 --fp_mode=relaxed 以采用严格的 ANSI 模式来启用宽松浮点模式。

--fp_reassoc={on|off}
启用或禁用浮点算术的重新组合。如果设置了 --strict_ansi，则设置 --fp_reassoc=off，因为浮点算术的重新关联是违反 ANSI 要求的。

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。

--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。

--no_const_clink
通知编译器不要为常量全局数组生成条件链接 (.clink) 指令。默认情况下，这些数组放置在 .const 子段中并有条件地链接。

--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组合。

3.3.4 运行时模型选项

这些选项专用于 TMS302C6000 工具集。有关更多信息，请参阅参考的章节。节 3.3.12 中列出了 TMS320C6000 专用汇编器选项。

C6000 编译器目前仅支持使用 ELF 目标文件格式和 DWARF 调试格式的嵌入式应用程序二进制接口 (EABI) ABI。有关 EABI 的详细信息，请参阅《*C6000 嵌入式应用程序二进制接口应用报告*》(SPRAB89)。如果希望支持传统的 COFF ABI，请使用 C6000 v7.4.x 代码生成工具，并参阅 SPRU187 和 SPRU186 以查看相关文档。

--assume_control_regs_read
告知编译器假设在程序中的某个位置读取 FP 和 SAT 控制位。因此，编译器不会推测可能设置 FP 或 SAT 控制位的指令。有关更多信息，请参阅节 8.6.15。

--advice:performance
生成编译时优化建议。请参阅节 3.14。

--big_endian
以大端格式生成代码。默认情况下生成小端代码。

--common={on|off}
当为 on（默认设置）时，未初始化的文件范围变量作为通用符号发出。当为 off 时，不会创建通用符号。允许创建通用符号的好处是生成的代码可以删除未使用的变量，否则会增加 .bss 段的大小。（大于 32 字节的未初始化变量通过放置在可以在链接时省略的单独子段中被单独地优化。）如果变量已分配到 .bss 以外的段或具有指定的存储体，则变量不能作为通用符号。

--debug_software_pipeline
生成详细的软件流水线报告。请参阅节 4.6.2。

--disable_software_pipeline
关闭软件流水线。请参阅节 4.6.1。

--fp_not_associative
编译器不会对浮点运算进行重新排序。请参阅节 4.13。

--interrupt_threshold=*n*
指定中断阈值 *n*。该阈值设置了编译器可以禁用中断的最大周期数。请参阅节 3.12。

--mem_model:const=*type*
允许创建与--mem_model:data 选项远远无关的常量对象。*type* 可以是 data、far 或 far_aggregates。请参阅 节 8.1.4.3

--mem_model:data=*type*	指定数据访问模型为 *type* far、far_aggregates 或 near。默认为 far_aggregates。请参阅节 8.1.4.1。
--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。
--silicon_version=*num*	选择目标 CPU 版本。请参阅节 3.3.5。
--speculate_loads=*n*	指定推测负载字节计数阈值。允许推测执行具有边界地址的负载。请参阅节 4.6.3.1。
--speculate_unknown_loads	允许推测执行具有边界地址的负载。
--static_template_instantiation	根据需要通过解析器对当前文件中的所有模板实体进行实例化。这些实例化也获得了内部（静态）链接。此选项可能会稍微提高编译速度。
--use_const_for_alias_analysis	使用常量来消除指针的歧义。
--wchar_t={32|16}	设置 C/C++ 类型 wchar_t 的大小（以位为单位）。默认情况下，编译器生成 16 位 wchar_t。16 位 wchar_t 对象与 32 位 wchar_t 对象不兼容；如果将这两个对象组合在一起，则会产生错误。因为 Linux 使用 32 位扩展字符，指定 --linux 选项时，则意味着 --wchar_t=32。

3.3.5 选择目标 CPU 版本（--silicon_version 选项）

--silicon_version 选项控制目标专用指令和对齐方式的使用。此选项的别名为 -mv。如果未使用此选项，编译器会默认为 C6400+ 部件生成代码。

指定器件的系列，例如 --silicon_version=6400+ 或 --silicon_version=6740。

目标 CPU 版本选项包括：

- -mv6400+ 或 -mv64+
- -mv6740
- -mv6600

如果希望支持 C6200、C6400、C6700 或 C6700+ 目标，请使用 C6000 v7.4.x 代码生成工具，并参阅 SPRU187 和 SPRU186 以查看相关文档。C6000 v8.x 代码生成工具不再支持这些目标。

3.3.6 符号调试和分析选项

下述选项用于选择符号调试或分析：

--profile:breakpt	禁用在使用基于断点的分析器时可能会导致错误行为的优化。
--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 4.17。有关 DWARF 格式的详细信息，请参阅 *DWARF 调试标准*。
--symdebug:dwarf_version={2|3}	在指定 --symdebug:dwarf（默认值）时，指定待生成的 DWARF 调试格式版本（2 或 3）。默认情况下，编译器生成 DWARF 版本 3 的调试信息。有关 TI 扩展到 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。

3.3.7 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 、线性汇编文件或目标文件。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
.sa	线性汇编文件

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 3.3.8。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 3.3.11。

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：

```
cl6x *.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。

3.3.8 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：

--ap_file=*filename*	用于线性汇编源文件
--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：

```
cl6x --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。

3.3.9 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 3.3.10。

3.3.10 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：

--ap_extension=*new extension*	用于线性汇编源文件
--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：

```
cl6x --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：

```
cl6x --asm_extension=rrr --obj_extension=o fit.rrr
```

3.3.11 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：

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

```
cl6x --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：

```
cl6x --list_directory=d:\listing
```

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

```
cl6x --obj_directory=d:\object
```

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

```
cl6x --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：

```
cl6x --pp_directory=d:\preproc
```

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

```
cl6x --temp_directory=d:\temp
```

3.3.12 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《*TMS320C6000 汇编语言工具用户指南*》。

--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。

--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。

--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。
--machine_regs	将寄存器操作数显示为汇编文件中的机器寄存器以用于调试。
--no_compress	防止在汇编器中进行压缩。在可能/有利的情况下，压缩将 32 位指令更改为 16 位指令。
--no_reload_errors	关闭汇编代码中所有与重新加载相关的循环缓冲区错误消息。
--strip_coff_underscore	帮助将手工编码的汇编从 COFF 过滤到 EABI。尽管 COFF 输出不再受支持，但此选项仍可用作 COFF ABI 到 ELF EABI 的迁移辅助工具。对于 COFF ABI，编译器在所有 C/C++ 标识符的开头加下划线。对于 EABI，链接时符号与 C/C++ 标识符名称相同。此选项根据需要从旧符号引用中删除下划线前缀。

3.4 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。

3.4.1 设置默认编译器选项 (C6X_C_OPTION)

您可能会发现，使用 C6X_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 C6X_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 C6X_C_OPTION 环境变量并进行处理。

下表展示了如何设置 C6X_C_OPTION 环境变量。为操作系统选择命令：

操作系统	输入
UNIX (Bourne shell)	**C6X_C_OPTION="** *option*$_1$ [*option*$_2$ ...]**"; export C6X_C_OPTION**
Windows	**set C6X_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 C6X_C_OPTION 环境变量，如下所示：

```
set C6X_C_OPTION=--quiet --src_interlist --run_linker
```

命令行或 C6X_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 C6X_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 C6X_C_OPTION 设置如上所示：

```
cl6x *c                          ; compiles and links
cl6x --compile_only *.c          ; only compiles
cl6x *.c --run_linker lnk.cmd    ; compiles and links using a command file
cl6x --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 3.3。有关编译器选项的详细信息，请参阅 *TMS320C6000 汇编语言工具用户指南*》中的链接器说明一章。

3.4.2 命名一个或多个备用目录 (C6X_C_DIR)

链接器使用 C6X_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)	**C6X_C_DIR="** *pathname*$_1$ **;** *pathname*$_2$ **;..."; export C6X_C_DIR**
Windows	**set C6X_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 C6X_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 C6X_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 C6X_C_DIR`
Windows	`set C6X_C_DIR=`

3.5 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。

3.5.1 预先定义的宏名称

编译器维护并识别表 3-29 中列出的预定义宏名称。

表 3-29. 预定义 C6000 宏名称

宏名称	说明
_BIG_ENDIAN	如果选择了大端模式（使用了 --big_endian 选项），则已定义；否则未定义。
__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	如果选择了小端模式（未使用 --big_endian 选项），则已定义；否则未定义。
__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 7.1。
__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_32BIT_LONG__	如果“long”类型为 32 位宽，则定义为 1；否则未定义。
__TI_40BIT_LONG__	如果未定义 __TI_32BIT_LONG__，则定义为 1；否则未定义。
__TI_C99_COMPLEX_ENABLED__	如果启用了复杂数据类型，则定义为 1。尽管数学运算仅在包含 complex.h 时才可用，但情况总是如此。

表 3-29. 预定义 C6000 宏名称 (continued)

宏名称	说明
__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。
__TI_EABI__	始终定义为 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。
__TI_WCHAR_T_BITS__	定义为 wchar_t 类型。
__TIME__ (1)	以“*hh:mm:ss*”形式扩展到编译时间
_TMS320C6X	始终已定义
_TMS320C6400_PLUS	如果目标是 C6400+、C6740 或 C6600，则已定义
_TMS320C6740	如果目标是 C6740 或 C6600，则已定义
_TMS320C6600	如果目标是 C6600，则已定义
__TMS320C6X__	始终定义为 _TMS320C6x 的备用名称
__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 3-29 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");
```

3.5.2 #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 C6X_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 C6X_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 3.5.2.1。有关输入文件目录的更多信息，请参阅节 3.4.2。

3.5.2.1 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	`cl6x --include_path=/tools/files source.c`
Windows	`cl6x --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 C6X_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 C6X_C_DIR：

```
C6X_C_DIR "/usr/include;/usr/ucb"; export C6X_C_DIR
```

或使用以下命令调用编译器：

```
cl6x --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
```

3.5.3 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。

3.5.4 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。

3.5.5 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。

3.5.6 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。

3.5.7 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。

3.5.8 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。

3.5.9 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。

3.5.10 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。

3.6 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。

3.7 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。
- **建议**提供有关推荐用法的信息。有关详细信息，请参阅节 4.15。

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。

3.7.1 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 3.7。
--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）。
--section_sizes={on|off}	生成段大小信息，包括含可执行代码和常量、常量或初始化数据（全局和静态变量）以及未初始化数据的段的大小。段大小信息在汇编阶段和链接阶段输出。此选项应与编译器选项一同放置在命令行上（即 --run_linker 或 --z 选项之前）。
--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。

3.7.2 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。

3.8 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。

3.9 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	

3.10 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 3-30 中列出的标识符之一开头。

表 3-30. 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 3-31 中定义的诊断标识符。

表 3-31. 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 3-31 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 3.7。

3.11 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 3.11.2）。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。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 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 4.5。
- 除非 --opt_level=off，否则 pragma FUNC_ALWAYS_INLINE（节 7.9.10）和等效的 `always_inline` 属性（节 7.14.2）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 7.9.8）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。

3.11.1 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，内在函数都是内联的。

有关内在函数的详细信息以及内在函数列表，请参阅节 8.6.6。除了所列出的这些之外，abs 和 memcpy 也是作为内在函数实现的。

3.11.2 内联限制

编译器会根据节 3.11 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

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。

3.11.3 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 3-1 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 3-1. 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);
   ...
}
```

3.11.4 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 3-2 所示。
- 在 .c 或 .cpp 文件中创建相同版本的函数定义，如示例 3-3 所示。

在以下示例中，strlen 函数有两个定义。第一个定义（示例 3-2）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 3-3）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 3-2. 头文件 string.h

```
/****************************************************************************/
/* string.h  vx.xx   (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 3-3. 库定义文件

```
/****************************************************************************/
/*  strlen                                                                  */
/****************************************************************************/
#undef _INLINE
#include <string.h>
{
_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;
}
```

3.12 中断灵活性选项（--interrupt_threshold 选项）

在 C6000 架构上，不能在分支的延迟时隙中接受中断。在某些情况下，编译器可以生成在多个潜在的周期内不能被中断的代码。对于给定的实时系统，可以硬性规定中断禁用的时间。

--interrupt_threshold=*n* 选项指定中断阈值 *n*。该阈值指定编译器可以禁用中断的最大周期数。如果省略 *n*，编译器会假设代码永不中断。在 Code Composer Studio 中，要指定代码永不中断，请选中“Interrupt Threshold”复选框，并将“Build Options”对话框“Compiler”选项卡上的“Advanced”类别中的文本框留空。

如果未指定 --interrupt_threshold=*n* 选项，则仅在软件流水线循环周围显式禁用中断。使用 --interrupt_threshold=*n* 选项时，则编译器分析循环结构和循环计数器，以确定执行循环所需的最大周期数。如果可以确定最大循环数小于阈值，则编译器会生成最快/最佳的循环版本。如果循环小于六个周期，就不会发生中断，因为循环总是在分支的延迟时隙内执行。否则，编译器将生成一个可以中断的循环（但仍然生成正确的结果：单赋值代码），这在大多数情况下会降低循环的性能。

--interrupt_threshold=*n* 选项无法理解内存系统的影响。在确定循环的最大执行周期数时，编译器不会计算使用慢速片外内存或内存条冲突的影响。建议使用保守的阈值来调整内存系统的影响。

有关更多信息，请参阅节 7.9.13 或《*TMS320C6000 编程指南*》。

备注

RTS 库文件不是使用 --interrupt_threshold 选项构建的

编译器附带的运行时支持库文件不是使用中断灵活性选项构建的。请参阅自述文件以了解如何为您的版本构建运行时支持库文件。请参阅节 9.4，以使用中断灵活性选项构建您自己的运行时支持库文件。

备注

带有--interrupt_threshold 选项的特殊情况

--interrupt_threshold=0 选项生成与未使用 --interrupt_threshold 选项时相同的代码来禁用软件流水线循环周围的中断。

--interrupt_threshold 选项（省略阈值）意味着没有添加代码来禁用软件流水线循环周围的中断，这意味着不能安全地中断代码。此外，循环性能不会降低，因为编译器不会通过确保循环内核中至少有一个周期不在分支指令的延迟时隙中来尝试使循环可中断。

3.13 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：

```
cl6x --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 4.16。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。

```
_main:
            STW     .D2     B3,*SP--(12)
            STW     .D2     A10,*+SP(8)
;-----------------------------------------------------------------------------
;   5 | printf("Hello, world\n");
;-----------------------------------------------------------------------------
            B       .S1     _printf
            NOP             2
            MVKL    .S1     SL1+0,A0
            MVKH    .S1     SL1+0,A0
||          MVKL    .S2     RL0,B3
            STW     .D2     A0,*+SP(4)
||          MVKH    .S2     RL0,B3
RL0:        ; CALL OCCURS
;-----------------------------------------------------------------------------
;   6 | return 0;
;-----------------------------------------------------------------------------
            ZERO    .L1     A10
            MV      .L1     A10,A4
            LDW     .D2     *+SP(8),A10
            LDW     .D2     *++SP(12),B3
            NOP             4
            B       .S2     B3
            NOP             5
            ; BRANCH OCCURS
```

3.14 生成和使用性能建议

在某些情况下，如果用户通过在代码中提供附加信息来辅助编译器，编译器可以进行更好的优化。编译器可以通过发出“建议”来提示采取某些措施以提高性能。要获得此建议，请使用 --advice:performance 选项：

```
cl6x --advice:performance -o3 filename.c
```

此性能建议分为 3 种不同类型：

- 使用正确编译器选项的建议
- 防止软件流水线不合格的建议
- 提高循环性能的建议

有关使用性能建议来优化代码的更多详细信息，请参阅节 4.15

3.15 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该 ABI 的任何系统上运行。

C6000 编译器仅支持嵌入式应用程序二进制接口 (EABI) ABI，这种接口仅适用于使用 ELF 目标文件格式和 DWARF 调试格式的目标文件。如果希望支持旧版 COFF ABI，请使用 C6000 v7.4.x 代码生成工具，并参阅 SPRU187 和 SPRU186 来获取文档。

没有 COFF 到 ELF 的自动转换工具；您将需要对汇编代码进行重新编译或重新汇编。请参阅《*TMS320C6000 EABI 迁移指南*》(SPRAB90)，了解从 COFF 迁移到 ELF 的好处以及迁移策略和问题的链接。

3.16 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 7.9.26。

章节 4

优化您的代码

编译器工具可以通过简化循环、软件流水线处理、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。

4.1 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），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 4.18.3）
  - 将变量分配给寄存器（节 4.18.13）
  - 执行循环旋转（节 4.18.10）
  - 消除未使用的代码
  - 简化表达式和语句（节 4.18.5）
  - 扩展对声明的内联函数的调用（节 4.18.6）
- **--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 4.18.4）
  - 删除未使用的赋值（节 4.18.4）
  - 消除局部公用表达式
- **--opt_level=2** 或 **-O2** 执行所有 --opt_level=1 (-O1) 优化，加上：
  - 执行软件流水线（节 4.6）
  - 执行循环优化
  - 消除全局公用子表达式（节 4.18.4）
  - 消除全局未使用的赋值（节 4.18.4）
  - 将循环中的数组引用转换为递增的指针形式（节 4.18.8）
  - 执行循环展开（节 7.9.33）
- **--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 4.4）
  - 简化返回值从未使用过的函数（节 4.4）
  - 内联函数对小函数的调用（节 3.11 和节 4.5）
  - 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 4.4）
  - 执行其他优化（节 4.3 和节 4.4）

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 3.11。

调试默认启用，并且优化级别不受调试信息生成的影响。

备注

不要降低优化级别以控制代码大小：要减小代码大小，请不要降低优化级别。相反，请使用 --opt_for_space 选项来控制代码大小/性能权衡。更高的优化级别（--opt_level 或 -O）与偏高的 --opt_for_space 级别相结合会产生最小的代码大小。有关更多信息，请参阅 节 4.9。

备注

--opt_level= *n* (-O *n*) 选项适用于汇编优化器：还应该将 --opt_level=*n* (-O) 选项与汇编优化器一起使用。汇编优化器不会执行本文描述的所有优化，但重要优化（例如软件流水线和循环展开）需要 --opt_level 选项。

4.2 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 4

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

4.3 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 4-1 中列出的选项与 --opt_level=3 一起使用以执行指定的优化：

表 4-1. 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 4.3.1
希望编译多个源文件	--program_level_compile	节 4.4

备注

请勿降低优化级别以控制代码大小

当试图减小代码大小时，不要降低优化级别，因为您可能会看到代码大小增加。相反，请使用 --opt_for_space 选项来控制代码。

4.3.1 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 4-2 选择相应的级别以附加到该选项。

表 4-2. 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

4.4 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。

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，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 4.3.1。

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。

4.4.1 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 4-3 选择合适的级别以附加到 --call_assumptions 选项。

表 4-3. 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 4-4 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 4-4. 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 4.4.2。

4.4.2 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 7.9.12）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 4.4.1。

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，可以获得最佳结果。

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++ 函数到应用程序入口点 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，可以实现最佳优化。*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。

4.5 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* 指定自动内联的大小阈值。此选项仅控制未明确声明为内联的函数的内联。

当未使用 --auto_inline 选项时，编译器根据优化级别和优化目标 (性能与代码大小) 设置大小限制。如果 -auto_inline size 参数设置为 0，则禁用自动内联扩展。如果 --auto_inline size 参数设置为非零整数，则编译器自动内联任何小于 ***size*** 的函数。 (这是对以前版本的更改；以前的版本会内联那些函数大小与函数调用次数的乘积小于 ***size*** 的函数。新方案更简单，但通常会对给定的 ***size*** 值进行更多的内联。)

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_info=1 或 --gen_opt_info=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。当使用 --auto_inline 时，编译器不会试图阻止导致编译时间或大小过度增长的内联；故请小心使用。

当未使用 --auto_inline 选项时，在特定调用点内联函数的决策是基于试图优化效益和成本的算法。编译器在调用点内联符合条件的函数，直至达到有关大小或编译时间的限制。

内联行为因指定的编译时选项而异：

- 如果编译的是代码大小而非性能时，则代码大小限制较小。--auto_inline 选项覆盖此大小限制。
- 在 --opt_level=3 时，编译器自动内联小函数。

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 3.11。

备注

有些函数不能被内联：如要考虑内联调用点，内联函数必须是合法的，并且不得以某种方式禁用内联。请参阅节 3.11.2 中的内联限制。

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。

4.6 优化软件流水线

软件流水线调度循环中的指令，以便循环的多次迭代能够并行执行。在优化级别 --opt_level=2（即 -O2）和 --opt_level=3（即 -O3）上，编译器通常试图对循环进行软件流水线处理。--opt_for_space 选项还会影响编译器试图对循环进行软件流水线处理的决定。通常，在使用 --opt_level=2 或 --opt_level=3 选项时，代码大小和性能会更好。（请参阅节 4.1。）

图 4-1 说明了进行了软件流水线处理的循环。循环的阶段由 A、B、C、D 和 E 表示。在此图中，一次最多可以执行 5 次循环迭代。阴影区域表示循环*内核*。在循环内核中，所有五个阶段都是并行执行的。内核上方的区域称为*流水线循环序言*，内核下方的区域称为*流水线循环结语*。

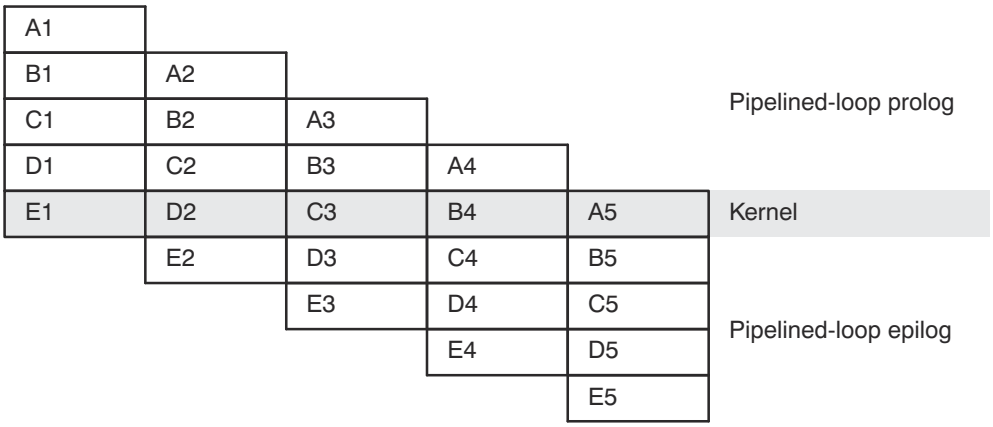


图 4-1. 进行了软件流水线处理的循环

如果您在线性汇编输入文件中输入指令注释，编译器会将注释连同附加信息一起移至输出文件。它会将一个二元组 <x, y> 附加到注释，从而规定指令在软件流水线上执行的循环的迭代和周期。基于零的数字 x 表示在循环内核第一次执行期间指令所进行的迭代。基于零的数字 y 表示在循环的单次迭代中指令调度周期。

有关软件流水线的更多信息，请参阅《*TMS320C6000 程序员指南*》。

4.6.1 关闭软件流水线（--disable_software_pipeline 选项）

在优化级别 --opt_level=2（即 -O2）和 -O3 上，编译器试图对循环进行软件流水线处理。出于调试原因，您可能不希望您的循环被软件流水线化。由于代码不是按顺序显示的，因此软件流水线循环有时难以调试。--disable_software_pipeline 选项同时影响已编译的 C/C++ 代码和汇编优化代码。

备注

软件流水线可能会增加代码大小

不使用 SPLOOP 的软件流水线可能会显著增加代码大小。为了控制软件流水线循环的代码大小，建议使用 --opt_for_space 选项而不是 --disable_software_pipeline 选项。--opt_for_space 选项能够在必要时禁用非 SPLOOP 软件流水线以节省代码大小，但它不会影响 SPLOOP 功能（节 4.8）。SPLOOP 不会显著增加代码大小，但可以大大加快循环速度。使用 --disable_software_pipeline 选项会禁用所有软件流水线，包括 SPLOOP。

4.6.2 软件流水线信息

编译器将软件流水线循环信息嵌入到 .asm 文件中。此信息用于优化 C/C++ 代码或线性汇编代码。

在循环之前，软件流水线信息以注释的形式出现在 .asm 文件中，对于汇编优化器，该信息在工具运行时显示。示例 4-1 展示了为每个循环生成的信息。

--debug_software_pipeline 选项添加了附加信息（该附加信息显示循环内核每个周期的寄存器使用情况），并显示软件流水线循环的单次迭代的指令顺序。

备注

有关软件流水线信息的更多详情

请参阅《*TMS320C6000 编程人员指南*》，详细了解每个循环之前“软件流水线信息”注释块中可能出现的信息和消息。

示例 4-1. 软件流水线信息

```
;*--------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Known Minimum Trip Count         : 2
;*      Known Maximum Trip Count         : 2
;*      Known Max Trip Count Factor      : 2
;*      Loop Carried Dependency Bound(^) : 4
;*      Unpartitioned Resource Bound     : 4
;*      Partitioned Resource Bound(*)    : 5
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     2        3
;*      .S units                     4        4
;*      .D units                     1        0
;*      .M units                     0        0
;*      .X cross paths               1        3
;*      .T address paths             1        0
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        1     (.L or .S unit)
;*      Addition ops (.LSD)          6        3     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             3        4
;*      Bound(.L .S .D .LS .LSD)     5*       4
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 5  Register is live too long
;*         ii = 6  Did not find schedule
;*         ii = 7  Schedule found with 3 iterations in parallel
;*      done
;*
;*      Epilog not entirely removed
;*      Collapsed epilog stages     : 1
;*
;*      Prolog not removed
;*      Collapsed prolog stages     : 0
;*
;*      Minimum required memory pad : 2 bytes
;*
;*      Minimum safe trip count     : 2
;*
;*--------------------------------------------------------------------------*
```

4.6.2.1 软件流水线信息术语

软件流水线信息中出现下述定义的术语。有关每个术语的更多信息，请参阅《*TMS320C6000 程序员指南*》。

- **循环展开因子。**专为提高性能而展开的循环的次数。
- **已知最小循环计数。**循环执行的最小次数。
- **已知最大循环计数。**循环执行的最大次数。
- **已知最大循环计数因子。**总能平均划分循环计数的因子。此信息可用于在可能的情况下展开循环。
- **循环标签。**在线性汇编输入文件中为循环指定的标签。C/C++ 代码不存在此字段。
- **循环携带依赖限制。**最大循环携带路径的距离。当循环的一次迭代写入必须在未来迭代中读取的值时，便会出现循环携带路径。作为循环携带限制的一部分的指令用 ^ 符号标记。
- **启动间隔** (ii)。循环的连续迭代开始之间的周期数。启动间隔越小，执行循环所需的周期就越少。
- **资源限制。**使用最多的资源限制了最小启动间隔。如果 4 条指令需要一个 .D 单元，则它们至少需要两个周期来执行（4 条指令/2 个并行 .D 单元）。
- **未分区资源限制。**在循环中的指令被划分到特定的一侧之前的最佳资源限制值。
- **分区资源限制 (*)。**在指令被划分之后的资源限制值。
- **资源分区。**下表总结了如何划分指令。此信息可用于在编写线性程序集时帮助分配功能单元。每个表条目都有对于 A 侧和 B 侧功能单元的值。星号用于标记那些决定资源限制值的条目。表格条目代表以下术语：
  - **.L 单元**是需要 .L 单元的指令总数。
  - **.S 单元**是需要 .S 单元的指令总数。
  - **.D 单元**是需要 .D 单元的指令总数。

  - **.M 单元**是需要 .M 单元的指令总数。
  - **.X 交叉路径**是 .X 交叉路径的总数。
  - **.T 地址路径**是地址路径的总数。
  - **长读取路径**是长读取端口路径的总数。
  - **长写入路径**是长写入端口路径的总数。
  - **逻辑运算 (.LS)** 是可以使用 .L 或 .S 单元的指令总数。
  - **加法运算 (.LSD)** 是可以使用 .L、.S 或 .D 单元的指令总数。
- **Bound(.L .S .LS)。**由使用 .L 和 .S 单元的指令数确定的资源限制值。其计算公式如下：

  Bound(.L .S .LS) = ceil((.L + .S + .LS) / 2)
- **Bound(.L .S .D .LS .LSD)**。由使用 .D、.L 和 .S 单元的指令数确定的资源限制值。其计算公式如下：

  Bound(.L .S .D .LS .LSD) = ceil((.L + .S + .D + .LS + .LSD) / 3)
- **最低要求的内存垫。**启用推测执行时读取的字节数。有关更多信息，请参阅节 4.6.3。

4.6.2.2 不符合软件流水线的循环的消息循环

如果循环完全不符合软件流水线的要求，则会显示以下消息：

- 错误的循环结构。此错误非常罕见，可能源于以下原因：
  - 在 C 代码内循环中插入了 asm 语句
  - 被用作线性汇编优化器的输入的并行指令
  - 复杂的控制流，例如 GOTO 语句和中断
- 循环包含调用。有时，编译器可能无法内联处于循环中的函数调用。由于编译器无法内联函数调用，因此无法对循环进行软件流水化处理。
- 太多指令。到软件流水线的循环中有太多指令。
- 禁用软件流水线。软件流水已被命令行选项禁用，例如，当使用 --disable_software_pipeline 选项、不使用 --opt_level=2（或 -O2）或 --opt_level=3（或 -O3）选项或使用 --opt_for_space=2 或 --opt_for_space=3 选项时。
- 未初始化行程计数器。行程计数器可能未设置为初始值。
- 抑制以防止代码扩展。由于 --opt_for_space=1 选项，软件流水线可能被抑制。当使用 --opt_for_space=1 选项时，在不太乐观的情况下会禁用软件流水线以减小代码大小。如需启用流水线，请使用 --opt_for_space=0 或完全省略 --opt_for_space 选项。
- 循环携带的依赖性约束太大。如果循环具有复杂的循环控制，请根据节 4.6.3.2 中的建议尝试使用 --speculate_loads。
- 无法识别跳变计数器。循环行程计数器无法被识别或在循环体中使用不当。

4.6.2.3 流水线故障消息

当编译器或汇编优化器正在处理软件流水线并且失败时，可能会出现以下消息：

- 地址增量过大。必须调整地址寄存器的偏移量，因为该偏移量超出了 C6000 的偏移寻址模式的范围。必须最小化地址寄存器偏移量。
- 不能分配机器寄存器。找到了软件流水线调度，但其不能为该调度分配机器寄存器。简化循环可能会有所帮助。

  显示在给定的 ii 中找到的调度的寄存器使用情况。在编写线性汇编时可以使用此信息，以平衡寄存器文件两侧的寄存器压力。例如：

```
ii = 11 Cannot allocate machine registers
Regs Live Always : 3/0  (A/B-side)
Max Regs Live : 20/14
Max Condo Regs Live : 2/1
```

  - **寄存器始终活跃中**在整个循环体的持续时间内必须分配给寄存器的值的数量。这意味着，这些值必须始终被分配给寄存器，用于为循环找到的任何给定调度。
  - **最大寄存器活跃中**循环中任何给定周期内必须分配给寄存器的最大值的数量。这表示所找到的调度所需的最大寄存器数。

  - **最大条件寄存器活跃中。**循环内核中任何给定周期内必须分配给条件寄存器的最大寄存器数量。
- **周期数太高。**从未收益有了编译器为循环找到的调度，使用非软件流水线版本会更有效。
- 没有找到调试编译器无法在给定的 ii（迭代间隔）中找到软件流水线的调度。应简化循环和/或消除循环携带依赖项。
- 并行迭代**> 最小或最大循环计数**.找到了软件流水线调度，但该调度的并行迭代次数多于最小或最大循环行程计数。必须启用冗余循环或传递行程信息。
- 超出推测阀值有必要大胆地加载超出当前由 --speculate_loads 选项指定的阈值。必须按照位于汇编文件中的软件流水线反馈中的建议增大 --speculate_loads 阈值。
- 寄存器活跃太久寄存器必须具有能存在（活跃）超过 ii 个周期的值。可以插入 MV 指令来分割太长的寄存器寿命。

  如果正在使用汇编优化器，则在此失败消息之后会列出用于定义和使用活跃太久的寄存器的指令所在的 .sa 文件行号。例如：

```
    ii = 9  Register is live too long
    |10| -> |17|
```

  这表示定义寄存器值的指令位于 .sa 文件中的第 10 行，使用寄存器值的指令位于第 17 行。
- 太多的谓词活跃在一侧上 C6000 具有可用于条件指令的谓词或条件寄存器。有六个谓词寄存器。A 侧有三个，B 侧有三个。有时，特定的分区和调度组合需要的寄存器不止这些。
- N 次迭代并行下找到的调试（这不是失败消息。）在并行执行 N 个迭代的情况下，找到软件流水线调度。
- 循环中使用的行程变量-不能调整循环计数循环行程计数器在循环中除了作为循环行程计数器之外还有其他用途。
- 对不规则循环的不安全调试“不规则”循环是具有已知迭代次数的非倒计数循环，例如 while 循环。不规则循环可能要求转换指令执行的次数多于循环所要求的次数。此错误表示编译器无法找到一个具有可以安全地过度执行、使用谓词进行保护或在循环后撤消其影响的指令的调度。尝试将循环重写为倒计数循环。还可以尝试增加 --speculate_loads (-mh) 选项。

4.6.2.4 由 --debug_software_pipeline 选项生成寄存器使用表

--debug_software_pipeline 选项在生成的汇编文件中放置额外的软件流水线反馈。该信息包括软件流水线循环的单个调度迭代视图。

如果给定循环的软件流水线成功，并且在编译过程中使用了 --debug_software_pipeline 选项，那么将在生成的汇编代码中的软件流水线信息注释块中添加寄存器使用表。

每行上的数字代表循环内核中的周期号。

每一列代表 TMS320C6000 上的一个寄存器。寄存器标记在寄存器使用表的前三行，应逐列读取。

表条目中的 * 表示列标头所指示的寄存器位于内核执行数据包中，内核执行包由标记每一行的周期号指示。

寄存器使用表的一个示例如下：

```
;*      Searching for software pipeline schedule at
;*      ii = 15 Schedule found with 2 iterations in parallel
;*
;*      Register Usage Table:
;*          +----------------------------------+
;*          |AAAAAAAAAAAAAAAA|BBBBBBBBBBBBBBBB|
;*          |0000000000111111|0000000000111111|
;*          |0123456789012345|0123456789012345|
;*          |----------------+----------------|
;*       0: |***   ****      |*** ******      |
;*       1: |****  ****      |*** ******      |
;*       2: |****  ****      |*** ******      |
;*       3: | **  *****      |*** ******      |
;*       4: | **  *****      |*** ******      |
;*       5: | **  *****      |*** ******      |
;*       6: | **  *****      |**********      |
;*       7: |***  *****      |** *******      |
;*       8: |**** *****      |***********     |
;*       9: |**********      |** ********     |
;*      10: |***********     |** *********    |
;*      11: |***********     |** *********    |
;*      12: |**********      |***********    |
;*      13: |**** *****      |** ******* *    |
;*      14: |***  *****      |*** ****** *    |
;*          +----------------------------------+
;*
```

此示例显示在循环内核的第 0 个周期（第一个执行数据包）上，寄存器 A0、A1、A2、A6、A7、A8、A9、B0、B1、B2、B4、B5、B6、B7、B8 和 B9 在这个周期内全都处于激活状态。

4.6.3 折叠 序言和结语以改善性能和代码大小

当循环为软件流水线时，通常需要序言和结语。序言用于启动循环，结语用于结束循环。

通常，循环必须执行最少次数的迭代才能安全地执行软件流水线版本。如果已知的最小循环计数太小，则添加冗余循环或禁用软件流水线。折叠循环的序言和结语可以减少安全执行流水线循环所需的最小循环计数。

折叠还可以大大缩减代码大小。这种代码大小增长的部分原因是冗余循环。其余原因是由于序言和结语。

软件流水线循环的 序言和结语最多由 长度为 *ii* 的 *p*-1 个阶段组成，其中 *p* 是稳态期间并行执行的迭代次数，*ii* 是流水线循环体的循环时间。在序言和结语折叠期间，编译器会试图折叠尽可能多的阶段。但是，过度折叠会对性能产生负面影响。因此，默认情况下，编译器会试图在不牺牲性能的情况下折叠尽可能多的阶段。当--opt_for_space=2 或 --opt_for_space=3 选项被调用时，编译器越来越倾向于优化代码大小而不是性能。

4.6.3.1 推测执行

当 prolog 和 epilog 被折叠时，指令可能被推测执行，从而导致加载超出循环中显式读取的范围两端的地址。默认情况下，编译器无法推测加载，因为这可能会导致读取非法的内存位置。有时，编译器可以预测这些加载以防止过度执行。然而，这会增加寄存器的压力，并可能减少可以执行的折叠总量。

使用 --speculate_loads=*n* 选项时，推测阈值从默认值 0 增加到 *n*。当阈值为 *n* 时，编译器可以允许推测性地执行加载，因为它读取的内存位置在循环内显式读取的某个位置之前或之后不超过 *n* 个字节。如果省略 *n*，编译器会假定推测阈值是无限的。如需在 Code Composer Studio 中指定此值，请选中“Speculate Threshold”复选框，并将编译器选项的高级类别上的选项构建对话框中的文本框留空。

折叠序言和结语通常可以减少最小安全行程计数。如果已知的最小行程计数小于最小安全行程计数，则需要一个冗余循环。否则，必须抑制流水线。下述两个值都可以在软件流水线循环之前的注释块中找到。

```
;*Known Minimum Trip Count: 1
....
;*Minimum safe trip count: 7
```

如果最小安全行程计数大于已知的最小行程计数，强烈建议使用 --speculate_loads，不仅是为了代码大小，也是为了性能。

使用 --speculate_loads 时，必须确保潜在推测的加载不会导致非法读取。这可以通过在两个方向上根据需要使用内存垫填充数据段和/或栈来实现。给定的软件流水线循环所需的内存垫也会在该循环的注释块中提供。

```
;*Minimum required memory pad: 8 字节
```

4.6.3.2 选择最佳阈值

当循环是软件流水线循环时，循环前的注释块提供以下信息：

- 此循环需要内存填充
- 实现此软件流水线调度和折叠级别所需的最小值 *n*
- 建议使用更大的 *n* 值，以允许额外的折叠

此信息在注释块中显示如下：

```
;*Minimum required memory pad : 5 bytes
;*Minimum threshold value     : --speculate_loads=7
;*
;*For further improvement on this loop, try option --speculate_loads=14
```

为了安全起见，此示例循环要求在此循环中引用的数组数据前后至少填充 5 个字节。此填充可以包含其他程序数据。此填充不会被修改。在许多情况下，阈值 (即，实现特定调度和级别折叠所需的 --speculate_loads 参数的最小值) 与填充相同。但是，如果不同，注释块也会包含最小阈值。在这种循环中，阈值必须至少为 7 才能实现这种级别的折叠。

编译器和链接器可以通过 --speculate_loads 选项的 *auto* 参数 (即 *--speculate_loads=auto* 或 *-mh=auto*) 提供自动加载推测。通过 *auto* 参数可以更轻松地使用推测加载优化并从中受益。此选项可以生成最多 256 字节的推测加载，超出可以为编译器分配的内存。

此外，编译器将信息传递给链接器以帮助自动确保所需的前置填充和后置填充：

- 如果在编译时已知推测加载缓冲区的符号，则链接器将确保符号指向的对象具有所需的填充，以便推测加载访问合法内存。
- 如果在编译期间不知道符号信息，则链接器将确保数据段的放置方式允许合法地访问超出数据段边界的数据。链接器通过简单地填充数据段所在的内存范围的开头和结尾来实现该目标。

但是，也可以通过 *--speculate_loads=n* 选项明确指定推测加载阈值，其中 *n* 至少是所需的最小填充 (如前所述)，但还需要考虑更大的阈值是否有助于进一步折叠。如果适用，还会提供此信息。例如，在上述注释块中，阈值 14 可能有助于进一步折叠。如果为 --speculate_loads 选择 *auto* 参数，编译器将自动考虑更大的阈值。

4.7 冗余循环

在循环终止之前，循环会迭代一定的次数。迭代次数称为*循环计数*。用于计算迭代次数的变量是*循环计数器*。当行程计数器达到等于循环计数的限制时，循环终止。代码生成工具使用计数来确定循环是否可以流水线化。软件流水线循环的结构要求执行最少的循环迭代次数（最小循环计数）即可装填或准备流水线。

软件流水线循环的最小循环计数由并行执行的迭代数量设置。在图 4-1 中，最小循环计数为 5。在下述示例中，A、B 和 C 是软件流水线中的指令，因此该单周期软件流水线循环的最小循环计数为 3。

```
A
B     A
C     B     A     ←三个并行迭代 = 最小循环计数
      C     B
            C
```

当代码生成工具无法确定循环的循环计数时，默认情况下会生成两个循环和控制逻辑。第一个循环不是流水线循环，如果运行时循环计数小于循环的最小安全循环计数，则会执行该循环。第二个循环是软件流水线循环，如果运行时循环计数大于或等于最小循环计数，则会执行该循环。在任意给定时间，其中一个循环是*冗余循环*。例如：

```
foo(N)  /* N 是循环计数 */
{
   for (I=0; I < N; I++) /* I 是循环计数 */
}
```

找到循环的软件流水线后，编译器将 foo() 转换如下，假定循环的最小循环计数为 3。随后将生成两个版本的循环，并使用以下比较来确定应执行哪个版本：

```
foo(N)
{
   if (N < 3)
   {
      for (I=0; I < N; I++)    /* 非流水线版本 */
   }
   else
   }
      for (I=0; I < N; I++)    /* 流水线版本    */
   }
}
foo(50); /* 执行软件流水线循环           */
foo(2);  /* 执行循环（非流水线）*/
```

可以使用 --program_level_compile --opt_level=3（请参阅节 4.4）或使用 MUST_ITERATE pragma（请参阅节 7.9.22）来帮助编译器避免产生冗余循环。

备注

关闭冗余循环：指定任何 --opt_for_space 选项都会关闭冗余循环。

4.8 通过 SPLOOP 使用循环缓冲区

循环缓冲区提高了软件流水线循环的性能并减小其代码大小。循环缓冲区具有以下优点：

- 代码大小。单次循环迭代存储在程序存储器中。
- 中断延迟。在循环缓冲区外执行的循环是可中断的。
- 提高具有未知循环计数的循环的性能并消除冗余循环。
- 减少或消除对推测加载的需求。
- 减少功耗。

在软件流水线循环的开头找到 SPLOOP(D/W) 而在结尾找到 SPKERNEL 时，便可以判定编译器正在使用循环缓冲区。有关 SPLOOP 相关的信息，请参阅《*TMS320C64x/C64x+ CPU 和指令集参考指南*》。

当不使用 --opt_for_space 选项时，如果编译器可以在没有循环缓冲区的情况下找到更快的软件流水线循环，则编译器将不会使用循环缓冲区。使用 --opt_for_space 选项时，编译器将尽可能使用循环缓冲区。

发生以下任一情况时，编译器不会为循环缓冲区 (SPLOOP/D/W) 生成代码：

- ii（启动间隔）> 14 个周期
- （单次迭代的）动态长度 > 48 个周期
- 优化器完全展开循环
- 代码包含使正常软件流水线不合格的元素（循环中的调用、循环中的复杂控制代码等）。有关更多信息，请参阅《*TMS320C6000 程序员指南*》。

4.9 减小代码大小（--opt_for_space（或 -ms）选项）

使用 --opt_level=*n* 选项（或 -O*n*）就是在通知编译器优化您的代码。*n* 的值越高，编译器在优化代码方面投入的精力就越多。但是，可能仍需要告知编译器您的优化优先级是什么。默认情况下，当指定 --opt_level=2 或 -opt_level=3 时，编译器主要优化性能。（在较低的优化级别下，优先优化的是编译时间和调试难度。）可以使用代码大小标志 --opt_for_space=*n* 来调整性能和代码大小之间的优先级。对于 --opt_for_space=0、--opt_for_space=1、--opt_for_space=2 和 --opt_for_space=3 选项越来越倾向于代码大小而不是性能。

当同时指定 --silicon_version=6400+ 与 --opt_for_space 选项时，代码将根据压缩进行调整。也就是说，需要调整更多的指令，以便在汇编时更有可能从 32 位指令转换为 16 位指令。

建议不要将代码大小标志用于性能最关键的代码。除了性能最关键的代码外，建议对所有代码使用 --opt_for_space=0 或 --opt_for_space=1。对于很少执行的代码，建议使用 --opt_for_space=2 或 --opt_for_space=3。如果需要最小代码大小，则应使用 --opt_for_space=2 或 --opt_for_space=3。通常建议将代码大小标志与 --opt_level=2 或 --opt_level=3 结合使用。

备注

禁用代码大小优化或降低优化级别

如果降低优化和/或不使用代码大小标志，则会禁用代码大小优化并牺牲性能。

备注

--opt_for_space 选项等效于 --opt_for_space=0

如果在未指定代码大小级别编号的情况下使用 --opt_for_space，则选项级别默认为 --opt_for_space=0。

4.10 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。

4.10.1 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。

4.10.1.1 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 pdd6x 的工具对生成的 PDAT 文件进行后处理。pdd6x 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 4.10.2），供反馈制导优化的第 2 阶段使用。

4.10.1.2 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。

4.10.1.3 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 pdd6x 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。

4.10.1.4 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。

```
cl6x -mv6400+ --opt_level=2 --gen_profile_info foo.c --run_linker --output_file=foo.out
     --library=lnk.cmd --library=rts64plus.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 pdd6x 以创建与--use_profile_info 一起使用的分析信息（PRF）文件。

```
pdd6x -e foo.out -o pprofout.prf pprofout.pdat
```

4. 使用分析反馈文件重新编译。

```
cl6x -mv6400+ --opt_level=2 --use_profile_info=pprofout.prf foo.c --run_linker
     --output_file=foo.out --library=lnk.cmd --library=rts64plus.lib
```

4.10.1.5 .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。

4.10.1.6 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。

4.10.1.7 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 --opt_for_space (-ms) 代码大小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。

4.10.1.8 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。

4.10.2 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 pdd6x 的工具，该工具用于对分析数据 (PDAT) 文件进行后处理。pdd6x 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 pdd6x 的讨论，请参阅节 4.10.1。使用以下语法调用 pdd6x 工具：

pdd6x -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 pdd6x 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，pdd6x 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 pdd6x 才能理解这种格式。不应以任何方式修改此文件。

4.10.3 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。

4.10.4 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

4.11 使用配置文件信息获得更好的程序缓存布局并分析代码覆盖率

可从路径分析器获得两种不同类型的分析信息：代码覆盖率信息和调用图信息。

程序缓存布局工具可帮助您在应用程序中实现更高的程序指令缓存效率。程序缓存布局是控制代码段在内存中的相对位置的工艺，旨在最大程度地减少程序指令缓存中发生冲突缺失的情况。

4.11.1 背景和动机

有效利用程序指令缓存是从 C6000 获得最佳性能的重要因素。专用程序指令缓存 (L1P) 提供了快速指令获功能，但*缓存缺失*的代价可能非常高。某些应用程序（例如 h264）会因为 L1P 缓存缺失而花费 30% 以上的处理器时间。当未能从 L1P 读取指令，并且进程需要从下一级内存访问指令时，就会发生缓存缺失。对 L2 或外部存储器的请求比从 L1P 访问的延迟更长。

谨慎放置代码段可以大幅减少缓存缺失的次数。C6000 L1P 对代码放置特别敏感，因为它是*直接映射*的。

许多 L1P 缓存缺失都是*冲突缺失*。当缓存最近逐出了某个代码块但随后又再次需要该代码块时，就会发生冲突缺失。在程序指令缓存中，当两个经常执行的代码块（通常来自不同的函数）交错执行并映射到相同的缓存行时，通常会发生这种情况。

例如，假设在函数 A 的循环内部调用了函数 B，再假设函数 A 的循环代码被映射到与每次调用 B 时都会执行的函数 B 的代码块相同的缓存行中。每次从该循环中调用函数 B 时，函数 A 中的循环代码都会被映射到同一缓存行的函数 B 中的代码从缓存中逐出。更糟糕的是，当 B 返回到 A 时，A 中的循环代码将映射到同一缓存行的函数 B 中的代码逐出。

每次循环迭代都会导致两次程序指令缓存冲缺失。如果循环被多次遍历，则因程序指令缓存停顿而丢失的处理器周期数会变得非常大。

通过更智能地放置同时处于活动状态的函数，可以避免许多程序指令缓存缺失。使用代码放置策略可以显著地提高程序指令缓存效率。这些策略利用检测应用程序运行期间收集的动态配置文件信息。

程序缓存布局工具 (clt6x) 以加权调用图的形式获取动态配置文件信息，然后创建可作为链接器输入的首选函数顺序命令文件，以指导函数子段的放置。

可以使用程序缓存布局工具帮助改善程序的局域性，并减少应用程序运行期间发生的 L1P 缓存冲突缺失的次数，从而提高应用程序的性能。

4.11.2 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。

4.11.2.1 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 pdd6x 的工具对生成的 PDAT 文件进行后处理。pdd6x 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 4.10.2），供反馈制导优化的第 2 阶段使用。

4.11.2.2 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：

```
cl6x --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/c64p/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。

4.11.3 您期待看到哪些性能改进？

如果您的应用程序没有受到 L1P 缓存使用效率低下的影响，那么程序缓存布局功能不会对应用程序的性能产生任何影响。在将程序缓存布局工具应用于您的应用程序之前，请分析应用程序中的 L1P 缓存性能。

4.11.3.1 评估 L1P 缓存性能

评估应用程序的 L1P 缓存使用效率不仅有助于确定应用程序是否可以从使用程序缓存布局中受益，而且还可以粗略估计应用程序缓存布局能够合理提高多少性能。

有几种资源有助于评估应用程序中的 L1P 缓存使用情况。一种方法是使用 Code Composer Studio (CCS) 中的函数分析功能。

由于 L1P 缓存缺失而导致的 CPU 停顿周期数可以合理地估算出能够在应用程序中使用程序缓存布局工具来恢复的 CPU 周期数的上限值。请注意，程序缓存布局对性能的影响会因应用程序中运行的不同数据集而有所不同。

4.11.4 程序缓存布局相关的特征和功能

代码生成工具提供了一些可与程序缓存布局工具 clt6x 结合使用的特征和功能。这些特征和功能综述如下：

4.11.4.1 路径分析器

代码生成工具包括路径分析实用程序 pprof6x，该程序是从编译器 cl6x 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 pprof6x 实用程序：

cl6x --gen_profile ... *file.c*
cl6x --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 4.10。

4.11.4.2 分析选项

路径分析实用程序 pprof6x 将代码覆盖率或加权调用图分析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（无论是代码覆盖率还是加权调用图分析）。如果尝试在同一输出 CSV 文件中混合代码覆盖率和加权调用图分析信息的行为被检测到，则 pprof6x 将发出致命错误并中止。

--analyze=callgraph	指示编译器生成加权调用图分析信息。
--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。

4.11.4.3 环境变量

为了协助管理输出 CSV 分析文件，pprof6x 支持以下环境变量：

TI_WCGDATA	允许为所有加权调用图分析信息指定单个输出 CSV 文件。如果文件已存在，新信息将附加到此环境变量标识的 CSV 文件中。
TI_ANALYSIS_DIR	指定输出分析文件将在其内生成的目录。同一个环境变量可同时用于代码覆盖率信息和加权调用图信息（pprof6x 生成的所有分析文件都将写入 TI_ANALYSIS_DIR 环境变量指定的目录）。

备注

TI_COVDIR 环境变量

在生成代码覆盖率分析时，仍然支持现有的 TI_COVDIR 环境变量，但当存在已定义的 TI_ANALYSIS_DIR 环境变量时会覆盖 TI_COVDIR 环境变量。

4.11.4.4 程序缓存布局工具 clt6x

程序缓存布局工具根据输入加权调用图 (WCG) 信息创建了首选函数顺序命令文件。语法为：

clt6x *CSV files with WCG info* **-o forder.cmd**

4.11.4.5 连接器

编译器确定函数放置的相对优先顺序的依据是调用链接器期间遇到的 --preferred_order 选项的顺序。语法为：

--preferred_order= *function specification*

4.11.4.6 链接器命令文件运算符 unordered()

新的链接器命令文件关键字 unordered 放宽了放置在输出段上的约束，其规范包括显式列表，该显式列表显示哪些输入段包含在输出段。语法为：

unordered()

4.11.5 程序指令缓存布局开发流程

一旦确定您的应用程序正在低效率地使用程序指令缓存时，可以选择在您的开发中包含程序缓存布局工具，以试图恢复一些由于程序指令缓存冲突缺失而停止的 CPU 周期。

4.11.5.1 收集动态分析信息

程序缓存布局工具 clt6x 采用加权调用图形式的动态分析信息来生成首选函数顺序命令文件。该文件可用于在重新创建应用程序的过程中在链接时引导函数布置。

有几种方式可以收集该动态分析信息。例如，如果在硬件上运行应用程序，也许可以收集 PC 间断跟踪。然后可以对间断跟踪进行后处理，从而构建 clt6x 的加权调用图输入信息。

此处介绍的用于收集动态分析信息的方法依赖于 C6000 代码生成工具中的路径分析功能。具体的工作原理如下：

1. 使用 --gen_profile_info 选项创建检测的应用程序。

   使用 --gen_profile_info 指示编译器沿着每个函数的执行路径将计数器嵌入到代码中。

   只需编译，请使用：

   cl6x *options* **--gen_profile_info** *src_file(s)*

   要编译和链接，请使用：

   cl6x *options* **--gen_profile_info** *src_file(s)* **-run_linker --library** *lnk.cmd*

2. 运行检测的应用程序以生成 .pdat 文件。

   当应用程序运行时，通过 --gen_profile_info 嵌入应用程序的计数器会跟踪通过函数遍历特定执行路径的次数。在这些计数器中收集的数据写入名为 pprofout.pdat 的分析数据文件。

   自动生成分析数据文件。

3. 解码分析数据文件。

   一旦有了分析数据文件后，该文件将通过分析数据解码器工具 pdd6x 进行解码，如下所示：

   pdd6x -e= *instrumented app out file* **-o=pprofout.prf pprofout.pdat**

   使用 pdd6x 生成 .prf 文件，然后将其输入到应用程序的重新编译中。该应用程序使用分析信息生成加权调用图输入数据。

4. 使用解码的分析信息生成加权调用图输入。

   --analyze 编译器选项能行编译器生成加权调用图或代码覆盖率分析信息。相应的语法如下：

--analyze=callgraph	指示编译器生成加权调用图信息。
--analyze=codecov	指示编译器生成代码覆盖率信息。此选项替换了先前的 --codecov 选项。

   编译器还支持新的 --analyze_only 选项，该选项指示编译器在生成分析信息后停止编译。此选项替换了先前的 --onlycodecov 选项。

   要利用收集到的动态分析信息，请使用 --analyze=callgraph 选项和 --use_profile_info 选项重新编译应用程序的源代码：

   cl6x *options* **-mo --analyze=callgraph --use_profile_info=pprofout.prf** *src_file(s)*

   使用 -mo 指示编译器为每个函数生成代码到自己的子段中。此选项为链接器提供了直接控制给定函数的代码位置的方法。

   编译器生成 CSV 文件，该文件包含在命令行上指定的每个源文件的加权调用图信息。如果这样的 CSV 文件已经存在，则新的调用图分析信息将附加到现有的 CSV 文件中。然后这些 CSV 文件将输入到缓存布局工具 (clt6x) 中，为应用程序生成首选函数顺序命令文件。

   更多有关编译器生成的 CSV 文件（包含加权调用图信息）内容的详细信息，请参阅节 4.11.6。

4.11.5.2 从动态分析信息中生成首选功能顺序

此时，编译器已经为重新编译应用程序的命令行中指定的每个 C/C++ 源文件生成了一个 CSV 文件。每个 CSV 文件都包含有关 C/C++ 源文件中定义的每个函数中所有调用点的加权调用图信息。

程序缓存布局工具 clt6x 将这些 CSV 文件中的所有加权调用图信息收集到单个合并的加权调用图中。对加权调用图进行处理生成首选函数顺序命令文件。将该文件输入链接器中以指导应用程序源文件中定义的函数的放置位置。clt6x 的语法如下所示：

```
clt6x *.csv -o forder.cmd
```

clt6x 的输出是一个包含一系列 **--preferred_order=** *function specification* 选项的文本文件。默认情况下，输出文件的名称为 forder.cmd，但您可以使用 -o 选项指定自己的文件名。函数在此文件中出现的顺序是由 clt6x 确定的首选函数顺序。

通常，在首选函数顺序列表中，一个函数与另一个函数的接近程度反映了这两个函数相互调用的频率。如果列表中的两个函数彼此非常接近，则链接器会将此情况解释为建议将这两个函数放置在非常靠近的位置。当两个函数同时处于活动状态时，放在一起的函数不太可能在运行时出现创建缓存冲突缺失。总体效果应该是程序指令缓存效率和性能都得到提高。

4.11.5.3 在重新构建的应用程序中使用首选函数顺序

最后，在重新构建应用程序期间，由 clt6x 生成的首选函数顺序命令文件会馈送到链接器中，如下所示：

```
cl6x options --run_linker *.obj forder.cmd -l lnk.cmd
```

首选函数顺序命令文件 forder.cmd 包含 --preferred_order=*function specification* 选项列表。链接器按照 --preferred_order 选项在链接器调用期间遇到的顺序，对函数的相对位置进行优先排序。

每个 --preferred_order 选项都包含一个函数规格。函数规格可以简单地描述全局函数的函数名称，也可以提供定义函数的路径名和源文件名。包含路径和文件名信息的函数规格用于区分具有相同函数名的静态函数。

--preferred_order 选项由链接器解释为指导函数相对于彼此放置的建议。它们不是明确的放置指令。如果在链接器命令文件 SECTIONS 指令中明确提及了某个目标文件或输入段，则链接器命令文件中指定的放置指令优先于来自--preferred_order 选项的任何建议，该选项与目标文件或输入段中定义的函数相关联。

如节 4.11.7 所述，通过将 unordered() 运算符应用于输出规格，可以放宽此优先级。

4.11.6 带有加权调用图 (WCG) 信息的逗号分隔值 (CSV) 文件

编译器在 **--analyze=callgraph --use_profile_info** 选项组合下生成的 CSV 文件格式如下：

```
"caller","callee","weight" [CR][LF]
caller spec,callee spec,call frequency [CR][LF]
caller spec,callee spec,call frequency [CR][LF]
caller spec,callee spec,call frequency [CR][LF]
...
```

请牢记以下要点：

- CSV 文件的第 1 行是标题行。该行指定 CSV 文件剩余部分的每一行中每个字段的含义。如果 CSV 文件包含加权调用图信息，每行都将有一个调用方函数规范，后跟一个被调用方函数规范，再跟一个无符号整数，该整数指定在运行时执行调用的次数。
- 在 CSV 文件中，可能存在调用方和被调用方函数规范在多行中相同的情况。当调用方函数针对被调用方函数有多个调用点时，就会发生这种情况。在 clt6x 创建的合并加权调用图中，具有相同调用方和被调用方函数规范的每一行权重都将被加到一起。
- 编译器使用路径分析工具生成的 CSV 文件将不包含关于间接函数调用或对运行时支持辅助函数（如 _remi 或 _divi）的调用的信息。但是，也可以使用另一种方法（如前面提到的 PC 间断跟踪）收集有关此类调用的信息。
- 这些 CSV 文件的格式符合逗号分隔值 (CSV) 文件的 RFC-4180 规范。更多有关此规范的详细信息，请参阅 http://tools.ietf.org/html/rfc4180。

4.11.7 链接器命令文件运算符 - unordered()

可以在链接器命令文件中使用 unordered() 运算符。此运算符的作用是放宽放置在输出段规范上的约束，其中该规范明确地说明了输出段的内容。

请考虑输出段规格示例：

```
SECTIONS
{
  .text:
  {
    file.obj(.text:func_a)
    file.obj(.text:func_b)
    file.obj(.text:func_c)
    file.obj(.text:func_d)
    file.obj(.text:func_e)
    file.obj(.text:func_f)
    file.obj(.text:func_g)
    file.obj(.text:func_h)
    *(.text)
  } > PMEM
  ...
}
```

在此 SECTIONS 指令中，.text 的规范明确地说明了函数在输出段中的布局顺序。因此，默认情况下，链接器将完全按照规定的顺序对 func_a 到 func_h 进行布局，而不管任何其他放置优先级标准（例如由 --preferred_order 选项枚举的首选函数顺序列表）。

unordered() 运算符可用于放宽对“.text”输出段中函数放置的约束，以便可以由其他放置优先级标准决定放置。

unordered() 运算符可以应用于输出段，如示例 4-2 所示。

示例 4-2. unordered() 运算符的输出段

```
SECTIONS
{
  .text: unordered()
  {
    file.obj(.text:func_a)
    file.obj(.text:func_b)
    file.obj(.text:func_c)
    file.obj(.text:func_d)
    file.obj(.text:func_e)
    file.obj(.text:func_f)
    file.obj(.text:func_g)
    file.obj(.text:func_h)
    *(.text)
  } > PMEM
  ...
}
```

因此，鉴于此--preferred_order 选项列表：

- --preferred_order="func_g"
- --preferred_order="func_b"
- --preferred_order="func_d"
- --preferred_order="func_a"
- --preferred_order="func_c"
- --preferred_order="func_f"
- --preferred_order="func_h"
- --preferred_order="func_e"

.text 输出段中的函数放置将遵循此首选函数顺序列表。此放置将反映在链接器生成的映射文件中，如下所示：

示例 4-3. 为 示例 4-2 生成的链接器映射文件

```
SECTION ALLOCATION MAP
output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
.text       0    00000020    00000120
                  00000020    00000020     file.obj (.text:func_g:func_g)
                  00000040    00000020     file.obj (.text:func_b:func_b)
                  00000060    00000020     file.obj (.text:func_d:func_d)
                  00000080    00000020     file.obj (.text:func_a:func_a)
                  000000a0    00000020     file.obj (.text:func_c:func_c)
                  000000c0    00000020     file.obj (.text:func_f:func_f)
                  000000e0    00000020     file.obj (.text:func_h:func_h)
                  00000100    00000020     file.obj (.text:func_e:func_e)
```

4.11.7.1 关于 Dot (.) 出现 unordered() 的表达式

unordered() 运算符应该考虑的另一个方面是，即使该运算符使链接器放松由输出段内容的显式规格施加的约束，unordered() 运算符仍将慎重对待 dot (.) 表达式在此类规格中的位置。

请考虑示例 4-4 中的输出段规格

示例 4-4. 关于 a 位置表达式

```
SECTIONS
{
  .text: unordered()
  {
    file.obj(.text:func_a)
    file.obj(.text:func_b)
    file.obj(.text:func_c)
    file.obj(.text:func_d)
    .+= 0x100;
    file.obj(.text:func_e)
    file.obj(.text:func_f)
    file.obj(.text:func_g)
    file.obj(.text:func_h)
    *(.text)
  } > PMEM
  ...
}
```

在示例 4-4 中，点 (.) 表达式 “.+= 0x100;” 将输出段中两组函数的显式规格分隔开来。在这种情况下，链接器将遵循点 (.) 表达式相对于表达式任一侧函数的指定位置。也就是说，unordered() 运算符将允许由首选函数顺序列表指导 func_a 到 func_d 的相对位置，但这些函数都不会放置在由点 (.) 表达式创建的漏洞之后。同样，unordered() 运算符允许由首选函数顺序列表影响 func_e 到 func_h 的相对位置，但这些函数都不会放置在由点 (.) 表达式创建的漏洞之前。

4.11.7.2 GROUP 和 UNION

unordered() 运算符只能应用于输出段。这包括 GROUP 或 UNION 指令的成员。

示例 4-5. 将 unordered()应用于 GROUP

```
SECTIONS
{
  GROUP
  {
    .grp1:  {
      file.obj(.grp1:func_a)
      file.obj(.grp1:func_b)
      file.obj(.grp1:func_c)
      file.obj(.grp1:func_d)
    } unordered()
    .grp2:  {
      file.obj(.grp2:func_e)
      file.obj(.grp2:func_f)
      file.obj(.grp2:func_g)
      file.obj(.grp2:func_h)
    }
    .text:  { *(.text) }
  } > PMEM
  ...
}
```

示例 4-5 中的 SECTIONS 指令将 unordered() 运算符应用于 GROUP 的第一个成员。然后，.grp1 输出段布局会受到其他放置优先级标准（如首选函数顺序列表）的影响，而 .grp2 输出段将按照明确指定的方式进行布局。

unordered() 运算符不能应用于整个 GROUP 或 UNION。尝试这样做将导致链接器命令文件语法错误并中止链接。

4.11.8 注意事项

请牢记程序缓存布局开发流程的一些行为特征和限制：

- **生成路径分析数据文件 (.pdat)。**当运行一个收集路径分析数据的应用程序时（在构建期间使用 --gen_profile_info 编译器选项），该应用程序将使用运行时支持库中的函数将信息写入路径分析数据文件（上面教程中的 pprofout.pdat）。如果在应用程序开始运行时已经存在路径分析数据文件，则生成的任何新路径分析数据都将被附加到现有文件中。为防止组合应用程序单独运行产生的路径分析数据，需要重命名上次运行应用程序产生的路径分析数据文件，或者在再次运行应用程序之前删除该文件。
- **路径分析机制无法识别的间接调用。**当使用可用的路径分析机制从路径分析数据中收集加权调用图信息时，pprof6x 不能识别间接调用。间接调用点不会在由 pprof6x 生成的 CSV 输出文件中表示。可以通过在相关的 CSV 文件中引入有关间接调用站点的信息来解决这种限制。如果采用这种方式，请务必遵循调用图分析 CSV 文件的格式（“调用方”、“被调用方”、“调用频率”）。如果能够从 PC 跟踪中获得加权调用图信息到调用图分析 CSV 文件中，则此限制将不再适用（因为 PC 跟踪总是可以识别出间接调用的被调用方）。
- **与单个函数关联的多个 --preferred_order 选项。**在链接应用程序期间，可能需要向链接器输入多个首选函数顺序命令文件。例如，您可能已经为应用程序使用的一个或多个对象库开发或接收了单独的首选函数顺序命令文件。在这种情况下，可能会在多个首选函数顺序命令文件中指定一个函数。如果发生这种情况，链接器将仅接受指定函数的 --preferred_order 选项的第一个实例。

4.12 指示是否使用了某些别名技术

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进行优化。编译器谨慎执行其行为。

以下几节将介绍一些可能在代码中使用的别名技术。根据 ISO C 标准，这些技术是有效的，并被 C6000 编译器接受；但是，它们会阻止优化器对代码进行全面优化。

4.12.1 采用某些别名时使用 --aliased_variables 选项

在优化调用时，编译器假定，如果将局部变量的地址传递给函数，则该函数通过指针写入来更改局部变量。这使得局部变量的地址在返回后无法在其他地方使用。例如，被调用函数不能将局部变量的地址分配给全局变量，也不能返回局部变量的地址。

如果代码以这种方式使用别名并使用了优化，则必须使用 --aliased_variables 选项。例如，假设您的代码类似于以下代码，其中局部变量 x 的地址传递给函数 f()，该函数将 glob_ptr 别名设置为该地址并返回该地址。如果该示例要通过优化进行编译，则需要 --aliased_variables 选项，以便函数 f() 能够成功执行其操作。

```
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;
}
```

4.12.2 使用 --no_bad_aliases 选项来指示未采用这些技术

--no_bad_aliases 选项通知编译器可以对如何在代码中使用别名做出某些假设。这些假设允许编译器改进优化。--no_bad_aliases 选项还指定循环不变计数器的增量和减量是非零值。循环不变是指表达式的值在循环内不会改变。

- --no_bad_aliases 选项指示代码不使用节 4.12.1 中所述的别名技术。如果代码使用了该技术，*不要* 使用 --no_bad_aliases 选项。必须使用 --aliased_variables 选项进行编译。

  不要 将 --aliased_variables 选项与 --no_bad_aliases 选项一起使用。如果这样做，--no_bad_aliases 选项会覆盖 --aliased_variables 选项。
- --no_bad_aliases 选项指示指向字符类型的指针 *不会* 为另一种类型的对象设置别名（指向该对象）。也就是说，忽略 ISO 规范第 3.3 节中给出的这些类型的通用别名规则的特殊例外情形。如果有类似于以下示例的代码，请 *不要* 使用 --no_bad_aliases 选项：

  ```
  {
      long  l;
      char *p = (char *) &l;
      p[2] = 5;
  }
  ```

- --no_bad_aliases 选项指示如果 P 和 Q 这两个指针是运行时由同一调用激活的同一函数的不同参数，则对 P 和 Q 的间接引用就不是别名。如果有类似于以下示例的代码，请 *不要* 使用 --no_bad_aliases 选项：

  ```
  g(int j)
  {
      int a[20];
      f(&a, &a)         /* Bad */
      f(&a+42, &a+j)    /* Also Bad */
  ```

```
}
f(int *ptr1, int *ptr2)
{
   ...
}
```

- --no_bad_aliases 选项指示数组引用 A[E1]..[En] 中的每个下标表达式计算为小于相应声明数组边界的非负值。如果有类似于以下示例代码，请*不要* 使用 --no_bad_aliases：

```
static int ary[20][20];
int g()
{
    return f(5, -4); /* -4 is a negative index */
    return f(0, 96); /* 96 exceeds 20 as an index */
    return f(4, 16); /* This one is OK */
}
int f(int I, int j)
{
    return ary[i][j];
}
```

  在此示例中，ary[5][-4]、ary[0][96] 和 ary[4][16] 访问相同的内存位置。 --no_bad_aliases 选项只接受引用 ary[4][16] -，因为它的两个索引都在边界 (0..19) 内。
- --no_bad_aliases 选项指示循环计数器的循环不变计数器增量和减量是非零值。循环不变指表达式的值在循环内不会改变。

如果代码*不* 包含上述任何别名技术，那么应该使用 --no_bad_aliases 选项来提升代码的优化水平。但是，必须谨慎使用 --no_bad_aliases 选项；如果代码中出现了这些别名技术并且使用了 --no_bad_aliases 选项，则可能会出现意想不到的结果。

4.12.3 将 --no_bad_aliases 选项与汇编优化器一起使用

--no_bad_aliases 选项允许汇编优化器假设线性汇编代码中没有内存别名；即，没有内存引用相互依赖。但是，汇编优化器仍能识别使用 .mdep 指令指出的任何内存依赖项。有关 .mdep 指令的更多信息，请参阅节 5.4。

4.13 防止重新排列关联浮点运算

编译器可以自由地重新排列关联浮点运算。如果不希望编译器重新排列关联浮点运算，请使用 --fp_not_associative 选项。指定 --fp_not_associative 选项可能会降低性能。

4.14 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。

4.15 使用性能建议优化您的代码

您可以使用由编译器生成的性能建议来优化您的代码。为了获得性能建议，请使用以下选项进行编译：

--advice:performance	指示编译器发送建议到 stdout 中（默认）。
--advice:performance_file	指示编译器发送建议到文件中。
--advice:performance_dir	指示编译器发送建议到特定目录中的文件中。

备注

如果请求了建议文件，但又没有建议，则不会创建建议文件；相反，编译器会打印一条消息到 stdout：

```
"filename.c": advice #27004: 未生成性能建议
```

示例 1：下述示例发送输出建议到 stdout 中（默认）：

```
cl6x -mv6400+ -o2 -k --advice:performance func.c
```

```
"func.c", line 10: advice #30006: Loop at line 8 cannot be scheduled efficiently
   as it contains a function call ("_init").Try making "_init" an inline
   function.
"func.c", line 12: advice #30000: Loop at line 8 cannot be scheduled efficiently
   as it contains a function call ("_calculate").Try to inline call or
   consider rewriting loop.
```

请注意，防止“软件流水线不合格”的建议（如上文所示）也会打印在 .asm 文件中。因此，func.asm 将包含：

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*      Disqualified loop: Loop contains a call
;*          Loop at line 8 cannot be scheduled efficiently as it contains a
;*          function call ("_init").Try making "_init" an inline function.
;*      Disqualified loop: Loop contains non-pipelinable instructions
;*      Disqualified loop: Loop contains a call
;*          Loop at line 8 cannot be scheduled efficiently as it contains a
;*          function call ("_calculate").Try to inline call or consider
;*          rewriting loop.
;*      Disqualified loop: Loop contains non-pipelinable instructions
;*----------------------------------------------------------------------------*
```

示例 2：下述示例发送输出建议到名为 filename.advice 的文件中：

```
cl6x -mv6400+ --advice:performance --advice:performance_file=filename.advice func.c
```

```
;******************************************************************************
;* TMS320C6x C/C++ Codegen         Unix v7.5.0P12047 (a0322878 - Feb 16 2012) *
;* Date/Time created: Thu Feb 16 10:26:02 2012                                *
;*                                                                            *
;* Warning: This file is auto generated by the compiler and can be            *
;* overwritten during the next compile.                                       *
;*                                                                            *
;******************************************************************************
;* User Options: --silicon_version=6400+

"func.c": advice #27000: Detecting compilation without optimization.Use
   optimization option -o2 or higher.
```

示例 3：下述示例使用不同的选项发送输出建议到 mydir 目录中名为 myfile.adv 的文件中。

使用 --advice:performance_file 和 --advice:performance_dir 选项：

```
cl6x -mv6400+ -o2 -k --advice:performance_file=myfile.adv --advice:performance_dir=mydir basicloop.c
```

仅使用 --advice:performance_file 选项来指定完整的路径名称：

```
cl6x -mv6400+ -o2 -k --advice:performance_file=mydir/myfile.adv basicloop.c
```

如果同时指定 --advice_dir 选项和完整的路径名称，则忽略 --advice:performance_dir 选项，并在完整的路径名称建议文件中生成建议。另外，请注意目录“mydir”必须已经存在，才能在其中创建建议文件。

4.15.1 Advice #27000

```
advice #27000: 检测未经优化的编译。使用 -o2 或更高级别优化。
```

您的编译是在没有任何优化选项（-o0 及以上）的情况下完成的。这阻止了编译器使用其强大的优化技术，因为 -o (--opt_level) 选项是大多数其他优化的基础。使用 -o2（或更高级别）优化可以大幅度提高性能。软件流水线循环优化需要优化选项 -o2，这对于获得良好的性能至关重要。

C/C++ 编译器能够执行各种优化，但是您需要在命令行上指定优化选项才能执行这些优化。调用优化的最简单方法是在编译器命令行上指定 --opt_level=n 选项。您可以使用 -On 为 --opt_level 选项设置别名。n 表示优化级别（0、1、2 和 3），其控制优化的类型和程度。有关优化选项的更多信息，请参阅节 4.1 中的“调用优化”。

4.15.2 Advice #27001 提高优化级别

```
advice #27001: Detecting compilation with low optimization level.
       Use optimization option -o2 or higher.
```

您的编译使用低级别优化选项（-o1 及以下），这会阻止编译器使用其最强大的优化技术。

C/C++ 编译器能够执行各种优化，但您可以控制这些优化的级别。高级别优化在优化器中执行，特定于目标的低级别优化在代码生成器中执行。您必须使用高级别优化才能实现最优代码。您可以通过在编译器命令行上指定 --opt_level=n 选项来调用优化。

请参阅节 4.1 中的“调用优化”，以了解有关优化选项的更多信息。有关 Advice #27000 的信息，另请参阅节 4.15.1。

4.15.3 Advice #27002 不要关闭软件流水线

```
advice #27002: Detecting compilation with "-mu" which turns off
       software pipelining.要进行优化，请关闭该选项
```

正在使用 -mu 完成编译，这会关闭软件流水线。软件流水线是实现良好性能的重要优化技术。发出此建议是为了提醒您不要使用编译器选项 -mu。-mu 是一个不错的调试选项，但建议不要将此选项用于生产代码，因为它会对性能产生负面影响。

通常，为了实现最大性能，应避免在生产代码中使用以下内容：

- -g: 使用调试信息进行编译不再影响优化代码的能力。然而，由于代码重构和其他转换，高级别的优化确实会使调试代码变得更加困难。如果您仍处于调试阶段，您可能希望使用较低级别的优化。对于生产代码，可以在启用或禁用包含调试信息的情况下使用高级别的优化。
- -ss: 将源代码交叉列入汇编文件中。与 -g 一样，此选项会对性能产生负面影响。
- -mu: 关闭软件流水线（其是实现良好性能的重要优化技术）。这是一个不错的调试选项，但建议不要将此选项用在生产代码中，因为它会对性能产生负面影响。

4.15.4 Advice #27003 避免使用调试选项进行编译

```
advice #27003: Detecting compilation with debug option "-g",
       which hinders optimization.To optimize, remove -g or
    compile with --optimize_with_debug_option.
```

早期版本中提供此建议，其中包含的调试信息会影响优化代码的能力。调试信息不再影响优化，并且 --optimize_with_debug 选项已被弃用。另请参阅节 4.15.3 中的 Advice #27002。

4.15.5 Advice #27004 未生成性能建议

```
advice #27004: 未生成性能建议
```

编译器检测到编译正由 --advice:performance 选项完成，但编译器没有待报告的建议。发出此建议是为了提醒您没有建议发出，也不会创建建议文件（如果已请求）。

4.15.6 Advice #30000 防止由于调用导致循环不合格

```
advice #30000: Loop at line 10 cannot be scheduled efficiently,
         as it contains a function call ("function_name").
         尝试内联调用或考虑重写循环.
```

编译器尝试在优化级别 --opt_level=3 (或 -O3) 上执行软件流水线循环优化。如果循环中有调用，编译器会尝试完全内联被调用的函数，但有时难以实现。如果编译器无法内联被调用的函数，则无法执行软件流水线。这会严重降低循环的性能。

在下面的测试用例中，对函数“func2”的调用会阻止软件流水线。内联函数“func2”或重写循环来避免函数调用可以避免流水线不合格。如果循环流水线成功，您可能会看到性能得以提高。

```
void func1(int *p, int *q, int n)
{
       unsigned int i;

       for (i = 0; i < n; i++)
       {
           int t = func2(i);

           p[i] = q[i] + t;
       }
}
```

4.15.7 Advice #30001 防止由于 rts 调用导致循环不合格

```
advice #30001: Loop at line 18 cannot be scheduled efficiently, as it
                  contains conversion from "type-a" to "type-b".
                  Try to use "suggested" type.
```

编译器可以在运行时支持库 (RTS) 中插入对特殊函数的调用，以支持 ISA 本机不支持的运算。例如，虽然浮点 ISA 支持在浮点和有符号整数值之间的指令转换，但不支持在浮点和无符号整数值之间的转换。如果在浮点表达式中使用无符号变量，编译器将生成对 RTS 例程的调用来执行该函数。这样的调用将禁用软件流水线。

可以将代码中的无符号变量更改为有符号变量，从而防止这种情况发生。然后，编译器将能够使用本机硬件而不是添加特殊的函数调用，因此可以获得更好的性能。

4.15.8 Advice #30002 防止由于 asm 语句导致循环不合格

```
advice #30002: 第 8 行的循环不能被有效地调度，因为其包含一条 asm() 语句。尝试将 asm() 语句替换为 C 或内在语句。
```

在 C 代码循环中插入 asm 语句将使软件流水线的循环不合格。软件流水线是实现良好性能的重要优化技术。没有软件流水线，性能可能会下降。

将 asm() 语句替换为本地 C 语言或调用内在函数以防止这种情况发生。

4.15.9 Advice #30003 防止复杂条件导致的循环不合格

```
advice #30003: Loop at line 8 cannot be scheduled efficiently, as it
                  contains complex conditional expression.Try to simplify
                  condition.
```

代码在循环中包含一个复杂的条件表达式，可能是一个较大的“if”子句，这会妨碍优化。编译器将优化小的“if”语句（带有较短或空“if”和“else”块的“if”语句）。编译器不会优化大的“if”语句，循环主体中如此大的 if 语句将取消循环用于软件流水的资格。软件流水优化很关键，没有这项优化可能会导致性能降低。

在下面的示例中，示例 1 将使用流水线，但示例 2 不会：

示例 1：

```
for (i=0; i < N; i++)
{
        if (!flag)        {
           //statements
        }
        else              {
          x[i] = y[i];
        }
}
```

示例 2：

```
for (i = 0; i < n; i++)
{
        if (!flag)        {
           //statements
        }
        else              {
           if (flag == 1) x[i] = y[i];
        }
}
```

示例 1 的性能将明显优于示例 2，因为它成功地使用了流水线。但是，如果修改代码以消除嵌套的“if”，则示例 2 也可以使用流水线：

```
for (i = 0; i < n; i++)
{
        if (!flag)        {
           //statements
        }
        else              {
           p = (flag == 1);
           x[i] = !p * x[i] + p * y[i] ;
        }
}
```

4.15.10 Advice #30004 防止由于 switch 语句导致循环不合格

```
advice #30004: 第 257 行的循环不能被有效地安排，因为其包含一条切换语句。尝试重写循环。
```

循环中有一条切换语句。循环中的切换语句将使软件流水线的循环不合格。软件流水线是重要的优化技术，没有这项优化，性能可能会下降。

尝试重写没有切换语句的循环。

4.15.11 Advice #30005 防止因算术运算导致循环不合格

```
advice #30005: 第 5 行的循环不能被有效地安排，因为其包含一条 "除法"运算。如果可能的话，使用更简单的运算重写。
```

编译器可以在运行时支持库 (RTS) 中插入对特殊函数的调用，以支持 ISA 本身不支持的运算。例如，编译器调用 __c6xabi_divi() 函数来执行 32 位整数除法运算。这些函数被称为编译器辅助函数，并在循环主体中生成函数调用。在下面的示例中，编译器将通过调用编译器帮助函数“_divi”来完成除法运算：

```
void func(float *p, float n)
{
        int i;

        for (i = 1; i < 1000; i++)    {
            p[i] /= n;
        }
}
```

但是，如果修改此循环（如下所示），则循环按以下方式执行：

```
void func_adjusted(float *p, float n)
{
        int i;

        float inv = 1/n;

        for (i = 1; i < 1000; i++)     {
            p[i] *= inv;
        }
}
```

4.15.12 Advice #30006 防止由于调用导致循环不合格 (2)

```
advice #30006: 第 22 行的循环不能被有效调度，因为其包含一个函数调用（"function_name"）。尝试将 function_name 设置为内联函数。
```

为了提高性能，在优化级别 --opt_level=2 (-O2) 和 --opt_level=3 (-O3) 上，编译器尝试对循环进行软件流水线处理。有时编译器可能无法内联循环中的函数调用。由于编译器无法内联函数调用，因此无法对循环进行软件流水线处理，也无法有效地调度循环。

例如，在下面的测试用例中，对函数“func2”的调用阻止了软件流水线：

```
void func1(int *p, int *q, int n)
{
       unsigned int i;

       for (i = 0; i < n; i++)       {
           int t = func2(i);

           ; other operations
       }
}
int function func2() { ...}
```

但是，如果函数 func2 被内联，则可以节省函数调用的开销。编译器可以自由地使用周围代码在上下文中优化函数。自动内联由“内联”关键字控制；使用该关键字可以启用特定函数的内联：

```
inline int function func2() { ...}
```

另请参阅节 4.15.6 中的 Advice #30000。

4.15.13 Advice #30007 防止由于 rts 调用导致循环不合格 (2)

```
advice #30007: Attempting to use floating-point operation "__mpyd" on
                  fixed-point device, at line 5 (there may be other instances
                  of this).Such calls reduce loop performance; use fixed point
                  operation if possible.
```

编译器在运行时支持库 (RTS) 中插入对特殊函数的调用，以支持指令集架构 (ISA) 本机不支持的运算。例如，定点 ISA 不支持浮点指令，编译器将生成对 RTS 例程的调用来执行浮点运算。在下面的测试用例中，浮点乘法对于定点器件不可用：

```
void func(float *p, float *q, int n)
{
        unsigned int i;

        for (i = 1; i < n; i++)
        {
            p[i] = (q[i] * 12.4) / p[i - 1];
        }
}
```

如果为 C6400+ 进行编译（编译器选项 -mv6400+），编译器将使用 RTS 调用来执行运算。这样的调用将禁用软件流水线。您可以重写该运算，或使用定点运算来防止这种情况发生。

另请参阅节 4.15.7 中的 Advice #30001。

4.15.14 Advice #30008 改进循环；使用 restrict 进行限定

```
advice #30008: 如果 inp1、inp2 没有访问相同的内存位置，可以考虑在 inp1、inp2 的定义中添加限制限定符。
```

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。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被定义。

若要查看有关使用 restrict 的更多信息，请参阅节 7.5.6

4.15.15 Advice #30009 改进循环；添加 MUST_ITERATE pragma

```
advice #30009: If you know that this loop will always execute at a
                  multiple of <2> and at least <2> times, try adding
                  "#pragma MUST_ITERATE(2, ,2)" just before the loop.
```

C6000 架构被划分为几乎对称的两部分。在 asm 文件的软件流水线信息中显示的资源细目是在编译器已经将指令划分到 A 侧或 B 侧之后计算的。如果资源不平衡（即，一侧的某些资源比另一侧的资源使用得更多），则软件流水线会受到资源的限制，循环无法有效调度。如果编译器具有关于循环次数的信息，则可以展开循环以平衡资源使用，并获得更好的流水线。可以使用“MUST_ITERATE”pragma 向编译器提供循环计数的信息。

要查看更多有关使用 MUST_ITERATE pragma 的信息，请参阅节 7.9.22

4.15.16 Advice #30010 改进循环；添加 MUST_ITERATE pragma (2)

```
advice #30010: If you know that this loop will always execute at least
                  <2> times, try adding "#pragma MUST_ITERATE(2)" just before the loop.
```

请参阅节 4.15.15 中的 Advice #30009。

4.15.17 Advice #30011 改进循环；添加 _nasssert()

```
advice #30011: Consider adding assertions to indicate n-byte alignment
                  of variables input1, input2, output if they are actually n-byte
                  aligned: _nassert((int)(input1) % 8 == 0)。
```

大多数循环都有内存访问指令。编译器尝试使用更宽的加载指令和对齐的内存访问而不是非对齐的内存访问，以减少/平衡用于内存访问指令的资源。让编译器知道可以安全使用“更宽”加载的方式之一是使用关键字“_nassert”。

要了解更多有关使用 _nassert 关键字的信息，请参阅节 8.6.11。

4.16 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。

```
_main:
;** 5-----------------------    printf("Hello, world\n");
;** 6-----------------------    return 0;
           STW     .D2     B3,*SP--(12)
.line3
           B       .S1     _printf
           NOP             2
           MVKL    .S1     SL1+0,A0
           MVKH    .S1     SL1+0,A0
||         MVKL    .S2     RL0,B3
           STW     .D2     A0,*+SP(4)
||         MVKH    .S2     RL0,B3
RL0:       ; CALL OCCURS
.line4
           ZERO    .L1     A4
.line5
           LDW     .D2     *++SP(12),B3
           NOP             4
           B       .S2     B3
           NOP             5
           ; BRANCH 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。

```
_main:
;** 5-----------------------    printf("Hello, world\n");
;** 6-----------------------    return 0;
           STW     .D2     B3,*SP--(12)
;------------------------------------------------------------------------------
;   5 | printf("Hello, world\n");
;------------------------------------------------------------------------------
           B       .S1     _printf
           NOP             2
           MVKL    .S1     SL1+0,A0
           MVKH    .S1     SL1+0,A0
||         MVKL    .S2     RL0,B3
           STW     .D2     A0,*+SP(4)
||         MVKH    .S2     RL0,B3
RL0:       ; CALL OCCURS
;------------------------------------------------------------------------------
;   6 | return 0;
;------------------------------------------------------------------------------
           ZERO    .L1     A4
           LDW     .D2     *++SP(12),B3
           NOP             4
           B       .S2     B3
           NOP             5
           ; BRANCH OCCURS
```

4.17 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。

如果在调试代码中的循环时遇到问题，可以使用 --disable_software_pipeline 选项关闭软件流水线。有关更多信息，请参阅节 4.6.1。

4.17.1 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 ）。

如果您有基于断点的分析器，请使用 --profile:breakpt 选项与 --opt_level 选项。--profile:breakpt 选项禁用在使用基于断点的分析器时会导致错误行为的优化。

4.18 正在执行什么类型的优化？

TMS320C6000 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 4.18.1
别名消歧	节 4.18.2
分支优化和控制流简化	节 4.18.3
数据流优化 • 复制传播 • 通用子表达式消除 • 冗余分配消除	节 4.18.4
表达式简化	节 4.18.5
函数的内联扩展	节 4.18.6
函数符号别名	节 4.18.7
归纳变量和强度降低	节 4.18.8
循环不变量代码运动	节 4.18.9
循环旋转	节 4.18.10
矢量化	节 4.18.11
指令调度	节 4.18.12

C6000 专用优化	请参阅
寄存器变量	节 4.18.13
寄存器跟踪/定位	节 4.18.14
软件流水线	节 4.18.15

4.18.1 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和软件流水线，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。

4.18.2 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。

4.18.3 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。

4.18.4 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。

4.18.5 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。

4.18.6 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 3.11。

这种类型的优化通过 --opt_level=0 和更高的优化设置来启用。

4.18.7 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 7.14.2。

4.18.8 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。

4.18.9 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。

4.18.10 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。

4.18.11 向量化 (SIMD)

编译器可转换循环，使循环使用的指令一次操作多个数据，从而显著提高性能。

4.18.12 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流水线延迟。它还可用于缩减代码大小。

4.18.13 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。

4.18.14 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。

4.18.15 软件流水线

软件流水线是一种用于从循环调度的技术，以便并行执行循环的多次迭代。有关更多信息，请参阅节 4.6。

这种类型的优化通过 --opt_level=2 和更高的优化设置来启用。

This page intentionally left blank.

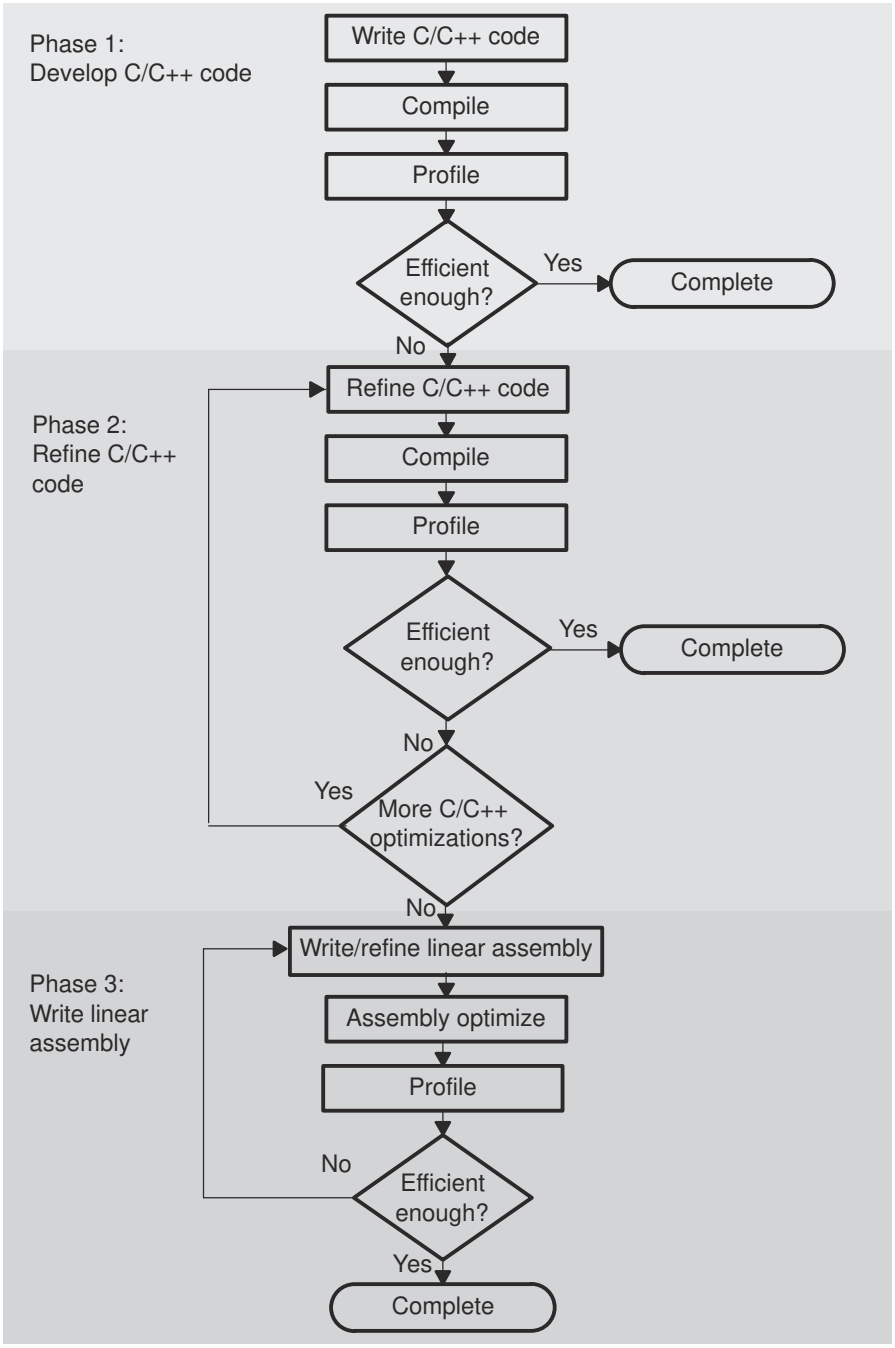
章节 5

使用汇编优化器

汇编优化器允许在编写汇编代码时无需关注 C6000 流水线结构或分配寄存器。它接受*线性汇编代码*，这是可能已执行寄存器分配且未调度的汇编代码。汇编优化器通过分配寄存器和循环优化将线性汇编转换为高度并行汇编。

5.1 可提高性能的代码开发流程

如果您在编写和调试代码时遵循此流程，则可以用 C6000 代码发挥出卓越性能：



C6000 的代码开发分为三个阶段：

- **第 1 阶段：用 C 语言编写**

  您可以在不了解 C6000 的情况下为第 1 阶段开发 C/C++ 代码。使用 --opt_level=3 选项，而不使用任何 --debug 选项进行编译。识别 C/C++ 代码中的任何低效区域。有关调试和分析优化代码的更多信息，请参阅节 4.17。要提高代码的性能，请继续执行第 2 阶段。
- **第 2 阶段：优化您的 C/C++ 代码**

在第 2 阶段，使用本书中介绍的内在函数和编译器选项来改进您的 C/C++ 代码。检查更改后的代码的性能。有关完善 C/C++ 代码的提示，请参阅 *TMS320C6000 编程人员指南*。如果您的代码仍然不如您希望的高效，请继续执行第 3 阶段。

- **第 3 阶段：编写线性汇编文件**

  在此阶段，您从 C/C++ 代码中提取时间关键区域，并以线性汇编语言重新编写代码。您可以使用汇编优化器来优化此代码。在编写第一遍线性汇编代码时，不应担心流水线结构或分配寄存器。稍后，在优化线性汇编代码时，您可能希望向代码添加更多详细信息，例如分区寄存器。

  在此阶段提高性能所需的时间比第 2 阶段要长，因此请尝试在使用第 3 阶段之前尽可能地优化您的代码。然后，在此阶段，您应该有较小的代码段可以使用。

5.2 关于汇编优化器

如果您在使用了所有可用的 C/C++优化之后对 C/C++ 代码的性能不满意，则可以使用汇编优化器来简化为 C6000 编写汇编代码的过程。

汇编优化器会执行多项任务，包括：

- （可选）对指令和/或寄存器进行分区
- 使用 C6000 的指令级并行性来调度指令以更大限度地提高性能
- 确保指令符合 C6000 延迟要求
- （可选）为源代码分配寄存器

与 C/C++ 编译器一样，汇编优化器会执行软件流水线。*软件流水线* 是用于调度循环中的指令以使循环的多个迭代能够并行执行的一种技术。代码生成工具会尝试使用您的输入以及从程序收集的信息对代码进行软件流水线处理。有关详细信息，请参阅节 4.6。

要调用汇编优化器，请使用编译器程序 (cl6x)。当其中一个输入文件具有 *.sa* 扩展名时，编译器程序会自动调用汇编优化器。您可以指定 C/C++ 源文件以及线性汇编文件。更多有关编译器程序的信息，请参阅章节 3。

5.3 编写线性汇编需要了解的内容

通过使用 C6000 分析工具，您可以识别代码中需要重写为线性汇编语言的时间关键段。为汇编优化器编写的源代码与汇编源代码类似。但是，线性汇编代码不需要进行分区、调度或寄存器分配。目的是让汇编优化器为您确定此信息。当您编写线性汇编代码时，您需要了解以下各项：

- **汇编优化器指令**

  您的线性汇编文件可以是线性汇编代码段和常规汇编源的组合。使用汇编优化器指令将汇编优化器代码与常规汇编代码区分开来，并向汇编优化器提供有关代码的其他信息。节 5.4 中介绍了汇编优化器指令。
- **影响汇编优化器的功能的选项**

  表 5-1 中的编译器选项会影响汇编优化器的行为。

表 5-1. 影响汇编优化器的选项

选项	效果	请参阅
--ap_extension	更改汇编优化器源文件的默认扩展名	节 3.3.10
--ap_file	更改汇编优化器源文件的标识方式	节 3.3.8
--disable_software_pipelining	关闭软件流水线	节 4.6.1
--debug_software_pipeline	生成详细的软件流水线信息	节 4.6.2
--interrupt_threshold=*n*	指定中断阈值	节 3.12
--keep_asm	保留汇编语言 (.asm) 文件	节 3.3.2
--no_bad_aliases	假设没有存储器别名使用	节 4.12.3
--opt_for_space=*n*	在四个级别（*n*=0、1、2 或 3）上控制代码大小	节 4.9
--opt_level=*n*	提高优化级别（*n*=0、1、2 或 3）	节 4.1
--quiet	抑制进度消息	节 3.3.2
--silicon_version=*n*	选择目标版本	节 3.3.5
--skip_assembler	仅编译或汇编优化（不汇编）	节 3.3.2
--speculate_loads=*n*	允许对具有分界地址范围的加载进行推测执行	节 4.12.3

- **TMS320C6000 指令**

  当您编写线性汇编代码时，您的代码*不*需要指示以下内容：
  - 流水线延迟
  - 寄存器的使用
  - 正在使用哪个单元

  与其他代码生成工具一样，您可能需要修改线性汇编代码，直到对其性能感到满意。如果这样做，您可能需要为线性汇编添加更多细节。例如，您可能需要对某些寄存器进行分区或分配。

  备注

  请勿将已调度的汇编代码用作源代码

  汇编优化器假定输入文件中的指令按照您希望它们发生的逻辑顺序（即线性汇编代码）放置。并行指令是非法的。

  如果编译器无法使您的指令呈线性（非并行），则会生成错误消息。编译器假定指令按照其在文件中出现的顺序发生。已调度代码是非法的（即使是非并行调度代码）。编译器可能不会检测到已调度代码，但产生的输出可能不是您想要的。
- **线性汇编源语句语法**

线性汇编源程序包含源语句，这些源语句可以包含汇编优化器指令、汇编语言指令和注释。更多有关源语句元素的信息，请参阅节 5.3.1。

- **指定寄存器或寄存器边**

  寄存器可以显式分配给用户符号。或者，可以将符号分配给 A 侧或 B 侧，让编译器执行实际的寄存器分配。有关指定寄存器的信息，请参阅节 5.3.2。

- **指定功能单元**

  功能单元说明符在线性汇编代码中是可选的。数据路径信息会予以考虑；单元信息会被忽略。

- **源代码注释**

  汇编优化器会将输入线性汇编指令的注释附加到输出文件中。它会将一个二元组 <x, y> 附加到注释，从而规定指令在软件流水线上执行的循环的迭代和周期。从零开始的数字 x 表示在内核第一次执行期间指令所进行的迭代。从零开始的数字 y 表示在循环的单次迭代中进行指令调度的周期。有关使用源代码注释和生成的汇编优化器输出的说明，请参阅节 5.3.4。

5.3.1 线性汇编源语句格式

源语句可以包含五个有序字段（标签、助记符、单元说明符、操作数列表和注释）。源语句的一般语法如下：

label[:]	对于所有汇编语言指令和大多数（但不是全部）汇编优化器指令，标签都是可选的。使用时，标签必须从源语句的第 1 列开始。标签后面可以跟一个冒号。
[*register*]	方括号 ([ ]) 表示内容是条件指令。机器指令助记符会根据括号中寄存器的值来执行；有效的寄存器名称为 A0、A1、A2、B0、B1、B2 或符号。
助记符[mnemonic]	助记符是机器指令（如 ADDK、MVKH、B）或汇编优化器指令（如 .proc、.trip）
单元说明符	借助可选的单位说明符，您可以指定函数单元操作数。仅使用指定的单元侧；忽略其他规格。首选方法是指定寄存器侧。
操作数列表	并非所有指令都需要操作数列表。操作数可以是符号、常量或表达式，并且必须用逗号分隔。
注释	注释为可选项。从第 1 列开始的注释必须用分号或星号开头，但在任何其他列中开始的注释都必须以分号开头。

C6000 汇编优化器每行最多读取 200 个字符。超过 200 个字符的部分将被截断。为了正确汇编，请将源语句的操作部分（即注释以外的所有内容）长度保持在 200 个字符以内。您的注释可以超出字符限制，但截断部分不包括在 .asm 文件中。

在编写线性汇编代码时，请遵循以下准则：

- 所有语句都必须以标号、空白、星号或分号开头。
- 标签是可选的；如果使用，则必须从第 1 列开始。
- 每个字段必须有一个或多个空格。制表符被解析为空白。您必须用空格将操作数列表与前面的字段分开。
- 注释为可选项。在第 1 列中开始的注释可以用星号或分号 (* 或 ;) 开头，但在任何其他列中开始的注释都*必须*以分号开头。
- 如果您设置条件指令，则寄存器必须括在方括号中。
- 助记符不能在第 1 列中开始，否则它会被解析为标签。

有关 C6000 指令语法的信息，包括条件指令、标签和操作数，请参阅 *TMS320C6000 汇编语言工具用户指南*。

5.3.2 线性汇编的寄存器规格

C6000 中只有两条交叉路径。这会将 C6000 限制为每个周期从每个数据路径的相反寄存器文件中读取一个源。编译器必须为每个指令选择一边；这称为分区。

建议您不要手动对线性汇编源代码进行初始分区。这使编译器能够更自由地对代码进行分区和优化。如果编译器在软件流水线循环中找不到理想分区，则可以手动对足够的指令进行分区，以通过对寄存器进行分区来强制进行理想分区。

汇编优化器为您选择一个寄存器，这样它的使用就与为对值进行操作的指令选择的功能单元一致。

可以使用两条指令直接对寄存器进行分区。**.rega** 指令将符号名称限制为 A 侧寄存器。**.regb** 指令将符号名称限制为 B 侧寄存器。有关这些指令的更多详细信息，请参阅 .rega/.regb 主题。**.reg** 指令允许您对存储在寄存器中的值使用描述性名称。有关 .reg 指令的更多详细信息和示例，请参阅 .reg 主题。

示例 5-1 是一个手工编码的线性汇编程序，用于计算点积；与示例 5-2 (用于说明 C 代码) 进行比较。

示例 5-1. 用于计算点积的线性汇编代码

```
_dotp: .cproc a_0, b_0
       .rega     a_4, tmp0, sum0, prod1, prod2
       .regb     b_4, tmp1, sum1, prod3, prod4
       .reg      cnt, sum
       .reg      val0, val1
       ADD    4, a_0, a_4
       ADD    4, b_0, b_4
       MVK    100, cnt
       ZERO   sum0
       ZERO   sum1
loop:  .trip   25
       LDW          *a_0++[2], val0      ; load a[0-1]
       LDW          *b_0++[2], val1      ; load b[0-1]
       MPY          val0, val1, prod1    ; a[0] * b[0]
       MPYH         val0, val1, prod2    ; a[1] * b[1]
       ADD          prod1, prod2, tmp0   ; sum0 += (a[0]*b[0]) +
       ADD          tmp0, sum0, sum0     ;          (a[1]*b[1])
       LDW          *a_4++[2], val0      ; load a[2-3]
       LDW          *b_4++[2], val1      ; load b[2-3]
       MPY          val0, val1, prod3    ; a[2] * b[2]
       MPYH         val0, val1, prod4    ; a[3] * b[3]
       ADD          prod3, prod4, tmp1   ; sum1 =+ (a[2]*b[2]) +
       ADD          tmp1, sum1, sum1     ;          (a[3]*b[3])
[cnt]  SUB    cnt, 4, cnt        ; cnt -= 4
[cnt]  B      loop               ; if (cnt!=0) goto loop
       ADD    sum0, sum1, sum    ; compute final result
       .return sum
       .endproc
```

示例 5-2 是用于计算点积的精练 C 代码。

示例 5-2. 用于计算点积的 C 代码

```
int dotp(short a[], shortb[])
{
    int sum0 = 0;
    int sum1 = 0;
    int sum, I;
    for (I = 0; I < 100/4; I +=4)
    {
        sum0 += a[i]   * b[i];
        sum0 += a[i+1] * b[i+1];
        sum1 += a[i+2] * b[i+2];
        sum1 += a[i+3] * [b[i+3];
    }
    return
}
```

仍然可以使用通过分区指令间接对寄存器进行分区的旧方法。侧边和功能单元说明符仍可用于指令。但是，功能单元说明符 (.L/.S/.D/.M) 会被忽略。如果有，侧边说明符会转换为对应符号名称的分区约束。例如：

```
      MV  .1     x, y     ; translated to .REGA y
      LDW .D2T2 *u, v:w   ; translated to .REGB u, v, w
```

在线性汇编器中，您还可以使用 .cproc 和/或 .reg 指令指定寄存器对，如示例 5-3 所示：

示例 5-3. 指定寄存器对

```
   .global foopair
foopair: .cproc q1:q0,s0
   .reg r1:r0
   ADD q1:q0, s0, r1:r0
   .return r1:r0
   .endproc
```

在示例 5-3 中，表达式“q1:q0”表示线性汇编函数的第一个参数是寄存器对。根据 C 调用约定，“q1:q0”符号对映射到寄存器对“a5:a4”。当寄存器对语法被用作 .reg 指令的参数时（如图所示），这意味着当编译器处理线性汇编器源代码并分配寄存器对符号映射到“r1:r0”的实际寄存器时，两个寄存器符号被限制为对齐的寄存器对，如图所示。

编译器支持四倍字寄存器语法（仅限 C6600），以便在线性汇编和汇编源代码中指定 128 位指令的 128 位操作数。示例 5-4 说明了如何指定四倍字寄存器：

示例 5-4. 指定四倍字寄存器（仅限 C6600）

```
    .global fooquad
fooquad: .cproc q3:q2:q1:q0, s3:s2:s1:s0
    .reg r3:r2:r1:r0
    QMPY32 s3:s2:s1:s0, q3:q2:q1:q0, r3:r2:r1:r0
    .return r3:r2:r1:r0
    .endproc
```

在示例 5-4 中，表达式“q3:q2:q1:q0”表示线性汇编函数的第一个参数是四倍字寄存器。根据 C 调用约定，四元组“q3:q2:q1:q0”符号映射到四倍字寄存器“a7:a6:a5:a4”。当一个四倍字寄存器语法被用作 .reg 指令的参数时（如图所示），这意味着当编译器处理线性汇编器源代码并分配四倍字寄存器符号映射到“r3:r2:r1:r0”的实际寄存器时，四个寄存器符号被限制为对齐的四倍字寄存器，如图所示。

5.3.3 线性汇编的功能单元规格

直接对寄存器进行分区的能力已弃用了指定功能单元的功能。（有关详细信息，请参阅节 5.3.2。）虽然可以在线性汇编中使用单元说明符字段，但编译器仅使用寄存器侧信息。

通过在汇编器指令后面加上句点 (.) 和功能单元说明符来指定函数单元。可以在单一指令周期中将一条指令分配给每个功能单元。共有八个功能单元，每种功能类型两个，另外还有两个地址路径。每种功能类型的两个功能单元的不同之处在于其使用数据路径 A 还是 B。

.D1 和 **.D2**	数据/加法/减法运算
.L1 和 **.L2**	算术逻辑单元 (ALU)/compares/long 数据算术
.M1 和 **.M2**	乘法运算
.S1 和 **.S2**	Shift/ALU/branch/域操作
.T1 和 **.T2**	地址路径

有几种方法可以输入线性汇编中归档的单元说明符。其中，仅识别和使用特定的寄存器侧信息：

- 用户可以指定特定功能单元（例如 .D1）。
- 您可以指定 .D1 或 .D2 功能单元，后跟 T1 或 T2，以指定非存储器操作数位于特定的寄存器侧。T1 指定 A 侧，T2 定 B 侧。例如：

```
   LDW  .D1T2    *A3[A4], B3
   LDW  .D1T2    *src, dst
```

- 用户可以仅指定数据路径（例如 .1），由汇编优化器指定功能类型（例如 .L1）。

更多有关功能单元的信息，请参阅 *TMS320C6000 CPU 和指令集参考指南*。

5.3.4 使用线性汇编源代码注释

您在线性汇编中的注释可以从任何列开始并延伸到源代码行的末尾。注释可以包含任何 ASCII 字符，包括空格。注释打印在线性汇编源列表中，但不会影响线性汇编。

仅包含注释的源语句是有效的。如果它从第 1 列开始，则可以分号 (;) 或星号 (*) 开头。从行中任何其他位置开始的注释必须以分号开头。星号仅在出现在第 1 列中时才标识注释。

汇编优化器会调度指令；也就是说，它会重新排列指令。独立注释被移动到指令块的顶部。指令语句末尾的注释与指令一起保留。

示例 5-5 显示了包含注释的名为 Lmac 的函数的代码。

示例 5-5. 显示注释的 Lmac 函数代码

```
Lmac:   .cproc   A4,B4

        .reg    t0,t1,p,i,sh:sl

        MVK     100,i
        ZERO    sh
        ZERO    sl

loop:   .trip   100

        LDH           *a4++, t0       ; t0 = a[i]
        LDH           *b4++, t1       ; t1 = b[i]
        MPY           t0,t1,p         ; prod = t0 * t1
        ADD           p,sh:sl,sh:sl   ; sum += prod
[I]     ADD           -1,i,i          ; --I
[I]     B             loop            ; if (I) goto loop

        .return sh:sl

        .endproc
```

5.3.5 汇编文件保留您的符号寄存器名称

在输出汇编文件中，寄存器操作数包含您的符号名称。这有助于您调试线性汇编文件，并将线性汇编输出的代码段封装到汇编文件中。

汇编函数开头的 .map 指令（请参阅 .map 主题）会将符号名称与实际寄存器相关联。换句话说，符号名称成为实际寄存器的别名。可在汇编和线性汇编代码中使用 .map 指令。

当编译器将用户符号拆分为两个符号且每个符号都映射到不同的机器寄存器时，符号名称的实例会附加一个后缀以生成唯一名称，从而使每个唯一名称都与一个机器寄存器相关联。

例如，如果编译器在某些指令中将符号名称 y 与 A5 关联，在另外一些指令中将符号名称 y 与 B6 关联，则输出汇编代码可能如下所示：

```
    .MAP y/A5
    .MAP y'/B6
    ...
    ADD .S2X y, 4, y'  ; Equivalent to add A5, 4, B6
```

要使用符号名称禁用此格式并使用实际寄存器显示汇编指令，请使用 --machine_regs 选项进行编译。

5.4 汇编优化器指令

汇编优化器指令为汇编优化过程提供数据并对其进行控制。汇编优化器会优化过程中包含的线性汇编代码；即 .proc 和 .endproc 指令中或 .cproc 和 .endproc 指令中的代码。如果不在线性汇编文件中使用 .cproc/.proc 指令，则汇编优化器不会优化代码。本节介绍了这些指令以及可与汇编优化器一起使用的其他指令。

表 5-2 总结了汇编优化器指令。它提供了每个指令的语法，每个指令的说明以及您应记住的任何限制。有关详细信息，请参阅特定指令主题。

在表 5-2 和详细指令主题中，使用了以下参数术语：

参数 符号变量名称或机器寄存器

memref 用于存储器引用的符号（不是寄存器）

register 机器（硬件）寄存器

符号[symbol] 符号用户名或符号寄存器名称

变量 符号变量名称或机器寄存器

表 5-2. 汇编优化器指令摘要

句法	说明	限制
.call [*ret_reg* =] *func_name* (*argument*$_1$, *argument*$_2$, ...)	调用函数	仅在过程中有效
.circ *symbol*$_1$ */ register*$_1$ [, *symbol*$_2$ */ register*$_2$]	声明循环寻址	必须手动插入循环寻址的设置/拆卸代码。仅在过程中有效
label .cproc [*argument*$_1$ [, *argument*$_2$, ⋯]]	启动 C/C++ 可调用过程	必须与 .endproc 一起使用
.endproc	结束 C/C++ 可调用过程	必须与 .cproc 一起使用
.endproc [*variable*$_1$ [, *variable*$_2$,⋯]]	结束过程	必须与 .proc 一起使用
.map *symbol*$_1$ */ register*$_1$ [, *symbol*$_2$ */ register*$_2$]	将符号分配给寄存器	必须使用实际的机器寄存器
.mdep [*memref*$_1$[, *memref*$_2$]]	表示存储器依赖	仅在过程中有效
.mptr {*variable*|*memref*}, *base* [+ *offset*] [, *stride*]	避免存储器组冲突	仅在过程中有效
.no_mdep	函数中没有存储器别名	仅在过程中有效
.pref *symbol / register*$_1$[/*register*$_2$ /...]	将符号分配给集合中的寄存器	必须使用实际的机器寄存器
label .proc [*variable*$_1$ [, *variable*$_2$, ⋯]]	启动过程	必须与 .endproc 一起使用
.reg *symbol*$_1$[, *symbol*$_2$,⋯]	声明变量	仅在过程中有效
.rega *symbol*$_1$[, *symbol*$_2$,⋯]	将符号分区到 A 侧寄存器	仅在过程中有效
.regb *symbol*$_1$[, *symbol*$_2$,⋯]	将符号分区到 B 侧寄存器	仅在过程中有效
.reserve [*register*$_1$ [, *register*$_2$,⋯]]	阻止编译器分配寄存器	仅在过程中有效
.return [*argument*]	将值返回到过程	仅在 .cproc 过程中有效
label .trip *min*	指定行程计数值	仅在过程中有效
.volatile *memref*$_1$[, *memref*$_2$,⋯]	指定存储器参考易失性	如果在中断期间可以修改引用，则使用 --interrupt_threshold=1

.call

调用函数

语法

.call [*ret_reg*=] *func_name* ([*argument*$_1$, *argument*$_2$,...])

说明

使用 **.call** 指令调用函数。或者，您可以指定一个分配了调用结果的寄存器。该寄存器可以是符号寄存器或机器寄存器。.call 指令遵循与 C/C++ 编译器相同的寄存器和函数调用惯例。有关信息，请参阅节 8.3 和节 8.4。不支持替代寄存器或函数调用惯例。

您不能调用具有可变数量参数的函数，例如 printf。不执行错误检查以确保传递正确数量和/或类型的参数。您不能通过 .call 指令传递或返回结构。

以下是 .call 指令参数的说明：

ret_reg	（可选）分配了调用结果的符号/机器寄存器。如果未指定，汇编优化器会假定调用会用结果覆盖寄存器 A5 和 A4。
func_name	要调用的函数的名称，或用于间接调用的符号/机器寄存器的名称。不允许使用寄存器对。被调用函数的标签必须在文件中定义。如果函数的代码不在文件中，则必须使用 .global 或 .ref 指令定义标签（有关详细信息，请参阅 *TMS320C6000 汇编语言工具用户指南*）。如果您要调用 C/C++ 函数，则必须使用该函数的适当链接名。有关更多信息，请参阅节 7.12。
arguments	（可选）作为参数传递的符号/机器寄存器。参数按此顺序传递，不能是常量、存储器引用或其他表达式。

默认情况下，编译器会生成 near 调用，如果 near 调用无法到达其目标，链接器会使用 trampoline。要强制执行 far 调用，必须将函数的地址显式加载到寄存器中，然后发出间接调用。例如：

```
MVK   func,reg
MVKH func,reg
.call reg(op1)            ; forcing a far call
```

如果要使用 * 进行间接访问，则必须遵守 C/C++ 语法规则，并使用以下替代语法：

.call [*ret_reg* =] (* *ireg*)([*arg1, arg2,...*])

例如：

```
.call  (*driver)(op1, op2) ; indirect call
.reg   driver
.call  driver(op1, op2)    ; also an indirect call
```

以下是使用 .call 语法的其他有效示例。

```
.call  fir(x, h, y)           ; void function
.call  minimal( )             ; no arguments
.call  sum = vecsum(a, b)     ; returns an int
.call  hi:lo = _atol(string)  ; returns a long
```

由于您可以在可使用符号寄存器的任何位置使用机器寄存器名称，因此您或许可以更改函数调用约定。例如：

```
.call   A6 = compute()
```

结果似乎在 A6 中返回，而不是在 A4 中返回。这不正确。使用机器寄存器不会覆盖调用约定。从 *compute* 函数返回在 A4 中返回的结果后，MV 指令会将结果传输到 A6。

.call (continued)

调用函数

示例

下面是一个完整的 .call 示例：

```
        .global _main
        .global _puts, _rand, _ltoa
        .sect   ".const"
string1:        .string "The random value returned is ", 0
string2:        .string " ", 10, 0  ; '10' == newline
        .bss    charbuf, 20
        .text
_main:  .cproc
        .reg    random_value, bufptr, ran_val_hi:ran_val_lo
        .call   random_value = _rand()               ; get a random value
        MVKL    string1, bufptr                      ; load address of string1
        MVKH    string1, bufptr
        .call   _puts(bufptr)                        ; print out string1
        MV      random_value, ran_val_lo
        SHR     ran_val_lo, 31, ran_val_hi           ; sign extend random value
        .call   _ltoa(ran_val_hi:ran_val_lo, bufptr) ; convert it to a string
        .call   _puts(bufptr)                        ; print out the random
value
        MVKL    string2, bufptr                      ; load address of string2
        MVKH    string2, bufptr
        .call   _puts(bufptr)                        ; print out a newline
        .endproc
```

.circ

声明循环寄存器

语法 **.circ** *symbol*$_1$ /*register*$_1$ [, *symbol*$_2$ /*register*$_2$, ...]

说明 **.circ** 指令为机器寄存器分配符号寄存器名称，并声明符号寄存器可用于循环寻址。然后，编译器将变量分配给寄存器，并确保在这种情况下所有代码转换都是安全的。必须插入用于循环寻址的设置/拆卸代码。

符号[symbol]	要分配给寄存器的有效符号名称。该变量最长为 128 个字符，并且必须以字母开头。该变量的其余字符可以是字母数字字符、下划线 (_) 和美元符号 ($) 的组合。
register	要分配变量的实际寄存器的名称。

编译器假设使用显式声明的循环寻址变量作为地址指针来推测任何负载是安全的，并且可以利用该假设来执行优化。

当使用 .circ 指令声明符号时，无需使用 .reg 指令声明该符号。

.circ 指令等效于使用带循环声明的 .map。

示例 此处将符号名称 Ri 分配给实际的机器寄存器 Mi，并声明 Ri 可能用于循环寻址。

```
   .CIRC R1/M1、R2/M2 ...
```

.cproc/.endproc

定义 C 可调用过程

语法

label **.cproc** [*argument*$_1$ [**,** *argument*$_2$ **,** ⋯]]

.endproc

说明

使用 **.cproc/.endproc** 指令对来分隔代码段，您希望汇编优化器优化该代码段并将其视为 C/C++ 可调用函数。该代码段称为过程。cproc 指令与 .proc 指令的相似之处在于，您在段的开头使用 .cproc，在段的末尾使用 .endproc。通过这种方法，您可以设置要优化的汇编代码段，例如函数。这些指令必须成对使用；如果没有相应的 .endproc,则不要使用 .cproc。使用 .cproc 指令指定标签。线性汇编文件中可以有多个过程。

.cproc 指令与 .proc 指令的不同之处在于，编译器将 .cproc 区域视为 C/C++ 可调用函数。汇编优化器会在 .cproc 区域中自动执行一些操作，以使函数符合 C/C++ 调用约定和 C/C++ 寄存器使用约定。

这些操作包括：

- 当您使用入口保存寄存器（A10 到 A15 和 B10 到 B15）时，汇编优化器会将这些寄存器保存在栈上并在过程结束时恢复其原始值。
- 如果编译器无法将机器寄存器分配给使用 .reg 指令指定的符号寄存器名称（请参阅 .reg 主题），它会使用本地临时栈变量。使用 .cproc，编译器管理栈指针并确保在栈上为这些变量分配空间。

如需更多信息，请参阅节 8.3 和节 8.4。

使用可选的 *argument* 表示函数参数。参数条目与 C/C++ 函数中声明的参数非常相似。.cproc 指令的参数可以是以下类型：

- **机器寄存器名称**。如果您指定机器寄存器名称，则其在参数列表中的位置必须与 C 的参数传递约定相对应（请参阅节 8.4）。例如，C/C++ 编译器将第一个参数传递给寄存器 A4 中的函数。这意味着 .cproc 指令中的第一个参数必须是 A4 或符号名称。最多可将十个参数与 .cproc 指令一同使用。
- **变量名称**。如果您指定变量名称，则汇编优化器会确保将变量名称分配给适当的参数传递寄存器，或将参数传递寄存器复制到为变量名称分配的寄存器。例如，C/C++ 调用中的第一个参数在寄存器 A4 中传递，因此如果指定以下 .cproc 指令：

```
frame   .cproc arg1
```

汇编优化器要么将 arg1 分配给 A4，要么将 arg1 分配给不同的寄存器（例如 B7）和 MV A4，B7 是自动生成的。

.cproc/.endproc (continued)

定义 C 可调用过程

- **寄存器对**。寄存器对指定为 arghi:arglo，表示 40 位参数或 64 位类型的双精度参数。

  例如，.cproc 定义如下：

```
_fcn:  .cproc  arg1, arg2hi:arg2lo, arg3, B6, arg5, B9:B8
       ...
       .return res
       ...
       .endproc
```

  对应于声明如下的 C 函数：

```
int fcn(int arg1, long arg2, int arg3, int arg4, int arg5, long arg6);
```

  在此示例中，.cproc 的第四个参数是寄存器 B6。这是允许的，因为在 B6 中传递了 C/C++ 调用约定中的第四个参数。.cproc 的第六个参数是实际的寄存器对 B9:B8。这是允许的，因为 C/C++ 调用约定中的第六个参数在 B8 或 B9:B8 中传递了 long。
- **四倍字寄存器**（仅限 C6600）。四倍字寄存器指定为 r3:r2:r1:r0，表示 128 位类型__x128_t。请参阅示例 5-4。

如果从 C++ 源代码调用过程，则必须为过程标签使用适当的链接名。否则，您可以使用 extern C 声明强制使用 C 命名约定。请参阅 节 7.12 和 节 8.6 了解详情。

当 .endproc 与 .cproc 指令一起使用时，它不能有参数。.cproc 区域的 *live out* 集由出现在 .cproc 区域中的任何 .return 指令确定。（如果值是在过程之前或过程中定义的，并且用作过程的输出，则它为 *live out*。）从 .cproc 区域返回值由 .return 指令处理。返回分支是在 .cproc 区域中自动生成的。有关更多信息，请参阅 .return 主题。

只有过程中的代码会被优化。汇编优化器会将过程之外的任何代码复制到输出文件，而不会对其进行修改。请参阅节 5.4.1，了解 .cproc 区域中不能出现的指令类型列表。

示例

以下是使用了 .cproc 和 .endproc 的示例：

```
_if_then: .cproc   a, cword, mask, theta
        .reg    cond, if, ai, sum, cntr
        MVK         32,cntr             ; cntr = 32
        ZERO        sum                 ; sum = 0

LOOP:
        AND         cword,mask,cond     ; cond = codeword & mask
 [cond] MVK         1,cond              ; !(!(cond))
        CMPEQ       theta,cond,if       ; (theta == !(!(cond)))
        LDH         *a++,ai             ; a[i]
 [if]   ADD         sum,ai,sum          ; sum += a[i]
 [!if]  SUB         sum,ai,sum          ; sum -= a[i]
        SHL         mask,1,mask         ; mask = mask << 1
 [cntr] ADD         -1,cntr,cntr        ; decrement counter
 [cntr] B           LOOP                ; for LOOP

        .return sum
        .endproc
```

.map

将变量分配给寄存器

语法 **.map** *symbol*$_1$ */ register*$_1$ [**,** *symbol*$_2$ */ register*$_2$, ...]

说明 **.map** 指令为机器寄存器分配符号名称。符号存储在替代符号表中。符号名称和实际寄存器之间的关联在每个线性汇编函数的开头和结尾被擦除。可以在汇编和线性汇编文件中使用 .map 指令。

变量	要分配给寄存器的有效符号名称。有效符号最长可以包含 128 个字符，并且必须以字母开头。该变量的其余字符可以是字母数字字符、下划线 (_) 和美元符号 ($) 的组合。
register	要分配变量的实际寄存器的名称。

当使用 .map 指令声明符号时，无需使用 .reg 指令声明该符号。

示例 此处，.map 指令用于将 x 分配给寄存器 A6，将 y 分配给寄存器 B7。这些符号与 move 语句一同使用。

```
    .map x/A6, y/B7
    MV   x, y             ; equivalent to MV A6, B7
```

.mdep

表示存储器依赖

语法 **.mdep** *memref*$_1$ **,** *memref*$_2$

说明 **.mdep** 指令用于标识特定的存储器依赖。

以下是 .mdep 指令参数的说明：

memref	符号参数是存储器引用的名称。

用于命名存储器引用的符号具有与任何汇编符号相同的语法限制。（更多有关符号的信息，请参阅 *TMS320C6000 汇编语言工具用户指南*。）它与符号寄存器位于相同的空间中。您不能对符号寄存器使用相同的名称，也不能对存储器引用进行批注。

.mdep 指令会告知汇编优化器两个存储器引用之间存在依赖关系。

.mdep 指令仅在过程中有效，即在出现 .proc 和.endproc 指令对或 .cproc 和 .endproc 指令对时有效。

示例 在下面的示例中，使用 .mdep 表示两个存储器引用之间的依赖关系。

```
    .mdep ld1, st1
    LDW   *p1++{ld1}, inp1   ;memory reference "ld1"
;other code ...
    STW   outp2, *p2++{st1}  ;memory reference "st1"
```

.mptr

避免存储器组冲突

语法

.mptr {*variable* | *memref*}, *base* [**+** *offset*] [**,** *stride*]

说明

.mptr 指令将寄存器与允许汇编优化器自动确定两个存储器操作是否存在存储器组冲突的信息相关联。如果汇编优化器确定两个存储器操作存在存储器组冲突，则不会并行调度它们。

在以下情况下，会发生存储器组冲突：在给定周期中对单个存储器组进行两次访问导致存储器停止，从而在从存储器读取第二个值的同时停止一个周期的所有流水线操作。更多有关存储器组冲突的信息，包括如何使用 .mptr 指令来防止冲突，请参阅节 5.5。

以下是 .mptr 指令参数的说明：

variable|*memref*	用于标识依赖关系中所涉及加载或存储的寄存器符号或存储器引用的名称。
base	关联相关存储器访问的符号地址
失调电压	起始基符号的偏移量（以字节为单位）。offset 是一个可选参数，默认为 0。
stride	寄存器循环以字节为单位递增。stride 是一个可选参数，默认为 0。

.mptr 指令会告知汇编优化器，当 *symbol* 或 *memref* 用作 LD(B/BU)(H/HU)(W) 或 ST(B/H/W) 指令中的存储器指针时，它会初始化为指向 *base* + *offset*，并在每次循环中按 *stride* 递增。

.mptr 指令仅在过程中有效，即在出现 .proc 和.endproc 指令对或 .cproc 和 .endproc 指令对时有效。

用于基符号名称的*符号地址* 位于与所有其他标签分开的命名空间中。这意味着符号寄存器或汇编标签的名称可以与存储器组基本名称相同。例如：

```
     .mptr  Darray,Darray
```

示例

在下面的示例中，使用 .mptr 来避免存储器组冲突。

```
_blkcp:   .cproc I
          .reg   ptr1, ptr2, tmp1, tmp2
          MVK    0x0, ptr1              ; ptr1 = address 0
          MVK    0x8, ptr2              ; ptr2 = address 8
loop:     .trip  50
          .mptr  ptr1, a+0, 4
          .mptr  foo, a+8, 4
                                        ; potential conflict
          LDW    *ptr1++, tmp1          ; load  *0, bank 0
          STW    tmp1, *ptr2++{foo}     ; store *8, bank 0
   [I]    ADD    -1,i,i                 ; I--
   [I]    B      loop                   ; if (!0) goto loop
          .endproc
```

.no_mdep

函数中没有存储器别名

语法 **.no_mdep**

说明 **.no_mdep** 指令会告知汇编优化器，该函数内不会发生存储器依赖关系，但使用 .mdep 指令指向的依赖关系除外。

示例 以下是使用 .no_mdep 的示例。

```
fn:   .cproc     dst, src, cnt
      .no_mdep   ;no memory aliasing in this function
      ...
      .endproc
```

.pref

将变量分配给集合中的寄存器

语法 **.pref** *symbol* / *register*$_1$ [/*register*$_2$...]

说明 **.pref** 指令传达将变量分配给寄存器列表之一的首选项。该首选项仅在 .cproc 或 .proc 区域中使用，.pref 指令在相应区域中声明，并且仅在相应区域结束前有效。

符号[symbol]	要分配给寄存器的有效符号名称。有效符号最长可以包含 128 个字符，并且必须以字母开头。该符号的其余字符可以是字母数字字符、下划线 (_) 和美元符号 ($) 的组合。
register	要分配变量的实际寄存器列表。

不保证将该符号分配给指定组中的任何寄存器。编译器可能会忽略该首选项。

当使用 .pref 指令声明符号时，无需使用 .reg 指令声明该变量。

示例 此处 x 优先分配给 A6 或 B7。但是，编译器将 x 分配给 B3 (例如) 是正确的。

```
  .PREF x/A6/B7; 首选将 x 分配给 A6 或 B7
```

.proc/.endproc

定义过程

语法

label **.proc** [*variable*$_1$ **[,** *variable*$_2$ **,** ⋯]]

.endproc [*register*$_1$ **[,** *register*$_2$ **,** ⋯]]

说明

使用 **.proc/.endproc** 指令对来分隔您希望汇编优化器优化的代码段。该代码段称为过程。在段的开头使用 .proc，在段的末尾使用 .endproc。通过这种方法，您可以设置要由编译器优化的未调度汇编指令的段。这些指令必须成对使用；如果没有相应的 .endproc，则不要使用 .proc。使用 .proc 指令指定标签。线性汇编文件中可以有多个过程。

使用 .proc 指令中的可选 *variable* 参数指示哪些寄存器处于 live in 状态，并使用 .endproc 指令的可选寄存器参数指示每个过程中哪些寄存器处于 live out 状态。*variable* 可以是实际寄存器或符号名称。例如：

```
   .PROC x, A5, y, B7
...
   .ENDPROC y
```

如果值在过程之前已被定义并用作过程的输入，则它为 *live in*。如果值是在过程之前或过程中定义的，并且用作过程的输出，则它为 *live out*。如果您没有使用 .endproc 指令指定任何寄存器，则假定没有寄存器处于 live out 状态。

只有过程中的代码会被优化。汇编优化器会将过程之外的任何代码复制到输出文件，而不会对其进行修改。

请参阅节 5.4.1，了解 .proc 区域中不能出现的指令类型列表。

示例

下面是使用 .proc 和 .endproc 的块移动示例：

```
move   .proc A4, B4, B0
       .no_mdep
loop:
       LDW    *B4++, A1
       MV     A1, B1
       STW    B1, *A4++
       ADD    -4, B0, B0
 [B0]  B      loop
       .endproc
```

.reg

声明符号寄存器

语法

.reg *symbol*$_1$ [, *symbol*$_2$, ⋯]

说明

.reg 指令允许您对存储在寄存器中的值使用描述性名称。汇编优化器为您选择一个寄存器，这样它的使用就与为对值进行操作的指令选择的功能单元一致。

.reg 指令仅在过程中有效，即在出现 .proc 和.endproc 指令对或 .cproc 和 .endproc 指令对时有效。

明确声明寄存器对（或 C6600 的四倍字寄存器）是可选的。只有当寄存器应作为一对分配时，才有必要这样做，但它们不是这样使用的。最好使用对/四倍字语法声明寄存器对和四倍字寄存器。下面是一个声明寄存器对的例子：

```
   .regA7:A6
```

示例 1

此示例使用与 .proc/.endproc 所示的块移动示例相同的代码，但使用了 .reg 指令：

```
move   .cproc dst, src, cnt
       .reg tmp1, tmp2
loop:
       LDW    *src++, tmp1
       MV     tmp1, tmp2
       STW    tmp2, *dst++
       ADD    -4, cnt, cnt
[cnt]  B      loop
```

请注意该示例与 .proc 示例不同：使用 .reg 声明的符号寄存器被分配为机器寄存器。

示例 2

以下示例中的代码无效，因为 .reg 指令定义的变量不能在定义的过程之外使用：

```
move   .proc A4
       .reg tmp
       LDW    *A4++, top
       MV     top, B5
       .endproc
       MV top, B6    ; WRONG: top is invalid outside of the procedure
```

.rega/.regb

直接对寄存器进行分区

语法

.rega *symbol*$_1$ [, *symbol*$_2$, ⋯]

.regb *symbol*$_1$ [, *symbol*$_2$, ⋯]

说明

可以通过两条指令直接对寄存器进行分区。**.rega** 指令用于将符号名称限制到 A 侧寄存器。**.regb** 指令用于将符号名称限制到 B 侧寄存器。例如：

```
      .REGA y
      .REGB u, v, w
      MV    x, y
      LDW  *u, v:w
```

.rega 和 .regb 指令仅在过程中有效，即在出现 .proc 和.endproc 指令对或 .cproc 和 .endproc 指令对时有效。

当使用 .rega 或 .regb 指令声明符号时，无需使用 .reg 指令声明该符号。

仍然可以使用通过分区指令间接对寄存器进行分区的旧方法。侧边和功能单元说明符仍可用于指令。但是，功能单元说明符 (.L/.S/.D/.M) 和交叉路径信息会被忽略。如果有，侧边说明符会转换为对应符号名称的分区约束。例如：

```
      MV  .1X     z, y     ; translated to .REGA y
      LDW .D2T2 *u, v:w    ; translated to .REGB u, v, w
```

.reserve

保留寄存器

语法 **.reserve** [*register*$_1$ [, *register*$_2$, ⋯]]

说明 **.reserve** 指令会阻止汇编优化器在 .proc 或 .cproc 区域中使用指定的*寄存器*。

如果 .proc 或 .cproc 区域中显式分配了 .reserved 寄存器，则汇编优化器也可以使用该寄存器。例如，变量 tmp1 可以分配给寄存器 A7，即使它在 .reserve 列表中，因为 A7 在 ADD 指令中明确定义：

```
     .cproc
     .reserve  a7
     .reg      tmp1
     ....
     ADD       a6, b4, a7
     ....
     .endproc
```

备注

保留寄存器 A4 和 A5

当位于包含 .call 语句的 .cproc 区域内时，无法在.reserve 语句中指定 A4 和 A5。调用约定要求 A4 和 A5 用作 .call 语句的返回寄存器。

示例 1 此示例中的 .reserve 可保证汇编优化器不会对变量 tmp1 到 tmp5 使用 A10 到 A13 或 B10 到 B13：

```
test .proc    a4, b4
     .reg     tmp1, tmp2, tmp3, tmp4, tmp5
     .reserve a10, a11, a12, a13, b10, b11, b12, b13
     .....
     .endproc a4
```

示例 2 如果可用的寄存器池受到过度限制，汇编优化器可能会生成效率较低的代码。此外，可用寄存器池可能会受到限制，从而无法进行分配，并生成错误消息。例如，以下代码会生成错误，因为所有条件寄存器都已保留，但变量 tmp 需要条件寄存器：

```
     .cproc ...
     .reserve a1,a2,b0,b1,b2
     .reg tmp
     ....
[tmp] ....
     ....
     .endproc
```

.return

将值返回到 C 可调用过程

语法

.return [*argument*]

说明

.return 指令函数与 C/C++ 代码中的 return 语句等效。根据 C/C++ 调用约定，它将可选参数放置在相应的寄存器中，以便返回值（请参阅节 8.4）。

可选 *argument* 具有以下含义：

- 零参数表示 .cproc 区域没有返回值，类似于 C/C++ 代码中的 void 函数。
- 参数表示 .cproc 区域具有 32 位返回值，类似于 C/C++ 代码中的 int 函数。
- 格式为 hi:lo 的寄存器对表示 .cproc 区域具有 40 位长、64 位长或 64 位类型的 double 返回值；类似于 C/C++ 代码中的 long/long long/double 函数。

.return 指令的参数可以是符号寄存器名称或机器寄存器名称。

在 .cproc 区域中的所有 return 语句必须与返回值的类型一致。在同一 .cproc 区域中将 .return arg 与 .return hi:lo 混合是不合法的。

.return 指令是无条件的。若要执行条件 .return，只需在 .return 周围使用条件分支。汇编优化器会删除分支并生成适当的条件代码。例如，要在条件 cc 为真时返回，将返回编码为：

```
[!cc]  B   around
       .return
around:
```

示例

此示例使用符号寄存器 tmp 和机器寄存器 A5 作为 .return 参数：

```
     .cproc ...
     .reg tmp
     ...
     .return tmp= legal symbolic name
     ...
     .return a5 = legal actual name
```

.trip

指定行程计数值

语法

label **.trip** *minimum value* [**,** *maximum value*[**,** *factor*]]

说明

.trip 指令用于指定行程计数的值。*trip count* 指示循环迭代的次数。.trip 指令仅在过程中有效。以下是 .trip 指令参数的说明：

label	标签表示循环的开始。这是必需参数。
minimum value	循环可以迭代的最小次数。这是必需参数。默认为 1。
maximum value	循环可以迭代的最大次数。maximum value 是一个可选参数。
因数	用于确定循环迭代次数的系数，以及 *minimum value* 和 *maximum value*。*factor* 为 2 表示循环始终执行偶数次，允许编译器展开一次；这样可以提高性能。在本示例中，循环会执行 8 的整数倍次，总数介于 8 次和 48 次之间： `loop:  .trip 8, 48, 8` 当指定最大值时，系数是可选的。

如果汇编优化器无法确保行程计数足够大，无法对循环进行流水线处理以实现卓越性能，则会生成同一循环的流水线版本和非流水线版本。这使其中一个循环成为*冗余循环*。进行流水线处理或未进行流水线处理的循环是基于行程计数与循环可以并行执行的迭代次数的比较执行的。如果行程计数大于或等于并行迭代的次数，则执行进行流水线处理的循环；否则执行未进行流水线处理的循环。更多有关冗余循环的信息，请参阅节 4.7。

您无需为每个循环指定 .trip 指令；但是，如果您知道循环迭代了一定次数，则应使用 .trip。这通常意味着不会生成冗余循环（除非最小值确实很小），从而节省代码大小和执行时间。

如果知道循环每次调用时总是执行相同的次数，则还应定义最大值（其中最大值等于最小值）。编译器现在可以展开循环，从而提高性能。

当您使用中断灵活性选项 (--interrupt_threshold=*n*) 进行编译时，使用 .trip 最大值允许编译器确定循环可以执行的最大周期数。然后，编译器将该值与 --interrupt_threshold 选项给出的阈值进行比较。有关更多信息，请参阅节 3.12。

示例

.trip 指令指出，调用 w_vecsum 例程时，循环将执行 16、24、32、40 或 48 次。

```
w_vecsum:   .cproc  ptr_a, ptr_b, ptr_c, weight, cnt
            .reg    ai, bi, prod, scaled_prod, ci
            .no_mdep
loop:       .trip 16, 48, 8
            ldh     *ptr_a++, ai
            ldh     *ptr_b++, bi
            mpy     weight, ai, prod
            shr     prod, 15, scaled_prod
            add     scaled_prod, bi, ci
            sth     ci, *ptr_c++
  [cnt]     sub     cnt, 1, cnt
  [cnt]     b       loop
            .endproc
```

.volatile

将存储器引用声明为易失性

语法 **.volatile** *memref*$_1$ [, *memref*$_2$, ⋯]

说明 **.volatile** 指令可让您将存储器引用指定为易失性。不会删除易失性加载和存储。易失性加载和存储不会相对于其他易失性加载和存储进行重新排序。

如果 .volatile 指令引用的存储器位置在中断期间可以修改，请使用 --interrupt_threshold=1 选项进行编译，以确保可以中断引用易失性存储器位置的所有代码。

示例 st 和 ld 存储器引用被指定为易失性。

```
    .volatile st, ld
    STW  W, *X{st}          ; volatile store
    STW  U, *V
    LDW *Y{ld}, Z           ; volatile load
```

5.4.1 过程中不允许使用的指令

在 .cproc 或 .proc topic 区域中不允许使用以下类型的指令：

- 可以读取堆栈指针（寄存器 B15），但不能写入。在 .proc 或 .cproc 区域中不允许使用写入 B15 的指令。栈空间可由 .proc 或 .cproc 区域中的汇编优化器分配，以存储临时值。要分配此存储区域，栈指针在进入该区域时递减，在退出该区域时递增。由于栈指针可以在进入该区域时更改值，因此汇编优化器不允许使用更改栈指针寄存器的代码。
- 在 .proc 或 .cproc 区域中不允许使用间接分支，因此不能绕过 .proc 或 .cproc 区域退出协议。下面是间接分支的示例：

```
       B   B4<=   illegal
```

- 不允许直接分支到 .proc 或 .cproc 区域中未定义的标签，因此不能绕过 .proc 或 .cproc 区域退出协议。下面是 .proc 区域之外的直接分支示例：

```
       .proc
       ...
       B    outside     = illegal
       .endproc
   outside:
```

- 不允许直接分支到与 .proc 指令关联的标签。如果需要将分支返回线性汇编函数的起始位置，则使用 .call 指令。下面是直接分支到 .proc 指令标签的示例：

```
    _func:  .proc
            ...
            B _func    <= illegal
            ...
            .endproc
```

- .if/.endif 循环必须完全位于 proc 或 .cproc 区域内部或外部。不允许 .if/.endif 循环的一部分位于 .proc 或 .cproc 区域内，而另一部分位于 .proc 或 .cproc 区域之外。以下是两个合法 .if/.endif 循环的示例。第一个循环位于 .cproc 区域外，第二个循环位于 .proc 区域内：

```
       .if
       .cproc
       ...
       .endproc
       .endif
       .proc
       .if
       ...
       .endif
       .endproc
```

这些非法示例 .if/.endif 循环部分位于 .cproc 或 .proc 区域内部，部分位于 .cproc 或 .proc 区域外部：

```
       .if
       .cproc
       .endif
       .endproc
       .proc
       .if
       ...
       .else
       .endproc
       .endif
```

- 以下汇编指令不能从线性汇编中使用：
  - EFI
  - SPLOOP、SPLOOPD 和 SPLOOPW 以及所有其他与环路缓冲器相关的指令
  - ADDKSP 和 DP 相对寻址

5.5 避免与汇编优化器发生存储器组冲突

C6000 系列的内部存储器因器件而异。请参阅相应的器件数据表，以确定特定器件中的存储器空间。本节讨论了如何编写代码以避免存储器组冲突。

大多数 C6000 器件使用交错式存储器组方案，如图 5-1 所示。图中的每个数字代表一个字节地址。来自地址 0 的加载字节 (LDB) 指令会加载组 0 中的字节 0。来自地址 0 的加载半字 (LDH) 会加载字节 0 和 1 中的半字值，这两个字节也位于组 0 中。来自地址 0 的加载字 (LDW) 会加载组 0 和 1 中的字节 0 至 3。

由于每个存储器组是单端口存储器，因此每个周期只允许对每个存储器组进行一次访问。在给定周期内对单个组进行两次访问会导致存储器停止，从而在从存储器读取第二个值的同时使所有流水线操作暂停一个周期。每个周期允许进行两次存储器操作，只要它们不访问同一个存储器组，就不会发生任何停止。

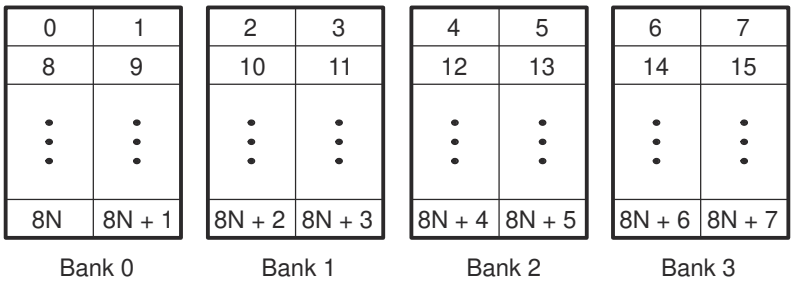


图 5-1. 4 组交错存储器

对于具有多个存储器空间的器件（图 5-2），对一个存储器空间中组 0 的访问不会干扰对另一个存储器空间中组 0 的访问，并且不会发生流水线停滞。

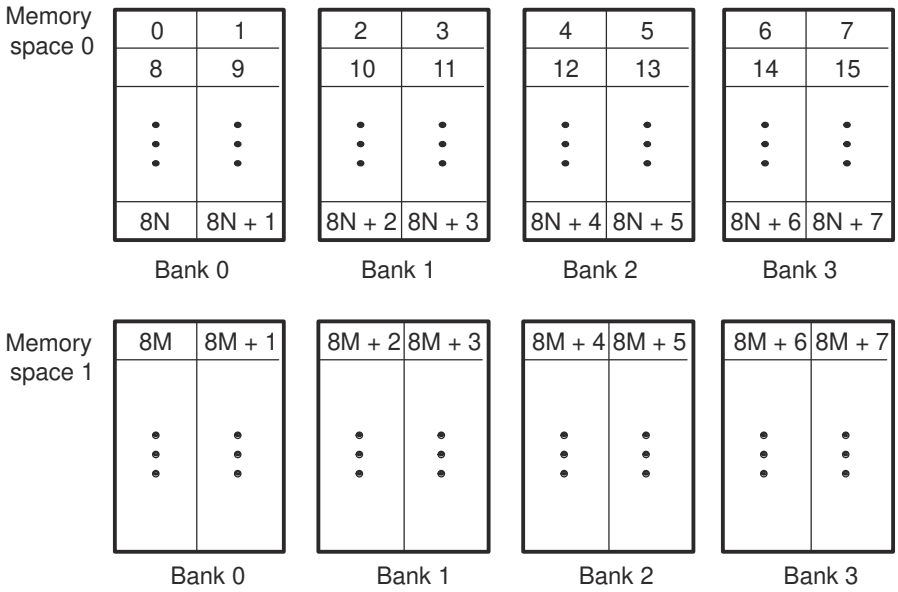


图 5-2. 具有两个存储器空间的 4 组交错存储器

5.5.1 防止存储器组冲突

汇编优化器假设存储器操作不存在组冲突。如果确定两个存储器操作在任何循环迭代上都存在组冲突，则*不*会并行调度操作。汇编优化器仅检查其尝试软件流水线的那些循环的存储器组冲突。

存储器组分析所需的信息指示基址、偏移、跨度、宽度和迭代增量。宽度由存储器访问类型隐式确定。迭代增量由汇编优化器在为软件流水线构建调度时确定。基址、偏移和跨度由 load 和 store 指令和/或 .mptr 指令提供。

通过使用 .mptr 指令，线性汇编中的 LD(B/BU)(H/HU)(W) 或 ST(B/H/W) 操作可以隐式具有与其关联的存储器组信息。.mptr 指令将寄存器与允许汇编优化器自动确定两个存储器操作是否存在组冲突的信息相关联。如果汇编优化器确定两个存储器操作存在存储器组冲突，则不会在软件流水线循环中并行调度这两个操作。语法为：

```
.mptr variable , base + offset , stride
```

例如：

```
.mptr a_0,a+0,16
.mptr a_4,a+4,16
LDW *a_0++[4], val1  ; base=a, offset=0, stride=16
LDW *a_4++[4], val2  ; base=a, offset=4, stride=16
.mptr dptr,D+0,8
LDH *dptr++, d0      ; base=D, offset=0, stride=8
LDH *dptr++, d1      ; base=D, offset=2, stride=8
LDH *dptr++, d2      ; base=D, offset=4, stride=8
LDH *dptr++, d3      ; base=D, offset=6, stride=8
```

在此示例中，每次存储器访问后 dptr 的偏移都会更新。仅当指针被常量修改时，偏移才会更新。在前/后递增/递减寻址模式下，会发生此操作。

有关更多信息，请参阅 .mptr 主题。

示例 5-6 显示了从正在进行软件流水线的循环中提取的加载和存储。

示例 5-6. 指定存储器组信息的加载和存储指令

```
.mptr   Ain,IN,-16
.mptr   Bin,IN-4,-16

.mptr   Aco,COEF,16
.mptr   Bco,COEF+4,16

.mptr   Aout,optr+0,4
.mptr   Bout,optr+2,4
LDW          *Ain--[2],Ain12          ; IN(k-I) & IN(k-I+1)
LDW          *Bin--[2],Bin23          ; IN(k-I-2) & IN(k-I-1)
LDW          *Ain--[2],Ain34          ; IN(k-I-4) & IN(k-I-3)
LDW          *Bin--[2],Bin56          ; IN(k-I-6) & IN(k-I-5)

LDW          *Bco++[2],Bco12          ; COEF(I) & COEF(I+1)
LDW          *Aco++[2],Aco23          ; COEF(I+2) & COEF(I+3)
LDW          *Bco++[2],Bin34          ; COEF(I+4) & COEF(I+5)
LDW          *Aco++[2],Ain56          ; COEF(I+6) & COEF(I+7)

STH          Assum,*Aout++[2]         ; *oPtr++ = (r >> 15)
STH          Bssum,*Bout++[2]         ; *oPtr++ = (I >> 15)
```

5.5.2 避免存储器组冲突的点积示例

示例 5-7 中的 C 代码实现了点积函数。内部循环展开一次，以利用 C6000 对单个 32 位寄存器中的两个 16 位数据项进行操作的能力。LDW 指令用于加载两个连续的短整型值。示例 5-8 中的线性汇编指令实现了 dotp 循环内核。示例 5-9 显示了由汇编优化器确定的循环内核。

对于此循环内核，有两个与数组 a[ ] 和 b[ ] 相关的限制：

- 由于正在使用 LDW，因此数组必须对齐以从字边界开始。

- 为了避免存储器组冲突，一个数组必须在组 0 中开始，而另一个数组必须在组 2 中开始。如果它们在同一组中开始，则每个周期都会发生存储器组冲突，并且由于存储器组停止，环路会每两个周期（而不是每个周期）计算一次结果。例如：

  组冲突：

```
      MVK   0, A0
   || MVK   8, B0
      LDW   *A0, A1
```

  无组冲突：

```
      MVK   0, A0
   || MVK   4, B0
      LDW   *A0, A1
   || LDW   *B0, B1
```

示例 5-7. 点积的 C 代码

```
int dot(short a[], short b[])
{
    int sum0 = 0, sum1 = 0, sum, I;
    for (I = 0; I < 100/2; I+= 2)
    {
        sum0 += a[i] * b[i];
        sum1 += a[i + 1] * b[i + 1];
    }
    return sum0 + sum1;
}
```

示例 5-8. 点积的线性汇编

```
_dot:  .cproc  a, b
        .reg   sum0, sum1, I
        .reg    val1, val2, prod1, prod2
        MVK     50,i ; I = 100/2
        ZERO    sum0 ; multiply result = 0
        ZERO    sum1 ; multiply result = 0
loop:   .trip 50
        LDW     *a++,val1        ; load a[0-1]  bank0
        LDW     *b++,val2        ; load b[0-1]  bank2
        MPY     val1,val2,prod1  ; a[0] * b[0]
        MPYH    val1,val2,prod2  ; a[1] * b[1]
        ADD     prod1,sum0,sum0  ; sum0 += a[0] * b[0]
        ADD     prod2,sum1,sum1  ; sum1 += a[1] * b[1]
    [I] ADD     -1,i,i           ; I--
    [I] B       loop             ; if (!I) goto loop
        ADD     sum0,sum1,A4     ; compute final result
        .return A4
        .endproc
```

示例 5-9. 点积软件流水线内核

```
L2:     ; PIPED LOOP KERNEL
          ADD    .L2     B7,B4,B4           ; |14| <0,7>  sum0 += a[0]*b[0]
||        ADD    .L1     A5,A0,A0           ; |15| <0,7>  sum1 += a[1]*b[1]
||        MPY    .M2X    B6,A4,B7           ; |12| <2,5>  a[0] * b[0]
||        MPYH   .M1X    B6,A4,A5           ; |13| <2,5>  a[1] * b[1]
|| [ B0]  B      .S1     L2                 ; |18| <5,2>  if (!I) goto loop
|| [ B0]  ADD    .S2     0xffffffff,B0,B0   ; |17| <6,1>  I--
||        LDW    .D2T2   *B5++,B6           ; |10| <7,0>  load a[0-1] bank0
||        LDW    .D1T1   *A3++,A4           ; |11| <7,0>  load b[0-1] bank2
   || LDW   *B0, B1
```

并不总是能够完全控制数组和其他存储器对象的对齐方式。当将指针传递给函数时尤其如此，并且每次调用该函数时该指针可能具有不同的对齐方式。解决此问题的方法是编写一个不能有存储器命中的点积例程。这样就不需要数组使用不同的存储器组。

如果点积循环内核展开一次，则在循环内核中执行四条 LDW 指令。假设只知道数组 a 和 b 的组对齐是字对齐（其他一无所知），则可以对数组访问做出的唯一安全假设是 a[0-1] 不能与 a[2-3] 发生冲突，而 b[0-1] 不能与 b[2-3] 发生冲突。示例 5-10 显示了展开的循环内核。

示例 5-10. 示例 5-8 中的点积展开以防止存储器组冲突

```
_dotp2: .cproc   a_0, b_0
        .reg    a_4, b_4, sum0, sum1, I
        .reg    val1, val2, prod1, prod2
        ADD     4,a_0,a_4
        ADD     4,b_0,b_4
        MVK     25,i    ; I = 100/4
        ZERO    sum0    ; multiply result = 0
        ZERO    sum1    ; multiply result = 0
        .mptr   a_0,a+0,8
        .mptr   a_4,a+4,8
        .mptr   b_0,b+0,8
        .mptr   b_4,b+4,8

loop:   .trip   25
        LDW     *a_0++[2],val1  ; load a[0-1] bankx
        LDW     *b_0++[2],val2  ; load b[0-1] banky
        MPY     val1,val2,prod1 ; a[0] * b[0]
        MPYH    val1,val2,prod2 ; a[1] * b[1]
        ADD     prod1,sum0,sum0 ; sum0 += a[0] * b[0]
        ADD     prod2,sum1,sum1 ; sum1 += a[1] * b[1]
        LDW     *a_4++[2],val1  ; load a[2-3] bankx+2
        LDW     *b_4++[2],val2  ; load b[2-3] banky+2
        MPY     val1,val2,prod1 ; a[2] * b[2]
        MPYH    val1,val2,prod2 ; a[3] * b[3]
        ADD     prod1,sum0,sum0 ; sum0 += a[2] * b[2]
        ADD     prod2,sum1,sum1 ; sum1 += a[3] * b[3]
    [I] ADD     -1,i,i          ; I--
    [I] B       loop            ; if (!0) goto loop
         ADD     sum0,sum1,A4    ; compute final result
        .return A4
        .endproc
```

目标是查找以下指令并行的软件流水线：

```
    LDW *a0++[2],val1 ; load a[0-1]  bankx
||  LDW *a2++[2],val2 ; load a[2-3]  bankx+2
    LDW *b0++[2],val1 ; load b[0-1]  banky
||  LDW *b2++[2],val2 ; load b[2-3]  banky+2
```

示例 5-11. 从示例 5-9 展开的点积内核

```
L2:      ; PIPED LOOP KERNEL
   [ B1]  SUB    .S2    B1,1,B1         ; <0,8>
||         ADD    .L2    B9,B5,B9        ; |21| <0,8>  ^  sum0 += a[0] * b[0]
||         ADD    .L1    A6,A0,A0        ; |22| <0,8>  ^  sum1 += a[1] * b[1]
||         MPY    .M2X   B8,A4,B9        ; |19| <1,6>  a[0] * b[0]
||         MPYH   .M1X   B8,A4,A6        ; |20| <1,6>  a[1] * b[1]
|| [ B0]  B      .S1    L2              ; |32| <2,4>  if (!I) goto loop
|| [ B1]  LDW    .D1T1  *A3++(8),A4     ; |24| <3,2>  load a[2-3] bankx+2
|| [ A1]  LDW    .D2T2  *B6++(8),B8     ; |17| <4,0>  load a[0-1] bankx
   [ A1]  SUB    .S1    A1,1,A1         ; <0,9>
||         ADD    .L2    B5,B9,B5        ; |28| <0,9>  ^  sum0 += a[2] * b[2]
||         ADD    .L1    A6,A0,A0        ; |29| <0,9>  ^  sum1 += a[3] * b[3]
||         MPY    .M2X   A4,B7,B5        ; |26| <1,7>  a[2] * b[2]
||         MPYH   .M1X   A4,B7,A6        ; |27| <1,7>  a[3] * b[3]
|| [ B0]  ADD    .S2    -1,B0,B0        ; |31| <3,3>  I--
|| [ A1]  LDW    .D2T2  *B4++(8),B7     ; |25| <4,1>  load b[2-3] banky+2
|| [ A1]  LDW    .D1T1  *A5++(8),A4     ; |18| <4,1>  load b[0-1] banky
```

如果没有示例 5-10 中的 .mptr 指令，a[0-1] 和 b[0-1] 的加载将并行调度，而 a[2-3] 和 b[2-3] 的加载也可能并行调度。这会导致每个周期中有 50% 的几率会发生存储器冲突。但是，示例 5-11 中显示的循环内核不会发生存储器组冲突。

在示例 5-8 中，如果使用了 .mptr 指令来指定 a 和 b 指向不同的基址，则汇编优化器不会找到 1 周期循环内核的调度，因为总是会发生存储器组冲突。但是，它会找到一个 2 周期循环内核的调度。

5.5.3 索引指针的存储器组冲突

在确定索引存储器访问的存储器组冲突时，有时需要指定一对存储器访问始终发生冲突，或指定它们从不发生冲突。这可以通过使用具有 0 跨度的 .mptr 指令来实现。

跨度为 0 表示无论迭代增量如何，存储器访问之间都存在常量关系。本质上，汇编优化器仅使用基址、偏移和宽度来确定存储器组冲突。请记住，跨度是可选的，默认为 0。

在示例 5-12 中，.mptr 指令用于指定哪些存储器访问发生冲突，哪些从不发生冲突。

示例 5-12. 对索引指针使用 .mptr

```
      .mptr a,RS
      .mptr b,RS
      .mptr c,XY
      .mptr d,XY+2
      LDW     *a++[i0a],A0  ; a and b always conflict with each other
      LDW     *b++[i0b],B0  ;
      STH     A1,*c++[i1a]    ; c and d never conflict with each other
      STH     B2,*d++[i1b]    ;
```

5.5.4 存储器组冲突算法

汇编优化器使用以下过程来确定两个存储器访问指令是否可能存在存储器组冲突：

1. 如果这两个访问都没有存储器组信息，它们不会发生冲突。
2. 如果两个访问不具有相同的基址，它们会发生冲突。
3. 偏移、跨度、访问宽度和迭代增量用于确定是否会发生存储器组冲突。汇编优化器使用对访问模式的直接分析，并确定它们是否访问过同一相对组。跨度和偏移值始终以字节表示。

   迭代增量是在软件流水线中调度的存储器引用的循环迭代中的差值。例如，假设有三个指令 A、B、C 和一个具有单周期内核的软件流水线，A 和 C 的迭代增量为 2：

```
   A
   B   A
   C   B   A
       C   B
           C
```

5.6 存储器别名消歧

当两条指令可以访问同一存储器位置时，会发生存储器别名使用。此类存储器引用被称为有歧义。存储器别名消歧是确定何时不可能出现此类歧义的过程。当您无法确定两个存储器引用是否存在歧义时，您便认为它们有歧义。这与两个指令引用之间具有存储器依赖的说法相同。

指令之间的依赖限制了指令调度，包括软件流水线调度。一般而言，依赖越少，选择调度的自由度越大，最终调度的执行效果越好。

5.6.1 汇编优化器如何处理存储器引用（默认）

汇编优化器假定存储器引用设有别名，除非它可以其他方式证明。

由于汇编优化器中的别名分析非常有限，因此这种假设通常过于保守。在这种情况下，由于假定的存储器别名，额外的指令依赖关系可能会导致汇编优化器发出并行性较低且性能不好的指令调度。为了处理这些情况，汇编优化器提供了一个选项和两条指令。

5.6.2 使用 --no_bad_aliases 选项处理存储器引用

在汇编优化器中，--no_bad_aliases 选项意味着没有存储器引用相互依赖。--no_bad_aliases 选项对于 C/C++ 编译器来说意义不同。C/C++ 编译器会解释 -no_bad_aliases 开关，以指示不会发生几种特定的存储器别名使用的情况。更多有关使用 --no_bad_aliases 选项的信息，请参阅节 4.12.2。

5.6.3 使用 .no_mdep 指令

您可以在 .(c)proc 函数中的任何位置指定 .no_mdep 指令。无论何时使用它，您都保证该函数内不会发生存储器依赖。

备注

存储器依赖异常

对于 --no_bad_aliases 和 .no_mdep 这两种方法，汇编优化器会识别您使用 .mdep 指令指出的任何存储器依赖。

5.6.4 使用 .mdep 指令来识别特定的存储器依赖关系

使用 .mdep 指令，您可以为每个存储器引用添加名称注释来标识特定的存储器依赖关系，并使用这些名称与 .mdep 指令来指示实际依赖关系。对存储器引用进行批注需要在汇编流中的存储器引用旁边添加信息。紧随存储器引用后添加以下内容：

{ *memref* **}**

memref 具有与任何汇编符号相同的语法限制。（更多有关符号的信息，请参阅 *TMS320C6000 汇编语言工具用户指南*。）它与符号寄存器位于同一命名空间中。您不能对符号寄存器使用相同的名称，也不能对存储器引用进行批注。

示例 5-13. 对存储器引用进行批注

```
     LDW    *p1++ {ld1}, inp1  ;name memory reference "ld1"
;other code ...
     STW    outp2, *p2++ {st1} ;name memory reference "st1"
*<The directive to indicate...:
.mdep ld1, st1 <<bold>>
```

在上一个示例中指示特定存储器依赖关系的指令如下：

```
.mdep ld1, st1
```

这意味着每当 ld1 在位置 X 访问存储器时，在稍后的代码执行时间，st1 也可能访问位置 X。这相当于在这两条指令之间添加了依赖关系。就软件流水线而言，这两条指令必须保持相同的顺序。ld1 引用必须始终出现在 st1 引用之前；甚至不能并行调度指令。

请务必注意指令从 ld1 到 st1 的方向感应。相反，从 st1 到 ld1，并不暗示。就软件流水线而言，虽然每个 ld1 必须在每个 st1 之前发生，但从迭代 n+1 到迭代 n 中的 st1 调度 ld1 仍然是合法的

示例 5-14 是软件流水线的图片，其中包含不同列中两个不同迭代的指令。在实际指令序列中，同一水平线上的指令是并行的。

示例 5-14. 使用 .mdep ld1、st1 的软件流水线

```
      iteration n             iteration n+1
      -------------           -------------
      LDW { ld1 }
      ...                     LDW { ld1 }
      STW { st1 }             ...
                              STW { st1 }
*<If that schedule...>
   .mdep   st1, ld1
```

如果该调度不起作用，因为迭代 n st1 可能写入迭代 n+1 ld1 应该读取的值，那么您必须注意从 st1 到 ld1 的依赖关系。

```
.mdep st1, ld1
```

这两个指令共同强制使用示例 5-15 中所示的软件流水线。

示例 5-15. 使用 .mdep st1、ld1 和 .mdep ld1、st1 的软件流水线

```
      iteration n              iteration n+1
      -------------            -------------
      LDW { ld1 }
      ...
      STW { st1 }
                               LDW { ld1 }
                               ...
                               STW { st1 }
<Indexed addressing,...>
    .mdep   ld1, st1
    .mdep   st1, ld1
```

索引寻址 *+base[index] 是寻址模式的一个很好的示例，在该模式下，您通常不知道存储器访问的相对顺序，除非它们有时访问同一个位置。要正确地为此例建模，您需要在两个方向上记录依赖关系，并且需要使用这两个指令。

```
.mdep ld1, st1 .mdep st1, ld1
```

5.6.5 存储器别名示例

以下是使用 .mdep 和 .no_mdep 指令的存储器别名示例。

- **示例 1**

  .mdep r1、r2 指令会声明 LDW 必须在 STW 之前。在这种情况下，src 和 dst 可能指向同一个数组。

```
    fn:     .cproc      dst, src, cnt
            .reg        tmp
            .no_mdep
            .mdep       r1, r2
            LDW         *src{r1}, tmp
            STW         cnt, *dst{r2}
            .return     tmp
            .endproc
```

- **示例 2**

  这里，.mdep r2、r1 表示 STW 必须发生在 LDW 之前。由于 STW 在代码中位于 LDW 之后，因此依赖关系跨循环迭代。STW 指令写入一个值，该值可在下一次迭代时由 LDW 指令读取。在这种情况下，将创建一个 6 周期循环。

```
    fn:     .cproc     dst, src, cnt
            .reg        tmp
            .no_mdep
            .mdep       r2, r1
    LOOP:   .trip       100
            LDW         *src++{r1}, tmp
            STW         tmp, *dst++{r2}
     [cnt]  SUB         cnt, 1, cnt
     [cnt]  B           LOOP
            .endproc
```

备注

存储器依赖/组冲突

不要将存储器别名消歧与存储器组冲突的处理混为一谈。这两者看起来可能相似，因为它们都涉及存储器引用以及这些存储器引用对指令调度的影响。别名消歧属于正确性问题，组冲突属于性能问题。与组冲突相比，存储器依赖对指令调度的影响要大得多。最好将这两个主题分开。

备注

易失性引用

对于易失性引用，请使用 .volatile 而不是 .mdep。

This page intentionally left blank.

章节 6

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《*TMS320C6000 汇编语言工具用户指南*》。

6.1 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。

6.1.1 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：

cl6x --run_linker {**--rom_model | --ram_model**} *filenames*
[*options*] [**--output_file=** *name.out*] **--library=** *library* [*lnk.cmd*]

cl6x --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 cl6x --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 6.3.4。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。（《*TMS320C6000 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《*TMS320C6000 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：

```
cl6x --run_linker --ram_model prog1 prog2 prog3 --output_file=prog.out
     --library=rts6600.lib
```

6.1.2 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：

cl6x *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 6.1.1 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 线性汇编文件或编译器选项。节 3.2 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：

```
cl6x prog1.c prog2.c prog3.c --run_linker --rom_model --output_file=prog.out --library=rts6600.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 cl6x --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 6.3.4。

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. C6X_C_OPTION 环境变量中 --run_linker 选项后面的参数

6.1.3 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 C6X_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。

6.2 链接器代码优化

6.2.1 条件链接

使用 ELF 条件链接时，除非引用了该代码或数据段中的至少一个符号，否则链接中不会包含该代码或数据段。

可以使用 RETAIN pragma（节 7.9.30）迫使将包含特定符号的段包含在链接中。可以使用 CODE_SECTION（节 7.9.3）和 DATA_SECTION（节 7.9.6）pragma 迫使将符号分配在特定段中。必须在声明之外的语句中引用该符号，才能迫使将该段包含在链接中。

6.2.2 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。

但是，请注意，如果所有或几乎所有函数都被引用，则使用 --gen_func_subsections 编译器选项可能会导致整体代码大小增大。这是因为任何包含代码的段都必须与 32 字节边界对齐以支持 C6000 分支机制。当不使用 --gen_func_subsections 选项时，源文件中的所有函数通常都放在对齐的公共段中。当使用 --gen_func_subsections 时，源文件中定义的每个函数都放在唯一的段中。每个唯一的段都需要对齐。如果链接需要文件中的所有函数，则代码大小可能会因各个子段的额外对齐填充而增大。因此，--gen_func_subsections 编译器选项有利于与库一起使用，在这些库中，任何一个可执行文件中通常只使用文件中有限数量的函数。如果不使用 --gen_func_subsections 选项，替代方法是将每个函数放在其自己的文件中。

如果未使用此选项，则默认为“off”。如果使用了此选项但既未指定“on”也未指定“off”，则默认为“on”。

6.2.3 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。

6.3 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《*TMS320C6000 汇编语言工具用户指南*》中的链接器说明。

6.3.1 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。

6.3.1.1 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 6.3.4。

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 6-1. 使用 --issue_remarks 选项

```
cl6x --silicon_version=6400+ --issue_remarks main.c --run_linker --rom_model
<Linking>
remark: linking in "libc.a"
remark: linking in "rts64plus.lib" in place of "libc.a"
```

6.3.1.2 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 C6X_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。

cl6x --run_linker {**--rom_model** | **--ram_model**} *filenames* **--library=** *libraryname*

6.3.1.3 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。

6.3.2 运行时初始化

C/C++ 程序在开始执行程序之前需要初始化运行时环境。该初始化由*引导程序*进行。引导程序负责创建堆栈、初始化全局变量并调用 main() 函数。引导程序应该是程序的入口点，通常应该是 RESET 中断处理程序。引导程序负责执行以下任务：

1. 通过初始化 SP 来设置堆栈
2. 设置数据页指针 DP（对于有一个 DP 的架构）
3. 设置配置寄存器
4. 处理 *.cinit* 表以自动初始化全局变量（使用 --rom_model 选项时）
5. 处理 *.pinit* 表以构造全局 C++ 对象。

6. 使用合适的参数调用 main() 函数
7. 当 main() 返回时调用 exit()

当编译 C/C++ 程序并使用 --rom_model 或 --ram_model 时，链接器会自动查找名为 _c_int00 的引导程序。运行时支持库在 boot.c.obj 中提供了一个示例 _c_int00，它会执行所需的任务。如果使用运行时支持库的引导程序，应将 _c_int00 设置为入口点。

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 6.3.4。

6.3.3 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 8.9.2.6 讨论了全局构造函数表的格式。

6.3.4 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 8.9.2.4 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 8.9.2.3）。
- 在*加载时*初始化全局变量。使用 --ram_model 链接器选项（请参阅节 8.9.2.5）。

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 6.1）。

有关 EABI 使用 --rom_model 和 --ram_model 的链接约定的信息，请分别参阅节 8.9.2.3 和节 8.9.2.5。

6.3.5 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 8.1.1。

编译器创建两种基本类型的段：初始化段和未初始化段。表 6-1 总结了初始化段。表 6-2 总结了未初始化段。

表 6-1. 由编译器创建的初始化段

名称	内容
.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 3.6。
.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 *TMS320C6000 汇编语言工具用户指南*）
.c6xabi.exidx	用于异常处理的索引表；只读（请参阅 --exceptions 选项）。
.c6xabi.extab	用于异常处理的展开指令；只读（请参阅 --exceptions 选项）。
.cinit	除非指定了 --rom_mode 链接器选项，否则编译器不会生成 .cinit 段。如果指定了 --rom_mode，链接器就会创建此段，其中包含用于显式初始化的全局变量和静态变量的表。

表 6-1. 由编译器创建的初始化段 (continued)

名称	内容
.const	全局和静态常量变量，包括字符串常量以及局部变量的初始化值。
.data	显式初始化的全局和静态非常量变量。
.fardata	显式初始化的远非常量全局和静态变量。
.init_array	启动时要调用的构造函数表。
.name.load	段 *名称* 的压缩图像；只读 (有关复制表的信息，请参阅 *TMS320C6000 汇编语言工具用户指南*。)
.neardata	显式初始化的近非常量全局和静态变量。
.ovly	复制除引导时间 (.binit) 复制表以外的表。只读数据。
.ppdata	用于基于编译器的分析的数据表 (请参阅 --gen_profile_info 选项) 。
.ppinfo	用于基于编译器的分析的相关性表 (请参阅 --gen_profile_info 选项) 。
.rodata	具有 近 和 常量 限定符的全局和静态变量。
.switch	大型切换语句的跳转表。
.text	用于可执行代码的标准默认段。--gen_func_subsections 选项使代码放置在每个函数 func() 单独的 .text:*func* 段中。
.TI.crctab	生成的 CRC 校验表。只读数据。

表 6-2. 由编译器创建的未初始化段

名称	内容
.bss	未初始化全局和静态变量
.cio	运行时支持库中 stdio 函数的缓冲区
.far	声明为远的全局和静态变量
.stack	栈
.sysmem	用于动态内存分配 (malloc 等) 的内存池 (堆)

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。除代码段外，编译器创建的初始化段和未初始化段不能分配到内部程序内存中。

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 *TMS320C6000 汇编语言工具用户指南*。

6.3.6 链接器命令文件示例

链接器命令文件 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 ，且该命令文件 列出了几个链接器选项：

--rom_model 通知链接器在运行时使用自动初始化功能。

--heap_size 通知链接器将 C 堆大小设置为 0x2000 字节。

--stack_size 通知链接器将栈大小设置为 0x0100 字节。

--library 通知链接器使用存档库文件 rts64plus.lib 进行输入。

要链接该程序，请使用以下语法：

cl6x --run_linker *object_file(s)* **--output_file=** *outfile* **--map_file=** *mapfile* **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《*TMS320C6000 汇编语言工具用户指南*》。

链接器命令文件

```
--rom_model
--heap_size=0x2000
```

```
--stack_size=0x0100
--library=rts64plus.lib
MEMORY
{
    VECS:   o = 0x00000000     l = 0x000000400 /* reset & interrupt vectors     */
    PMEM:   o = 0x00000400     l = 0x00000FC00 /* intended for initialization   */
    BMEM:   o = 0x80000000     l = 0x000010000 /* .bss, .sysmem, .stack, .cinit */
}
SECTIONS
{
    vectors     >       VECS
    .text       >       PMEM
    .data       >       BMEM
    .stack      >       BMEM
    .bss        >       BMEM
    .sysmem     >       BMEM
    .cinit      >       BMEM
    .const      >       BMEM
    .cio        >       BMEM
    .far        >       BMEM
}
```

章节 7

C/C++ 语言实现

受 C6000 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 C6000 支持的 C++ 语言由 ANSI/ISO/IEC 14882:2014 标准定义，但有一些例外。

7.1 TMS320C6000 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 7.14）。

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 7.13。

*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 unsigned short（16 位），但如果设置 --wchar_t=32 选项，也可以是 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 7.4，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 7.13。

7.1.1 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 7.4 (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 7.4。(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 8。有关数据类型详细信息，请参阅节 7.3。(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 7.4。（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

- C6000 支持额外的整数类型 __int40_t 和无符号 __int40_t，它们是有符号和无符号 40 位整数类型。(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)

- __int40_t 和无符号 __int40_t 的秩低于 long long 的秩。__int40_t 和无符号 __int40_t 的秩大于长整型的秩。(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)
- 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。
- 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 7.3 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 3.11.2。(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 shot、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。(6.7.2.1)
- 位字段在一个单元内按字节顺序分配。请参阅节 8.2.2。(6.7.2.1)
- 结构体的非位字段成员按照节 8.2.1 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 7.3.1 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 3.5.2。(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 3.5.2。(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 3.5.2。(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 7.9。(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 7.13.1 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 7.3 中进行了介绍。（5.2.4.2，7.18.2，7.18.3）
- 节 8.2.1 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 8.2.1。(6.5.3.4)

7.2 TMS320C6000 C++ 的特征

根据 ANSI/ISO/IEC 14882:2014 标准 (C++14) 中的定义，C6000 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 7.6。
- 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 2014 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。
- typeinfo 标头中不支持 bad_cast 或 bad_type_id。
- 仅部分支持目标特定类型的常量表达式。
- 不支持新的字符类型（在 C++11 标准中引入）。
- 不支持 Unicode 字符串文字（在 C++11 标准中引入）。
- 不支持文字中的通用字符名称（在 C++11 标准中引入）。
- 不支持原子操作（在 C++11 标准中引入）。
- 不支持原子性和内存模型的数据依赖项排序（在 C++11 标准中引入）。
- 不支持在信号处理程序中允许原子性（在 C++11 标准中引入）。
- 不支持强比较和交换（在 C++11 标准中引入）。
- 不支持双向围栏（在 C++11 标准中引入）。
- 不支持内存模型（在 C++11 标准中引入）。

- 不支持传播异常（在 C++11 标准中引入）。
- 不支持线程本地存储（在 C++11 标准中引入）。
- 不支持动态初始化和并发销毁（在 C++11 标准中引入）。

如果与使用旧版编译器编译的 C++ 目标文件或库链接，为了支持 C++14 所做的更改可能会导致“未定义的符号”错误。如果出现这样的链接时错误，请使用 --no_demangle 命令行选项重新编译 C++ 代码。如果任何未定义的符号名称以 _Z 或 _ZVT 开头，请重新编译整个应用，包括目标文件和库。如果没有库的源代码，请下载库的新编译版本。

7.3 数据类型

表 7-1 列出了 C6000 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。

节 8.2.1 中介绍了数据类型的存储和对齐。

表 7-1. TMS320C6000 C/C++ 数据类型

			范围	
类型	**大小**	**表示**	**最小值**	**最大值**
char、signed char	8 位	ASCII	-128	127
unsigned char	8 位	ASCII	0	255
_Bool、bool	8 位	ASCII	0 (false)	1 (true)
short	16 位	二进制	-32 768	32 767
unsigned short、wchar_t (1)	16 位	二进制	0	65 535
int、signed int	32 位	二进制	-2 147 483 648	2 147 483 647
unsigned int	32 位	二进制	0	4 294 967 295
long、signed long	32 位	二进制	-2 147 483 648	2 147 483 648
unsigned long	32 位	二进制	0	4 294 967 295
__int40_t	40 位	二进制	-549 755 813 888	549 755 813 887
unsigned __int40_t	40 位	二进制	0	1 099 511 627 775
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 (2)	不尽相同	二进制	不尽相同	不尽相同
float	32 位	IEEE 32 位	1.175 494e-38(3)	3.40 282 346e+38
float complex (4)	64 位	2 个 IEEE 32 位数组	1.175 494e-38 分别适用于实部和虚部	3.40 282 346e+38 分别适用于实部和虚部
double	64 位	IEEE 64 位	2.22 507 385e-308(3)	1.79 769 313e+308
double complex (4) (5)	128 位	2 个 IEEE 64 位数组	2.22 507 385e-308 分别适用于实部和虚部	1.79 769 313e+308 分别适用于实部和虚部
long double	64 位	IEEE 64 位	2.22 507 385e-308(3)	1.79 769 313e+308
long double complex (4) (5)	128 位	2 个 IEEE 64 位数组	2.22 507 385e-308 分别适用于实部和虚部	1.79 769 313e+308 分别适用于实部和虚部
指针、引用、指向数据成员的指针	32 位	二进制	0	0xFFFFFFFF

(1) 这是 wchar_t 的默认类型。您可以使用 --wchar_t 选项将 wchar_t 类型更改为 32 位 unsigned int 类型。
(2) 有关枚举类型大小的详细信息，请参阅节 7.3.1。
(3) 数字是最低精度。
(4) 若要使用复杂数据类型，必须包含 <complex.h> 头文件。请参阅节 7.5.1，以了解有关复杂数据类型的更多信息。
(5) 仅 C6600

有符号类型的负值用 2 的补码表示。

C、C99 和 C++ 的这些附加类型被定义为标准类型的同义词：

```
typedef unsigned int  size_t;
typedef int           ptrdiff_t;
typedef unsigned int  wchar_t;
typedef unsigned int  wint_t;
typedef char *        va_list;
```

7.3.1 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，编译器选择此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：

- unsigned char
- signed char
- unsigned short
- signed short
- unsigned int
- signed int
- unsigned long long
- igned long long

会跳过“long”类型，因为其与“int”类型大小相同。

例如，下述枚举类型将会以“unsigned char”作为其基础类型：

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

但下述类型将会以“signed char”作为其基础类型：

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

而下述类型将会以“signed short”作为其基础类型：

```
enum ss { a, b, c, d = -1, e = UCHAR_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。

7.3.2 矢量数据类型

C/C++ 编译器支持在 C/C++ 源文件中使用 TI 向量数据类型。矢量数据类型很有用，因为其可以对处理内核中的自然向量宽度加以利用。矢量数据类型可直接利用该架构上可用的 SIMD 指令，矢量数据类型还提供了从矢量数据对象的抽象模型到寄存器中该数据对象的物理表示的更直接映射。

向量数据类型与数组类似，因为矢量包含特定数量的指定类型元素。但矢量长度只能为 2、3、4、8 或 16。作用在矢量上的内在函数会在可能的情况下进行优化，以便在器件上利用高效的单指令多数据 (SIMD) 指令。

您可以使用 --vectypes 编译器选项来支持矢量数据类型。

C6000 编程模型支持的所有矢量数据类型和相关的内置函数都安装在 C6000 CGT 的“包含”子目录下的“c6x_vec.h”头文件中。任何使用矢量数据类型或任何相关内置函数的 C/C++ 源文件必须在源文件中包含以下内容：

```
#include <c6x_vec.h>
```

矢量类型名称连接元素类型名称和表示矢量长度的数字。生成的矢量包含规定数量的指定类型元素。

向量数据类型和运算的实现严格遵循 OpenCL C 语言规范。有关 OpenCL 向量数据类型和操作的详细说明，请参阅 1.2 版本的 OpenCL 规范，该规范可从 Khronos OpenCL 工作组获取。1.2 版规范的第 6.1.2 节详细说明了 OpenCL C 编程语言中支持的内置向量数据类型。C6000 编程模型提供下述内置向量数据类型：

表 7-2. 矢量数据类型

类型	说明	最大元素数
char*n*	由 *n* 个 8 位有符号整数值组成的矢量。	16
uchar*n*	由 *n* 个 8 位无符号整数值组成的矢量。	16
short*n*	由 *n* 个 16 位有符号整数值组成的矢量。	8
ushort*n*	由 *n* 个 16 位无符号整数值组成的矢量。	8
int*n*	由 *n* 个 32 位有符号整数值组成的矢量。	4
uint*n*	由 *n* 个 32 位无符号整数值组成的矢量。	4
longlong*n*	由 *n* 个 64 位有符号整数值组成的矢量。	2
ulonglong*n*	由 *n* 个 64 位无符号整数值组成的矢量。	2
float*n*	由 *n* 个 32 位单精度浮点值组成的矢量。	4
double*n*	由 *n* 个 64 位双精度浮点值组成的矢量。	8

n 为矢量长度 2、3、4、8 或 16。

例如，"uchar8" 是由 8 个无符号字符组成的矢量；其长度为 8，大小为 64 位。"float4" 是由 4 个浮点元素组成的矢量；其长度为 4，大小为 128 位。

矢量类型与边界对齐，边界等于矢量元素的总大小，最多为 64 位。任何总大小超过 64 位的矢量类型都与 64 位边界对齐（8 字节）。例如，short2 总大小为 32 位，与 4 字节边界对齐。longlong2 总大小为 128 位，与 8 字节边界对齐。

备注

为避免混淆 C6000 对 32 位和 64 位 long 的定义，不提供基本 long long 和无符号 long（“ulong”）的矢量类型。如果要使用标准 long 和 ulong 类型，可以创建一个简单的预处理器宏，例如：`#define long2 longlong2` 或 `#define long2 int2`，具体取决于您要使用的元素类型和大小。

代码生成工具还提供了表示复数类型矢量的扩展。前缀 'c' 用于指示复数类型名称。每种复数类型的矢量元素都包含一个实部和一个虚部（实部占据存储器的低位地址）。复数矢量类型如下：

表 7-3. 复数矢量数据类型

类型	说明	最大元素数
cchar*n*	由 *n* 对 8 位有符号整数值组成的矢量。	8
cshort*n*	由 *n* 对 16 位有符号整数值组成的矢量。	4
cint*n*	由 *n* 对 32 位有符号整数值组成的矢量。	2
clonglong*n*	由 *n* 对 64 位有符号整数值组成的矢量。	1
cfloat*n*	由 *n* 对 32 位浮点值组成的矢量。	2
cdouble*n*	由 *n* 对 64 位浮点值组成的矢量。	1

n 为矢量长度 1、2、4 或 8。请注意，16 不是复数矢量类型的有效矢量长度。例如，“cfloat2”是由 2 个浮点值组成的复数矢量。其长度为 2，大小为 128 位。每个“cfloat2”矢量元素都包含一个实部和一个虚部。

有关与矢量数据类型搭配使用的运算符和内置函数的信息，请参阅 节 7.15。

7.4 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。

7.5 关键字

C6000 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、__near、__far、__cregister、和 __asm。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

7.5.1 complex 关键字

若要使用复杂数据类型，必须包含 <complex.h> 头文件。如果包含此头文件，则所有 C/C++ 模式都可以使用 complex 支持，包括宽松和严格 ANSI 模式以及 C89 和 C99。<complex.h> 头文件实现了复杂数据类型的数学运算和函数支持。

Complex 类型实现为包含两个元素的一个数组。例如，对于以下声明，变量存储为包含两个浮点值的一个数组。数字的实部存储在 x._Vals[0] 中，而数字的虚部存储在 x._Vals[1] 中。

```
float complex x;
```

7.5.2 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。您可以将常量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 far 常量的全局对象放置在 .const 段中。链接器从 ROM 或闪存中分配 .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：

- 如果对象定义中也指定了 *volatile*。例如，`volatile const int x`。假设将 Volatile 关键字分配给 RAM。（不允许程序修改 const volatile 对象，但可能会修改程序外部的内容。）
- 如果对象具有自动存储功能（在栈上分配）。
- 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 const 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 const 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：

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

7.5.3 __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。

在对象上使用 __cregister 关键字时，编译器会将对象的名称与标准控制寄存器列表进行比较（请参阅表 7-4）。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 7-4. 面向 C64x+、C6740 和 C6600 的控制寄存器

寄存器	说明
AMR	寻址模式寄存器
CSR	控制状态寄存器
DNUM	DSP 内核数寄存器
ECR	异常清除寄存器
EFR	异常标志寄存器
GFPGFR	伽罗瓦域乘法控制寄存器
GPLYA	GMPY A 侧多项式寄存器
GPLYB	GMPY B 侧多项式寄存器
ICR	中断清除寄存器
IER	中断启用寄存器
IERR	内部异常报告寄存器
IFR	中断标志寄存器。（IFR 为只读。）
ILC	内部环路计数寄存器
IRP	中断返回指针寄存器
ISR	中断设置寄存器
ISTP	中断服务表指针寄存器
ITSR	中断任务状态寄存器
NRP	非屏蔽性中断返回指针寄存器
NTSR	NMI/例外任务状态寄存器
PCE1	程序计数器，E1 阶段
REP	受限入口点地址寄存器
RILC	重新加载内部环路计数寄存器
SSR	饱和状态寄存器
TSCH	时间戳计数器（高 32）寄存器
TSCL	时间戳计数器（低 32 位）寄存器
TSR	任务状态寄存器

表 7-5 中列出的附加控制寄存器用于 C6740 和 C6600 器件上的浮点运算：

表 7-5. C6740 和 C6600 的附加控制寄存器

寄存器	说明
FADCR	浮点加法器配置寄存器
FAUCR	浮点辅助配置寄存器
FMCR	浮点乘法器配置寄存器

__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 7-4 中的控制寄存器，必须按如下方式声明每个寄存器。c6x.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 ;
```

声明寄存器后，就可以直接使用寄存器名称。有关控制寄存器的详细信息，请参阅 *TMS320C64x/C64x+ DSP CPU 和指令集参考指南* (SPRU732)、*TMS320C66x DSP CPU 和指令集参考指南* (SPRUGH7) 或 *TMS320C674x DSP CPU 和指令集参考指南* (SPRUFE8)。

有关声明和使用控制寄存器的示例，请参阅示例 7-1。

示例 7-1. 定义和使用控制寄存器

```
extern __cregister volatile unsigned int AMR;
extern __cregister volatile unsigned int CSR;
extern __cregister volatile unsigned int IFR;
extern __cregister volatile unsigned int ISR;
extern __cregister volatile unsigned int ICR;
extern __cregister volatile unsigned int IER;
extern __cregister volatile unsigned int FADCR;
extern __cregister volatile unsigned int FAUCR;
extern __cregister volatile unsigned int FMCR;
main()
{
   printf("AMR = %x\n", AMR);
}
```

7.5.4 __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 7.9.20 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。

7.5.5 __near 和 __far 关键字

C6000 C/C++ 编译器使用 __near 和 __far 关键字扩展 C/C++ 语言，以指定如何访问全局变量和静态变量，以及如何调用函数。

在语法上，__near 和 __far 关键字被视为存储类说明符。可以出现在存储类说明符和类型之前、之后或之间。通常，变量声明中只允许使用一个存储类说明符。但是，如果两个存储类说明符中的有一个是 __near 或 __far，则可以在单个声明中使用两个存储类说明符。

以下示例是 __near 和 __far 与其他存储类说明符的合规组合：

```
__far static int x;
static __near int x;
static int __far x;
__far int foo();
static __far int foo();
```

7.5.5.1 near 和 far 数据对象

可以通过以下两种方式访问全局和静态数据对象：

__near 关键字 编译器假定可以相对于数据页指针访问数据项。例如：

```
   LDW    *+dp(_address),a0
```

__far 关键字 编译器无法通过 DP 访问数据项。如果程序数据总量大于 DP 允许的偏移量 (32K)，则可能需要这样做。例如：

```
   MVKL   _address, a1
   MVKH   _address, a1
   LDW    *a1,a0
```

与 near 和 far 声明保持一致。如果对象被定义为 far，则其他 C 文件或头文件中针对该对象的所有外部声明也必须包含 __far 关键字，否则您可能会遇到编译器或链接器错误。如果对象被定义为 near，您可以安全地在其他 C 文件或头文件中将其声明为 __far，但您对该变量的数据访问速度会变慢。

如果您使用 DATA_SECTION pragma，则对象被指示为 far 变量，并且不能被覆盖。如果您在另一个文件中引用此对象，则在另一个源文件中声明此对象时需要使用 *extern __far*。这确保了对变量的访问，因为变量可能不在 .bss 段中。相关详细信息，请参阅节 7.9.6。

备注

在汇编代码中定义全局变量

如果您还使用 .usect 指令在汇编代码中定义了一个全局变量（该变量未在 .bss 段中分配），或者您使用 #pragma DATA_SECTION 指令将变量分配到单独的段；并且您想在 C 代码中引用该变量，则必须将该变量声明为 extern __far。这确保编译器不会尝试通过数据页指针生成对变量的非法访问。

当数据对象未指定 __near 或 __far 关键字时，编译器将使用 far 访问聚合数据，并使用 near 访问非聚合数据。更多有关数据存储模式和数据访问控制方式的信息，请参阅节 8.1.4.1。

7.5.5.2 near 和 far 函数调用

可以通过以下两种方式之一调用函数调用：

__near 关键字 编译器会假定调用的目标在调用方的 ± 1 M 字以内。这里编译器使用相对于 PC 的分支指令。

```
   B      _func
```

__far 关键字 您告知编译器调用不在 ± 1 M 字以内。

```
   MVKL   _func, al
  MVKH    _func, al
   B      _func
```

默认情况下，编译器会生成小存储器模式代码，这意味着系统在处理每个函数调用时都会认为它被声明为 near，除非它实际上被声明为 far。更多有关函数调用的信息，请参阅节 8.1.5。

7.5.6 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某些代码段，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 7.13。

在以下示例中，restrict 关键字用于告诉编译器，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。1999 版的 ANSI/ISO C 标准中描述了 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;
  }
}
```

7.5.7 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;
        }
    }
}
```

使用 volatile 进行编译时，应使用 --interrupt_threshold=1 选项。

7.6 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 6.3.1.1。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。这段代码会略微增加程序的大小。此外，即使从未引发异常，执行时间开销也很小，并且异常处理表的数据大小略有增加，

有关运行时库的详细信息，请参阅节 9.1。

7.7 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 8.3。

7.8 __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 7.14.2。

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 *TMS320C6000 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 8.6.5。

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。

7.9 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 7.9.1)
- CODE_ALIGN (请参阅节 7.9.2)
- CODE_SECTION (请参阅节 7.9.3)
- DATA_ALIGN (请参阅节 7.9.4)
- DATA_MEM_BANK (请参阅节 7.9.5)
- DATA_SECTION (请参阅节 7.9.6)
- diag_suppress、diag_remark、diag_warning、diag_error、diag_default、diag_push、diag_pop (请参阅节 7.9.7)
- FORCEINLINE (请参阅节 7.9.8)
- FORCEINLINE_RECURSIVE (请参阅节 7.9.9)
- FUNC_ALWAYS_INLINE (请参阅节 7.9.10)
- FUNC_CANNOT_INLINE (请参阅节 7.9.11)
- FUNC_EXT_CALLED (请参阅节 7.9.12)
- FUNC_INTERRUPT_THRESHOLD (请参阅节 7.9.13)
- FUNC_IS_PURE (请参阅节 7.9.14)
- FUNC_IS_SYSTEM (请参阅节 7.9.15)
- FUNC_NEVER_RETURNS (请参阅节 7.9.16)
- FUNC_NO_GLOBAL_ASG (请参阅节 7.9.17)
- FUNC_NO_IND_ASG (请参阅节 7.9.18)
- FUNCTION_OPTIONS (请参阅节 7.9.19)
- INTERRUPT (请参阅节 7.9.20)
- LOCATION (请参阅节 7.9.21)
- MUST_ITERATE (请参阅节 7.9.22)
- NMI_INTERRUPT (请参阅节 7.9.23)
- NOINIT (请参阅节 7.9.24)
- NOINLINE (请参阅节 7.9.25)
- NO_HOOKS (请参阅节 7.9.26)
- once (请参阅节 7.9.27)
- pack (请参阅节 7.9.28)
- PERSISTENT (请参阅节 7.9.24)
- PROB_ITERATE (请参阅节 7.9.29)
- RETAIN (请参阅节 7.9.30)
- SET_CODE_SECTION (请参阅节 7.9.31)
- SET_DATA_SECTION (请参阅节 7.9.31)
- STRUCT_ALIGN (请参阅节 7.9.32)
- UNROLL (请参阅节 7.9.33)

提供了额外的 pragma 来支持 OpenMP API (请参阅节 8.10.1)。有关这些 pragma 的描述，请参阅 TI OpenMP 加速器模型在线文档。

大多数 pragma 适用于函数。除了 DATA_MEM_BANK pragma 外，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。

7.9.1 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《*TMS320C6000 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 7.14.2）具有与 CALLS pragma 相同的效果，语法如下：

```
__attribute__((calls("function_1","function_2",..., "function_n")))
```

7.9.2 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 7.14.2。

C 中 pragma 的语法为：

#pragma CODE_ALIGN (*func* **,** *constant* **)**

C++ 中 pragma 的语法为：

#pragma CODE_ALIGN (*constant* **)**

7.9.3 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 7.14.2。

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

```
#pragma CODE_SECTION(fn, "my_sect")
int fn(int x)
{
    return x;
}
```

以下示例 C 代码将生成下列汇编代码：

```
    .sect    "my_sect"
    .global  _fn
;******************************************************************************
;* FUNCTION NAME: _fn                                                         *
;*                                                                            *
;*   Regs Modified     : SP                                                   *
;*   Regs Used         : A4,B3,SP                                             *
;*   Local Frame Size  : 0 Args + 4 Auto + 0 Save = 4 byte                    *
;******************************************************************************
_fn:
;** --------------------------------------------------------------------------*
           RET     .S2     B3                ; |6|
           SUB     .D2     SP,8,SP           ; |4|
           STW     .D2T1   A4,*+SP(4)        ; |4|
           ADD     .S2     8,SP,SP           ; |6|
           NOP             2
           ; BRANCH OCCURS                   ; |6|
```

7.9.4 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 7.14.4。

C 中 pragma 的语法为：

```
#pragma DATA_ALIGN ( symbol , constant )
```

C++ 中 pragma 的语法为：

```
#pragma DATA_ALIGN ( constant )
```

7.9.5 DATA_MEM_BANK Pragma

DATA_MEM_BANK pragma 可将符号或变量与指定的内部数据存储器组边界对齐。*constant* 指定要启动变量的特定存储器组。（有关内存库的图形表示，请参阅节 5.5。）C6400+、C6740 和 C6600 器件包含 8 个存储器组。*constant* 可以是 0、2、4 或 6。

C 中 pragma 的语法为：

```
#pragma DATA_MEM_BANK ( symbol , constant )
```

C++ 中 pragma 的语法为：

```
#pragma DATA_MEM_BANK ( constant )
```

只有全局变量可以与 DATA_MEM_BANK pragma 对齐。

DATA_MEM_BANK pragma 使您能够对齐任何可以保存类型大小为 *symbol* 的数据的数据存储器组上的数据。如果您需要以特定方式对齐数据，以避免出现存储器组冲突（在手动编码和用零填充的汇编代码中，并且必须考虑代码中的填充），则此 pragma 会非常有用。

当使用填充将数据对齐到正确的存储器组时，此 pragma 会少量增加数据存储器中使用的空间量。

DATA_MEM_BANK pragma 的常量参数值为 0 会导致起始地址的最后五位为 0x00。对于值 2，起始地址的最后五位将为 0x08 (0b01000)。对于值 4，起始地址的最后五位将为 0x10 (0b10000)。对于值 6，起始地址的最后五位将为 0x18 (0b11000)。

示例 7-2 中的代码使用 DATA_MEM_BANK pragma 指定 x、y、z、w 和 zz 阵列的对齐方式。然后，该代码将值分配给所有的数组元素，并打印每个数组的起始地址。

示例 7-2. 使用 DATA_MEM_BANK Pragma

```
#include <stdio.h>
#pragma DATA_MEM_BANK (x, 2)
short x[100];
#pragma DATA_MEM_BANK (z, 0)
#pragma DATA_SECTION (z, ".z_sect")
short z[100];
#pragma DATA_MEM_BANK (w, 4)
#pragma DATA_SECTION (w, ".w_sect")
short w[100];
#pragma DATA_MEM_BANK (zz, 6)
#pragma DATA_SECTION (zz, ".zz_sect")
short zz[100];
static short my_count = 0;
void main()
{
   int i;
   #pragma DATA_MEM_BANK (y, 4)
   short y[100];
   for (i = 0; i < 100; i++)
   {
      w[i] = my_count++;
      x[i] = my_count++;
      y[i] = my_count++;
      z[i] = my_count++;
      zz[i] = my_count++;
   }
   printf("address of  w: 0x%08lx\n", (unsigned long)w);
   printf("address of  x: 0x%08lx\n", (unsigned long)x);
   printf("address of  y: 0x%08lx\n", (unsigned long)y);
   printf("address of  z: 0x%08lx\n", (unsigned long)z);
   printf("address of zz: 0x%08lx\n", (unsigned long)zz);
}
```

示例输出如下：

```
address of  w: 0x00006a70
address of  x: 0x80009468
address of  y: 0x80005f10
address of  z: 0x00006b60
address of zz: 0x00006978
```

7.9.6 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 .bss 段的区域，此 pragma 很有用。如果使用 DATA_SECTION pragma 来分配全局变量，并且希望在 C 代码中引用该变量，则必须将该变量声明为 extern far。

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 7.14.4。

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 7-3 到示例 7-5 演示了 DATA_SECTION pragma 的用法。

示例 7-3. 使用 DATA_SECTION Pragma C 源文件

```
#pragma DATA_SECTION(bufferB, "my_sect")
char bufferA[512];
char bufferB[512];
```

示例 7-4. 使用 DATA_SECTION Pragma C++ 源文件

```
char bufferA[512];
#pragma DATA_SECTION("my_sect")
char bufferB[512];
```

示例 7-5. 使用 DATA_SECTION Pragma 汇编源文件

```
            .global _bufferA
            .bss    _bufferA,512,4
            .global _bufferB
```

7.9.7 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。

7.9.8 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 3.11。

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。

7.9.9 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 3.11。

7.9.10 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 3.11。

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 7.14.2。

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

谨慎使用 FUNC_ALWAYS_INLINE Pragma

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。

7.9.11 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 3.11。

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 7.14.2。

C 中 pragma 的语法为：

```
#pragma FUNC_CANNOT_INLINE ( func )
```

C++ 中 pragma 的语法为：

```
#pragma FUNC_CANNOT_INLINE
```

7.9.12 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++ 函数可能通过手工编码的程序集而不是通过 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 4.4.2。

7.9.13 FUNC_INTERRUPT_THRESHOLD Pragma

编译器允许在函数内的阈值周期内，在软件流水线循环周围禁用中断。这将为单个函数实现 --interrupt_threshold 选项（请参阅节 3.12）。FUNC_INTERRUPT_THRESHOLD pragma 始终覆盖 --interrupt_threshold=*n* 命令行选项。小于 0 的阈值假定函数从未中断，这相当于无限大的中断阈值。

C 中 pragma 的语法为：

#pragma FUNC_INTERRUPT_THRESHOLD (*func* **,** *threshold* **)**

C++ 中 pragma 的语法为：

#pragma FUNC_INTERRUPT_THRESHOLD (*threshold* **)**

以下示例演示了不同阈值的用法：

- 函数 foo( ) 必须至少每 2000 个周期可中断一次：

```
#pragma FUNC_INTERRUPT_THRESHOLD (foo, 2000)
```

- 函数 foo( ) 必须始终是可中断的。

```
#pragma FUNC_INTERRUPT_THRESHOLD (foo, 1)
```

- 函数 foo( ) 从未中断。

```
#pragma FUNC_INTERRUPT_THRESHOLD (foo, -1)
```

7.9.14 FUNC_IS_PURE Pragma

FUNC_IS_PURE pragma 向编译器指定命名函数没有副作用。这允许编译器执行以下操作：

- 如果不需要函数的值，则删除对该函数的调用
- 删除重复的函数

此 pragma 必须出现在针对函数进行的任何声明或引用之前。在 C 中，参数 *func* 是函数的名称。在 C++ 中，pragma 应用于下一个声明的函数。

C 中 pragma 的语法为：

#pragma FUNC_IS_PURE (*func* **)**

C++ 中 pragma 的语法为：

#pragma FUNC_IS_PURE

7.9.15 FUNC_IS_SYSTEM Pragma

FUNC_IS_SYSTEM pragma 向编译器指定命名函数具有 ANSI/ISO 标准为具有该名称的函数定义的行为。

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* 是要作为 ANSI/ISO 标准函数处理的函数的名称。在 C++ 中，pragma 应用于所声明的下一个函数。

C 中 pragma 的语法为：

```
#pragma FUNC_IS_SYSTEM ( func )
```

C++ 中 pragma 的语法为：

```
#pragma FUNC_IS_SYSTEM
```

7.9.16 FUNC_NEVER_RETURNS Pragma

FUNC_NEVER_RETURNS pragma 向编译器指定函数不会返回其调用方。

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* 是不返回的函数的名称。在 C++ 中，pragma 应用于所声明的下一个函数。

C 中 pragma 的语法为：

```
#pragma FUNC_NEVER_RETURNS ( func )
```

C++ 中 pragma 的语法为：

```
#pragma FUNC_NEVER_RETURNS
```

7.9.17 FUNC_NO_GLOBAL_ASG Pragma

FUNC_NO_GLOBAL_ASG pragma 向编译器指定，该函数不对所命名的全局变量赋值，也不包含 asm 语句。

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* 是不进行赋值的函数的名称。在 C++ 中，pragma 适用于下一个声明的函数。

C 中 pragma 的语法为：

```
#pragma FUNC_NO_GLOBAL_ASG ( func )
```

C++ 中 pragma 的语法为：

```
#pragma FUNC_NO_GLOBAL_ASG
```

7.9.18 FUNC_NO_IND_ASG Pragma

FUNC_NO_IND_ASG pragma 向编译器指定该函数不通过指针进行赋值，也不包含 asm 语句。

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* 是不进行赋值的函数的名称。在 C++ 中，pragma 应用于所声明的下一个函数。

C 中 pragma 的语法为：

```
#pragma FUNC_NO_IND_ASG ( func )
```

C++ 中 pragma 的语法为：

```
#pragma FUNC_NO_IND_ASG
```

7.9.19 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。

7.9.20 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。在 C++ 中，pragma 应用于下一个声明的函数。

C 中 pragma 的语法为：

```
#pragma INTERRUPT ( func )
```

C++ 中 pragma 的语法为：

```
#pragma INTERRUPT
void func ( void )
```

```
__attribute__((interrupt )) void func ( void )
```

该函数的代码将通过 IRP（中断返回指针）返回。

```
#pragma INTERRUPT ( func , {HPI|LPI} )
```

```
#pragma INTERRUPT ( {HPI|LPI} )
```

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。

7.9.21 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 7.14.2。

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 7.14.4）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 7.9.24。

7.9.22 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（即软件流水线和嵌套循环变换）。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 和 PROB_ITERATE 等其他 pragma。

7.9.22.1 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];
    }
    ...
}
```

7.9.22.2 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，编译器也尝试生成软件流水循环。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，编译器将尝试生成软件流水循环：

```
#pragma MUST_ITERATE(8, 8)
for(i2 = ipos[2]; i2 < 40; i2 += 5)  { ...
```

通常，如果使用 MUST_ITERATE pragma 优化循环执行，则会在优化代码前附加 DINT 指令，执行循环代码后，循环终止时会执行 RINT 指令。

7.9.23 NMI_INTERRUPT Pragma

NMI_INTERRUPT pragma 让您可以直接使用 C 代码来处理不可屏蔽的中断。在 C 语言中，参数 *func* 是函数的名称。在 C++ 中，pragma 应用于所声明的下一个函数。

C 中 pragma 的语法为：

```
#pragma NMI_INTERRUPT( func )
```

C++ 中 pragma 的语法为：

```
#pragma NMI_INTERRUPT
```

为该函数生成的代码将通过 NRP 返回，而对于通过中断关键字或 INTERRUPT pragma 声明的函数，则是通过 IRP 返回。

除了为 C 程序系统复位中断预留的名称 _c_int00，中断的名称（函数）无需遵循命名规则。

7.9.24 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。

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 7.14.4。

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;
```

7.9.25 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 3.11。

7.9.26 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 3.16。

7.9.27 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。

7.9.28 pack Pragma

pack pragma 可用于控制类、结构或联合类型中的字段对齐。该 pragma 在 C/C++ 语言中的语法可以是以下任何一种：

```
#pragma pack ( n )
```

上述形式的 pack pragma 影响文件中此 pragma 后面的所有类、结构或联合类型声明。它会强制将每个字段的最大对齐设置为 *n* 指定的值。*n* 的有效值为 1、2、4、8 和 16 字节。

```
#pragma pack ( push, n )
#pragma pack ( pop )
```

上述形式的 pack pragma 仅影响 push 和 pop 指令之间的类、结构和联合类型声明。（如果 pop 指令前面没有 push 指令，则会导致编译器发出警告诊断消息。）所有已声明字段的最大对齐为 *n*。*n* 的有效值为 1、2、4、8 和 16 字节。

```
#pragma pack ( show )
```

上述形式的 pack pragma 会向 stderr 发送警告诊断消息来记录 pack pragma 堆栈的当前状态。您可以在调试时使用这种形式。

有关各个打包字段的更多信息，请参阅节 7.14.5。

7.9.29 PROB_ITERATE Pragma

PROB_ITERATE pragma 为编译器指定循环的某些属性。您可以断言这些属性在常见情况下为 true。PROB_ITERATE pragma 能够帮助编译器选择最优循环和循环变换（也即软件流水线和嵌套循环变换）。仅当未使用 MUST_ITERATE pragma 时或 PROB_ITERATE 参数比 MUST_ITERATE 参数具有更多限制时，PROB_ITERATE 才有用。

PROB_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 和 PROB_ITERATE 和 PROB_ITERATE 等其他 pragma。该 pragma 的 C 和 C++ 语法为：

#pragma PROB_ITERATE(*min* **,** *max* **)**

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 7.9.22.1。

[[TI::prob_iterate(*min*, *max* **)]]**

其中，min 和 max 是循环在常见情况下的最小和最大行程计数。行程计数是指循环的迭代次数。这两个参数都是可选的。

例如，PROB_ITERATE 可应用于大多数情况下执行八次迭代（但有时可能执行超过或少于八次迭代）的循环：

```
#pragma PROB_ITERATE(8, 8)
```

如果仅知道预期的最小行程计数（例如 5 次），则该 pragma 与以下所示类似：

```
#pragma PROB_ITERATE(5)
```

如果仅知道预期的最大行程计数（例如 10 次），则该 pragma 与以下所示类似：

```
#pragma PROB_ITERATE(, 10)  /* Note the blank field for min */
```

7.9.30 RETAIN Pragma

RETAIN pragma 可以应用于代码或数据符号。

它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。

RETAIN pragma 与使用 `retain` 函数或变量属性的效果相同。请分别参阅节 7.14.2 和节 7.14.4。

C 中 pragma 的语法为：

#pragma RETAIN (*symbol* **)**

C++ 中 pragma 的语法为：

#pragma RETAIN

7.9.31 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。

7.9.32 STRUCT_ALIGN Pragma

STRUCT_ALIGN pragma 类似于 DATA_ALIGN，但可应用于结构体或联合体类型或 typedef，由通过该类型创建的任何符号来继承。STRUCT_ALIGN pragma 仅在 C 语言中受支持。

该 pragma 的语法为：

#pragma STRUCT_ALIGN(*type* **,** *constant expression* **)**

此 pragma 保证指定类型或指定 typedef 基本类型的对齐至少等于该表达式的对齐。（该对齐可能大于编译器所需的字节数。）该对齐必须为 2 的幂。*type* 必须为类型或 typedef 名称。如果是类型，则它必须为结构体标签或联合体标签。如果是 typedef，则其基本类型必须为结构体标签或联合体标签。

请注意，虽然某个类型（或该类型的 typedef）的顶级对象会按要求对齐，但是该类型不会通过填充达到对齐（这在结构体中很常见），并且对齐也不会应用到数组和父级结构体等派生类型。如果您要填充结构体或联合体，以便也将个别元素对齐和/或使对齐应用到派生类型，请按照节 7.14.5 中所述使用“aligned”类型属性。

ANSI/ISO C 会声明 typedef 只是类型（例如结构体）的别名，因此该 pragma 可应用到结构体、结构体的 typedef 或从它们派生的任何 typedef，并会影响对应基本类型的所有别名。

该示例会对齐页面边界上的任何 st_tag 结构体变量：

```
typedef struct st_tag
{
   int   a;
   short b;
} st_typedef;
#pragma STRUCT_ALIGN (st_tag, 128)
#pragma STRUCT_ALIGN (st_typedef, 128)
```

任何将 STRUCT_ALIGN 与基本类型（int、short、float）或变量结合使用的情况都会导致错误。

7.9.33 UNROLL Pragma

UNROLL pragma 向编译器指定了循环应该展开的次数。UNROLL pragma 对帮助编译器利用 SIMD 指令很有用。另外，在需要通过非展开循环更好地利用软件流水线资源的情况下，该 pragma 也很有用。

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 和 PROB_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 7.9.22.1。

[[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。

7.10 _Pragma 运算符

C6000 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。

7.11 应用程序二进制接口

应用程序二进制接口 (ABI) 定义单独编写、单独编译或汇编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 允许将符合 ABI 的目标文件链接在一起，而不管其来源如何，并使生成的可执行文件能够在支持该 ABI 的任何系统上运行。它定义了从 C 符号名称生成的链接名称。它还定义了目标文件格式和调试格式，并记录系统的初始化方式。如果是 C++，它则定义了对 C++ 名称的处理和异常处理支持。

C6000 编译器和链接器现在仅支持嵌入式应用程序二进制接口 (EABI) ABI，该接口仅适用于使用 ELF 目标文件格式和 DWARF 调试格式的目标文件。如果希望支持传统 COFF ABI，请使用 C6000 v7.4.x 代码生成工具，并参阅 SPRU187 和 SPRU186 文档。

EABI 使用 ELF 目标文件格式，支持早期模板实例化和导出内联函数等现代语言功能。节 8.9.2 中描述了有关 EABI 模式的 TI 特定信息。

有关 C6000 EABI 的低层面细节，请参阅《*C6000 嵌入式应用程序二进制接口*》(SPRAB89)。

7.12 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。

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。

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。

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 10，了解详情。

改编算法遵循 Itanium C++ ABI (http://www.codesourcery.com/cxx-abi/abi.html) 中的描述。

`int foo(int i) { }` 将改编为 _Z3fooi

7.13 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 7.13.1。）

- ANSI/ISO C11（--c11，请参阅节 7.13.2）
- ISO C++14（--c++14 ，用于所有 C++ 文件 ，请参阅节 7.2。）

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 7.13.3。）

7.13.1 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
  - FP_ILOGB0 宏命令/FP_ILOGBNAN 宏命令
  - MATH_ERRNO 宏命令
  - copysign()
  - float_t 类型/double_t 类型
  - math_errhandling()
  - signbit()
  - 以下 C99 函数集不支持“long double”类型。支持使用浮点和 long double 类型的 C89 数学函数。（段号来自 C99 标准。）
    - 7.12.4: 三角函数
    - 7.12.5: 双曲函数
    - 7.12.6: 指数和对数函数
    - 7.12.7: 幂函数和绝对值函数
    - 7.12.9: 最近整数函数
    - 7.12.10: 余数函数
  - expm1()
  - ilogb()/log1p()/logb()
  - scalbn()/scalbln()
  - cbrt()
  - hypot()
  - erf()/erfc()
  - lgamma()/tgamma()
  - nearbyint()
  - rint()/lrint()/llrint()
  - lround()/llround()
  - remainder()/remquo()
  - nan()
  - nextafter()/nexttoward()
  - fdim()/fmax()/fmin()/fma()
  - isgreater()/isgreaterequal()/isless()/islessequal()/islessgreater()/isunordered()
- 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

7.13.2 C11 支持 (--c11)

编译器支持 ISO 标准化的 2011 年标准 C。但是，除了节 7.13.1 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

7.13.3 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 7.14 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。

7.14 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 7.14.6)
- __has_attribute

7.14.1 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 7-6 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 7-6. 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

表 7-6. GCC 语言扩展 (continued)

扩展	说明
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
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 7.14.6）
矢量扩展	使用矢量运算（OpenCL 语法，请参见节 7.15）
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

7.14.2 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
- visibility
- warn_unused_result

支持以下其他 TI 特定函数：

- retain

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 7.9.2。

always_inline 函数属性与 FUNC_ALWAYS_INLINE pragma 具有相同的效果。请参阅 节 7.9.10

calls 属性与 CALLS pragma 具有相同的效果，相关描述请参阅 节 7.9.1。

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 7.9.20。

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 7.8。

noinline 函数属性与 FUNC_CANNOT_INLINE pragma 具有相同的效果。请参阅 节 7.9.11

retain 属性与 RETAIN pragma (节 7.9.30) 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 7.9.3

7.14.3 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::prob_iterate
- TI::unroll

有关使用 for 循环属性的示例，请参阅节 7.9.22.1。

7.14.4 变量属性

支持下述变量属性：

- aligned

- deprecated
- location
- mode
- noinit
- packed
- persistent
- retain
- section
- transparent_union
- unused
- used

在变量上使用的 **aligned** 属性与 DATA_ALIGN pragma 的效果相同。请参阅 节 7.9.4

location 属性与 LOCATION pragma 的效果相同。请参阅节 7.9.21。例如：

```
__attribute__((location(0x100))) extern struct PERIPH peripheral;
```

noinit 和 **persistent** 属性适用于 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《*TMS320C6000 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 7.9.24。

packed 属性可应用于结构体或联合体中的单个字段。只有当硬件支持非对齐访问时，用于结构体或联合体字段的 packed 属性才可适用。

retain 属性与 RETAIN pragma (节 7.9.30) 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 7.9.6

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 ）。

7.14.5 类型属性

编译器支持以下类型属性：

- aligned
- deprecated
- packed
- transparent_union
- unused
- visibility

使用 **aligned** 类型属性时，编译器会根据需要对结构体、联合体或其他类型的单个元素进行填充（这在结构体中很常见），从而实现所有元素按指定方式对齐。此外，任何派生类型都具有相同的对齐。例如：

```
    struct __attribute__((aligned(32))) myStruct { char c1; int i; char c2; };
```

结构体和联合体类型都支持 **packed** 属性。如果使用了 --relaxed_ansi 选项，则它仅适用于对未对齐访问提供硬件支持的目标架构。

压缩结构的成员在存储时会尽可能靠近彼此，并会忽略通常为保持字对齐而添加的额外填充字节。例如，假定一个 4 个字节的字大小通常在成员 c1 和 i 之间具有 3 个填充字节，在成员 c2 后具有另外 3 个填充字节，因此总大小为 12 个字节：

```
struct unpacked_struct { char c1; int i; char c2;};
```

不过，压缩结构的成员是字节对齐的。因此，以下示例中成员之间或之后没有任何填充字节，总共为 6 个字节：

```
struct __attribute__((__packed__)) packed_struct { char c1; int i; char c2; };
```

因此，数组中的压缩结构会压缩在一起，数组元素之间没有填充字节。

在位字段上使用“packed”会覆盖位字段的 EABI 要求。对于*非压缩位字段*，位字段的声明类型会用于容器类型。对于*打包位字段*，不管声明类型如何，都会使用最小整型（bool 除外）。对于*非压缩易失性位字段*，位字段必须使用大小与声明类型相同的访问来进行访问。对于*压缩易失性位字段*，访问必须与实际容器类型具有相同的大小，并可能与声明类型的大小不同；另外，实际容器可能不对齐，并且可能跨多个对齐的容器边界，因此访问压缩易失性位字段时可能需要多次进行存储器存取。这还可能会影响结构体的总体大小；例如，如果该结构体仅包含位字段，它可能与位字段的声明类型不一样大。对于*压缩和非压缩位字段*，位字段都是位对齐，并与以下相邻位字段压缩在一起：没有填充字节，完全包含在至少是字节对齐的整数容器中；并且不会改变相邻非位字段结构成员的对齐方式。 有关位字段布局的说明，请参阅节 8.2.2。

“packed”属性只能应用于结构体或联合体类型的原始定义。它不能通过 typedef 用于已定义的非压缩结构，也不能用于结构体或联合体对象的声明。因此，任意给定结构体或联合体类型都只能是压缩或非压缩，并且该类型的所有对象都会继承其 packed 或 non-packed 属性。

“packed”属性不能递归应用到压缩结构体中包含的结构体类型。因此，在以下示例中，成员会保留与上方第一个示例相同的内部布局。c 和 s 之间没有填充字节，因此 s 在未对齐的边界上：

```
struct __attribute__((__packed__)) outer_packed_struct { char c; struct unpacked_struct s; };
```

以隐式或显式方式将压缩结构体成员的地址作为指针投射到除无符号字符以外的任意非紧凑类型都是非法的。在以下示例中，p1、p2 和对 foo 的调用都是非法的。

```
void foo(int *param);
struct packed_struct ps;
int *p1 = &ps.i;
int *p2 = (int *)&ps.i;
foo(&ps.i);
```

不过，以显式方式将压缩结构体成员的地址作为指针投射到无符号字符则是合法的。

```
unsigned char *pc = (unsigned char *)&ps.i;
```

TI 编译器还支持枚举类型的 **unpacked** 属性，让您可以指示表现形式为不小于 int 的整型；也就是说，它不是 *packed*。

7.14.6 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_memcpy()
- __builtin_return_address()

__builtin_frame_address() 函数始终返回 0，除非参数是常数零。

__builtin_return_address() 函数始终返回零。

7.15 向量数据类型的运算和函数

C/C++ 编译器支持在 C/C++ 源文件中使用 TI 向量数据类型。节 7.3.2 中对向量数据类型进行了介绍。这些向量数据类型对于并行编程应用很有用。您可以使用 --vectypes 编译器选项来启用对向量数据类型的支持。

向量数据类型和运算的实现严格遵循 OpenCL C 语言规范。有关 OpenCL 向量数据类型和运算的详细说明，请参阅 OpenCL 规范 1.2 版，该规范可从 Khronos OpenCL 工作组获取。

可以对向量执行各种类型的运算。包括向量字面量和串联运算符 (节 7.15.1)、一元和二元运算符 (节 7.15.2)、用于组件访问的混合运算符 (节 7.15.3) 以及转换运算符 (节 7.15.4)。此外，还提供了几个内置函数 (节 7.15.7) 来处理向量类型。

除了向量运算之外，还提供 printf() 支持以输出向量数据。请参阅 OpenCL 规范 1.2 版的第 6.12.13 节，了解使用 printf() 设置向量数据类型格式的详细信息。有关向量数据类型的 OpenCL 规范的例外情况，请参阅节 7.15.6。

7.15.1 向量字面量和串联

您可以使用字面量或标量变量，指定用于向量初始化或分配的值。如果分配给向量的所有值均是常数，得到的结果就是向量字面量。否则向量的值在运行时确定。

例如，在以下声明中分配给 vec_a 和 vec_b 的值是向量字面量，并且在编译时已知：

```
short4 vec_a = (short4)(1, 2, 3, 4);
float2 vec_b = (float2)(3.2, -2.3);
```

以下语句会将向量的所有元素初始化为相同的值，在本例中该值为 1。

```
ushort4 myushort4 = (ushort4)(1);
```

较短的向量可以串联到一起，组成较长的向量。在以下示例中，两个 int 变量串联为一个 int2 变量。在以下函数中 myvec 的值直到运行时才进行解析：

```
void foo(int a, int b)
{
    int2 myvec = (int2)(a, b);
    ...
}
```

在以下示例中，两个 int2 变量串联为一个 int4 变量，并传递到外部函数：

```
extern void bar(int4 v4);
void foo(int a, int b)
{
    int2 myv2_a = (int2)(a, 1);
    int2 myv2_b = (int2)(b, 2);
    int4 myv4 = (int4)(myv2_a, myv2_b);
    bar(myv4);
}
```

7.15.2 向量的一元和二进制运算符

当为向量应用一元运算符（例如负号：-）和二进制运算符（例如 +）时，运算符会应用到向量中的每个元素。也就是说，生成的向量中的每个元素都是将运算符应用于源向量中对应元素的结果。

表 7-7. 各个向量类型支持的一元运算符

运算符	说明
-	取反
~	按位补码
!	逻辑非（仅限整型向量）

下面的示例中声明了一个被称为 pos_i4 的 int4 向量，并将它初始化为值 1、2、3 和 4。然后，它使用取反运算符来将另一个 int4 向量 neg_i4 的值初始化为 -1、-2、-3 和 -4。

```
int4 pos_i4 = (int4)(1, 2, 3, 4);
int4 neg_i4 = -pos_i4;
```

表 7-8. 各个向量类型支持的二进制运算符

运算符	说明
+、-、*、/	算术运算符（复数向量中也受支持）
=、+=、-=、*=、/=、	赋值运算符
%	取模运算符（仅限整型向量）
&、|、^、<<、>>	按位运算符
>、>=、==、!=、<=、<	关系运算符
++、--	增量/减量运算符（前缀和后缀；仅限整型向量；复数向量的实部中也受支持）
&&、||	逻辑运算符（仅限整型向量）

当二进制运算符与 TI 向量类型一起使用时，每个操作数向量类型中的元素类型和元素数量必须相同。对于算术二进制运算符（例如 +、-），结果类型与操作数类型相当。

向量二进制逻辑运算符产生的向量类型与带符号整数元素的向量操作数具有相同的元素数量。例如，如果 == 运算符比较两种 float4 类型，则生成的类型将是 int4。比较两个 double8 类型会得到一个 long8 类型。向量二进制逻辑运算符会在每个结果向量通道中产生 -1（表示 true）或 0（表示 false）。

下面的示例在类型为 int4 的向量上使用 =、++ 和 + 运算符。假定 iv4 参数最初包含 (1, 2, 3, 4)。在退出 foo() 时，iv4 将包含 (3, 4, 5, 6)。

```
void foo(int4 iv4)
{
    int4 local_iva = iv4++;                 /* local_iva = (1, 2, 3, 4) */
    int4 local_ivb = iv4++;                 /* local_ivb = (2, 3, 4, 5) */

    int4 local_ivc = local_iva + local_ivb; /* local_ivc = (3, 5, 7, 9) */
}
```

算术运算符和增量/减量运算符可以与复数向量类型搭配使用。增量/减量运算符会加上或减去 1+0i。

下面的示例中将类型为 cfloat2 的复数向量相乘和相除。如需详细了解复数乘法和除法的规则，请参阅 C99 C 语言规范的“附件 G”。

```
void foo()
{
    cfloat2 va = (cfloat2) (1.0, -2.0, 3.0, -4.0);
    cfloat2 vb = (cfloat2) (4.0, -2.0, -4.0, 2.0);
    /* vc = (0.0, -10.0), (-4.0, 22.0) */
    cfloat2 vc = va * vb;
    /* vd = (0.4, -0.3), (-1.0, 0.5) */
    cfloat2 vd = va / vb;
    ...
}
```

在 C64+ 和 C6740 上，前面示例中的 * 和 / 运算符会调用一个内置函数来执行复数乘法和除法运算。在 C6600 上，编译器会生成一个 CMPYSP 指令来执行复数乘法或除法运算。

7.15.3 矢量的混合运算符

编程模型实现方案支持以下“混合”运算符。这些运算符用作变量名的后缀。这些运算符可用于赋值运算符的任意一侧（左侧或右侧）。在赋值的左侧使用时，每个分量必须是唯一可识别的。

.x、.y、.z 或 .w 访问长度小于等于 4 的矢量元素。

```
char4 my_c4 = (char4)(1, 2, 3, 4);
char tmp = my_c4.y * my_c4.w;
/* ".y" accesses 2nd element ".w" accesses 4th element
 * tmp = 2 * 4 = 8; */
```

.s0、.s1、...、.s9、.sa、...、.sf 访问向量中多达 16 个元素中的一个。

```
uchar16 ucvec16 = (uchar16)(1, 2, 3, 4, 5, 6, 7, 8,
                            9, 10, 11, 12, 13, 14, 15, 16 );
uchar8 ucvec8 = (uchar8)(2, 4, 6, 8, 10, 12, 14, 16);
int tmp = ucvec16.sa * ucvec8.s7;
/* ".sa" is 11th element of ucvec16
 * ".s7" is 8th element of ucvec8
 * tmp = 11 * 16 = 176;     */
```

.even、.odd 访问矢量的偶数或奇数元素，其中第零个元素为偶数。

```
ushort4 usvec4 = (ushort4)(1, 2, 3, 4);
ushort2 usvecodd = usvec4.odd;   /* usvecodd = (ushort2)(2, 4); */
ushort2 usveceven = usvec4.even; /* usveceven = (ushort2)(1, 3); */
```

.hi、.lo 使用 .hi 访问矢量上半部分的元素，或使用 .lo 访问矢量下半部分的元素。

```
ushort8 usvec8 = (ushort8)(1, 2, 3, 4, 5, 6, 7, 8);
ushort4 usvechi = usvec8.hi; /* usvechi = (ushort4)(5, 6, 7, 8);*/
ushort4 usveclo = usvec8.lo; /* usveclo = (ushort4)(1, 2, 3, 4); */
```

.r 访问复杂类型矢量中每个元素的实部。

```
cfloat2 cfa = (cfloat2)(1.0, -2.0, 3.0, -4.0);
float2 rfa = cfa.r;      /* rfa = (float2)(1.0, 3.0); */
```

.i 访问复杂类型矢量中每个元素的虚部。

```
cfloat2 cfa = (cfloat2)(1.0, -2.0, 3.0, -4.0);
float2 ifa = cfa.i;      /* ifa = (float2)(-2.0, -4.0); */
```

可以组合混合运算符来访问元素子集的子集。组合的结果必须明确定义。例如，在以下代码运行后，usvec4 包含（1、2、5、4）。

```
ushort4 usvec4 = (ushort4)(1, 2, 3, 4);

usvec4.hi.even = 5;
```

7.15.4 向量的转换函数

不能对向量数据类型使用标准类型转换。但是，可通过 convert_<*destination type*>(<*source type*>) 函数将一个向量类型对象的元素转换为另一个向量类型对象。每个元素都进行转换，而且源向量类型和目标向量类型必须具有相同的长度。也就是说，4 元素向量只能转换为其他类型的 4 元素向量。

以下示例使用两个串联的 INT 来初始化 short2 向量，以形成 int2 向量：

```
void foo(int a, int b)
{
    short2 svec2 = convert_short2((int2)(a, b));
    ...
}
```

如果向量元素存储的数据超出了可以存储在目标类型中的值的范围，默认情况下，该值将被截断。但是，如果将 _sat 修饰符（表示“饱和”）添加到函数名，则目标类型范围之外的值将设置为目标类型的最大值（或范围之外负值的最小值）。将整数向量转换为浮点向量时，_sat 修饰符无法使用。在下面的示例中，myint4 元素中存储的大于 32,767 的任何值在 myshort4 的相应元素中都设置为 32,767。

```
int4 myint4;
short4 myshort4 = convert_short4_sat( myint4 );
```

同样，在浮点和整数向量之间转换时，可以向函数名添加以下任一修饰符，以指定浮点值的四舍五入规则：

- **_rte** — 舍入到最近的偶数整数
- **_rtz** — 舍入到零（默认将浮点型转换为整数）
- **_rtp** — 舍入到正无穷大
- **_rtn** — 舍入到负无穷大

从整型转换为浮点型时，不需要四舍五入。从较大的双精度浮点值转换为较小的单精度浮点值则需要四舍五入。双精度到单精度转换的默认舍入方法是 _rte。

下面的示例将存储在 float4 中的数据转换为 int4。它使用 _rtp 修饰符，因此值向上舍入到正无穷大：

```
float4 myfloat4;
int4 myint4 = convert_int4_rtp( myfloat4 );
```

_sat 修饰符可以与舍入修饰符组合使用。以下示例将浮点值舍入为偶数，并将大于最大可能短整型值的值设置为最大值：

```
float4 myfloat4;
int4 myshort4 = convert_short4_sat_rte( myfloat4 );
```

如果启用了向量数据类型，还可以对标量（非向量）类型（如 short 和 int）使用 convert_<*type*>() 函数。结果与转换源类型的类型相同。不允许在标量类型和向量类型之间进行转换，因为源类型和目标类型必须包含相同数量的元素。

convert_<*destination type*>() 函数不可用于复数向量类型。

7.15.5 矢量的重新解释函数

as_<*destination type*>(<*source type object*>) 函数用于将对象的原始类型重新解释为另一种矢量类型。源类型和目标类型的位数必须相同。如果大小不同，则返回错误。

虽然算术转换由上一段中介绍的转换函数执行，但重新解释函数不执行算术转换。例如，假设浮点值 1.0 被重新解释为整数值。浮点值 1.0 以十六进制表示为 0x3f800000，由此得到的整数值为 1,065,353,216。

以下示例将 longlong 类型（64 位）的非矢量变量重新解释为 float2 矢量（2 个元素，每个元素 32 位）。mylonglong 的最低有效 32 位放在 fltvec2.s0 中，而 mylonglong 的最高有效 32 位放在 fltvec2.s1 中。不执行算术转换。

```
extern longlong mylonglong;
float2 fltvec2 = as_float2(mylonglong);
```

如果源类型和目标类型的大小不同，则会发生错误。

如果启用了矢量数据类型，还可以对标量（非矢量）类型使用 as_<*type*>() 函数。类型必须具有相同的位数。以下示例将浮点值重新解释为整数值。浮点值 1.0 以十六进制表示为 0x3f800000，由此得到的整数值为 1,065,353,216。

```
float myfloat = 1.0f;
myint = as_int(myfloat);
```

as_<*destination type*>() 函数不可用于复数矢量类型。

7.15.6 使用 printf() 设置矢量

除了矢量运算之外，还提供 printf() 支持以输出矢量数据。请参阅 1.2 版《OpenCL 规范》中的第 6.12.13 节，了解使用 printf() 设置矢量数据类型格式的详细信息。请注意，德州仪器 (TI) C6000 的实现方式与 OpenCL 规范在以下方面具有差别：

- **返回值。**OpenCL 规范指出 printf() 返回 0 或 -1。为了与 C99 规范就 scalar printf() 保持一致，vector printf() 的返回值是打印的字符数。
- **64 位整数的长度修饰符。**但 C6000 矢量类型不包含 long*n* 或 ulong*n* 矢量类型，因为 int 和 long 都是 32 位的。因此使用 %l (ell) 长度修饰符将造成不一致。请为 64 位 long long 和 unsigned long long 矢量类型使用 %ll (ell ell) 长度修饰符。

  即 `ll` 长度修饰符指定，以下 d、i、o、u、x 或 X 转换指定符适用于 longlong*n* 或 ulonglong*n* 参数。如 OpenCL 规范中所述，将 `l` 长度修饰符用于 64 位 double*n* 参数。

以下示例声明、初始化并打印四个 32 位浮点值的矢量，四个 8 位 unsigned char 值的矢量，以及两个 64 位 long long 值的矢量。

```
float4 f4 = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
uchar4 uc = (uchar4)(0xFA, 0xFB, 0xFC, 0xFD);
longlong2 bigNums = (longlong2)(600000000000, -600000000000);

printf("f4 = %2.2v4hlf\n", f);
printf("uc = %#v4hhx\n", uc);
printf("bigNums = %+v2lld\n", bigNums);
```

此示例打印包含浮点值的矢量，使用 `%2.2v4hlf` 格式字符串，输出至少 2 位宽度的值，精度为 2 (`2.2`)，矢量长度为 4 (`v4`)，具有 float*n* 长度修饰符 (`hl`)，使用 float 类型指定符 (`f`)。

此示例打印包含 uchar 值的矢量，使用 `%#v4hhx` 格式字符串，输出 0x 前缀 (#)，后跟长度为 4 的矢量 (v4)，具有 char*n* 或 uchar*n* 长度修饰符 (hh)，使用小写十六进制标记 (x)。

此示例打印包含 long long 值的矢量，使用 `%+v2lld` 格式字符串，输出具有 + 或 - 前缀的值 (+)，后跟长度为 2 的矢量 (v2)，具有 longlong*n* 或 ulonglong*n* 长度修饰符 (ll)，使用十进制标记 (d)。

```
/* Output */
f4 = 1.00,2.00,3.00,4.00
uc = 0xfa,0xfb,0xfc,0xfd
bigNums = +600000000000,-600000000000
```

7.15.7 内置矢量函数

表 7-9 列出了能够接受矢量参数并返回矢量结果的内置函数。除非另有规定，否则该函数应用于矢量中的每个元素。也就是说，结果矢量中的每个元素都是将函数应用于源矢量中相应元素的结果。除非另有规定，否则所有参数的矢量类型和返回的矢量必须相同。

表 7-9. 接受矢量参数的内置函数

函数	描述	支持的矢量类型
__hadd(*x*, *y*)	使用 (*x* + *y*) >> 1 返回平均值。中间和不会模溢出。*x*、*y* 的矢量类型以及返回的矢量必须相同。	short to short16 uchar to uchar16
__rhadd(*x*, *y*)	使用 (*x* + *y* + 1) >> 1 进行四舍五入返回平均值。中间和不会模溢出。	short to short16 uchar to uchar16
__max(*x*, *y*)	返回所返回矢量的每个元素中 *x* 和 *y* 的较大值。	short to short16 uchar to uchar16
__min(*x*, *y*)	返回所返回矢量的每个元素中 *x* 和 *y* 的较小值。	short to short16 uchar to uchar16
__add_sat(*x*, *y*)	返回 *x* + *y* 并使结果饱和。也就是说，如果存在溢出，则返回类型范围内的最大值。	short to short16 ushort to ushort16 uchar to uchar16 int to int16
__sub_sat(*x*, *y*)	返回 *x* - *y* 并使结果饱和。也就是说，如果存在溢出，则返回该类型在范围内的最大值。	short to short16 int to int16
__abs_diff(*x*, *y*)	使用 | *x* - *y* | 返回绝对差值，无溢出。	uchar to uchar16
__abs(*x*)	使用 | *x* | 返回绝对值	short to short16
__popcount(*x*)	返回 *x* 中的非零位的数量。	uchar to uchar16
__mpy_ext(*x*, *y*)	扩展精度乘法。	short to short16 ushort to ushort16 char to char16 uchar to uchar16
__mpy_fx_ext(*x*, *y*)	定点乘法。	short to short16
__conj_cmpy(*x*, *y*)	共轭复数乘法。	cfloat to cfloat8
__cmpy_ext(*x*, *y*)	扩展精度复数乘法。	cshort to cshort8
__cmpyr_fx(*x*, *y*)	涉及四舍五入的定点复数乘法。	cshort to cshort8
__cmpy_fx(*x*, *y*)	定点复数乘法。	cshort to cshort8 cint to cint8
__conj_cmpy_fx(*x*, *y*)	定点共轭复数乘法。（仅 C6600）	cint to cint8
__crot90(*x*)	旋转 90 度。（仅 C6600）	cshort to cshort8
__crot270(*x*)	旋转 270 度。（仅 C6600）	cshort to cshort8
__dot_ext(*x*, *y*)	使用 *x* • *y* 的扩展精度点积。	short2 因子给出 int 结果 char4 因子给出 int 结果 uchar4 因子给出 uint 结果 short4 因子给出 int 结果（仅 C6600） short4 和 ushort4 因子给出 int 结果（仅 C6600）

表 7-9. 接受矢量参数的内置函数 (continued)

函数	描述	支持的矢量类型
__dot_extll(*x*, *y*)	使用 *x* • *y* 的扩展精度点积。	short2 因子给出 longlong 结果 short4 因子给出 longlong 结果（仅 C6600） short4 和 ushort4 因子给出 longlong 结果（仅 C6600）
__dot_fx(*x*, *y*)	使用 *x* • *y* 的定点点积。	short2 • ushort2 = int
__gmpy(*x*, *y*)	伽罗瓦域乘法。	uchar4
__ddot_ext(*x*, *y*)	扩展精度双向点积。	short2 • char4 = int4 short8 • short8 = int2（仅 C6600） short8 • ushort8 = int2（仅 C6600）
__mpy_fx(*x*, *y*)	定点乘法。	short2 * int = int2 int4 * int4 = int4（仅 C6600）
__apply_sign(*x*, *y*)	使用 x 的符号位来确定是将 y 中的四个 16 位值乘以 1 还是 -1。产生四个有符号 16 位结果。此函数是 _dapys2() 内在函数的别名。（仅 C6600）	short4
__cmpy_conj_ext(*x*, *y*)	扩展精度复数乘法共轭。（仅 C6600）	cshort2 * cshort2 = cint2
__cmpy_conj_fx(*x*, *y*)	定点复数乘法共轭。（仅 C6600）	cshort2 * cshort2 = cshort2
__cmatmpy_ext(*x*, *y*)	扩展精度复数矩阵乘法。（仅 C6600）	cshort2 * cshort4 = cint2
__conj_cmatmpy_ext(*x*, *y*)	扩展精度复数矩阵乘法共轭。（仅 C6600）	cshort2 * cshort4 = cint2
__cmatmpy_fx(*x*, *y*)	定点复数矩阵乘法。（仅 C6600）	cshort2 * cshort4 = cshort2
__conj_cmatmpy_fx(*x*, *y*)	定点复数矩阵乘法共轭。（仅 C6600）	cshort2 * cshort4 = cshort2

所有支持的矢量内置函数的原型都列在 "c6x_vec.h" 头文件中，该文件位于“代码生成工具”安装程序的“包含”子目录中。有关矢量内置函数的完整列表，请参阅 "c6x_vec.h"。

以下示例 vbif_ex.c 文件使用带有矢量的 __add_sat() 和 __sub_sat() 内置函数：

```
#include <stdio.h>
#include <c6x_vec.h>
void print_short4(char *s, short4 v)
{
    printf("%s", s);
    printf(" <%d, %d, %d, %d>\n", v.x, v.y, v.z, v.w);
}
int main()
{
    short4 va = (short4) (1, 2, 3, -32766);
    short4 vb = (short4) (5, 32767, -13, 17);
    short4 vc = va + vb;
    short4 vd = va - vb;
    short4 ve = __add_sat(va, vb);
    short4 vf = __sub_sat(va, vb);
    print_short4("va=", va);
    print_short4("vb=", vb);
    print_short4("vc=(va+vb)=", vc);
    print_short4("vd=(va-vb)=", vd);
    print_short4("ve=__add_sat(va,vb)=", ve);
    print_short4("vf=__sub_sat(va,vb)=", vf);
    return 0;
}
```

编译示例如下：

```
%> cl6x -mv6400 --vectypes -o1 -k vbif_ex.c -z -o vbif_ex.out -llnk.cmd
```

请注意，lnk.cmd 文件包含对 rts6400.lib 的引用。rts6400.libs 库包含 c6x_veclib.obj，其定义了内置函数 __add_sat() 和 __sub_sat()。

运行此示例会生成以下输出：

```
va= <1, 2, 3, -32766>
vb= <5, 32767, -13, 17>
vc=(va+vb)= <6, -32767, -10, -32749>
vd=(va-vb)= <-4, -32765, 16, 32753>
ve=__add_sat(va,vb)= <6, 32767, -10, -32749>
vf=__sub_sat(va,vb)= <-4, -32765, 16, -32768>
```

This page intentionally left blank.

章节 8

运行时环境

本章介绍 TMS320C6000 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。

8.1 存储器模型

C6000 编译器将内存视为单线性块，该块被划分为代码子块和数据子块。C 程序生成的每个代码子块或数据子块都放置在其自己的连续内存空间中。编译器假定目标内存中有完整的 32 位地址空间可用。

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）。

8.1.1 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 *TMS320C6000 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：

- **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**包含基于主机的加载程序的命令参数。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 *TMS320C6000 汇编语言工具用户指南*。
  - 只有在使用 --rom_model 选项时，才会创建 **.cinit 段**。它包含显式初始化的全局变量和静态变量表。
  - **.init_array 段**包含用于调用全局构造函数的表。
  - **.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。
  - **.c6xabi.exidx 段**包含用于异常处理的索引表。**.c6xabi.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。
  - **.** *name* **.load 段**包含段 *名称* 的压缩映像。有关复制表的信息，请参阅 *TMS320C6000 汇编语言工具用户指南*。
  - **.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、浮点常量，以及使用 C/C++ 限定符 *far* 和 *const* 定义的数据（前提是该常量未定义为 *volatile* 或 节 7.5.2 中描述的异常之一）。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。
  - **.fardata 段**为初始化的非常量 far 全局变量和静态变量保留空间。此段是可写的。
  - **.neardata 段**为初始化的非常量 near 全局变量和静态变量保留空间。此段是可写的。
  - **.rodata 段**为 const near 全局变量和静态变量保留空间。
  - **.switch 段**包含用于大型 switch 语句的跳转表。
  - **.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。
- **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（除非指定了 --common=off），而不是放置在 .bss 中，以便将该变量从生成的应用中排除。
  - **.far 段**为已声明为 far 的全局变量和静态变量保留空间。
  - **.stack 段**为系统栈保留空间。
  - **.sysmem 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，编译器不会创建 .sysmem 段。

备注

仅使用程序存储器中的代码

除代码段外，已初始化和未初始化的段不能分配到内部程序存储器中。

汇编器会创建默认段 .text、.bss 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 7.9.3 和节 7.9.6）。

8.1.2 C/C++ 系统堆栈

C/C++ 编译器使用堆栈来：

- 保存函数返回地址
- 分配局部变量
- 传递参数给函数
- 保存临时结果

运行时堆栈从高位地址增长到低位地址。编译器使用 B15 寄存器来管理此堆栈。B15 是*栈指针* (SP)，指向堆栈上下一个未使用的位置。

链接器设置堆栈大小，创建全局符号 __TI_STACK_SIZE，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 字节。可以在链接时使用链接器命令中的 --stack_size 选项来更改堆栈大小。更多有关 --stack_size 选项的信息，请参阅《*TMS320C6000 汇编语言工具用户指南*》中的链接器描述章节。

在系统初始化时，SP 被设置为 .stack 段末尾（最高数字地址）之前的第一个 8 字节（64 位）对齐地址。由于堆栈的位置取决于 .stack 部分的分配位置，因此堆栈的实际地址是在链接时确定的。

C/C++ 环境在函数输入时自动递减 SP，以保留执行该函数所需的所有空间。堆栈指针在函数出口处递增，以将堆栈恢复到函数输入之前的状态。如果将汇编语言例程连接到 C/C++ 程序，请确保将堆栈指针恢复到函数输入之前的相同状态。

更多有关堆栈和堆栈指针的信息，请参阅节 8.4。

备注

未对齐的 SP 可能导致应用程序崩溃：无论 SP 是否对齐，HWI 调度器都会在中断调用期间使用 SP。因此，SP 绝不能错位，即使在 1 个周期内也不行。

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 3.16。

8.1.3 动态存储器分配

C6000 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 .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 .sysmem 段的大小。链接器还会创建一个全局符号 __TI_SYSMEM_SIZE，并为其分配一个等于堆大小的值（以字节为单位）。默认大小为 1K 字节。有关 --heap_size 选项的更多信息，请参阅 *TMS320C6000 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（.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 .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。

8.1.4 数据内存模型

多个选项扩展了 C6000 数据寻址模型。

8.1.4.1 确定数据地址模型

在 5.1.0 版本的编译器工具中，如果没有为对象指定近或 远关键字，编译器将生成对聚合数据的远访问和对所有其他数据的近访问。这意味着不能通过数据页 (DP) 指针访问结构、联合体、C++ 类和数组。

默认情况下，非聚合数据放置在 .bss 段中，并使用来自数据页指针（即 DP，为 B14）的相对偏移寻址进行访问。DP 指向 .bss 段的开头。与用于远数据访问的机制相比，通过数据页指针来访问数据通常更快，并且使用的指令更少。

如果要对聚合数据使用近访问，必须指定 --mem_model:data=near 选项，或使用近关键字声明您的数据。

从.bss 段开头如果有太多的静态和外部数据容纳在 15 位比例偏移中，则无法使用 --mem_model:data=near。如果 DP 相关数据访问无法到达，链接器发出错误消息。

--mem_model:data=*type* 选项控制数据的访问方式：

--mem_model:data=near	数据访问默认为 near
--mem_model:data=far	数据访问默认为 far
--mem_model:data=far_aggregates	对聚合数据的数据访问默认为 far，对非聚合数据的数据访问默认为 near。这是默认行为。

--mem_model:data 选项不影响对使用 near 或 far 关键字显式声明的对象的访问。

默认情况下，所有运行时支持数据都定义为 far。

更多有关对数据的近访问和远访问的信息，请参阅节 7.5.5。

8.1.4.2 DP 相对寻址与绝对寻址

编译器对 near (.bss) 数据使用 DP 相对寻址，而对所有其他 (far) 数据使用绝对寻址。

8.1.4.3 远常量对象

--mem_model:const 选项允许创建与 --mem_model:data 选项独立的常量对象。这就使得具有少量非常量数据和大量常量数据的应用程序能够将常量数据移出 .bss。此外，由于常量可以共享，而 .bss 不能，因此通过将常量数据移入 .const 来节省存储器。

--mem_model:const=*type* 选项具有以下值：

--mem_model:const=data	根据 --mem_model:data 选项放置常量对象。这是默认行为。
--mem_model:const=far	常量对象默认为 far，独立于--mem_model:data 选项。
--mem_model:const=far_aggregates	常量聚合对象默认为 far，标量常量默认为 near。

通过远关键字显式声明的或使用 --mem_model:const 隐式声明的远常量始终放置在 .const 段中。

8.1.5 函数调用的蹦床生成

C6000 编译器默认生成蹦床。蹦床是一种在链接时修改函数调用以到达通常太远的目标的方法。当函数调用与其目标相距超过 +/- 1M 条指令时，链接器将生成到达该目标的间接分支（或蹦床），并将函数调用重定向到指向蹦床。最终的结果是这些函数调用分支到蹦床，然后蹦床分支到最终目标。使用蹦床，不再需要指定内存模型选项来生成远调用。

8.1.6 位置无关数据

near 全局和静态数据存储在 .bss 段中。程序的所有 near 数据必须存入 32K 字节的内存中。此限制源自用于访问近数据的寻址模式，该模式与数据页指针 DP (B14) 之间的无符号偏移限制为 15 位。

对于某些应用程序，可能需要多个具有独立的 near 数据实例的数据页。例如，一个多通道应用程序可能具有运行在不同数据页上的同一程序的多个副本。该功能由 C6000 编译器的内存模型支持，称为位置无关数据。

位置无关数据意味着所有近数据访问都与数据页 (DP) 指针相关，从而允许在运行时更改 DP。编译器在三个方面实现了位置无关数据：

- 近直接内存访问

```
STW  B4,*DP(_a)
.global _a
.bss    _a,4,4
```

  所有近直接访问都与 DP 相关。
- near 间接内存访问

```
MVK (_a - $bss),A0
ADD DP,A0,A0
```

  表达式 (_a - $bss) 可以计算符号 _a 距 .bss 段开头始的偏移量。编译器在生成的汇编代码中定义全局 $bss。$bss 的值是 .bss 段的起始地址。

8.2 对象表示

本节说明如何对各种数据对象进行大小调整、对齐和访问。

8.2.1 数据类型存储

有关数据类型的一般信息，请参阅节 7.3。表 8-1 列出了各种数据类型的寄存器和内存存储空间：

表 8-1. 寄存器和内存中的数据表示

数据类型	寄存器存储	内存存储
char	寄存器的 0-7 位	8 位，与 8 位边界对齐
unsigned char	寄存器的 0-7 位	8 位，与 8 位边界对齐
short	寄存器的 0-15 位	16 位，与 16 位边界对齐
unsigned short	寄存器的 0-15 位	16 位，与 16 位边界对齐
int	整个寄存器	32 位，与 32 位边界对齐
unsigned int	整个寄存器	32 位，与 32 位边界对齐
enum (1)	整个寄存器或偶数/奇数寄存器对	32 位 (与 32 位边界对齐) 或 64 位 (与 64 位边界对齐)
float	整个寄存器	32 位，与 32 位边界对齐
float complex (4)	偶数/奇数寄存器对	64 位，与 32 位边界对齐
long	整个寄存器	32 位，与 32 位边界对齐
unsigned long	整个寄存器	32 位，与 32 位边界对齐
__int40_t	偶数/奇数寄存器对	64 位，与 64 位边界对齐
unsigned __int40_t	偶数/奇数寄存器对	64 位，与 64 位边界对齐
long long	偶数/奇数寄存器对	64 位，与 64 位边界对齐
unsigned long long	偶数/奇数寄存器对	64 位，与 64 位边界对齐
double	偶数/奇数寄存器对	64 位，与 64 位边界对齐
double complex (仅 C6600) (4)	四倍字寄存器(5)	128 位，与 64 位边界对齐
long double	偶数/奇数寄存器对	64 位，与 64 位边界对齐
long double complex (仅 C6600) (4)	四倍字寄存器(5)	128 位，与 64 位边界对齐
__x128_t (仅 C6600) (2)	四倍字寄存器(5)	128 位，与 64 位边界对齐
结构体	成员按其各自类型的要求存储。	存储器是最大成员类型边界对齐的倍数；成员根据其各自类型的要求进行存储和对齐。
数组	成员按其各自类型的要求存储。	成员按其各自类型的要求存储。(3)结构中的所有数组都根据数组中每个元素的类型对齐。
数据成员指针	寄存器的 0-31 位	32 位，与 32 位边界对齐
成员函数指针	组件按其各自类型的要求存储	32 位，与 32 位边界对齐

(1) 有关枚举类型大小的详细信息，请参阅节 7.3.1。
(2) 有关 __x128_t 容器类型的详细信息，请参阅节 8.6.7。
(3) 静态作用域数组与 64 位边界对齐。
(4) 要使用复杂的数据类型，必须包含 <complex.h> 头文件。有关如何存储复杂的数据类型更多信息，请参阅节 7.5.1。
(5) 四倍字寄存器仅支持 C66000。

8.2.1.1 char 和 short 数据类型（有符号和无符号）

char 和 unsigned char 数据类型作为单个字节存储在内存中；数据类型加载到寄存器的 0-7 位上， 并从这些位进行存储（请参阅图 8-1）。定义为 short 或 unsigned short 的对象作为两个字节存储在内存中，位于半字（2 字节）对齐的地址上；这些对象加载到寄存器的 0-15 位上， 并从这些位进行存储（请参阅图 8-1）。

在大端模式下，将内存的第一个字节（即较低地址）移动到寄存器的 8-15 位上，并将内存的第二个字节移动到 0-7 位上以使 2 字节对象加载到寄存器中。在小端模式下，将内存的第一个字节（即低地址）移动到寄存器的 0-7 位上，并将内存的第二个字节移动到-15 位上以使 2 字节对象加载到寄存器中。

图 8-1. Char 和 Short 数据存储格式

有符号 8 位 char

																								MS						LS
S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	I	I	I	I	I	I
31																								7						0

无符号 8 位 char

																								MS						LS
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	U	U	U	U	U	U	U
31																								7						0

有符号 16 位 short

																MS														LS
S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S	I	I	I	I	I	I	I	I	I	I	I	I	I	I
31																								15						0

无符号 16 位 short

																MS														LS
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U
31																								15						0

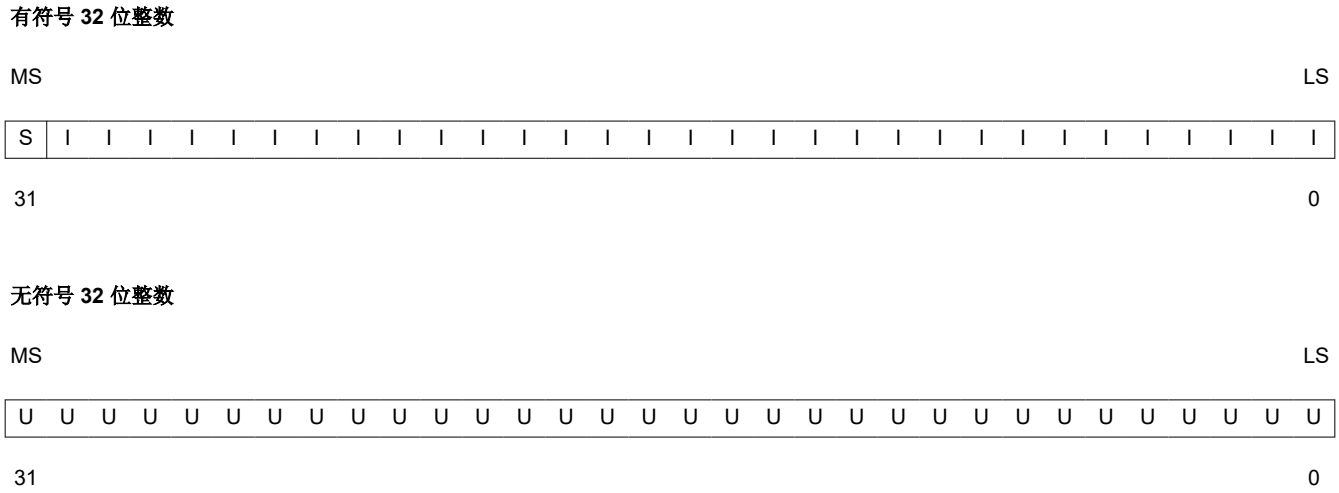
图例: S = 符号，I = 有符号整数，U = 无符号整数，MS = 最高有效，LS = 最低有效

8.2.1.2 enum、int 和 long 数据类型（有符号和无符号）

int 和 unsigned int 数据类型作为 32 位对象存储在内存中（请参阅图 8-2）。这些类型的对象被加载到寄存器的 0-31 位上，并从这些位进行存储。在大端模式下，通过将内存的第一个字节（即较低地址）移动到寄存器的 24-31 位 ，将内存的第二个字节移动到 16-23 位 ，将第三个字节移动到 8-15 位 ，并将第四个字节移到 0-7 位以使 4 字节对象加载到寄存器中。 。在小端模式下，通过将内存的第一个字节（即较低地址）移动到寄存器的 0-7 位，将第二个字节移动到 8-15 位，将第三个字节移动到 16-23 位，并将第四个字节移到 24-31 位以使 4 字节对象加载到寄存器中。

有关枚举类型大小的详细信息，请参阅节 7.3.1。

图 8-2. 32 位数据存储格式

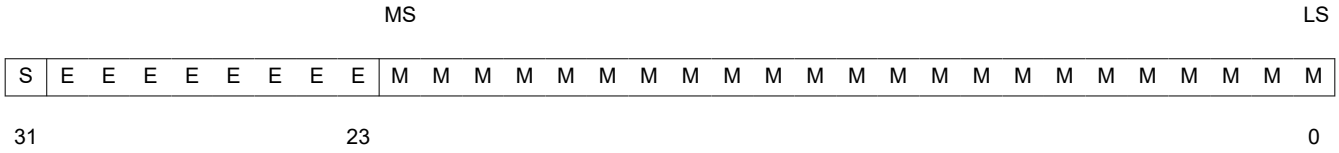


图例: S = 符号，U = 无符号整数，I = 有符号整数，MS = 最高有效，LS = 最低有效

8.2.1.3 浮点数据类型

浮点数据类型作为 32 位对象存储在内存中（请参阅图 8-3）。定义为浮点的对象被加载到寄存器的 0-31 位上，并从这些位进行存储。在大端模式下，通过将内存的第一个字节（即较低地址）移动到寄存器的 24-31 位，将内存的第二个字节移动到 16-23 位，将第三个字节移动到 8-15 位，并将第四个字节移到 0-7 位以使 4 字节对象加载到寄存器中。在小端模式下，通过将内存的第一个字节（即较低地址）移动到寄存器的 0-7 位，将第二个字节移动到 8-15 位，将第三个字节移动到 16-23 位，并将第四个字节移到 24-31 位以使 4 字节对象加载到寄存器中。

图 8-3. 单精度浮点字符数据存储格式

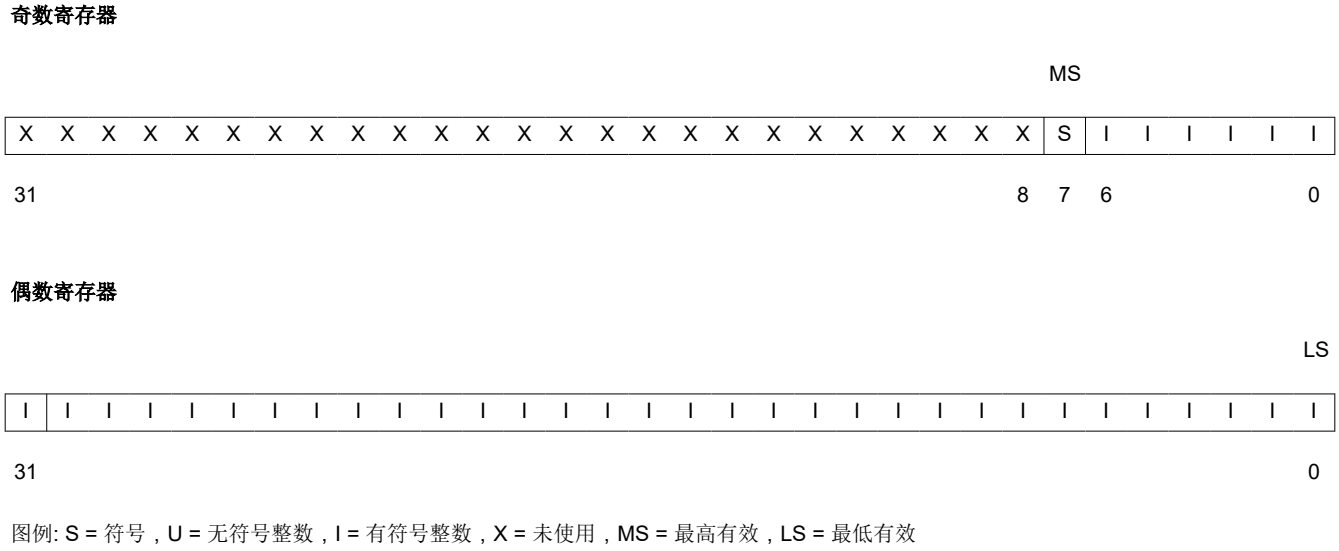


图例：S = 符号，M = 尾数，E = 指数，MS = 最高有效，LS = 最低有效

8.2.1.4 __int40_t 数据类型（有符号和无符号）

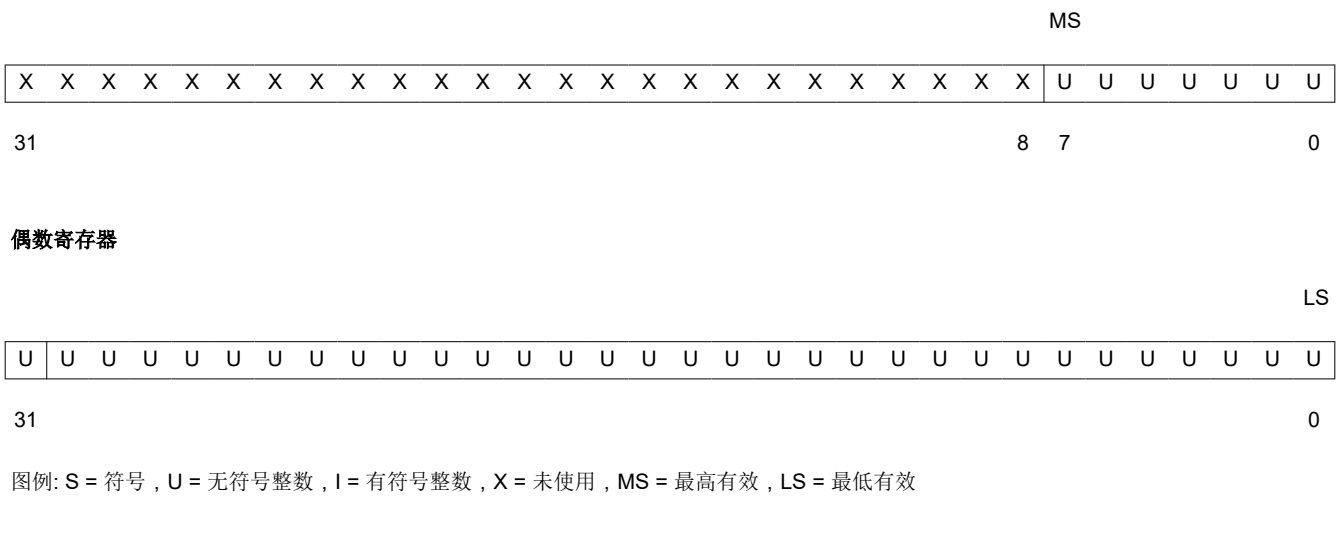
__int40_t 数据类型存储在一对奇数/偶数寄存器中（请参阅图 8-4），并且始终以“奇数寄存器:偶数寄存器”的成对格式（例如，A1:A0）进行引用。在小端模式下，较低地址加载到偶数寄存器中，较高地址加载到奇数寄存器中；如果数据是从位置 0 加载的，则 0 处的字节是偶数寄存器的最低字节。在大端模式下，较高地址加载到偶数寄存器中，较低地址加载到奇数寄存器中；如果数据是从位置 0 加载的，则 0 处的字节是奇数寄存器的最高字节，但会被忽略。

图 8-4. 40 位数据存储格式 - 有符号 __int40_t



图例: S = 符号，U = 无符号整数，I = 有符号整数，X = 未使用，MS = 最高有效，LS = 最低有效

图 8-5. 无符号 40 位 __int40_t

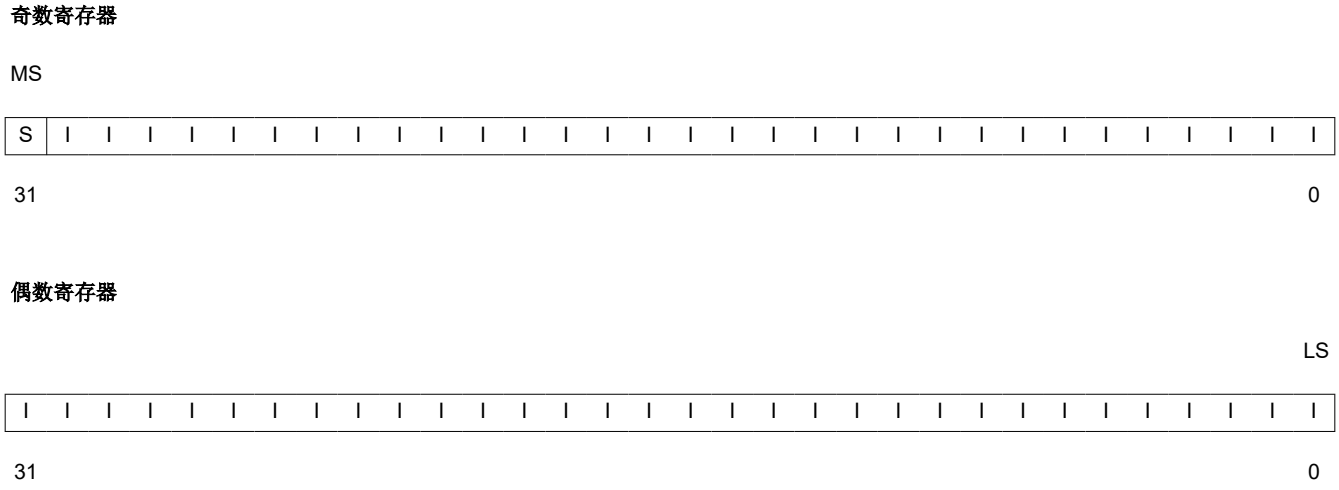


图例: S = 符号，U = 无符号整数，I = 有符号整数，X = 未使用，MS = 最高有效，LS = 最低有效

8.2.1.5 long long 数据类型（有符号和无符号）

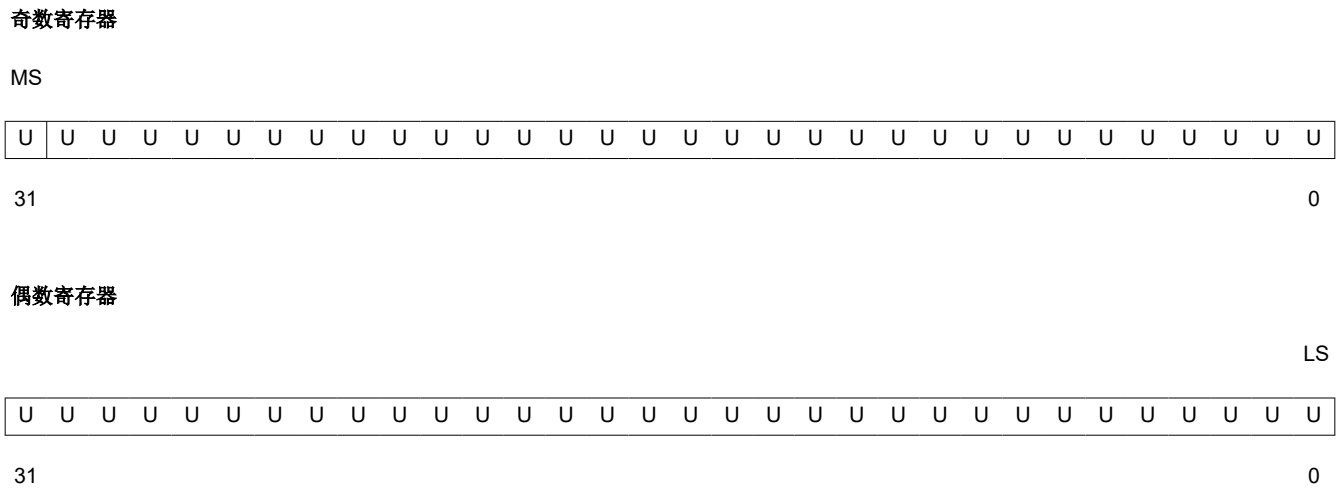
long long 和 unsigned long long 数据类型存储在一对奇数/偶数寄存器中（请参阅图 8-6），并且始终以“奇数寄存器:偶数寄存器”的成对格式（例如，A1:A0）被引用。在小端模式下，较低地址加载到偶数寄存器中，较高地址加载到奇数寄存器中；如果数据是从位置 0 加载的，则 0 处的字节是偶数寄存器的最低字节。在大端模式下，较高地址加载到偶数寄存器中，较低地址加载到奇数寄存器中；如果数据是从位置 0 加载的，则 0 处的字节是奇数寄存器的最高字节。

图 8-6. 64 位数据存储格式 - 有符号 64 位 long



图例: S = 符号，U = 无符号整数，I = 有符号整数，X = 未使用，MS = 最高有效，LS = 最低有效

图 8-7. 无符号 64 位 long

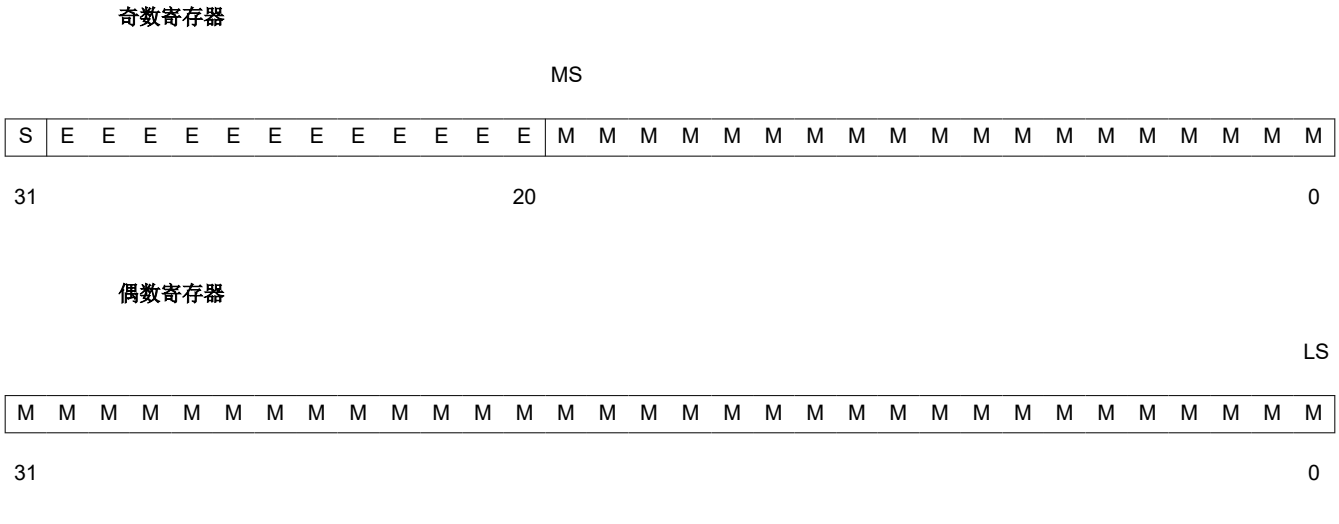


图例: S = 符号，U = 无符号整数，I = 有符号整数，X = 未使用，MS = 最高有效，LS = 最低有效

8.2.1.6 double 和 long double 数据类型

double 和 long double 数据类型存储在一对奇数/偶数寄存器中（请参阅图 8-8），并且只能以一种格式存在于寄存器中：以“奇数寄存器:偶数寄存器”的格式成对存在（例如，A1:A0）。奇数内存字包含符号位、指数以及尾数的最高有效部分。偶数内存字包含尾数的最低有效部分。在小端模式下，较低地址加载到偶数寄存器中，较高地址加载到奇数寄存器中。在大端模式下，较高地址加载到偶数寄存器中，较低地址加载到奇数寄存器中。在小端模式下，如果代码是从位置 0 加载的，则 0 处的字节是偶数寄存器的最低字节。在大端模式下，如果代码是从位置 0 加载的，则 0 处的字节是奇数寄存器的最高字节。

图 8-8. 双精度浮点数据存储格式



图例: S = 符号，M = 尾数，E = 指数，MS = 最高有效，LS = 最低有效

8.2.1.7 指向数据成员类型的指针

指向数据成员对象的指针像 unsigned int（32 位）整数类型一样存储在内存中。其值是类中数据成员的字节偏移加上 1。保留零值以表示 NULL 指针。

8.2.1.8 指向成员函数类型的指针

指向成员函数对象的指针的布局相当于：

```
struct __mptr {
   __vptp f;
   ptrdiff_t d;
};
```

其中，_vptp 是以下联合体：

```
union {
   void (*f) ();
   int 0;
}
```

如果参数 f 是非虚函数，则其为指向成员函数的指针。0 是类对象中虚函数指针的偏移量。参数 d 是要添加到该指针的类对象开头的偏移量。

8.2.1.9 结构和数组

结构与其具有最严格对齐要求的成员所需的边界对齐。例如，如果结构的成员所需的最大对齐是 16 位对齐（例如，short），则整个结构对齐到 16 位边界。如果结构包含需要 64 位对齐的类型（例如 double 或 long long），则该结构对齐到 64 位边界。

如果结构成员本身也是结构，则必须先确定内部结构的大小和对齐方式，然后才能确定外部结构的大小和对齐方式。

除非使用了 packed 属性，否则结构成员的大小和对齐方式与它们作为独立对象的大小和对齐方式相同。结构的数组成员按与其元素类型的对齐方式对齐；这可能与元素是独立的顶级（静态）对象时的对齐方式不同。

结构的大小总是等于结构对齐方式的倍数。有时，需要在最后一个成员之后填充，以将大小四舍五入为结构对齐方式的倍数。结构的大小包括成员之间任何必要的填充。例如，如果结构的最大成员是 float 类型，则结构的大小将是 32 位的倍数。

静态作用域数组（有时称为顶级数组）在 8 字节（64 位）边界上对齐。

8.2.2 位字段

位字段是唯一打包在字节中的对象。也就是说，两个位字段可存储在同一字节中。在 C 语言中，位字段的大小可以从 1 位到 64 位不等，在 C++语言中则可以更大。

对于大端模式，位字段按定义的顺序从最高有效位 (MSB) 到最低有效位 (LSB) 打包到寄存器中。位字段按从最高有效字节 (MSbyte) 到最低有效字节 (LSbyte) 的顺序打包到内存中。对于小端模式，位字段按照定义的顺序从 LSB 到 MSB 打包到寄存器中，并按照从 LSbyte 到 MSbyte 的顺序打包到内存中。

位字段的大小、对齐方式和类型遵循以下规则：

- 支持最长为 long long 类型的位字段。
- 位字段被视为声明的有符号或无符号类型。
- 包含位字段的结构的大小和对齐方式取决于位字段的声明类型。例如，考虑以下结构：

```
struct st
{
   int a:4
};
```

  此结构使用了 4 个字节并在 4 个字节处对齐。
- 未命名的位字段会影响结构或联合体的对齐方式。例如，考虑以下结构：

```
struct st
{
   char a:4;
   int :22;
};
```

  此结构使用了 4 个字节并在 4 字节边界处对齐。
- 根据位字段的声明类型访问声明为易失性的位字段。易失性位字段引用只生成一个对其存储的引用；多个易失性位字段访问不会被合并。

图 8-9 使用以下位字段定义说明位字段打包：

```
struct{
int A:7
int B:10
int C:3
int D:2
int E:9
}x;
```

A0 表示字段 A 的最低有效位；A1 表示下一个最低有效位，以此类推。同样，位字段在内存中的存储是通过逐字传输而不是逐位传输完成的。

图 8-9. 以大端格式和小端格式打包位字段

大端寄存器

MS LS

A	A	A	A	A	A	A	B	B	B	B	B	B	B	B	B	C	C	C	D	D	E	E	E	E	E	E	E	E	X
6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	2	1	0	1	0	8	7	6	5	4	3	2	1	0	X

31 0

大端内存

字节 0								字节 1								字节 2								字节 3							
A	A	A	A	A	A	A	B	B	B	B	B	B	B	B	B	C	C	C	D	D	E	E	E	E	E	E	E	E	X
6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	2	1	0	1	0	8	7	6	5	4	3	2	1	0	X

小端寄存器

MS LS

X	E	E	E	E	E	E	E	E	E	D	D	C	C	C	B	B	B	B	B	B	B	B	B	A	A	A	A	A	A	A	A
X	8	7	6	5	4	3	2	1	0	1	0	2	1	0	9	8	7	6	5	4	3	2	1	0	6	5	4	3	2	1	0

31 0

小端内存

字节 0								字节 1								字节 2								字节 3							
B	A	A	A	A	A	A	A	B	B	B	B	B	B	B	B	E	E	D	D	C	C	C	B	X	E	E	E	E	E	E	E
0	6	5	4	3	2	1	0	8	7	6	5	4	3	2	1	1	0	1	0	2	1	0	9	X	8	7	6	5	4	3	2

图例: X = 未使用，MS = 最高有效，LS = 最低有效

8.2.3 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 8.9。
- 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.const:string 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；编译器明确添加终止 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   ".const:string"
$C$SL5: .string "abc",0
```

字符串标签的形式为 CSLn，其中 C 是编译器生成的符号前缀，*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 CSL*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 .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 */
```

8.3 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。表 8-2 总结了编译器如何使用 TMS320C6000 寄存器。

表 8-2 中的寄存器可供编译器分配寄存器变量和临时表达式结果之用。如果编译器无法分配所需类型的寄存器，则会发生溢出。溢出是将寄存器的内容移动到存储器以释放寄存器用于其他目的的过程。

双精度型、长整型、超长整型或长双精度型的对象被分配到奇数/偶数寄存器对中，并且始终作为寄存器对引用（例如，A1:A0）。奇数寄存器包含符号位、指数和尾数的最高有效部分。偶数寄存器包含尾数的最低有效部分。如果第一个参数是双精度型、长整型、超长整型或长双精度型，则 A4 寄存器与 A5 一起用于传递第一个参数。第二个参数的 B4 和 B5 也是如此，依此类推。有关参数传递寄存器和返回寄存器的更多信息，请参阅节 8.4。

表 8-2. 寄存器的使用

寄存器	保留函数	特殊用途	寄存器	保留函数	特殊用途
A0	父级	–	B0	父级	–
A1	父级	–	B1	父级	–
A2	父级	–	B2	父级	–
A3	父级	结构寄存器（指向返回结构的指针）(1)	B3	父级	返回寄存器（返回到的地址）
A4	父级	参数 1 或返回值	B4	父级	参数 2
A5	父级	参数 1 或返回值，双精度、长整型和超长整型值返回 A4	B5	父级	参数 2，双精度、长整型和超长整型值返回 B4
A6	父级	参数 3	B6	父级	参数 4
A7	父级	参数 3，双精度、长整型和超长整型值返回 A6	B7	父级	参数 4，双精度、长整型和超长整型值返回 B6
A8	父级	参数 5	B8	父级	参数 6
A9	父级	参数 5，双精度、长整型和超长整型值返回 A8	B9	父级	参数 6，双精度、长整型和超长整型值返回 B8
A10	子级	参数 7	B10	子级	参数 8
A11	子级	参数 7，双精度、长整型和超长整型值返回 A10	B11	子级	参数 8，双精度、长整型和超长整型值返回 B10
A12	子级	参数 9	B12	子级	参数 10
A13	子级	参数 9，双精度、长整型和超长整型值返回 A12	B13	子级	参数 10，双精度、长整型和超长整型值返回 B12
A14	子级	–	B14	子级	数据页指针 (DP)
A15	子级	帧指针 (FP)	B15	子级	栈指针 (SP)
A16-A31	父级		B16-B31	父级	
ILC	子级	循环缓冲区计数器	NRP	父级	
IRP	父级		RILC	子级	循环缓冲区计数器

(1) 大小为 64 或更小的结构体通过寄存器中的值传递，而不是使用 A3 中的指针通过引用传递。

编译器不会保存或恢复所有其他控制寄存器。

编译器假设，表 8-2 中未列出的控制寄存器会影响具有默认值的编译代码。例如，编译器假设，所有启用了循环寻址的寄存器均可用于线性寻址（AMR 用于启用循环寻址）。启用循环寻址然后调用 C/C++ 函数而不将 AMR 恢复为默认设置违反了调用惯例。确定的是，对编译器所生成代码有影响的控制寄存器，在使用汇编代码调用 C/C++ 函数时便具有默认值。

汇编语言程序员必须知道链接器假定 B15 包含栈指针。链接器需要在它生成的 trampoline 代码中保存和恢复栈上的值。如果在汇编代码中不使用 B15 作为栈指针，则应使用禁用 trampoline 的链接器选项 --trampolines=off 。否则，trampoline 可能会破坏内存并覆盖寄存器值。

8.4 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 *C6000 嵌入式应用二进制接口应用报告* (SPRAB89)。

8.4.1 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. 传递给函数的参数被放置在寄存器或堆栈上。

   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：

   如果将参数传递给函数，则最多将前十个参数放置在寄存器 A4、B4、A6、B6、A8、B8、A10、B10、A12 和 B12 中。如果传递 long、long long、double 或 long double，它们将被放置在寄存器对 A5:A4、B5:B4、A7:A6 中，依此类推。

   C6600 __x128_t 类型对象与 64 位或 128 位边界对齐。（请参阅节 8.6.2 中的注释。）但是，对于 C6600，如果传递了多个 __x128_t 参数，则下一个 __x128_t 参数将在第一个可用四通道中传递，其中可用四通道列表的顺序如下：A7:A6:A5:A4、B7:B6:B5:B4、A11:A10:A9:A8、B11:B10:B9:B8。如果没有更多可用的四通道，则 __x128_t 进入堆栈。即使较早的 __x128_t 参数已放置在堆栈上，后续的 32 位、40 位或 64 位参数也可以采用第一个可用的寄存器或寄存器对。

   所有其余的参数都放在堆栈上（也就是说，假设 C6600 没有传递 __x128_t，堆栈指针指向下一个空闲位置；SP + *偏移*指向第十一个参数，依此类推）。放置在堆栈上的参数必须与适合其大小的值对齐。未在原型中声明且大小小于 int 大小的参数作为 int 传递。如果没有声明原型，则 float 类型参数将作为 double 型传递。

   结构体参数作为结构体的地址传递。由被调用函数来创建本地副本。

   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。

   图 8-10 显示了寄存器参数惯例。
2. 如果在调用后需要寄存器的值，则调用函数必须保存寄存器 A0-A9、B0-B9、A16-A31 和 B16-B31。该点应该通过将值推入堆栈来实现。有关寄存器惯例的详细信息，请参阅《*C6000 嵌入式应用二进制接口应用报告*》(SPRAB89A) 的第 3.2 节。
3. 调用方（父级）会调用函数（子级）。
4. 返回时，调用方通过添加到栈指针来收回参数所需的任何堆栈空间。只有在不是通过 C/C++ 代码编译的汇编程序中才需要此步骤。这是因为 C/C++ 编译器会在函数开始时分配所有调用所需的堆栈空间，并在函数结束时取消分配空间。

```
int func1( int a,    int b,      int c);
A4         A4        B4          A6
int func2( int a,    float b,    int c)         struct A d,     float e,   int f,   int g);
A4         A4        B4          A6             B6              A8         B8       A10
int func3( int a,    double b,   float c)       long double d);
A4         A4        B5:B4       A6             B7:B6
/*NOTE: The following function has a variable number of arguments. */
int vararg(int a,    int b,      int c,         int d);
A4         A4        B4          A6             stack
struct A func4(      int y);
A3                   A4
__x128_t func5(        __x128_t a);
A7:A6:A5:A4          A7:A6:A5:A4
void func6(int a,    int b,      __x128_t c);
           A4        B4          A11:A10:A9:A8
void func7(int a,    int b,      __x128_t c,    int d,          int e,     int f,   __x128_t g,  int h);
           A4        B4          A11:A10:A9:A8  A6              B6         B8       stack        B10
```

图 8-10. 寄存器参数惯例

8.4.2 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. 被调用函数（子级）在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。这种分配只在函数开始时发生一次，可能包括帧指针 (FP) 的分配。

   帧指针用于从堆栈中读取参数并处理寄存器溢出指令。如果任何参数被放置在堆栈上或者如果帧大小超过 128K 字节，帧指针 (A15) 将按以下方式分配：

   a. 旧 A15 被保存在堆栈上。
   b. 新的帧指针被设置为当前 SP (B15)。
   c. 通过将 SP 递减一个常数来分配帧。
   d. A15 (FP) 和 B15 (SP) 在此函数内的任何其他位置都不会递减。

   如果不满足上述条件，则不分配帧指针 (A15)。在这种情况下，通过从寄存器 B15 (SP) 中减去一个常数来分配帧。寄存器 B15 (SP) 在此函数内的任何其他位置都不会递减。
2. 如果被调用函数调用任何其他函数，则返回地址必须保存在堆栈上。否则，返回地址将留在返回寄存器 (B3) 中并被下一个函数调用覆盖。
3. 如果被调用函数修改了任何编号为 A10 到 A15 或 B10 到 B15 的寄存器，则必须将寄存器保存在其他寄存器或堆栈上。被调用函数可以修改任何其他寄存器而不保存寄存器。
4. 如果被调用函数需要一个结构体参数，其将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. 被调用函数执行函数的代码。

6. 如果被调用函数返回任何整数、指针或浮点类型，则返回值被放置在 A4 寄存器中。如果该函数返回任何双精度型、长双精度型、长整型或超长整型，则相应值被放置在 A5:A4 寄存器对中。对于 C6600，如果该函数返回 __x128_t，则该值被放置在 A7:A6:A5:A4 中。

   如果该函数返回一个结构体，则调用方会为该结构体分配空间并将返回空间的地址传递给 A3 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。
7. 恢复在步骤 3 中保存的任何编号为 A10 到 A15 或 B10 到 B15 的寄存器。
8. 如果 A15 用作帧指针 (FP)，则从堆栈中恢复 A15 的旧值。步骤 1 中分配给函数的空间在函数结束时通过向寄存器 B15 (SP) 添加常量来回收。
9. 该函数通过跳转到返回寄存器 (B3) 的值或返回寄存器的保存值来返回。

8.4.3 访问参数和局部变量

函数通过寄存器 A15 (FP) 或寄存器 B15 (SP) 间接访问其栈参数和本地非寄存器变量，其中一个寄存器指向栈顶。栈向较小的地址增长，因此通过 FP 或 SP 正偏移来访问函数的本地数据和参数数据。局部变量、临时存储器，以及为该函数调用的函数的栈参数所保留的区域，都是使用偏移量来访问的，该偏移量小于 FP 或 SP 在函数开头减去的常量。

访问传递到此函数的栈参数所需的偏移量大于或等于寄存器 FP 或 SP 在函数开头减去的常量。如果使用了优化或使用“寄存器”关键字定义了寄存器参数，编译器会尝试将寄存器参数保留在原始寄存器中。否则，会将参数复制到栈中，以释放这些寄存器供进一步分配。

有关是使用 FP 还是 SP 访问局部变量、临时存储器和栈参数的信息，请参阅节 8.4.2。更多有关 C/C++ 系统栈的信息，请参阅节 8.1.2。

8.5 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 *TMS320C6000 汇编语言工具用户指南* 中关于“链接器符号”的部分。

8.6 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 8.6.1）。
- 在 C/C++ 源代码中使用汇编语言变量和常量（请参阅节 8.6.3）。
- 使用直接嵌入 C/C++ 源代码中的内联汇编语言（请参阅节 8.6.5）。
- 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 8.6.6）。

8.6.1 使用汇编语言模块与 C/C++ 代码

只要遵循 节 8.4 中定义的调用惯例，以及 节 8.3 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 8.3 中罗列的寄存器惯例。
- 必须保留 A10 至 A15、B3 以及 B10 至 B15 寄存器，可能还需要保留 A3 寄存器。如通常使用栈，则无需显式保留栈。换而言之，您可以在函数中自由使用栈，只要在函数退出前将压入的全部内容弹出即可。您也可以自由使用所有其他寄存器，无需保留其内容。

- A10 至 A15 和 B10 至 B15 寄存器需要在函数返回之前恢复（即使上述任一寄存器正在传递参数也是如此）。
- 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 8.7。
- 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 8.4.1 中所述。

  请记住，C/C++ 编译器仅保留 A10 至 A15 和 B10 至 B15 寄存器。C/C++ 函数可更改任何其他寄存器，保存其内容（即在调用函数前将其内容压入栈），并在函数返回后恢复这些寄存器。
- 函数必须根据其 C/C++ 声明正确地返回值。整型值和 32 位浮点 (float) 值会返回 A4。双精度、长双精度、长整型和超长整型值会返回 A5:A4。C6600 __x128_t 的值会返回 A7:A6:A5:A4。通过将结构复制到 A3 中的地址即可将其返回。
- 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 7.12。
- 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。
- 可能需要保存 TSR 控制寄存器的 SGIE 位。请参阅 节 8.7.1 了解更多信息。
- 编译器假设，表 8-2 中未列出的控制寄存器对具有默认值的编译代码会产生效果。例如，编译器假设，所有启用了循环寻址的寄存器均可用于线性寻址（AMR 用于启用循环寻址）。启用循环寻址然后调用 C/C++ 函数而不将 AMR 恢复为默认设置违反了调用惯例。另外，启用循环寻址并启用中断会违反调用惯例。您必须确定，会影响编译器生成的代码的控制寄存器，在从汇编语言调用 C/C++ 函数时具有默认值。
- 汇编语言程序员必须知道链接器假定 B15 包含栈指针。链接器需要在它生成的 trampoline 代码中保存和恢复栈上的值。如果您在汇编代码中不使用 B15 作为栈指针，则应使用链接器选项禁用 trampolines：--trampolines=off。否则，trampoline 可能会破坏内存并覆盖寄存器值。
- 汇编代码将 B14 和/或 B15 用于本地化目的，而不是用于数据页指针和栈指针，会违反调用惯例。汇编编程器需要保护这些非标准使用 B14 和 B15 的领域，可关闭与这些代码相关的中断。中断处理例程需要栈（因此假设栈指针在 B15 中），因此此代码需要关闭中断。另外，中断服务例程可访问全局数据，也可能调用其他访问全局数据的函数，因此此特殊处理也适用于 B14。数据页指针和栈指针恢复后，可将中断打开。

8.6.2 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 8-1 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 8-2 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 8-1. 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 8-2. 由 示例 8-1 调用的汇编语言程序

```
    .global asmfunc
    .global gvar
asmfunc:
    LDW     *+b14(gvar),A3
    NOP     4
    ADD     a3,a4,a3
    STW     a3,*b14(gvar)
    MV      a3,a4
    B       b3
    NOP     5
```

在示例 8-1 的 C++ 程序中，asmfunc 的 extern 声明是可选的，因为返回类型是 int。与 C/C++ 函数一样，只有在返回非整数值或传递非整数参数时才需要声明汇编函数。

备注

SP 语义

栈指针必须始终采用 8 字节对齐方式。这是由 C 编译器和运行时支持库中的系统初始化代码自动执行的。任何手写汇编代码（启用了中断，或调用在 C 或线性汇编源中定义的函数）也应在栈上保存 8 字节的倍数。

备注

栈分配

即使编译器保证栈的双字对齐并且栈指针 (SP) 指向栈空间中的下一个空闲位置，但只有足够的保证空间在该位置存储一个 32 位字。被调用函数必须分配空间来存储双字。

备注

__x128_t 类型数据对象的对齐方式（仅限 C6600）

C6600 提供 128 位容器类型 __x128_t。此类型的全局数据对象与 16 字节边界（128 位）对齐。局部 __x128_t 变量在栈上分配，但不一定在 16 字节边界上对齐，因为它们的实际对齐方式取决于栈指针 (SP) 的对齐方式和局部 __x128_t 类型对象的 SP 相对偏移量。编译器将栈与 8 字节边界对齐。

8.6.3 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 .bss 段中定义的变量、未在 .bss 段中定义的变量，或者链接器符号。

8.6.3.1 访问汇编语言全局变量

从 .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. 当您使用 .usect 时，变量是在 .bss 以外的段中定义的，因此必须在 C 语言中声明为 far。
3. 使用 .def 或 .global 指令将定义设置为外部引用。
4. 在汇编语言中使用适当的链接名。
5. 在 C/C++ 语言中，将变量声明为 *extern* 并正常访问它。

示例 8-4 和示例 8-3 说明了如何访问 .bss 中定义的变量。

示例 8-3. 汇编语言变量程序

```
* Note the use of underscores in the following lines
        .bss     _var1,4,4      ; Define the variable
        .global  var1           ; Declare it as external
_var2   .usect   "mysect",4,4   ; Define the variable
        .global  _var2          ; Declare it as external
```

示例 8-4. C 程序从示例 8-3 中访问汇编语言

```
extern int var1;        /* 外部变量 */
extern far int var2;    /* 外部变量 */
var1 = 1;               /* 使用该变量  */
var2 = 1;               /* 使用该变量  */
```

8.6.3.2 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《*TMS320C6000 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”，了解更多使用 _symval 的示例。

有关符号和符号表的更多信息，请参阅《*TMS320C6000 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，如示例 8-5 和示例 8-6 中所示。

示例 8-5. 从 C 语言访问汇编语言常量

```
extern int table_size;          /*外部引用 */
#define TABLE_SIZE ((int) (&table_size))
       .                        /* 通过强制转换来隐藏地址 */
       .
       .
for (I=0; i<TABLE_SIZE; ++I)   /* 像普通符号一样使用 */
```

示例 8-6. 示例 8-5 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 8-5 中，使用了 int。您能够以类似的方式引用链接器定义的符号。

8.6.4 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包含声明和原型的 C 头文件。任何合法的 C/C++ 都可以在 .cdecls 块中使用，C/C++ 声明将导致自动生成合适的汇编语言，从而允许您在汇编语言代码中引用 C/C++ 构造。更多相关信息，请参阅《*TMS320C6000 汇编语言工具用户指南*》中的 C/C++ 头文件章节。

8.6.5 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。如需更多信息，请参阅 节 7.8。

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 asm 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。

8.6.6 使用内在函数访问汇编语言语句

C6000 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。

内在函数通过前导下划线指定，可像调用函数一样进行访问。例如：

```
int x1, x2, y;
y = _sadd(x1, x2);
```

备注

C 语言与汇编语言中的内在函数指令

在某些情况下，编译器可能不会使用与内在函数完全对应的汇编语言指令。在这类情况下，程序的含义不会改变。

列举内在函数的表格适用于以下器件系列：

表 8-3. 器件系列和内在函数表

系列	表 8-5	表 8-6	表 8-7
C6400+	是		
C6740	是	是	
C6600	是	是	是

表 8-4 对 C6000 内在函数进行了汇总，并说明了哪个器件支持哪个内在函数。

表 8-4. 各器件对 C6000 C/C++ 内在函数的支持

内在函数	C6400+	C6740	C6600
_abs	是	是	是
_abs2	是	是	是
_add2	是	是	是
_add4	是	是	是
_addsub	是	是	是
_addsub2	是	是	是
_amem2	是	是	是
_amem2_const	是	是	是
_amem4	是	是	是
_amem4_const	是	是	是
_amem8	是	是	是
_amem8_const	是	是	是
_amem8_f2	是	是	是
_amem8_f2_const	是	是	是
_amemd8	是	是	是
_amemd8_const	是	是	是
_avg2	是	是	是
_avgu4	是	是	是
_bitc4	是	是	是
_bitr	是	是	是
_ccmatmpy			是
_ccmatmpyr1			是
_ccmpy32r1			是
_clr	是	是	是
_clrr	是	是	是
_cmatmpy			是
_cmatmpyr1			是
_cmpeq2	是	是	是
_cmpeg4	是	是	是
_cmpgt2	是	是	是
_cmpgtu4	是	是	是
_cmplt2	是	是	是
_cmpltu4	是	是	是
_cmpy	是	是	是
_cmpy32r1			是
_cmpyr	是	是	是
_cmpyr1	是	是	是
_cmpysp			是
_complex_conjugate_mpysp			是
_complex_mpysp			是
_crot270			是
_crot90			是
_dadd			是
_dadd2			是

表 8-4. 各器件对 C6000 C/C++ 内在函数的支持 (continued)

内在函数	C6400+	C6740	C6600
_daddsp			是
_dadd_c			是
_dapys2			是
_davg2			是
_davgnr2			是
_davgnru4			是
_davgu4			是
_dccmpyr1			是
_dcmpeq2			是
_dcmpeq4			是
_dcmpgt2			是
_dcmpgtu4			是
_dccmpy			是
_dcmpy			是
_dcmpyr1			是
_dcrot90			是
_dcrot270			是
_ddotp4	是	是	是
_ddotp4h			是
_ddotph2	是	是	是
_ddotph2r	是	是	是
_ddotpl2	是	是	是
_ddotpl2r	是	是	是
_ddotpsu4h			是
_deal	是	是	是
_dinthsp			是
_dinthspu			是
_dintsp			是
_dintspu			是
_dmax2			是
_dmaxu4			是
_dmin2			是
_dminu4			是
_dmpy2			是
_dmpysp			是
_dmpysu4			是
_dmpyu2			是
_dmpyu4			是
_dmv	是	是	是
_dmvd			是
_dotp2	是	是	是
_dotp4h			是
_dotp4hll			是
_dotpn2	是	是	是
_dotpnrsu2	是	是	是

表 8-4. 各器件对 C6000 C/C++ 内在函数的支持 (continued)

内在函数	C6400+	C6740	C6600
_dotpnrus2	是	是	是
_dotprsu2	是	是	是
_dotpsu4	是	是	是
_dotpus4	是	是	是
_dotpsu4h			是
_dotpsu4hll			是
_dotpu4	是	是	是
_dpack2	是	是	是
_dpackh2			是
_dpackh4			是
_dpacklh2			是
_dpacklh4			是
_dpackl2			是
_dpackl4			是
_dpackx2	是	是	是
_dpint		是	是
_dsadd			是
_dsadd2			是
_dshl			是
_dshl2			是
_dshr			是
_dshr2			是
_dshru			是
_dshru2			是
_dsmpy2			是
_dspacku4			是
_dspint			是
_dspinth			是
_dssub			是
_dssub2			是
_dsub			是
_dsub2			是
_dsubsp			是
_dtol	是	是	是
_dtoll	是	是	是
_dxpnd2			是
_dxpnd4			是
_ext	是	是	是
_extr	是	是	是
_extu	是	是	是
_extur	是	是	是
_f2tol		是	是
_f2toll		是	是
_fabs		是	是
_fabsf		是	是

表 8-4. 各器件对 C6000 C/C++ 内在函数的支持 (continued)

内在函数	C6400+	C6740	C6600
_fdmvd_f2			是
_fdmv_f2	是	是	是
_ftoi	是	是	是
_gmpy	是	是	是
_gmpy4	是	是	是
_hi	是	是	是
_hill	是	是	是
_itod	是	是	是
_itof	是	是	是
_itoll	是	是	是
_labs	是	是	是
_land			是
_landn			是
_ldotp2	是	是	是
_lmbd	是	是	是
_lnorm	是	是	是
_lo	是	是	是
_loll	是	是	是
_lor			是
_lsadd	是	是	是
_lssub	是	是	是
_ltod	是	是	是
_lltod	是	是	是
_lltof2		是	是
_ltof2		是	是
_max2	是	是	是
_maxu4	是	是	是
_mfence			是
_min2	是	是	是
_minu4	是	是	是
_mem2	是	是	是
_mem2_const	是	是	是
_mem4	是	是	是
_mem4_const	是	是	是
_mem8	是	是	是
_mem8_const	是	是	是
_mem8_f2		是	是
_mem8_f2_const		是	是
_memd8	是	是	是
_memd8_const	是	是	是
_mpy	是	是	是
_mpy2ir	是	是	是
_mpy2ll	是	是	是
_mpy32	是	是	是
_mpy32ll	是	是	是

表 8-4. 各器件对 C6000 C/C++ 内在函数的支持 (continued)

内在函数	C6400+	C6740	C6600
_mpy32su	是	是	是
_mpy32u	是	是	是
_mpy32us	是	是	是
_mpyh	是	是	是
_mpyhill	是	是	是
_mpyihll	是	是	是
_mpyilll	是	是	是
_mpyhir	是	是	是
_mpyihr	是	是	是
_mpyilr	是	是	是
_mpyhl	是	是	是
_mpyhlu	是	是	是
_mpyhslu	是	是	是
_mpyhsu	是	是	是
_myphu	是	是	是
_mpyhuls	是	是	是
_mpyhus	是	是	是
_mpyidll		是	是
_mpylh	是	是	是
_mpylhu	是	是	是
_mpylill	是	是	是
_mpylir	是	是	是
_mpylshu	是	是	是
_mpyluhs	是	是	是
_mpysp2dp		是	是
_mpyspdp		是	是
_mpysu	是	是	是
_mpysu4ll	是	是	是
_mpyus4ll	是	是	是
_mpyu	是	是	是
_mpyu2			是
_mpyu4ll	是	是	是
_mpyus	是	是	是
_mvd	是	是	是
_nassert	是	是	是
_norm	是	是	是
_pack2	是	是	是
_packh2	是	是	是
_packh4	是	是	是
_packhl2	是	是	是
_packl4	是	是	是
_packlh2	是	是	是
_qmpy32			是
_qmpysp			是
_qsmpy32r1			是

表 8-4. 各器件对 C6000 C/C++ 内在函数的支持 (continued)

内在函数	C6400+	C6740	C6600
_rcpdp		是	是
_rcpsp		是	是
_rsqrdp		是	是
_rsqrsp		是	是
_rotl	是	是	是
_rpack2	是	是	是
_sadd	是	是	是
_sadd2	是	是	是
_saddsub	是	是	是
_saddsub2	是	是	是
_saddu4	是	是	是
_saddus2	是	是	是
_saddsu2	是	是	是
_sat	是	是	是
_set	是	是	是
_setr	是	是	是
_shfl	是	是	是
_shfl3	是	是	是
_shl2			是
_shlmb	是	是	是
_shr2	是	是	是
_shrmb	是	是	是
_shru2	是	是	是
_smpy	是	是	是
_smpy2ll	是	是	是
_smpy32	是	是	是
_smpyh	是	是	是
_smpyhl	是	是	是
_smpylh	是	是	是
_spack2	是	是	是
_spacku4	是	是	是
_spint		是	是
_sshl	是	是	是
_sshvl	是	是	是
_sshvr	是	是	是
_ssub	是	是	是
_ssub2	是	是	是
_sub2	是	是	是
_sub4	是	是	是
_subabs4	是	是	是
_subc	是	是	是
_swap2	是	是	是
_swap4	是	是	是
_unpkbu4			是
_unpkh2			是

表 8-4. 各器件对 C6000 C/C++ 内在函数的支持 (continued)

内在函数	C6400+	C6740	C6600
_unpkhu2			是
_unpkhu4	是	是	是
_unpklu4	是	是	是
_xorll_c			是
_xormpy	是	是	是
_xpnd2	是	是	是
_xpnd4	是	是	是

表 8-5 中列出的内在函数能够在所有 C6000 器件上使用。它们对应于所示的 C6000 汇编语言指令。更多信息，请参阅 *TMS320C6000 CPU 和指令集参考指南*。

如需特定于 C6740 和 C6600 的内在函数列表，请参阅表 8-6。如需特定于 C6600 的内在函数列表，请参阅表 8-7。

以下表格中列出的某些项目实际上已作为指向内在函数的宏命令在 c6x.h 头文件中定义。此头文件位于编译器的“include”目录中。您的代码必须包含此头文件，才能使用所述的宏命令。

表 8-5. TMS320C6000 C/C++ 编译器内在函数

C/C++ 编译器内在函数	汇编指令	说明
int **_abs** (int *src*); __int40_t **_labs** (__int40_t *src*);	**ABS**	返回 src 的饱和绝对值
int **_abs2** (int *src*);	**ABS2**	计算每个 16 位值的绝对值
int **_add2** (int *src1* , int *src2*);	**ADD2**	将 src1 的上半部分和下半部分加到 src2 的上半部分和下半部分并返回结果。下半部分相加的任何溢出均不影响上半部分相加。
int **_add4** (int *src1* , int *src2*);	**ADD4**	对几对打包的 8 位数字执行二进制补码加法
long long **_addsub** (int *src1* , int *src2*);	**ADDSUB**	并行执行加法和减法。
long long **_addsub2** (int *src1* , int *src2*);	**ADDSUB2**	并行执行 ADD2 和 SUB2。
ushort & **_amem2** (void **ptr*);	**LDHU** **STH**	允许对齐加载 2 个字节并将其存储至存储器。指针必须与两字节边界对齐。(1)
const ushort & **_amem2_const** (const void **ptr*);	**LDHU**	允许从存储器对齐加载 2 个字节。指针必须与两字节边界对齐。(1)
unsigned & **_amem4** (void **ptr*);	**LDW** **STW**	允许对齐加载 4 个字节并将其存储至存储器。指针必须与四字节边界对齐。(1)
const unsigned & **_amem4_const** (const void **ptr*);	**LDW**	允许从存储器对齐加载 4 个字节。指针必须与四字节边界对齐。(1)
long long & **_amem8** (void **ptr*);	**LDDW** **STDW**	允许对齐加载 8 个字节并将其存储至存储器。指针必须与八字节边界对齐。将使用 LDDW 或 STDW 指令。
const long long & **_amem8_const** (const void **ptr*);	**LDW/LDW** **LDDW**	允许从存储器对齐加载 8 个字节。指针必须与八字节边界对齐。(2)
__float2_t & **_amem8_f2**(void * *ptr*);	**LDDW** **STDW**	允许对齐加载 8 个字节并将其存储至存储器。指针必须与八字节边界对齐。这被定义为一个宏命令。必须包含 c6x.h。(2) (1)
const __float2_t & **_amem8_f2_const**(void * *ptr*);	**LDDW**	允许从存储器对齐加载 8 个字节。指针必须与八字节边界对齐。这被定义为一个宏命令。必须包含 c6x.h。(2) (1)
double & **_amemd8** (void **ptr*);	**LDDW** **STDW**	允许对齐加载 8 个字节并将其存储至存储器。指针必须与八字节边界对齐。(1) (2) 将使用 LDDW 或 STDW 指令。
const double & **_amemd8_const** (const void **ptr*);	**LDW/LDW** **LDDW**	允许从存储器对齐加载 8 个字节。指针必须与八字节边界对齐。(1) (2)
int **_avg2** (int *src1* , int *src2*);	**AVG2**	计算每对有符号 16 位值的平均值
unsigned **_avgu4** (unsigned *src1* , unsigned *src2*);	**AVGU4**	计算每对无符号 8 位值的平均值

表 8-5. TMS320C6000 C/C++ 编译器内在函数 (continued)

C/C++ 编译器内在函数	汇编指令	说明
unsigned **_bitc4** (unsigned *src*);	**BITC4**	对于 src 中的每个 8 位数量，该数目的 1 位会写入返回值中的对应位置
unsigned **_bitr** (unsigned *src*);	**BITR**	反转位的顺序
unsigned **_clr** (unsigned *src2* , unsigned *csta*, unsigned *cstb*);	**CLR**	清除 src2 中的指定字段。要清除字段的起始位和结束位分别由 csta 和 cstb 指定。
unsigned **_clrr** (unsigned *src2* , int *src1*);	**CLR**	清除 src2 中的指定字段。要清除字段的起始位和结束位由 src1 的低 10 位指定。
int **_cmpeq2** (int *src1* , int *src2*);	**CMPEQ2**	对每对 16 位值执行等式比较。等式结果被打包至返回值的两个最低有效位。
int **_cmpeq4** (int *src1* , int *src2*);	**CMPEQ4**	对每对 8 位值执行等式比较。等式结果被打包至返回值的四个最低有效位。
int **_cmpgt2** (int *src1* , int *src2*);	**CMPGT2**	比较每对有符号 16 位值。结果被打包至返回值的两个最低有效位。
unsigned **_cmpgtu4** (unsigned *src1* , unsigned *src2*);	**CMPGTU4**	比较每对无符号 8 位值。结果被打包至返回值的四个最低有效位。
int **_cmplt2** (int *src1* , int *src2*);	**CMPLT2**	交换操作数并调用 _cmpgt2。这被定义为一个宏命令。必须包含 c6x.h。
unsigned **_cmpltu4** (unsigned *src1* , unsigned *src2*);	**CMPLTU4**	交换操作数并调用 _cmpgtu4。这被定义为一个宏命令。必须包含 c6x.h。
long long **_cmpy** (unsigned *src1* , unsigned *src2*); unsigned **_cmpyr** (unsigned *src1* , unsigned *src2*); unsigned **_cmpyr1** (unsigned *src1* , unsigned *src2*);	**CMPY** **CMPYR** **CMPYR1**	执行各种复数乘法运算。
long long **_ddotp4** (unsigned *src1* , unsigned *src2*);	**DDOTP4**	同时执行两个 DOTP2 运算。
long long **_ddotph2** (long long *src1* , unsigned *src2*); long long **_ddotpl2** (long long *src1* , unsigned *src2*); unsigned **_ddotph2r** (long long *src1* , unsigned *src2*); unsigned **_ddotpl2r** (long long *src1* , unsigned *src2*);	**DDOTPH2** **DDOTPL2** **DDOTPH2R** **DDOTPL2**	对两对有符号 16 位打包值执行各种双点积运算。
unsigned **_deal** (unsigned *src*);	**DEAL**	将 src 的偶数位和奇数位分别提取到两个 16 位值中。
long long **_dmv** (int *src1* , int *src2*);	**DMV**	将 src1 置于超长整型值的 32 个 MSB 中，将 src2 置于超长整型值的 32 个 LSB 中。另请参阅 _itoll()。
int **_dotp2** (int *src1* , int *src2*); __int40_t **_ldotp2** (int *src1* , int *src2*);	**DOTP2** **DOTP2**	src1 和 src2 的有符号低 16 位值之积加上 src1 和 src2 的有符号高 16 位值之积。在使用 _dotp2 的情况下，有符号结果会被写入单个 32 位寄存器。在使用 _ldotp2 的情况下，有符号结果会被写入一个 64 位寄存器对。
int **_dotpn2** (int *src1* , int *src2*);	**DOTPN2**	src1 和 src2 的有符号高 16 位值之积减去 src1 和 src2 的有符号低 16 位值之积。
int **_dotpnrsu2** (int *src1* , unsigned *src2*);	**DOTPNRSU2**	src1 和 src2 的高 16 位值之积减去 src1 和 src2 的低 16 位值之积。src1 中的值被视为有符号打包数量；src2 中的值被视为无符号打包数量。2^15 相加，结果为向右移 16 位的符号。
int **_dotpnrus2** (unsigned *src1* , int *src2*);	**DOTPNRUS2**	交换操作数并调用 _dotpnrsu2。这被定义为一个宏命令。必须包含 c6x.h。
int **_dotprsu2** (int *src1* , unsigned *src2*);	**DOTPRSU2**	src1 和 src2 的低 16 位值之积加上 src1 和 src2 的高 16 位值之积。src1 中的值被视为有符号打包数量；src2 中的值被被视为无符号打包数量。2^15 相加，结果为移动 16 位的符号。
int **_dotpsu4** (int *src1* , unsigned *src2*); int **_dotpus4** (unsigned *src1* , int *src2*); unsigned **_dotpu4** (unsigned *src1* , unsigned *src2*);	**DOTPSU4** **DOTPUS4** **DOTPU4**	对于 src1 和 src2 中的每对 8 位值，将 src1 中的 8 位值乘以 src2 中的 8 位值。将四个乘积相加。 _dotpus4 被定义为一个宏命令。必须包含 c6x.h。
long long **_dpack2** (unsigned *src1* , unsigned *src2*);	**DPACK2**	并行执行 PACK2 和 PACKH2 运算。
long long **_dpackx2** (unsigned *src1* , unsigned *src2*);	**DPACKX2**	并行执行 PACKLH2 和 PACKX2 运算。
__int40_t **_dtol** (double *src*);		将双精度寄存器对重新解释为一个 __int40_t (存储为寄存器对) 。
long long **_dtoll** (double *src*);		将双精度寄存器对 src 重新解释为一个超长整型寄存器对。

表 8-5. TMS320C6000 C/C++ 编译器内在函数 (continued)

C/C++ 编译器内在函数	汇编指令	说明
int **_ext** (int *src2* , unsigned *csta* , unsigned *cstb*);	**EXT**	提取 src2 中的指定字段，并加符号扩展至 32 位。提取操作通过先向左移位再有符号向右移位来执行；csta 和 cstb 分别为向左移位和向右移位的位数。
int **_extr** (int *src2* , int *src1*);	**EXT**	提取 src2 中的指定字段，并加符号扩展至 32 位。提取操作通过先向左移位再有符号向右移位来执行；向左移位和向右移位的位数由 src1 的低 10 位指定。
unsigned **_extu** (unsigned *src2* , unsigned *csta* , unsigned *cstb*);	**EXTU**	提取 src2 中的指定字段，并加零扩展至 32 位。提取操作通过先向左移位再无符号向右移位来执行；csta 和 cstb 分别为向左移位和向右移位的位数。
unsigned **_extur** (unsigned *src2* , int *src1*);	**EXTU**	提取 src2 中的指定字段，并加零扩展至 32 位。提取操作通过先向左移位再无符号向右移位来执行；向左移位和向右移位的位数由 src1 的低 10 位指定。
__float2_t **_fdmv_f2**(float *src1* , float *src2*);	**DMV**	将 src1 置于 __float2_t 的 32 个 LSB 中并将 src2 置于 __float2_t 的 32 个 MSB 中。另请参阅 _itoll()。这被定义为一个宏命令。必须包含 c6x.h。
unsigned **_ftoi** (float *src*);		将浮点型中的位重新解释为无符号值。例如： _ftoi (1.0) == 1065353216U
unsigned **_gmpy** (unsigned *src1* , unsigned *src2*);	**GMPY**	执行伽罗瓦域乘法。
int **_gmpy4** (int *src1* , int *src2*);	**GMPY4**	对 src1 中的四个值与 src2 中的四个并行值执行伽罗瓦域乘法。四个乘积被打包至返回值中。
unsigned **_hi** (double *src*);		返回双精度寄存器对的高位（奇数）寄存器
unsigned **_hill** (long long *src*);		返回超长整型寄存器对的高位（奇数）寄存器
double **_itod** (unsigned *src2* , unsigned *src1*);		通过重新解释两个无符号值来构建一个新的双精度寄存器对，其中 src2 为高位（奇数）寄存器，而 src1 为低位（偶数）寄存器
float **_itof** (unsigned *src*);		将无符号型中的位重新解释为浮点值。例如： _itof (0x3f800000) = 1.0
long long **_itoll** (unsigned *src2* , unsigned *src1*);		通过重新解释两个无符号值来构建一个新的超长整型寄存器对，其中 src2 为高位（奇数）寄存器，而 src1 为低位（偶数）寄存器
unsigned **_lmbd** (unsigned *src1* , unsigned *src2*);	**LMBD**	搜索 src2 的最左侧 1 或 0，具体由 src1 的 LSB 决定。返回最多到变更位的位数。
unsigned **_lo** (double *src*);		返回一个双精度寄存器对的低位（偶数）寄存器
unsigned **_loll** (long long *src*);		返回超长整型寄存器对的低位（偶数）寄存器
double **_ltod** (__int40_t *src*);		将一个 __int40_t 寄存器对 src 重新解释为一个双精度寄存器对。
double **_lltod** (long long *src*);		将超长整型寄存器对 src 重新解释为一个双精度寄存器对。
int **_max2** (int *src1* , int *src2*); int **_min2** (int *src1* , int *src2*); unsigned **_maxu4** (unsigned *src1* , unsigned *src2*); unsigned **_minu4** (unsigned *src1* , unsigned *src2*);	**MAX2** **MIN2** **MAXU4** **MINU4**	将每对值中的较大值/较小值置于返回值中的相应位置。值可以是 16 位有符号值或 8 位无符号值。
ushort & **_mem2** (void * *ptr*);	**LDB/LDB** **STB/STB**	允许未对齐加载 2 个字节并将其存储至存储器(1)
const ushort & **_mem2_const** (const void * *ptr*);	**LDB/LDB**	允许将 2 个字节未对齐加载到存储器(1)
unsigned & **_mem4** (void * *ptr*);	**LDNW** **STNW**	允许未对齐加载 4 个字节并将其存储至存储器(1)
const unsigned & **_mem4_const** (const void * *ptr*);	**LDNW**	允许从存储器未对齐加载 4 个字节(1)
long long & **_mem8** (void * *ptr*);	**LDNDW** **STNDW**	允许未对齐加载 8 个字节并将其存储至存储器(1)
const long long & **_mem8_const** (const void * *ptr*);	**LDNDW**	允许从存储器未对齐加载 8 个字节(1)
double & **_memd8** (void * *ptr*);	**LDNDW** **STNDW**	允许未对齐加载 8 个字节并将其存储至存储器(2) (1)
const double & **_memd8_const** (const void * *ptr*);	**LDNDW**	允许从存储器未对齐加载 8 个字节(2) (1)

表 8-5. TMS320C6000 C/C++ 编译器内在函数 (continued)

C/C++ 编译器内在函数	汇编指令	说明
int **_mpy** (int *src1* , int *src2*); int **_mpyus** (unsigned *src1* , int *src2*); int **_mpysu** (int *src1* , unsigned *src2*); unsigned **_mpyu** (unsigned *src1* , unsigned *src2*);	**MPY** **MPYUS** **MPYSU** **MPYU**	将 src1 的 16 个 LSB 乘以 src2 的 16 个 LSB 并返回结果。值可以为有符号值或无符号值。
long long **_mpy2ir** (int *src1* , int *src2*);	**MPY2IR**	执行两个 16 x 32 乘法。两个结果都向右移 15 位以得到舍入结果。
long long **_mpy2ll** (int *src1* , int *src2*);	**MPY2**	返回 src1 和 src2 中低 16 位值和高 16 位值之积
int **_mpy32** (int *src1* , int *src2*);	**MPY32**	返回 32 x 32 乘法的 32 个 LSB。
long long **_mpy32ll** (int *src1* , int *src2*); long long **_mpy32su** (int *src1* , int *src2*); long long **_mpy32us** (unsigned *src1* , int *src2*); long long **_mpy32u** (unsigned *src1* , unsigned *src2*);	**MPY32** **MPY32SU** **MPY32US** **MPY32U**	返回 32 x 32 乘法的全部 64 位。值可以为有符号值或无符号值。
int **_mpyh** (int *src1* , int *src2*); int **_mpyhus** (unsigned *src1* , int *src2*); int **_mpyhsu** (int *src1* , unsigned *src2*); unsigned **_mpyhu** (unsigned *src1* , unsigned *src2*);	**MPYH** **MPYHUS** **MPYHSU** **MPYHU**	将 src1 的 16 个 MSB 乘以 src2 的 16 个 MSB 并返回结果。值可以为有符号值或无符号值。
long long **_mpyhill** (int *src1* , int *src2*); long long **_mpylill** (int *src1* , int *src2*);	**MPYHI** **MPYLI**	生成有符号 16 x 32 乘法。结果将保存在返回类型的低 48 位中。可以使用 src1 的高或低 16 位。
int **_mpyhir** (int *src1* , int *src2*); int **_mpylir** (int *src1* , int *src2*);	**MPYHIR** **MPYLIR**	生成有符号 16 x 32 乘法。结果会向右移 15 位。可以使用 src1 的高或低 16 位。
int **_mpyhl** (int *src1* , int *src2*); int **_mpyhuls** (unsigned *src1* , int *src2*); int **_mpyhslu** (int *src1* , unsigned *src2*); unsigned **_mpyhlu** (unsigned *src1* , unsigned *src2*);	**MPYHL** **MPYHULS** **MPYHSLU** **MPYHLU**	将 src1 的 16 个 MSB 乘以 src2 的 16 个 LSB 并返回结果。值可以为有符号值或无符号值。
long long **_mpyihll** (int *src1* , int *src2*); long long **_mpyilll** (int *src1* , int *src2*);	**MPYIH** **MPYIL**	交换操作数并调用 _mpyhill。这被定义为一个宏命令。必须包含 c6x.h。 交换操作数并调用 _mpylill。这被定义为一个宏命令。必须包含 c6x.h。
int **_mpyihr** (int *src1* , int *src2*); int **_mpyilr** (int *src1* , int *src2*);	**MPYIHR** **MPYILR**	交换操作数并调用 _mpyhir。这被定义为一个宏命令。必须包含 c6x.h。 交换操作数并调用 _mpylir。这被定义为一个宏命令。必须包含 c6x.h。
int **_mpylh** (int *src1* , int *src2*); int **_mpyluhs** (unsigned *src1* , int *src2*); int **_mpylshu** (int *src1* , unsigned *src2*); unsigned **_mpylhu** (unsigned *src1* , unsigned *src2*);	**MPYLH** **MPYLUHS** **MPYLSHU** **MPYLHU**	将 src1 的 16 个 LSB 乘以 src2 的 16 个 MSB 并返回结果。值可以为有符号值或无符号值。
long long **_mpysu4ll** (int *src1* , unsigned *src2*); long long **_mpyus4ll** (unsigned *src1* , int *src2*); long long **_mpyu4ll** (unsigned *src1* , unsigned *src2*);	**MPYSU4** **MPYUS4** **MPYU4**	对于 src1 和 src2 中的每个 8 位数值，执行 8 位 x 8 位乘法。四个 16 位结果被打包至一个 64 位结果。结果可以是有符号值或无符号值。 _mpyus4ll 被定义为一个宏命令。必须包含 c6x.h。
int **_mvd** (int *src2*);	**MVD**	使用乘法器流水线在四个周期内将数据从 src2 移到返回值
void **_nassert** (int *src*);		不生成任何代码。告诉优化器，通过 assert 函数声明的表达式为 true；这可以提示优化器什么样的优化可能有效。
unsigned **_norm** (int *src*); unsigned **_lnorm** (__int40_t *src*);	**NORM**	返回最多到 src 第一个非冗余符号位的位数。
unsigned **_pack2** (unsigned *src1* , unsigned *src2*); unsigned **_packh2** (unsigned *src1* , unsigned *src2*);	**PACK2** **PACKH2**	src1 和 src2 的下半字/上半字将保存在返回值中。
unsigned **_packh4** (unsigned *src1* , unsigned *src2*); unsigned **_packl4** (unsigned *src1* , unsigned *src2*);	**PACKH4** **PACKL4**	将备用字节打包至返回值。可以打包高位或低位字节。
unsigned **_packhl2** (unsigned *src1* , unsigned *src2*); unsigned **_packlh2** (unsigned *src1* , unsigned *src2*);	**PACKHL2** **PACKLH2**	src1 的上半字/下半字将保存在返回值的上半字中。src2 的下半字/上半字将保存在返回值的下半字中。
unsigned **_rotl** (unsigned *src1* , unsigned *src2*);	**ROTL**	将 src1 向左旋转 src2 中指定的量

表 8-5. TMS320C6000 C/C++ 编译器内在函数 (continued)

C/C++ 编译器内在函数	汇编指令	说明
int **_rpack2** (int *src1* , int *src2*);	**RPACK2**	将 src1 和 src2 向左移 1 位并进行饱和处理。移位后 src1 的 16 个 MSB 将保存在 32 位输出的 16 个 MSB 中。移位后 src2 的 16 个 MSB 将保存在 32 位输出的 16 个 LSB 中。
int **_sadd** (int *src1* , int *src2*); __int40_t **_lsadd** (int *src1* , __int40_t *src2*);	**SADD**	将 src1 和 src2 相加并对结果进行饱和处理。返回结果。
int **_sadd2** (int *src1* , int *src2*); int **_saddus2** (unsigned *src1* , int *src2*); int **_saddsu2** (int *src1* , unsigned *src2*);	**SADD2** **SADDUS2** **SADDSU2**	在 src1 和 src2 中的 16 位值对之间执行饱和加法。src1 的值可以是有符号值或无符号值。 _saddsu2 被定义为一个宏命令。必须包含 c6x.h。
long long **_saddsub** (unsigned *src1* , unsigned *src2*);	**SADDSUB**	并行执行饱和加法与饱和减法。
long long **_saddsub2** (unsigned *src1* , unsigned *src2*);	**SADDSUB2**	并行执行 SADD2 和 SSUB2。
unsigned **_saddu4** (unsigned *src1* , unsigned *src2*);	**SADDU4**	在 src1 和 src2 中的 8 位无符号值对之间执行饱和加法。
int **_sat** (__int40_t *src2*);	**SAT**	将 40 位长整型值转换为 32 位有符号整型值并在必要时使其饱和。
unsigned **_set** (unsigned *src2* , unsigned *csta* , unsigned *cstb*);	**SET**	将 src2 中的指定字段全部设置位 1 并返回 src2 值。要设置字段的起始位和结束位分别由 csta 和 cstb 指定。
unsigned **_setr** (unit *src2* , int *src1*);	**SET**	将 src2 中的指定字段全部设置位 1 并返回 src2 值。要设置字段的起始位和结束位由 src1 的低 10 位指定。
unsigned **_shfl** (unsigned *src2*);	**SHFL**	src2 的低 16 位将保存在偶数位处，而 src 的高 16 位将保存在奇数位处。
long long **_shfl3** (unsigned *src1* , unsigned *src2*);	**SHFL3**	从 src1 取出两个 16 位值并从 src2 取出 16 个 LSB 来执行 3 路交错，从而得到一个 48 位结果。
unsigned **_shlmb** (unsigned *src1* , unsigned *src2*); unsigned **_shrmb** (unsigned *src1* , unsigned *src2*);	**SHLMB** **SHRMB**	将 src2 向左/向右移动一个字节，并且 src1 的最高/最低有效字节会合并到最低/最高有效字节位置。
int **_shr2** (int *src1* , unsigned *src2*); unsigned **_shru2** (unsigned *src1* , unsigned *src2*);	**SHR2** **SHRU2**	对于 src1 中的每个 16 位数，该数均会以算术或逻辑方法向右移动 src2 位数。src1 可以包含有符号值或无符号值。
int **_smpy** (int *src1* , int *src2*); int **_smpyh** (int *src1* , int *src2*); int **_smpyhl** (int *src1* , int *src2*); int **_smpylh** (int *src1* , int *src2*);	**SMPY** **SMPYH** **SMPYHL** **SMPYLH**	将 src1 乘以 src2，再将结果向左移 1 位并返回结果。如果结果为 0x80000000，则将结果饱和处理为 0x7FFFFFFF
long long **_smpy2ll** (int *src1* , int *src2*);	**SMPY2**	对几对有符号打包 16 位值执行 16 位乘法，再向左多移动 1 位，并饱和处理为 64 位结果。
int **_smpy32** (int *src1* , int *src2*);	**SMPY32**	返回 32 x 32 乘法向左移 1 位后的 32 个 MSB。
int **_spack2** (int *src1* , int *src2*);	**SPACK2**	将两个有符号 32 位值饱和处理为 16 位值并打包到返回值
unsigned **_spacku4** (int *src1* , int *src2*);	**SPACKU4**	将四个有符号 16 位值饱和处理为 8 位值并打包到返回值
int **_sshl** (int *src2* , unsigned *src1*);	**SSHL**	将 src2 向左移动 src1 中所含内容对应的位数，将结果饱和处理为 32 位，然后返回结果
int **_sshvl** (int *src2* , int *src1*); int **_sshvr** (int *src2* , int *src1*);	**SSHVL** **SSHVR**	将 src2 向左/向右移动 src1 位。如果移动后的值大于 MAX_INT 或小于 MIN_INT，则对结果进行饱和处理。
int **_ssub** (int *src1* , int *src2*); __int40_t **_lssub** (int *src1* , __int40_t *src2*);	**SSUB**	从 src1 中减去 src2，对结果进行饱和处理，然后返回结果。
int **_ssub2** (int *src1* , int *src2*);	**SSUB2**	从 src1 的上半部分和下半部分中减去 src2 的上半部分和下半部分并对每个结果进行饱和处理。
int **_sub4** (int *src1* , int *src2*);	**SUB4**	对几对打包的 8 位值执行二进制补码减法
int **_subabs4** (int *src1* , int *src2*);	**SUBABS4**	计算每对打包无符号 8 位值之差的绝对值
unsigned **_subc** (unsigned *src1* , unsigned *src2*);	**SUBC**	条件减法除法步骤
int **_sub2** (int *src1* , int *src2*);	**SUB2**	从 src1 的上半部分和下半部分中减去 src2 的上半部分和下半部分并返回结果。下半部分减法中的借位不影响上半部分减法。
unsigned **_swap4** (unsigned *src*);	**SWAP4**	交换每个 16 位值内的字节对（字节序交换）。
unsigned **_swap2** (unsigned *src*);	**SWAP2**	调用 _packlh2。这被定义为一个宏命令。必须包含 c6x.h。

表 8-5. TMS320C6000 C/C++ 编译器内在函数 (continued)

C/C++ 编译器内在函数	汇编指令	说明
unsigned **_unpkhu4** (unsigned *src*);	**UNPKHU4**	将两个高位无符号 8 位值解压到无符号打包 16 位值
unsigned **_unpklu4** (unsigned *src*);	**UNPKLU4**	将两个低位无符号 8 位值解压到无符号打包 16 位值
unsigned **_xormpy** (unsigned *src1* , unsigned *src2*);	**XORMPY**	执行伽罗瓦域乘法
unsigned **_xpnd2** (unsigned *src*);	**XPND2**	src 的位 1 和位 0 会分别复制到结果的上半字和下半字。
unsigned **_xpnd4** (unsigned *src*);	**XPND4**	src 的位 3 和位 0 会复制到结果的字节 3 至 0。

(1) 更多信息，请参阅 *TMS320C6000 编程指南*。
(2) 有关操控 8 字节数据量的详细信息，请参阅节 8.6.10。

表 8-6 中列出的内在函数适用于 C6740 和 C6600 器件，但不适用于 C6400+ 器件。所列内在函数对应于所示的 C6000 汇编语言指令。更多信息，请参阅 *TMS320C6000 CPU 和指令集参考指南*。

如需通用 C6000 内在函数列表，请参阅表 8-5。如需特定于 C6600 的内在函数列表，请参阅表 8-7。

表 8-6. TMS320C6740 和 C6600 C/C++ 编译器内在函数

C/C++ 编译器内在函数	汇编指令	说明
int **_dpint** (double *src*);	**DPINT**	使用由 CSR 寄存器设置的舍入模式，将 64 位双精度值转换为 32 位有符号整型值。
__int40_t **_f2tol**(__float2_t *src*);		将一个 __float2_t 寄存器对 src 重新解释为一个 __int40_t (存储为寄存器对) 。这被定义为一个宏命令。必须包含 c6x.h。
__float2_t **_f2toll**(__float2_t *src*);		将一个 __float2_t 寄存器对重新解释为一个超长整型寄存器对。这被定义为一个宏命令。必须包含 c6x.h。
double **_fabs** (double *src*); float **_fabsf** (float *src*);	**ABSDP** **ABSSP**	返回 src 的绝对值。
__float2_t **_lltof2**(long long *src*);		将一个超长整型寄存器对重新解释为一个 __float2_t 寄存器对。这被定义为一个宏命令。必须包含 c6x.h。
__float2_t **_ltof2**(__int40_t *src*);		将一个 __int40_t 寄存器对重新解释为一个 __float2_t 寄存器对。这被定义为一个宏命令。必须包含 c6x.h。
__float2_t & **_mem8_f2**(void * *ptr*);	**LDNDW** **STNDW**	允许未对齐加载 8 个字节并将其存储至存储器。(1)这被定义为一个宏命令。必须包含 c6x.h。
const __float2_t & **_mem8_f2_const**(void * *ptr*);	**LDNDW** **STNDW**	允许从存储器未对齐加载 8 个字节。(1)这被定义为一个宏命令。必须包含 c6x.h。
long long **_mpyidll** (int *src1* , int *src2*);	**MPYID**	生成有符号整数乘法。结果被保存在一个寄存器对中。
double**_mpysp2dp** (float *src1* , float *src2*);	**MPYSP2DP**	生成双精度浮点乘法。结果被保存在一个寄存器对中。
double**_mpyspdp** (float *src1* , double *src2*);	**MPYSPDP**	生成双精度浮点乘法。结果被保存在一个寄存器对中。
double **_rcpdp** (double *src*);	**RCPDP**	计算近似的 64 位双精度倒数。
float **_rcpsp** (float *src*);	**RCPSP**	计算近似的 32 位浮点倒数。
double **_rsqrdp** (double *src*);	**RSQRDP**	计算近似的 64 位双精度平方根倒数。
float **_rsqrsp** (float *src*);	**RSQRSP**	计算近似的 32 位浮点平方根倒数。
int **_spint** (float *src*);	**SPINT**	使用由 CSR 寄存器设置的舍入模式，将 32 位浮点值转换为 32 位有符号整型值。

(1) 有关操控 8 字节数据量的详细信息，请参阅节 8.6.10。

仅 C6600 器件支持表 8-7 中列出的内在函数。这些内在函数是对表 8-5 和表 8-6 所列内在函数的补充。所列内在函数对应于所示的汇编语言指令。更多信息，请参阅 *TMS320C6000 CPU 和指令集参考指南*。

表 8-7. TMS320C6600 C/C++ 编译器内在函数

C/C++ 编译器内在函数	汇编指令	说明
	ADDDP	不存在内在函数。使用 C 语言原生函数 a + b，其中 a 和 b 为双精度值。
	ADDSP	不存在内在函数。使用 C 语言原生函数 a + b，其中 a 和 b 为浮点值。
	AND	不存在内在函数：使用 C 语言原生函数"a & b"，其中 a 和 b 为超长整型值。
	ANDN	不存在内在函数：使用 C 语言原生函数"a & ~b"，其中 a 和 b 为超长整型值。
	FMPYDP	不存在内在函数。使用 C 语言原生函数 a * b，其中 a 和 b 为双精度值。
	OR	不存在内在函数：使用 C 语言原生函数"a | b"，其中 a 和 b 为超长整型值。
	SUBDP	不存在内在函数。使用 C 语言原生函数 a - b，其中 a 和 b 为双精度值。
	SUBSP	不存在内在函数。使用 C 语言原生函数 a - b，其中 a 和 b 为浮点值。
	XOR	不存在内在函数：使用 C 语言原生函数"a ^ b"，其中 a 和 b 为超长整型值。另请参阅 _xorll_c()。
__x128_t **_ccmatmpy** (long long *src1* , __x128_t *src2*);	**CCMATMPY**	将共轭的 1x2 复数向量乘以 2x2 复数矩阵，生成两个 64 位结果。有关 __x128_t 容器类型的详细信息，请参阅节 8.6.7。
long long **_ccmatmpyr1** (long long *src1* , __x128_t *src2*);	**CCMATMPYR1**	将复共轭的 1x2 复数向量乘以 2x2 复数矩阵，生成两个 32 位复数结果。
long long **_ccmpy32r1** (long long *src1* , long long *src2*);	**CCMPY32R1**	32 位共轭复数乘以 Q31 数并进行舍入处理。
__x128_t **_cmatmpy** (long long *src1* , __x128_t *src2*);	**CMATMPY**	将 1x2 向量乘以 2x2 复数矩阵，生成两个 64 位复数结果。
long long **_cmatmpyr1** (long long *src1* , __x128_t *src2*);	**CMATMPYR1**	将 1x2 复数向量乘以 2x2 复数矩阵，生成两个 32 位复数结果。
long long **_cmpy32r1** (long long *src1* , long long *src2*);	**CMPY32R1**	32 位复数乘以 Q31 数字并进行舍入处理。
__x128_t **_cmpysp** (__float2_t *src1* , __float2_t *src2*);	**CMPYSP**	对两个复数的复数乘法执行乘法运算（另请参阅 _complex_mpysp 和 _complex_conjugate_mpysp。）
double **_complex_conjugate_mpysp** (double *src1* , double *src2*);	**CMPYSP DSUBSP**	通过执行 CMPYSP 和 DSUBSP 来执行复数共轭乘法。
double **_complex_mpysp** (double *src1* , double *src2*);	**CMPYSP DADDSP**	通过执行 CMPYSP 和 DADDSP 来执行复数乘法。
int **_crot90** (int *src*);	**CROT90**	将复数旋转 90 度。
int **_crot270** (int *src*);	**CROT270**	将复数旋转 270 度。
long long **_dadd** (long long *src1* , long long *src2*);	**DADD**	对有符号 32 位值进行两路 SIMD 加法，生成两个有符号 32 位结果。
long long **_dadd2** (long long *src1* , long long *src2*);	**DADD2**	对打包的有符号 16 位值进行四路 SIMD 加法，生成四个有符号 16 位结果。（两路 _add2）
__float2_t **_daddsp** (__float2_t *src1* , __float2_t *src2*);	**DADDSP**	对 32 位单精度数字进行两路 SIMD 加法。
long long **_dadd_c** (scst5 immediate *src1* , long long *src2*);	**DADD**	将两个有符号 32 位值与 src2 中的一个常数（-16 至 15）相加，生成两个有符号 32 位结果。
long long **_dapys2** (long long *src1* , long long *src2*);	**DAPYS2**	使用 src1 的符号位确定是将 src2 中的四个 16 位值乘以 1 还是 -1。产生四个有符号 16 位结果。（如果 src1 和 src2 是同一寄存器对，则相当于两路 _abs2。）
long long **_davg2** (long long *src1* , long long *src2*);	**DAVG2**	对有符号 16 位值求四路 SIMD 平均值并进行舍入处理。（两路 _avg2）
long long **_davgnr2** (long long *src1* , long long *src2*);	**DAVGNR2**	对有符号 16 位值求四路 SIMD 平均值但不进行舍入处理。
long long **_davgnru4** (long long *src1* , long long *src2*);	**DAVGNRU4**	对无符号 8 位值求八路 SIMD 平均值但不进行舍入处理。

表 8-7. TMS320C6600 C/C++ 编译器内在函数 (continued)

C/C++ 编译器内在函数	汇编指令	说明
long long **_davgu4** (long long *src1* , long long *src2*);	**DAVGU4**	对无符号 8 位值求八路 SIMD 平均值并进行舍入处理。（两路 _avgu4 ）
long long **_dccmpyr1** (long long *src1* , long long *src2*);	**DCCMPYR1**	对 src2 的共轭复数进行两路 SIMD 复数乘法并进行舍入处理 (_cmpyr1)。
unsigned **_dcmpeq2** (long long *src1* , long long *src2*);	**DCMPEQ2**	对有符号 16 位值进行四路 SIMD 比较。结果被打包至返回值的四个最低有效位。（两路 _cmpeq2 ）
unsigned **_dcmpeq4** (long long *src1* , long long *src2*);	**DCMPEQ4**	对无符号 8 位值进行八路 SIMD 比较。结果被打包至返回值的八个最低有效位。（两路 _cmpeq4 ）
unsigned **_dcmpgt2** (long long *src1* , long long *src2*);	**DCMPGT2**	对有符号 16 位值进行四路 SIMD 比较。结果被打包至返回值的四个最低有效位。（两路 _cmpgt2 ）
unsigned **_dcmpgtu4** (long long *src1* , long long *src2*);	**DCMPGTU4**	对无符号 8 位值进行八路 SIMD 比较。结果被打包至返回值的八个最低有效位。（两路 _cmpgtu4 ）
__x128_t **_dccmpy** (long long *src1* , long long *src2*);	**DCCMPY**	对两组打包的复数与 src2 的共轭复数执行两个复数乘法运算。
__x128_t **_dcmpy** (long long *src1* , long long *src2*);	**DCMPY**	对两组打包的复数执行两个复数乘法运算。（两路 SIMD _cmpy ）
long long **_dcmpyr1** (long long *src1* , long long *src2*);	**DCMPYR1**	两路 SIMD 复数乘法并进行舍入处理 (_cmpyr1)。
long long **_dcrot90** (long long *src*);	**DCROT90**	_crot90 的两路 SIMD 版本。
long long **_dcrot270** (long long *src*);	**DCROT270**	_crot270 的两路 SIMD 版本。
long long **_ddotp4h** (__x128_t *src1* , __x128_t *src2*);	**DDOTP4H**	对四组打包的 16 位值执行两个点积运算。（两路 _dotp4h ）
long long **_ddotpsu4h** (__x128_t *src1* , __x128_t *src2*);	**DDOTPSU4H**	对四组打包的 16 位值执行两个点积运算。（两路 _dotpsu4h ）
__float2_t **_dinthsp** (int *src*);	**DINTHSP**	将两个打包的有符号 16 位值转换为两个单精度浮点值。
__float2_t **_dinthspu** (unsigned *src*);	**DINTHSPU**	将两个打包的无符号 16 位值转换为两个单精度浮点值。
__float2_t **_dintsp**(long long *src*);	**DINTSP**	将两个 32 位有符号整数转换为两个单精度浮点值。
__float2_t **_dintspu**(long long *src*);	**DINTSPU**	将两个 32 位无符号整数转换为两个单精度浮点值。
long long **_dmax2** (long long *src1* , long long *src2*);	**DMAX2**	对 16 位有符号值进行四路 SIMD 取最大值，生成四个有符号 16 位结果。（两路 _max2 ）
long long **_dmaxu4** (long long *src1* , long long *src2*);	**DMAXU4**	对无符号 8 位值进行 8 路 SIMD 取最大值，生成八个无符号 8 位结果。（两路 _maxu4 ）
long long **_dmin2** (long long *src1* , long long *src2*);	**DMIN2**	对 16 位有符号值进行四路 SIMD 取最小值，生成四个有符号 16 位结果。（两路 _min2 ）
long long **_dminu4** (long long *src1* , long long *src2*);	**DMINU4**	对无符号 8 位值进行 8 路 SIMD 取最小值，生成八个无符号 8 位结果。（两路 _minu4 ）
__x128_t **_dmpy2** (long long *src1* , long long *src2*);	**DMPY2**	对 16 位有符号值进行四路 SIMD 乘法，生成四个有符号 32 位结果。（两路 _mpy2 ）
__float2_t **_dmpysp** (__float2_t *src1* , __float2_t *src2*);	**DMPYSP**	两路单精度浮点乘法，生成两个单精度结果。
__x128_t **_dmpysu4** (long long *src1* , long long *src2*);	**DMPYSU4**	对 8 位有符号值与 8 位无符号值执行八路 SIMD 乘法，生成八个有符号 16 位结果。（两路 _mpysu4 ）
__x128_t **_dmpyu2** (long long *src1* , long long *src2*);	**DMPYU2**	对 16 位无符号值进行四路 SIMD 乘法，生成四个无符号 32 位结果。（两路 _mpyu2 ）
__x128_t **_dmpyu4** (long long *src1* , long long *src2*);	**DMPYU4**	对 8 位有符号值进行八路 SIMD 乘法，生成八个有符号 16 位结果。（两路 _mpyu4 ）
long long **_dmvd** (int *src1* , int *src2*);	**DMVD**	将 src1 置于超长整型低位寄存器中，将 src2 置于超长整型高位寄存器中。执行四个周期。另请参阅 _dmv()、_fdmv_f2 和 _itoll()。
int **_dotp4h** (long long *src1* , long long *src2*);	**DOTP4H**	将两组四个有符号 16 位值相乘并返回 32 位和。
long long **_dotp4hll** (long long *src1* , long long *src2*);	**DOTP4H**	将两组四个有符号 16 位值相乘并返回 64 位和。

表 8-7. TMS320C6600 C/C++ 编译器内在函数 (continued)

C/C++ 编译器内在函数	汇编指令	说明
int **_dotpsu4h** (long long *src1* , long long *src2*);	**DOTPSU4H**	将四个有符号 16 位值与四个无符号 16 值相乘并返回 32 位和。
long long **_dotpsu4hll** (long long *src1* , long long *src2*);	**DOTPSU4H**	将四个有符号 16 位值与四个无符号 16 值相乘并返回 64 位和。
long long **_dpackh2** (long long *src1* , long long *src2*);	**DPACKH2**	两路 _packh2。
long long **_dpackh4** (long long *src1* , long long *src2*);	**DPACKH4**	两路 _packh4。
long long **_dpacklh2** (long long *src1* , long long *src2*);	**DPACKLH2**	两路 _packlh2。
long long **_dpacklh4** (unsigned *src1* , unsigned *src2*);	**DPACKLH4**	执行 _packl4 和 _packh4。_packl4 的输出保存在结果的低位寄存器中，而 _packh4 的输出保存在结果的高位寄存器中。
long long **_dpackl2** (long long *src1* , long long *src2*);	**DPACKL2**	两路 _packl2。
long long **_dpackl4** (long long *src1* , long long *src2*);	**DPACKL4**	两路 _packl4。
long long **_dsadd** (long long *src1* , long long *src2*);	**DSADD**	对有符号 32 位值进行两路 SIMD 饱和加法，生成两个有符号 32 位结果。（两路 _sadd ）
long long **_dsadd2** (long long *src1* , long long *src2*);	**DSADD2**	对有符号 16 位值进行四路 SIMD 饱和加法，生成四个有符号 16 位结果。（两路 _sadd2 ）
long long **_dshl** (long long *src1* , unsigned *src2*);	**DSHL**	使两个有符号 32 位值向左移位，位数等于 src2 参数中的单个值。
long long **_dshl2** (long long *src1* , unsigned *src2*);	**DSHL2**	使四个有符号 16 位值向左移位，位数等于 src2 参数中的单个值。（两路 _shl2 ）
long long **_dshr** (long long *src1* , unsigned *src2*);	**DSHR**	使两个有符号 32 位值向右移位，位数等于 src2 参数中的单个值。
long long **_dshr2** (long long *src1* , unsigned *src2*);	**DSHR2**	使四个有符号 16 位值向右移位，位数等于 src2 参数中的单个值。（两路 _shr2 ）
long long **_dshru** (long long *src1* , unsigned *src2*);	**DSHRU**	使两个无符号 32 位值向右移位，位数等于 src2 参数中的单个值。
long long **_dshru2** (long long *src1* , unsigned *src2*);	**DSHRU2**	使四个无符号 16 位值向右移位，位数等于 src2 参数中的单个值。（两路 _shru2 ）
__x128_t **_dsmpy2** (long long *src1* , long long *src2*);	**DSMPY2**	对有符号 16 位值执行四路 SIMD 乘法，再向左移 1 位并进行饱和处理，生成四个有符号 32 位结果。（两路 _smpy2 ）
long long **_dspacku4** (long long *src1* , long long *src2*);	**DSPACKU4**	两路 _spacku4。
long long **_dspint** (__float2_t *src*);	**DSPINT**	将两个打包的单精度浮点值转换为两个有符号 32 位值。
unsigned **_dspinth** (__float2_t *src*);	**DSPINTH**	将两个打包的单精度浮点值转换为两个打包的有符号 16 位值。
long long **_dssub** (long long *src1* , long long *src2*);	**DSSUB**	对 32 位有符号值进行两路 SIMD 饱和减法，生成两个有符号 32 位结果。
long long **_dssub2** (long long *src1* , long long *src2*);	**DSSUB2**	对有符号 16 位值进行四路 SIMD 饱和减法，生成四个有符号 16 位结果。（两路 _ssub2 ）
long long **_dsub** (long long *src1* , long long *src2*);	**DSUB**	对 32 位有符号值进行两路 SIMD 减法，生成两个有符号 32 位结果。
long long **_dsub2** (long long *src1* , long long *src2*);	**DSUB2**	对有符号 16 位值进行四路 SIMD 减法，生成四个有符号 16 位结果。（两路 _sub2 ）
__float2_t **_dsubsp** (__float2_t *src1* , __float2_t *src2*);	**DSUBSP**	对 32 位单精度数进行两路 SIMD 减法。
long long **_dxpnd2** (unsigned *src*);	**DXPND2**	将四个低位展开到四个 16 位字段。
long long **_dxpnd4** (unsigned *src*);	**DXPND4**	将八个低位展开到八个 8 位字段。
__float2_t **_fdmvd_f2**(float *src1* , float *src2*);	**DMVD**	将 src1 置于 __float2_t 的低位寄存器中并将 src2 置于 __float2_t 的高位寄存器中。执行四个周期。另请参阅 _dmv()、_dmvd() 和 _itoll()。这被定义为一个宏命令。必须包含 c6x.h。
int **_land** (int *src1* , int *src2*);	**LAND**	对 src1 和 src2 进行逻辑与运算。

表 8-7. TMS320C6600 C/C++ 编译器内在函数 (continued)

C/C++ 编译器内在函数	汇编指令	说明
int **_landn (**int *src1* , int *src2* **);**	**LANDN**	对 src1 进行逻辑与运算并对 src2 进行逻辑非运算；例如，src1 AND ~src2。
int **_lor (**int *src1* , int *src2* **);**	**LOR**	对 src1 和 src2 进行逻辑或运算。
void **_mfence();**	**MFENCE**	在存储器系统繁忙期间使 CPU 停顿。
long long **_mpyu2 (**unsigned *src1* , unsigned *src2* **);**	**MPYU2**	对 16 位无符号值进行两路 SIMD 乘法，生成两个无符号 32 位结果。
__x128_t **_qmpy32 (**__x128_t *src1* , __x128_t *src2* **);**	**QMPY32**	对 32 位有符号值进行四路 SIMD 乘法，生成四个 32 位结果。（四路 _mpy32 ）
__x128_t **_qmpysp (**__x128_t *src1* , __x128_t *src2* **);**	**QMPYSP**	进行四路 SIMD 32 位单精度乘法，生成四个 32 位单精度结果。
__x128_t **_qsmpy32r1 (**__x128_t *src1* , __x128_t *src2* **);**	**QSMPY32R1**	进行 4 路 SIMD 小数 32 位 x 32 位乘法，其中每个结果值均会向右移 31 位并进行舍入处理。在 Q31 小数体系中，这会将结果归一化到 -1 和 1 之间。
unsigned **_shl2 (**unsigned *src1* , unsigned *src2* **);**	**SHL2**	将两个有符号 16 位值向左移位，位数等于 src2 参数中的单个值。
long long **_unpkbu4 (**unsigned *src* **);**	**UNPKBU4**	将四个 8 位无符号值解压到四个 16 位无符号值。（另请参阅 _unpklu4 和 _unpkhu4 ）
long long **_unpkh2 (**unsigned *src* **);**	**UNPKH2**	将两个有符号 16 位值解压到两个有符号 32 位值。
long long **_unpkhu2 (**unsigned *src* **);**	**UNPKHU2**	将两个无符号 16 位值解压到两个无符号 32 位值。
long long **_xorll_c (**scst5 immediate *src1* , long long *src2* **);**	**XOR**	对 src1 与 src2 的高 32 位和低 32 位部分执行异或运算（对常数执行 SIMD 异或运算 ）。

8.6.7 __x128_t 容器类型

__x128_t 容器类型仅在为 C6600 编译时可用，它存储 128 位数据，在 C6600 上执行某些 SIMD 运算时必须使用它。另外，请注意前导的双下划线。使用 __x128_t 容器类型时，必须包含 c6x.h。

此类型可用于定义可与某些 C6600 内在函数一同使用的对象。（请参阅表 8-7。）对象可以使用各种内在函数进行填充和操作。此类型并非成熟的内置类型（如 long-long），因此不允许进行各种本机 C 运算。将此类型视为具有私有成员和特殊操作函数的结构。

当编译器将一个 __x128_t 对象放入寄存器文件中时，__x128_t 对象接受四个寄存器（寄存器 *quad*）。类型为 __x128_t 的对象与存储器中的 64 位边界对齐。

当 __x128_t 对象在栈上传递时，它被放置在相对于栈开始的 64 位边界上。（默认情况下，栈本身与 64 位边界对齐。）相关详细信息，请参阅节 8.6.2 中的注释。

支持以下操作：

- 声明一个 __x128_t 全局对象（例如：`__x128_t a;` ）。默认情况下，它将放在 .far 段中。
- 声明一个 __x128_t 局部对象（例如：`__x128_t a;` ）。它将被放在栈上。
- 声明一个 __x128_t 全局/局部指针（例如：`__x128_t *a;` ）。
- 声明一个 __x128_t 对象数组（例如：`__x128_t a[10];` ）。
- 将 __x128_t 类型声明为结构、类或联合的成员。
- 将一个 __x128_t 对象分配给另一个 __x128_t 对象。
- 将 __x128_t 对象传递给函数（包括可变参数函数）。（按值传递。）
- 从函数返回一个 __x128_t 对象。
- 使用 128 位操作内在函数设置和提取内容（请参阅表 8-8）。

不支持以下运算：

- 对 __x128_t 对象的本机类型运算，例如 +、-、* 等等。
- 将对象强制转换为 __x128_t 类型。
- 使用数组或结构表示法访问__x128_t 的元素。
- 将一个 __x128_t 对象传递给像 printf 这样的 I/O 函数。相反，使用适当的内在函数从 __x128_t 对象中提取值。

示例 8-7. __x128_t 容器类型

```
#include <c6x.h>
#include <stdio.h>
__x128_t mpy_four_way_example(__x128_t s, int a, int b, int c, int d)
{
  __x128_t t = _ito128(a, b, c, d);       // Pack values into a __x128_t
  __x128_t results = _qmpy32(s, t);       // Perform a four-way SIMD multiply

  int lowest32  = _get32_128(results, 0); // Extract lowest reg of __x128_t
  int highest32 = _get32_128(results, 3); // Extract highest reg of __x128_t
  printf("lowest  = %d\n", lowest32);
  printf("highest = %d\n", highest32);

  return results;
}
```

备注

在类型 __x128_t 或 __float2_t 中包含 c6x.h

使用 __x128_t 容器类型或 __float2_t typedef 或任何涉及 __float2_t 的内在函数时，必须包含 c6x.h。

表 8-8. Vector-in-Scalar 支持 C/C++ 编译器 v7.2 内在函数

C/C++ 编译器内在函数	说明
创建	
__x128_t **_ito128 (unsigned *src1*,unsigned** *src2* , unsigned *src3* , unsigned *src4* **);**	根据 (u)int (reg+3, reg+2, reg+1, reg+0) 创建 __x128_t
__x128_t **_fto128 (float** *src1* , float *src2* , float *src3* , float *src4* **);**	根据 float (reg+3, reg+2, reg+1, reg+0) 创建 __x128_t
__x128_t **_llto128 (**long long *src1* , long long *src2* **);**	根据两个 long long 创建 __x128_t
__x128_t **_dto128 (**double *src1* , double *src2* **);**	根据两个 double 创建 __x128_t
__x128_t **_f2to128(**__float2_t *src1* , __float2_t *src2* **);**	根据两个 __float2_t 对象创建 __x128_t。这被定义为一个宏命令。必须包含 c6x.h。
__x128_t **_dup32_128 (**int *src* **);**	通过复制 src1 创建 __x128_t
__float2_t **_ftof2(**float *src1* , float *src2* **);**	根据两个 float 创建 __float2_t。这被定义为一个宏命令。必须包含 c6x.h。
提取	
float **_hif (**double *src* **);**	从 double 提取高位 float
float **_lof (**double *src* **);**	从 double 提取低位 float
float **_hif2(**__float2_t *src* **);**	从 __float2_t 提取高位 float。这被定义为一个宏命令。必须包含 c6x.h。
float **_lof2(**__float2_t *src* **);**	从 __float2_t 提取低位 float。这被定义为一个宏命令。必须包含 c6x.h。
long long **_hi128 (**__x128_t *src* **);**	提取四倍寄存器的两个高位寄存器
double **_hid128 (**__x128_t *src* **);**	提取四倍寄存器的两个高位寄存器
__float2_t **_hif2_128(**__x128_t *src* **);**	提取四倍寄存器的两个高位寄存器。这被定义为一个宏命令。必须包含 c6x.h。
long long **_lo128 (**__x128_t *src* **);**	提取四倍寄存器的两个低位寄存器
double **_lod128 (**__x128_t *src* **);**	提取四倍寄存器的两个低位寄存器
__float2_t **_lof2_128(**__x128_t *src* **);**	提取四倍寄存器的两个低位寄存器。这被定义为一个宏命令。必须包含 c6x.h。
unsigned **_get32_128 (**__x128_t *src* **, 0);**	提取四倍寄存器的第一个寄存器 (base reg + 0)
unsigned **_get32_128 (**__x128_t *src* **, 1);**	提取四倍寄存器的第二个寄存器 (base reg + 1)
unsigned **_get32_128 (**__x128_t *src* **, 2);**	提取四倍寄存器的第三个寄存器 (base reg + 2)
unsigned **_get32_128 (**__x128_t *src* **, 3);**	提取四倍寄存器的第四个寄存器 (base reg + 3)
float **_get32f_128 (**__x128_t *src* **, 0);**	提取四倍寄存器的第一个寄存器 (base reg + 0)
float **_get32f_128 (**__x128_t *src* **, 1);**	提取四倍寄存器的第二个寄存器 (base reg + 1)
float **_get32f_128 (**__x128_t *src* **, 2);**	提取四倍寄存器的第三个寄存器 (base reg + 2)
float **_get32f_128 (**__x128_t *src* **, 3);**	提取四倍寄存器的第四个寄存器 (base reg + 3)

8.6.8 __float2_t 容器类型

应使用 __float2_t 容器类型（而不是双精度型）来存储两个浮点值。有一些操作内在函数可用于创建和操作具有 __float2_t 类型的对象（请参阅表 8-8）。在使用 __float2_t 或使用任何 __float2_t 操作内在函数时，必须包含运行时支持文件 c6x.h。

使用 __float2_t 类型的建议：

- 使用 __float2_t 存储两个浮点值。请勿使用双精度型。
- 使用超长整型来存储 64 位打包整数数据。请勿对打包整数数据使用 double 或 __float2_t。

8.6.9 使用内在函数进行中断控制和原子代码段

C/C++ 编译器支持三种内在函数，用于启用、禁用和恢复中断。语法为：

```
unsigned int        _disable_interrupts ( );
```

```
unsigned int        _enable_interrupts ( );
void                _restore_interrupts (unsigned int);
```

_disable_interrupts() 和 _enable_interrupts( ) 内在函数均返回 unsigned int，之后可传递给 _restore_interrupts( )，以恢复之前的中断状态。这些内在函数产生优化的边界，因此适用于实施临界（或原子）代码段。例如，

```
unsigned int restore_value;
restore_value = _disable_interrupts();
if (sem) sem--;
_restore_interrupts(restore_value);
```

示例代码禁用了中断，因此在 then 子句中修改 sem 之前，为条件子句读取的 sem 值不会改变。内在函数产生优化的边界，因此内存读取和写入 sem 不会越过 _disable_interrupts 或 _restore_interrupts 位置。

备注

覆盖 CSR

_restore_interrupts( ) 内在函数会用参数值覆盖 CSR 控制寄存器。在 _disable_interrupts( ) 内在函数或 _enable_interrupts( ) 内在函数之后，CSR 位的所有更改均会丢失。

_restore_interrupts( ) 内在函数不使用 RINT 指令。

8.6.10 使用未对齐的数据和 64 位值

C6000 支持未对齐的负载，并使用 _mem8、_memd8 和 _mem4 内在函数存储为 64 位和 32 位值。_lo 和 _hi 内在函数可用于从 64 位 double 变量中提取两个 32 位值。_loll 和 _hill 内在函数可用于从 64 位 long long 变量中提取两个 32 位值。

对于使用 64 位类型的内在函数，等效 C 类型为 long long。进行浮点转换时，不要使用 C 类型 double 变量或编译器调用运行时支持数学函数。以下是访问 64 位和 32 位值的方式：

- 若要在 C 代码中获取 long long 变量的高 32 位，请使用 >> 32 或 _hill( ) 内在函数。
- 若要在 C 代码中获取 long long 变量的低 32 位，请转换为 int 或 unsigned 类型，或使用 _loll 内在函数。
- 若要获得 double 变量（转换为 int）的高 32 位，请使用 _hi( )。
- 若要获得 double 变量（转换为 int）的低 32 位，请使用 _lo( )。
- 若要获得 __float2_t 的高 32 位，请使用 _hif2()。
- 若要获得 __float2_t 的低 32 位，请使用 _lof2()。
- 若要创建 long long 值，请使用 _itoll(int high32bits, int low32bits) 内在函数。
- 若要创建 __float2_t 值，请使用 _ftof2(float high32bits, float low32bits) 内在函数。

示例 8-8 展示了 _mem8 内在函数的用法。

示例 8-8. 使用 _mem8 内在函数

```
void alt_load_longlong_unaligned(void *a, int *high, int *low)
{
  long long p = _mem8(a);
  *high = p >> 32;
  *low  = (unsigned int) p;
}
```

8.6.11 通过 MUST_ITERATE 和 _nassert 来启用 SIMD 并扩展编译器对循环的了解

通过使用 MUST_ITERATE 和 _nassert，可确保循环执行特定的次数。

此示例告知编译器，确保循环可以正好运行 10 次：

```
#pragma MUST_ITERATE(10,10);
for (I = 0; I < trip_count; I++) { ...
```

MUST_ITERATE 还可用于指定循环计数的范围以及循环计数的系数。例如：

```
#pragma MUST_ITERATE(8,48,8);
for (I = 0; I < trip; I++) { ...
```

此示例告知编译器，循环执行 8 次到 48 次，循环变量是 8 的倍数（8、16、24、32、40、48）。编译器现在可以使用这些信息，通过展开尽可能地生成最佳循环，即使使用 --interrupt_threshold*n* 选项指定中断每 *n* 个周期发生一次。

*TMS320C6000 编程人员指南*指出，改进 C/C++ 代码的方式之一是使用 word 访问来操作 16 位数据，它们存储在 32 位寄存器的高位和低位中。示例展示了通过内在函数使用特定指令（例如 _mpyh），来强制转换为 int 指针。可使用 _nassert()实现自动化; 内在函数指定 16 位短数组在 32 位 (word) 边界对齐。

下述示例生成相同的汇编代码：

- **示例 1**

```
int dot_product(short *x, short *y, short z)
{
    int *w_x = (int *)x;
    int *w_y = (int *)y;
    int sum1 = 0, sum2 = 0, I;
    for (I = 0; I < z/2; I++)
    {
        sum1 += _mpy(w_x[i], w_y[i]);
        sum2 += _mpyh(w_x[i], w_y[i]);
    }
    return (sum1 + sum2);
}
```

- **示例 2**

```
int dot_product(short *x, short *y, short z)
{
    int sum = 0, I;
    _nassert (((int)(x) & 0x3) == 0);
    _nassert (((int)(y) & 0x3) == 0);
    #pragma MUST_ITERATE(20, , 4);
    for (I = 0; I < z; I++) sum += x[i] * y[i];
    return sum;
}
```

备注

_nassert 的 C++ 语法

在 C++ 代码中，_nassert 是标准命名空间的一部分。因此，正确的语法是 std::_nassert()。

8.6.12 对齐数据的方法

在下面的代码中，_nassert 会告知编译器，对于 f() 的每次调用，ptr 都与 8 字节边界对齐。这种断言往往会使编译器生成代码，以使用单个指令对多个数据值进行操作，也被称为 SIMD（单指令多数据）优化。

```
void f(short *ptr)
{
   _nassert((int) ptr % 8 == 0)
   ; a loop operating on data accessed by ptr
}
```

以下各小节介绍了可用于确保由 ptr 引用的数据已对齐的方法。您必须在代码中调用 f() 的每个位置使用这些方法之一。

8.6.12.1 数组的基址

ptr 之类的参数通常用于传递数组的基址，例如：

```
short buffer[100];
...
f(buffer);
```

这样的数组会自动与 8 字节边界对齐。无论数组是全局的、静态的还是本地的，都是如此。要在这些设备上实现 SIMD 优化只需要这种自动对齐。但您仍然需要包含 _nassert，因为在一般情况下，编译器无法保证 ptr 保存正确对齐的数组的地址。

如果始终将数组的基址传递给 ptr 之类的指针，那么可以使用以下宏命令来反映这一事实。

```
#if defined(_TMS320C6X)
    #define ALIGNED_ARRAY(ptr) _nassert((int) ptr % 8 == 0)
#else
    #define ALIGNED_ARRAY(ptr) /* empty */
#endif
void f(short *ptr)
{
   ALIGNED_ARRAY(ptr);
   ; a loop operating on data accessed by ptr
}
```

此宏命令适用于所有 C6000 设备，或者将代码移植到另一个目标。

8.6.12.2 相对于数组基址的偏移

一种更罕见的情况是传递相对于数组的偏移地址，例如：

```
f(&buffer[3]);
```

该代码将未对齐地址传递给 ptr，因此违反了 _nassert() 中编码的假设。对于这种情况没有直接的补救措施。尽量避免这种做法。

8.6.12.3 动态存储器分配

普通的动态存储器分配保证为本机类型的任何标量对象正确对齐所分配的存储器（例如，为长双精度型或超长整型正确对齐），但不保证任何更严格的对齐。例如：

```
buffer = calloc(100, sizeof(short))
```

若要获得更严格的对齐，请使用函数 memalign with the desired alignment。例如，若要获得 256 字节对齐，请使用以下语法：

```
buffer = memalign(256, 100 * sizeof(short);
```

如果使用 BIOS 存储器分配例程，请确保使用以下语法将对齐因子作为最后一个参数传递：

```
buffer = MEM_alloc( segid , 100 * sizeof(short), 256);
```

请参阅 *TMS320C6000 DSP/BIOS 帮助*，了解有关 BIOS 内存分配例程和 *segid* 参数的详细信息。

8.6.12.4 结构体或类的成员

作为结构体或类成员的数组仅根据数组所需的基本类型进行对齐。节 8.6.12.1 中所述的自动对齐不会出现。

示例 8-9. 结构中的数组

```
   struct s
   {
      ...
      short buf1[50];
      ...
   }  g;
...
f(g.buf1);
```

示例 8-10. 类中的数组

```
    class c
    {
       public :
          short buf1[50];
          void mfunc(void);
       ...
    };
    void c::mfunc()
    {
       f(buf1);
       ...
    }
```

若要对齐结构体中的数组，请将其放置在具有所需对齐方式的虚拟对象的联合体中。如果您想要做到 8 字节对齐，请使用“long long”虚拟字段。例如：

```
struct s
{
    union u
    {   long long dummy;  /* 8-字节对齐 */
        short buffer[50]; /* 也是 8-字节对齐 */
    } u;
    ...
};
```

如果要连续声明多个数组并保持给定的对齐方式，可以通过将数组大小（以字节为单位）保持为所需对齐方式的偶数倍来实现。例如：

```
struct s
{
    union u
    {   long long dummy;  /* 8-字节对齐       */
        short buffer[50]; /* 也是 8-字节对齐 */
        short buf2[50];   /* 4-字节对齐       */
        ...
    } u;
};
```

因为 buf1 的大小是 50 * 2 字节/short = 100 字节，并且 100 是 4（而非 8）的偶数倍，所以 buf2 只在 4 字节边界上对齐。将 buf1 填充到 52 个元素会使 buf2 采用 8 字节对齐方式。

在结构体或类中，无法强制执行大于 8 的数组对齐。出于 SIMD 优化的目的，这并非必需的。

备注

通过程序级优化对齐

在大多数情况下，程序级优化（参见节 4.4）需要在使用-pm -o3 选项的同时，通过一次调用编译器来编译所有源文件。这使编译器能够一次查看所有的源代码，从而启用了很少在其他情况下应用的优化。在这些优化中可以看到，例如，所有对函数 f() 的调用都是将数组的基地址传递给 ptr，因此 ptr 始终正确对齐以进行 SIMD 优化。在这种情况下，不需要 _nassert()。编译器会自动确定 ptr 必须被对齐，并生成优化的 SIMD 指令。

8.6.13 SAT 位副作用

如果发生饱和，则饱和内在函数运算会定义 SAT 位。通过访问控制状态寄存器 (CSR)，可以从 C/C++ 代码中设置和清除 SAT 位。编译器使用以下步骤生成访问 SAT 位的代码。

1. SAT 位因函数调用或函数返回而变为未定义。这意味着 CSR 中的 SAT 位是有效的，并且可以在 C/C++ 代码中读取，直到函数调用或函数返回为止。
2. 如果函数中的代码访问 CSR，则编译器假定 SAT 位在函数中有效，这意味着：
   - SAT 位由禁用软件流水线循环中断的代码维护。
   - 理论上，饱和指令不能被执行。
3. 如果中断服务例程修改了 SAT 位，则应编写该例程以保存和恢复 CSR。

8.6.14 IRP 和 AMR 规则

编译器对 IRP 和 AMR 控制寄存器做出了某些假设。假设应在所有程序中强制执行，如下所示：

1. 调用函数或从函数返回时，AMR 必须设置为 0。函数不必保存和恢复 AMR，但必须保证 AMR 在返回前为 0。
2. 启用中断时，AMR 必须设置为 0，或者在所有中断中都应使用 SAVE_AMR 和 RESTORE_AMR 宏命令（请参阅节 8.7.3）。
3. 只有当中断被禁用时，才能安全地修改 IRP。
4. 如果您将 IRP 用作临时寄存器，则必须保存和恢复 IRP 的值。

8.6.15 浮点和饱和控制寄存器副作用

在浮点架构上执行浮点运算时，或者执行饱和运算时，可能会设置某些控制寄存器中的状态位。特别是，可以在 FADCR、FAUCR、FMCR、CSR 和 SSR 寄存器（以下称为“状态位控制寄存器”）中设置状态位。通过写入或读取这些寄存器，可以从 C/C++ 代码中设置和清除这些位，如示例 7-1 中所示。

在某些情况下，编译器可能会*推测*指令。推理是一种优化技术，在该技术中，编译器会使指令执行的时间早于通常预期的时间，以提高性能。当编译器检测到函数中使用了可能写入状态位控制寄存器的指令，但外部嵌套函数（包括内联）中并没有读取相关控制寄存器时，编译器会假定可以自由推测指令。

如果程序使用可能写入状态位控制寄存器的指令，然后在函数的后面读取相关的控制寄存器（而不是在设置状态位控制寄存器的函数中），则可以使用 --assume_control_regs_read 选项来防止编译器推测可能会在状态位控制寄存器中设置状态位的指令。

如果中断服务例程修改了（或者可能修改）状态位控制寄存器中的任何位，则应写入中断服务例程以保存和恢复该浮点控制寄存器。

8.7 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。

8.7.1 保存 SGIE 位

编译器可以使用 DINT 和 RINT 指令来禁用和恢复围绕软件流水线循环的中断。这些指令使用 CSR 控制寄存器以及 TSR 控制寄存器中的 SGIE 位。因此，SGIE 位被视为调用时保存。如果有调用编译器生成代码的汇编代码，则应保存 SGIE 位（例如，保存到栈中）以供将来使用。SGIE 位应在从编译器生成代码返回时恢复。

8.7.2 在中断期间保存寄存器

当 C/C++ 代码被中断时，中断例程必须保存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。如果中断服务例程是用 C/C++ 编写并使用 __interrupt 关键字声明的，编译器会处理寄存器保留。如有需要，编译器会保存并恢复 ILC 和 RILC 控制寄存器。

8.7.3 使用 C/C++ 中断例程

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。C/C++ 中断例程可在栈上为局部变量分配多达 32K 的空间。例如：

```
__interrupt void example (void)
{
...
}
```

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尝试定义的寄存器。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，例程会保存所有可用寄存器。中断返回指针 (IRP) 的中断分支。请勿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。如需更多信息，请分别参见 节 7.9.20 和 节 7.5.4。

您应负责在中断中正确地处理 CSR 中的 AMR 控制寄存器和 SAT 位。默认情况下，编译器不会执行任何额外操作来保存/恢复 AMR 和 SAT 位。c6x.h 头文件中包含了处理 SAT 位和 AMR 寄存器的宏命令。

例如，在一些些手动汇编代码中使用循环寻址（即 AMR 不等于 0）。这些手动汇编代码可以被中断为 C 代码中断服务例程。C 代码中断服务例程假设 AMR 设置为 0。您需要定义一个无符号整型的本地临时变量，并在 C 中断服务例程的起始和终止处调用 SAVE_AMR 和 RESTORE_AMR 宏命令，以在 C 中断服务例程中正确地保存/恢复 AMR。

```
#include <c6x.h>
__interrupt void interrupt_func()
{
   unsigned int temp_amr;
   /* define other local variables used inside interrupt */
   /* save the AMR to a temp location and set it to 0 */
   SAVE_AMR(temp_amr);
   /* code and function calls for interrupt service routine */
   ...
   /* restore the AMR for your hand assembly code before exiting */
   RESTORE_AMR(temp_amr);
}
```

如果您需要在 C 中断服务例程中保存/恢复 SAT 位（即当在 C 中断服务例程中发生中断时，您正在执行饱和算术，该例程也可能执行一些饱和算术），可以通过与上述使用 SAVE_SAT 和 RESTORE_SAT 宏命令的示例类似的方式来完成。

如果需要，编译器会保存和恢复 ILC 和 RILC 控制寄存器。

对于浮点架构，您负责处理浮点控制寄存器 FADCR、FAUCR 和 FMCR。如果您正在从浮点控制寄存器中读取位，并且如果中断服务例程（或任何被调用的函数）执行浮点运算，则应保存和恢复相关的浮点控制寄存器。没有为这些寄存器提供宏命令，因为对 unsigned int 临时变量的简单赋值就足够了。

8.7.4 使用汇编语言中断例程

只要遵循与编译器相同的寄存器惯例，就可以利用汇编语言代码处理中断。像所有汇编函数一样，中断例程可使用堆栈、访问全局 C/C++ 变量并正常调用 C/C++ 函数。调用 C/C++ 函数时，请确保 表 8-2 中列出的所有寄存器已保存，因为 C/C++ 函数可以修改它们。

8.8 运行时支持算术例程

运行时支持库包含许多汇编语言函数，为 C/C++ 数学运算提供了 C6000 指令集不提供的算术例程，例如整数除法、整数求余数以及浮点运算。

这些例程遵循标准的 C/C++ 调用序列。编译器会在适用时自动添加这些例程；它们并非供您的程序直接调用。

`lib/src` 源目录中提供了这些函数的源代码。源代码中包含有用于描述函数运算的注释。您可以提取、检查和修改任何数学函数。不过，请务必遵循本章中概述的调用惯例和寄存器内容保存规则。表 8-9 总结了用于算术运算的运行时支持函数。

表 8-9. C6000 运行时支持算术函数

类型	函数	描述
float	__c6xabi_cvtdf (double)	将双精度型转换为浮点型
int	__c6xabi_fixdi (double)	将双精度型转换为带符号整数型
long	__c6xabi_fixdi (double)	将双精度型转换为长整型
long long	__c6xabi_fixdlli (double)	将双精度型转换为超长整型
uint	__c6xabi_fixdlli (double)	将双精度型转换为无符号整数型
ulong	__c6xabi_fixdul (double)	将双精度型转换为无符号长整型
ulong long	__c6xabi_fixdull (double)	将双精度型转换为无符号超长整型
double	__c6xabi_cvtfd (float)	将浮点型转换为双精度型
int	__c6xabi_fixfi (float)	将浮点型转换为带符号整数型
long	__c6xabi_fixfli (float)	将浮点型转换为长整型
long long	__c6xabi_fixflli (float)	将浮点型转换为超长整型
uint	__c6xabi_fixfu (float)	将浮点型转换为无符号整数型
ulong	__c6xabi_fixful (float)	将浮点型转换为无符号长整型
ulong long	__c6xabi_fixfull (float)	将浮点型转换为无符号超长整型
double	__c6xabi_fltid (int)	将带符号整数型转换为双精度型
float	__c6xabi_fltif (int)	将带符号整数型转换为浮点型
double	__c6xabi_fltud (uint)	将无符号整数型转换为双精度型
float	__c6xabi_fltuf (uint)	将无符号整数型转换为浮点型
double	__c6xabi_fltlid (long)	将带符号长整型转换为双精度型
float	__c6xabi_fltlif (long)	将带符号长整型转换为浮点型
double	__c6xabi_fltuld (ulong)	将无符号长整型转换为双精度型
float	__c6xabi_fltulf (ulong)	将无符号长整型转换为浮点型
double	__c6xabi_fltllid (long long)	将带符号超长整型转换为双精度型
float	__c6xabi_fltllif (long long)	将带符号超长整型转换为浮点型
double	__c6xabi_fltulld (ulong long)	将无符号超长整型转换为双精度型
float	__c6xabi_fltullf (ulong long)	将无符号超长整型转换为浮点型
double	__c6xabi_absd (double)	取双精度型绝对值
float	__c6xabi_absf (float)	取浮点型绝对值
long	__c6xabi_labs (long)	取长整型绝对值
long long	__c6xabi_llabs (long long)	取超长整型绝对值
double	__c6xabi_negd (double)	双精度型取反值
float	__c6xabi_negf (float)	浮点型取反值
long long	__c6xabi_negll (long)	超长整型取反值
ong long	__c6xabi_llshl (long long)	超长整型左移
long long	__c6xabi_llshr (long long)	超长整型右移
ulong long	__c6xabi_llshru (ulong long)	无符号超长整型右移
double	__c6xabi_addd (double, double)	双精度数加法

表 8-9. C6000 运行时支持算术函数 (continued)

类型	函数	描述
double	__c6xabi_cmpd (double, double)	双精度数比较
double	__c6xabi_divd (double, double)	双精度数除法
double	__c6xabi_mpyd (double, double)	双精度数乘法
double	__c6xabi_subd (double, double)	双精度数减法
float	__c6xabi_addf (float, float)	浮点数加法
float	__c6xabi_cmpf (float, float)	浮点数比较
float	__c6xabi_divf (float, float)	浮点数除法
float	__c6xabi_mpyf (float, float)	浮点数乘法
float	__c6xabi_subf (float, float)	浮点数减法
int	__c6xabi_divi (int, int)	带符号整数除法
nt	__c6xabi_remi (int, int)	带符号整数取余
uint	__c6xabi_divu (uint, uint)	无符号整数除法
uint	__c6xabi_remu (uint, uint)	无符号整数取余
long	__c6xabi_divli (long, long)	带符号长整数除法
long	__c6xabi_remli (long, long)	带符号长整数取余
ulong	__c6xabi_divul (ulong, ulong)	无符号长整数除法
ulong	__c6xabi_remul (ulong, ulong)	无符号长整数取余
long long	__c6xabi_divlli (long long, long long)	带符号超长整数除法
long long	__c6xabi_remlli (long long, long long)	带符号超长整数取余
ulong long	__c6xabi_mpyll (long long, long long)	无符号超长整数乘法
ulong long	__c6xabi_divull (ulong long, ulong long)	无符号超长整数除法
ulong long	__c6xabi_remull (ulong long, ulong long)	无符号超长整数取余

8.9 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，但通常由复位硬件导引至该函数。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。不过，这不会将硬件设置为在复位时自动导引至 c_int00（请参阅《*TMS320C64x/C64x+ DSP CPU 和指令集参考指南*》、《*TMS320C6740 CPU 和指令集参考指南*》或《*TMS320C66x+ DSP CPU 和指令集参考指南*》）。

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. 执行全局/静态变量的 C 自动初始化。如需更多信息，请参阅 节 8.9.2。
3. 通过将数据从初始化表复制到 .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）。
4. 从全局构造函数表调用文件域构建所需的 C++ 初始化例程。如需更多信息，请参阅 节 8.9.2.6。
5. 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。

8.9.1 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。

8.9.2 变量的自动初始化

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。

8.9.2.1 零初始化变量

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。

8.9.2.2 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 8.9.2.5。

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 8.9.2.3。

8.9.2.3 运行时变量自动初始化

在运行时自动初始化变量是自动初始化的默认方法。如需使用此方法，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 8-11 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。

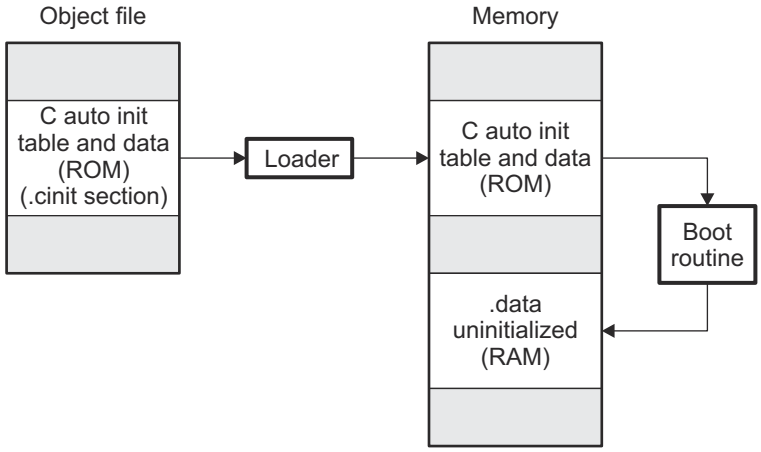


图 8-11. 运行时自动初始化

8.9.2.4 的自动初始化表

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。

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。

_TI_Handler_Table_Base:

32-bit handler 1 address
⋮
32-bit handler n address

_TI_Handler_Table_Limit:

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。

8.9.2.4.1 数据格式遵循的长度

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。

8.9.2.4.2 零初始化格式

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。

8.9.2.4.3 行程编码 (RLE) 格式

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。

8.9.2.4.4 Lempel-Ziv-Storer-Szymanski 压缩 (LZSS) 格式

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。

8.9.2.4.5 用于处理 C 自动初始化表的 C 代码示例

运行时支持引导例程具有处理 C 自动初始化表的代码。以下 C 代码说明了如何在目标上处理自动初始化表。

```
typedef void (*handler_fptr)(const unsigned char *in,
unsigned char *out);
#define HANDLER_TABLE   __TI_Handler_Table_Base
extern  unsigned int   HANDLER_TABLE;
extern  unsigned char *__TI_CINIT_Base;
extern  unsigned char *__TI_CINIT_Limit;
void auto_initialize()
{
   unsigned char **table_ptr;
   unsigned char **table_limit;
   /*-------------------------------------------------------------*/
   /* Check if Handler table has entries.                         */
   /*-------------------------------------------------------------*/
   if (&__TI_Handler_Table_Base >= &__TI_Handler_Table_Limit)
      return;
   /*-------------------------------------------------------------*/
   /* Get the Start and End of the CINIT Table.                   */
   /*-------------------------------------------------------------*/
   table_ptr   = (unsigned char **)&__TI_CINIT_Base;
   table_limit = (unsigned char **)&__TI_CINIT_Limit;
   while (table_ptr < table_limit)
   {
      /*----------------------------------------------------------*/
      /* 1.Get the Load and Run address.                          */
      /* 2.Read the 8-bit index from the load address.            */
      /* 3.Get the handler function pointer using the index from  */
      /*    handler table.                                        */
      /*----------------------------------------------------------*/
      unsigned char *load_addr   = *table_ptr++;
      unsigned char *run_addr    = *table_ptr++;
      unsigned char  handler_idx = *load_addr++;
      handler_fptr   handler     =
                  (handler_fptr)(&HANDLER_TABLE)[handler_idx];
      /*----------------------------------------------------------*/
      /* 4.Call the handler and pass the pointer to the load data */
      /*    after index and the run address.                      */
      /*----------------------------------------------------------*/
      (*handler)((const unsigned char *)load_addr, run_addr);
   }
}
```

8.9.2.5 在加载时初始化变量

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 8-12 演示了加载时变量的初始化。

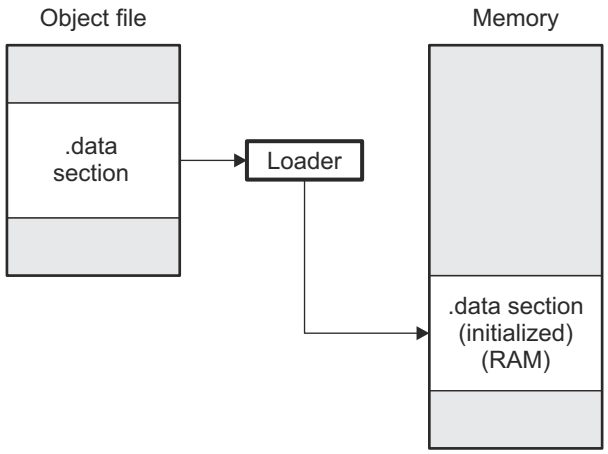


图 8-12. 加载时初始化

8.9.2.6 全局构造函数

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。

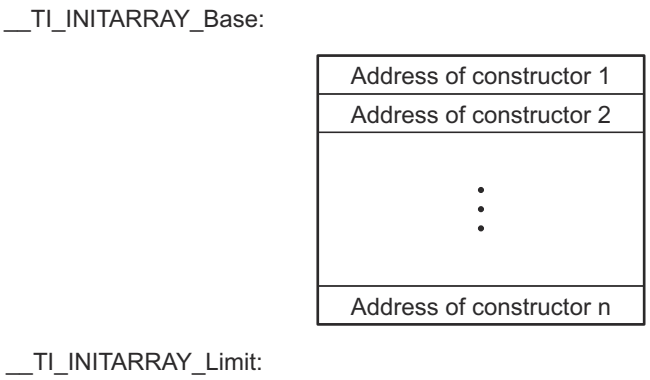


图 8-13. 构造函数表

8.10 支持多线程应用

编译器支持多线程应用的各种功能。这些功能假定有一个提供线程管理服务的底层运行时操作系统。

8.10.1 使用 OpenMP 进行编译

编译器实现了对 OpenMP 3.0 API 和部分 OpenMP 4.0 规范的支持。若要启用对 OpenMP 的支持，请使用 --openmp 或 --omp 编译器选项。

OpenMP 是共享内存并行编程的行业标准。其提供了可移植的高级编程结构，使用户能够以增量方式轻松地公开程序的任务和循环级并行性。使用 OpenMP，用户可以通过使用编译器指令注释程序代码来指定高级程序的并行化策略，这些指令指定了区域代码如何由一组线程执行。编译器会计算出计算到机器的详细映射。OpenMP 编程 API 使程序员能够执行以下操作：

- 创建和管理线程
- 分配和分发工作（任务）给线程
- 指定哪些数据是线程之间共享的，哪些数据是保密的
- 协调线程对共享数据的访问

OpenMP 是一种基于线程的编程语言。主线程执行程序的顺序部分。当主线程遇到并行区域时，主线程会分叉一组工作线程，与主线程并行执行。

OpenMP API 由指令 (#pragma)、函数调用和环境变量组成。编译器会将 OpenMP API 转换为多线程代码，并调用自定义运行时库，该库实现对线程管理、共享内存和同步的支持。有关 OpenMP API（包括 API 规范）的更多详细信息，请参阅 http://www.openmp.org。有关 TI 对 OpenMP 支持的详细信息，包括支持的 pragma 的描述，请参阅 TI OpenMP 加速器模型在线文档。

SYS/BIOS (OMP) 库的 OpenMP 运行时实现 OpenMP 解决方案堆栈。目前，仅在 SYS/BIOS 操作系统的 TI DSP 上支持 OpenMP。所有 OpenMP 程序都必须与 BIOS-MCSDK 2.1 中的 OMP 运行时库链接。

8.10.2 多线程运行时支持

使用 --openmp 编译会使链接器自动选择线程安全 RTS 库。如果您有一个非 OpenMP 多线程应用程序，则可以通过使用 --multithread 选项（而非 --openmp 选项）使应用程序与 RTS 库的线程安全版本链接在一起。

8.10.2.1 运行时线程安全

线程安全涉及线程私有数据的创建、初始化、维护和销毁。它还要求必须保护对在线程之间共享的数据的访问。也就是说，在给定的时间应该只允许一个线程访问一段共享数据。在具有私有数据缓存（例如 C6600 器件上的 L1D 缓存）的多核器件上需要解决的另一个问题是私有数据缓存和共享内存之间存在的共享数据副本必须保持一致。这意味着，如果线程将一段共享数据读入处理器上的私有数据缓存中，则在访问或修改数据之前，它必须使本地数据缓存中存在的该数据的任何本地副本无效。这可以确保当前正在执行的线程将只访问共享数据的最新可用副本。

8.10.2.2 线程创建、初始化和终止

线程库负责在创建线程时分配存储器的线程局部存储区，然后对驻留在该线程局部存储区中的任何线程私有数据对象进行初始化，而当线程终止时，线程库必须释放为线程私有数据分配的线程局部存储区。

8.10.2.3 线程局部存储 (TLS)

编译器支持使用 __thread 限定符来识别要向其提供线程私有存储空间的数据对象。若要访问已标识为线程局部变量的变量，编译器依赖对运行时操作系统线程库的支持来查找线程局部变量的运行时位置。具体来说，线程库必须提供 __c6xabi_get_tp() 函数的实现方案。运行时操作系统的线程库将会提供已代表当前所执行线程分配的 TLS 块的地址，然后编译器可以利用给定线程局部变量在 TLS 块中的位置信息来访问 TLS 块中的数据。

有关 TLS 数据对象的更多信息，请参阅《*C6000 嵌入式应用程序二进制接口应用报告*》(SPRAB89A)。

8.10.2.4 访问共享数据

对共享数据对象的访问必须在一个关键区域内得到保护，以防止第二个线程在第一个线程访问共享数据对象时进入代码的关键区域。我们还需要关注共享内存和专用数据缓存中可能存在的共享数据副本之间的数据一致性，如上文节 8.10.2.1 中所述。

This page intentionally left blank.

章节 9

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 9.1 和节 9.2 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 9.4 中对此过程进行了介绍。

9.1 C 和 C++ 运行时支持库

TMS320C6000 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。为每个目标 CPU 版本、大端字节序和小端字节序，以及 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 9.1.8。

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 7.1 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

9.1.1 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 6.3.1。

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《*TMS320C6000 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。

9.1.2 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 C6X_C_DIR 环境变量设为安装相关工具的 包含目录。

以下头文件提供 C 标准的 TI 扩展：

- c6x.h -- 提供内在函数定义。
- c6x_vec.h -- 提供对 TI 编译器使用的 OpenCL 样式矢量数据类型和内置函数的支持。
- cpy_tbl.h -- 声明 copy_in() RTS 函数，该函数用于在运行时将代码或数据从加载位置移动到单独的运行位置。此函数有助于管理叠加层。
- _data_synch.h -- 声明 RTS 库使用的函数来帮助实现共享数据同步。例如，在将本地数据缓存刷新到全局共享内存时使用这些函数。
- file.h -- 声明由 RTS 库中的低级别 I/O 函数使用的函数。
- gsm.h -- 提供由欧洲电信标准化协会 (ETSI) 定义的基本 DSP 运算和 GSM 数学运算。
- _lock.h -- 在声明系统范围的互斥锁时使用。此头文件已弃用；请改用 _reg_mutex_api.h 和 _mutex.h。
- memory.h -- 提供 C 标准不需要的 memalign() 函数。
- _mutex.h -- 声明 RTS 库使用的函数，以帮助实现 RTS 拥有的特定资源的互斥体。例如，这些函数用于堆或文件表分配。
- _pthread.h -- 声明低级别互斥体基础设施功能并提供对递归互斥体的支持。

- _reg_mutex_api.h -- 声明一个函数，RTOS 可以使用该函数来注册在 RTOS 中实现但由 RTS 的 _mutex.h 函数间接调用的底层锁定机制和/或线程 ID 机制。
- _reg_synch_api.h -- 声明一个函数，RTOS 可以使用该函数来注册在 RTOS 中实现但由 RTS 的 _data_synch.h 函数间接调用的底层缓存同步机制。
- strings.h -- 提供额外的字符串函数，包括 bcmp()、bcopy()、bzero()、ffs()、index()、rindex()、strcasecmp() 和 strncasecmp()。有关这些函数的详细信息，请参阅 v7.6 版本的说明。
- _tls.h -- 如果支持 __thread 限定符，则提供一些有用的宏命令来声明需要用于线程本地存储的对象。如果未启用 __thread 限定符，这些将成为普通声明。
- vect.h -- 提供支持 128 位向量的宏命令。

编译器随附了以下标准 C 头文件：assert.h、complex.h、ctype.h、errno.h、float.h、inttypes.h、iso646.h、limits.h、locale.h、math.h、setjmp.h、signal.h、stdarg.h、stdbool.h、stddef.h、stdint.h、stdio.h、stdlib.h、string.h、time.h、wchar.h 和 wctype.h。

编译器随附了以下标准 C++ 头文件：algorithm、bitset、cassert、cctype、cerrno、cfloat、ciso646、climits、clocale、cmath、complex、csetjmp、csignal、cstdarg、cstddef、cstdio、cstdlib、cstring、ctime、cwchar、cwctype、deque、exception、fstream、functional、hash_map、hash_set、iomanip、ios、iosfwd、iostream、istream、iterator、limits、list、locale、map、memory、new、numeric、ostream、queue、rope、set、sstream、stack、stdexcept、streambuf、string、strstream、typeinfo、utility、valarray 和 vector。

以下头文件用于较旧的 C++ 代码：fstream.h、iomanip.h、iostream.h、new.h、stdiostream.h、stl.h 和 strstream.h。

以下头文件供 TI 组件内部使用，不应直接包含在您的应用中：_data_synch.h、_fmt_specifier.h、_isfuncdcl.h、_isfuncdef.h、_mutex.h、_pthread.h、_tls.h、access.h、c60asm.i、cpp_inline_math.h、elf_linkage.h、elfnames.h、linkage.h、mathf.h、mathl.h、pprof.h、unaccess.h、wchar.hx、xcomplex、xdebug、xhash、xiosbase、xlocale、xlocinfo、xlocinfo.h、xlocmes、xlocmon、xlocnum、xloctime、xmemory、xstddef、xstring、xtree、xutility、xwcc.h、ymath.h 和 yvals.h

9.1.3 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\c6000_#.#.#\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 rts64plus.lib 库，或者构建新库。有关库命名的详细信息，请参阅节 9.1.8；有关构建的详细信息，请参阅节 9.4

9.1.4 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

9.1.5 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 unsigned short。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 7.4，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。

9.1.6 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 9-1. __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。

9.1.7 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。

9.1.8 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 6.3.1.1）。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：

rts*trg*[*endian*][*abi*][eh].lib

trg	该库用于构建的 C6000 架构器件系列。可以是以下情况之一：64plus、6740 或 6600。	
字节序	指示字节序：	
	（空）	小端字节序库
	e	大端字节序库
abi	指示使用的应用程序二进制接口 (ABI)。尽管不再支持 COFF 文件格式，但库文件名仍然包含“_elf”，以便与较旧 COFF 库中的 EABI 库区分开来。	
	_elf	EABI
eh	指示该库是否支持异常处理	
	（空）	不支持异常处理
	_eh	支持异常处理

9.2 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 *TMS320C6000 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）。

9.2.1 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：

```
cl6x main.c -z --heap_size=1000 --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。

9.2.1.1 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。

9.2.2 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 9.2.3。

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 9.2.5）。
- *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 9.2.3。

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 9.2.5。)

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。

9.2.3 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 9.2.5。）
- *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 9.2.3。

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。

9.2.4 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 9-1 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 9-1 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 9-1. 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"); }
```

9.2.5 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 9.2.2 部分中指定的接口来声明这些函数。用于运行 TMS320C6000 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 9-2 将执行以下操作：

- 将器件 *mydevice* 添加至器件表

add_device (continued)

向器件表添加器件

- 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 9-2 显示了为 C I/O 添加和使用器件：

示例 9-2. 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);
}
```

9.3 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。

9.4 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 rts64plus.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。

9.4.1 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。

9.4.2 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 9.4.2.2，以了解可能需要您这样做的情形。

9.4.2.1 通过链接器自动重建标准库

链接器主要通过 C6X_C_DIR 环境变量查找运行时支持库。通常，C6X_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 C6X_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=rts64plus.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/rts64plus.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 *TMS320C6000 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 C6X_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 C6X_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 9.4.2.2。
- 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。

9.4.2.2 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。）

9.4.2.2.1 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=rts64plus.lib
```

9.4.2.2.2 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。

9.4.2.2.3 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 rts64plus.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=rts64plus.lib --name=rts64plus_debug.lib --install_to=$Project/Debug --
extra_options="-g"
```

9.4.2.2.4 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 9-2 中所述。

```
mklib --help
```

表 9-2. 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=rts64plus.lib --index=$C_DIR/lib
```

构建类似 rts64plus.lib 的自定义库，但启用符号调试支持：

```
mklib --pattern=rts64plus.lib --extra_options="-g" --index=$C_DIR/lib --install_to=$Project/Debug --
name=rts64plus_debug.lib
```

9.4.3 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。

9.4.3.1 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。

9.4.3.2 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 C6X_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 9-2 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 10

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。

10.1 调用 C++ 名称还原器

调用 C++ 名称还原器的语法如下：

dem6x [*options*] [*filenames*]

dem6x	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 dem6x 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：

--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）

10.2 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。

```
class banana {
public:
    int calories(void);
    banana();
    ~banana();
};
int calories_in_a_banana(void)
{
    banana x;
    return x.calories();
}
```

编译器输出的汇编代码结果如下。

```
_calories_in_a_banana__Fv:
;** --------------------------------------------------------------------------*
           CALL    .S1     ___ct__6bananaFv
           STW     .D2T2   B3,*SP--(16)
           MVKL    .S2     RL0,B3
           MVKH    .S2     RL0,B3
           ADD     .S1X    8,SP,A4
           NOP             1
RL0:       ; CALL OCCURS
           CALL    .S1     _calories__6bananaFv
           MVKL    .S2     RL1,B3
           ADD     .S1X    8,SP,A4
           MVKH    .S2     RL1,B3
           NOP             2
RL1:       ; CALL OCCURS
           CALL    .S1     ___dt__6bananaFv
           STW     .D2T1   A4,*+SP(4)
           ADD     .S1X    8,SP,A4
           MVKL    .S2     RL2,B3
           MVK     .S2     0x2,B4
           MVKH    .S2     RL2,B3
RL2:       ; CALL OCCURS
           LDW     .D2T1   *+SP(4),A4
           LDW     .D2T2   *++SP(16),B3
           NOP             4
           RET     .S2     B3
           NOP             5
           ; BRANCH 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：

```
dem6x calories_in_a_banana.asm
```

运行 C++ 名称还原器后的结果如下所示。_ZN6bananaC1Ev、_ZN6banana8caloriesEv 和 _ZN6bananaD1Ev 中的链接名称已还原。

```
calories_in_a_banana():
;** --------------------------------------------------------------------------*
        CALL    .S1 banana::banana()
        STW     .D2T2 B3,*SP--(16)
        MVKL    .S2 RL0,B3
        MVKH    .S2 RL0,B3
        ADD     .S1X 8,SP,A4
        NOP 1
RL0:    ; CALL OCCURS
        CALL    .S1 banana::calories()
        MVKL    .S2 RL1,B3
        ADD .   S1X 8,SP,A4
        MVKH    .S2 RL1,B3
        NOP 2
RL1:    ; CALL OCCURS
        CALL    .S1 banana::~banana()
        STW     .D2T1 A4,*+SP(4)
        ADD     .S1X 8,SP,A4
        MVKL    .S2 RL2,B3
        MVK .   S2 0x2,B4
        MVKH .  S2 RL2,B3
RL2:    ; CALL OCCURS
        LDW     .D2T1 *+SP(4),A4
        LDW     .D2T2 *++SP(16),B3
        NOP 4
        RET .S2 B3
        NOP 5
        ; BRANCH OCCURS
```

This page intentionally left blank.

A.1 术语

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。

.bss 段[.bss section] 默认的目标文件段之一。使用汇编器 .bss 指令在存储器映射中保留指定量的空间，以便稍后用于存储数据。.bss 段未被初始化。

字节 根据 ANSI/ISO C，可容纳一个字符的最小可寻址单元。

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。不再支持该 ABI。
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 TMS320C6000 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。
内核	流水线循环序言和流水线循环结语之间的软件流水线循环主体。
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。
循环展开	一种扩展小循环的优化，使循环的每次迭代出现在代码中。虽然循环展开会增大代码大小，但可以提高代码性能。

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。另请参阅*汇编优化器*。
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。
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。
流水线	一种在第一条指令完成之前就开始执行第二条指令的技术。流水线中可以有几条指令，每条指令处于不同的处理阶段。
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。
软件流水线	C/C++ 优化器使用的一种技术，用于调度循环中的指令以使循环的多个迭代并行执行。
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。
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 ^。
循环计数	循环结束前执行的次数。
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。这些段是使用.bss 和 .usect 指令创建的。
无符号值	无论实际符号如何都会被当作非负数的值。

字 目标内存中的 32 位可寻址位置。

TEXAS INSTRUMENTS

Changes from JANUARY 17, 2023 to APRIL 6, 2023 (from Revision E (January 2023) to Revision F (April 2023)) **Page**

- 删除了 C6000 不支持的矢量构造函数样式语法的文档。 190

Changes from JANUARY 14, 2022 to JANUARY 17, 2023 (from Revision D (January 2022) to Revision E (January 2023)) **Page**

- 添加了 --assume_control_regs_read 选项 23
- --strict_compatibility 链接器选项不再起任何作用，已从文档中删除。 28
- 添加了 --fp_single_precision_constant 编译器选项。 31
- 添加了 --assume_control_regs_read 选项。 32
- 记录了 ptrdiff_t 和 size_t 类型的预定义宏。 37
- 更正了出现在其中的 --gen_cross_reference_listing 和 --asm_cross_reference_listing 选项的名称。 45
- 向 --opt_level 优化列表添加了指向其他文档的链接。 54
- 向某些优化说明添加了有关相应 --opt_level 的信息。 93
- 本机向量类型现在称为“TI 向量类型”。 149
- 更新了相关示例，以使用构造函数来初始化向量的元素，而不是使用强制转换/标量扩展语法。 190
- 更新了相关示例，以使用构造函数来初始化向量的元素，而不是强制转换/标量扩展语法。 192
- 添加了 --assume_control_regs_read 选项的影响说明。 246

Changes from JUNE 15, 2018 to JANUARY 14, 2022 (from Revision C (June 2018) to Revision D (January 2022)) **Page**

- 本文档修订版是对 v8.3.x 版本的更新。此前的修订版 SPRU514C 也是针对 v8.3.x 而发布的。 11
- 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
- C6000 编译器和链接器不再支持动态链接和动态加载 23
- 弃用 --small_enum 选项 23
- 不再支持 MISRA-C 检查 23
- 删除了不受支持的 --rom 链接器选项的文档。 28
- 不再支持 MISRA-C 检查 31
- 弃用 --small_enum 选项 32
- 添加了有关 --gen_func_subsections 选项的默认值的信息 136
- SET_DATA_SECTION pragma 优先于 --gen_data_subsections=on 选项。 136
- 更正了有关 --gen_data_subsections 选项的默认值的信息。 136
- 有文件证明不支持 C11 原子操作。 142
- 更新了有关枚举类型大小的信息。 148
- 新增了有关字符集和文件编码的信息。 150
- 不再支持 MISRA-C 检查的 pragma 159
- 阐明 --opt_level 和 FUNCTION_OPTIONS pragma 之间的交互。 170
- 为与 MUST_ITERATE pragma 对应的属性新增了 C++ 属性语法。 172
- SET_DATA_SECTION pragma 优先于 --gen_data_subsections=on 选项。 178

- 添加了与 UNROLL pragma 对应的各个属性的 C++ 属性语法。 179
- 有文件证明不支持 C11 原子操作。 183
- 增加了使用位置属性的示例。 186
- 更正了文档中 _avgu4、_dmvd、_spint 和 _ssub2 内在函数的参数。 222
- 更正了文档中 _labs 和 _lsadd 内在函数的返回类型。 222
- 更正了文档中 _dintsp 和 _maxu4 内在函数的汇编指令。 222
- 更正了文档中 _avgu4、_cmpgtu4、_rpack2 和 _subabs4 内在函数的说明。 222
- 阐明了有关字符串处理函数的信息。 261
- 添加了关于时间和时钟 RTS 函数的信息。 262

本文档介绍了适用于 TMS320C6000™ 处理器系列 C64x+、C6740 和 C6600 版本的 v8.x TI 代码生成工具。v8.0 和更高版本的 TI 代码生成工具不支持 C6200、C6400、C6700 和 C6700+ 版本。此外，v8.x 工具不再支持 COFF 目标文件格式和相关的 STABS 调试格式。如果使用某个旧版器件或者需要 COFF 支持，请使用 v7.4 的代码生成工具，并参考 SPRU187 和 SPRU186 文档。

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。

表 13-1. 修订历史记录

添加内容的版本	章节	位置	添加/修改/删除
SPRUI04C	使用编译器，C/C++ 语言	节 3.3 和节 7.2	编译器现在遵循 C++14 标准。此外，从例外列表中删除了几个旧的 C++ 功能，因为有多个版本已支持这些功能。
SPRUI04C	使用编译器和优化	表 3-12 和节 4.6.1	更正了 --disable_software_pipeline 选项的拼写。
SPRUI04C	使用编译器	节 3.3.1	添加了 --ecc=on 链接器选项，支持生成 ECC。请注意，现在 ECC 生成默认关闭。还添加了 --no_const_clink 选项，该选项会阻止编译器为常量全局数组生成 .clink 指令。
SPRUI04C	使用编译器，C/C++ 语言	节 3.11 和节 7.9	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。
SPRUI04C	C/C++ 语言	节 7.3.2 和节 7.15	对矢量数据类型的支持不再需要使用优化器。
SPRUI04C	C/C++ 语言	节 7.14.2 和节 7.14.4	添加了 retain 作为函数属性和变量属性。
SPRUI04C	C/C++ 语言	节 7.15.6	矢量数据类型可以与 printf() 一同使用，如 OpenCL 1.2 规范中所述。
SPRUI04C	运行时环境	节 8.6.2	阐明了有关 __x128_t 对象对齐的信息。
SPRUI04C	运行时支持函数	节 9.2.1.1	添加了有关在 format.h 中定义并由 printf() 等函数使用的 FORMAT_CONVERSION_BUFSIZE 宏命令的信息。
SPRUI04B	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。
SPRUI04B	使用编译器	节 3.3	添加了 --multithread 作为编译器和链接器选项。
SPRUI04B	使用编译器，C/C++ 语言	节 3.3.3	修改为指明：即使使用 CHECK_MISRA pragma 也需要 --check_misra 选项。
SPRUI04B	使用编译器	节 3.10	更正了文档以描述 ---gen_preprocessor_listing 选项。名称 --gen_parser_listing 不正确。
SPRUI04B	优化	节 4.10.3	更正了 _TI_start_pprof_collection() 和 _TI_stop_pprof_collection() 的函数名称。
SPRUI04B	链接 C/C++ 代码	节 6.3.5	将 .ovly 和 .TI.crctab 添加到了由编译器创建的初始化段列表中。
SPRUI04B	运行时环境	节 8.6.6	确定了定义为宏命令的内在函数，因此需要包含 c6x.h 头文件。
SPRUI04A	使用编译器	节 3.3 和节 6.2.3	添加了 --gen_data_subsections 选项。
SPRUI04A	运行时环境	节 8.9.1	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。

表 13-1. 修订历史记录 (continued)

添加内容的版本	章节	位置	添加/修改/删除
SPRUI04	引言	节 1.4	不再支持 COFF 目标文件格式和关联的 STABS 调试格式。C6000 编译器现在仅支持嵌入式应用二进制接口 (EABI) ABI，这种接口仅适用于使用 ELF 目标文件格式和 DWARF 调试格式的目标文件。已删除或简化了本文档中提及 COFF 格式的各段。如果需要 COFF 支持，请使用 7.4 版本的代码生成工具，并参考 SPRU187 和 SPRU186 文档。
SPRUI04	入门	章节 2	添加了入门章节，其中包含适用于新用户的入门信息。
SPRUI04	使用编译器	节 3.3.4	将 --ramfunc 选项添加到了编译器命令行选项中。
SPRUI04	使用编译器	节 3.3.5	不再支持 C6200、C6400、C6700 和 C6700+ 版本。删除或简化了本文档中提及这些器件的各段。如果您使用其中某个旧器件，请使用 7.4 版本的代码生成工具，并参考 SPRU187 和 SPRU186 文档。
SPRUI04	使用编译器	节 3.6	添加了有关向 main() 传递参数的技术的段。
SPRUI04	使用编译器	节 3.11	更正为使用 --opt_level=3 而不是 --opt_level=2 来执行自动内联。
SPRUI04	C/C++ 语言	节 7.9.7	添加了 diag_push 和 diag_pop 诊断消息 pragma。
SPRUI04	C/C++ 语言	节 7.9.24	添加了 NOINIT 和 PERSISTENT pragma。
SPRUI04	C/C++ 语言	节 7.14.2	添加了 ramfunc 函数属性。
SPRUI04	C/C++ 语言	节 7.3.2 和节 7.15	添加了矢量数据类型。
SPRUI04	运行时环境	节 8.5	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。
SPRUI04	运行时环境	节 8.6.6	更正了 _shr2 和 _shru2 内在函数的操作数。

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