C7000 优化 C/C++ 编译器

User's Guide

Literature Number: ZHCUAU3J
JANUARY 2018 – REVISED MARCH 2024

目录

插图清单

表格清单

This page intentionally left blank.

关于本手册

C7000 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
- 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。

本文档介绍了对 C7000™ 处理器系列的 C7100 型号的支持。

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：

cl7x [*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：

cl7x --run_linker {**--rom_model** | **--ram_model**} *filenames* [**--output_file=** *name.out*]
--library= *libraryname*

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

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™ 规范版本 1.2 (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：

SPRUIG5	***C6000 至 C7000 迁移用户指南***
SPRUIG4	***C7000 嵌入式应用二进制接口 (EABI) 用户指南***
SPRUIV4	***C7000 C/C++ 优化指南***
SPRUIG3	***C7000 VCOP Kernel-C 转换功能规范***
SPRUIG6	***C7000 主机仿真用户指南***
SPRUIU4	***C7x 指令指南*** （可通过 TI Field 应用工程师获得）
SPRUIP0	***C71x DSP CPU、指令集和矩阵乘法加速器技术参考手册*** （可通过 TI Field 应用工程师获得）
SPRUIQ3	***C71x DSP Corepac 技术参考手册*** （可通过 TI Field 应用工程师获得）
SPRAAB5	***DWARF 对 TI 目标文件的影响。***

商标

This page intentionally left blank.

章节 1

软件开发工具简介

C7000™ 由一套软件开发工具支持，其中包括优化 C/C++ 编译器、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。

C7000™ 代码生成工具类似于与为 TMS320C6000™ 提供的工具。有关差异和迁移的信息，请参阅 *C6000 到 C7000 迁移用户指南* (SPRUIG5)。

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。

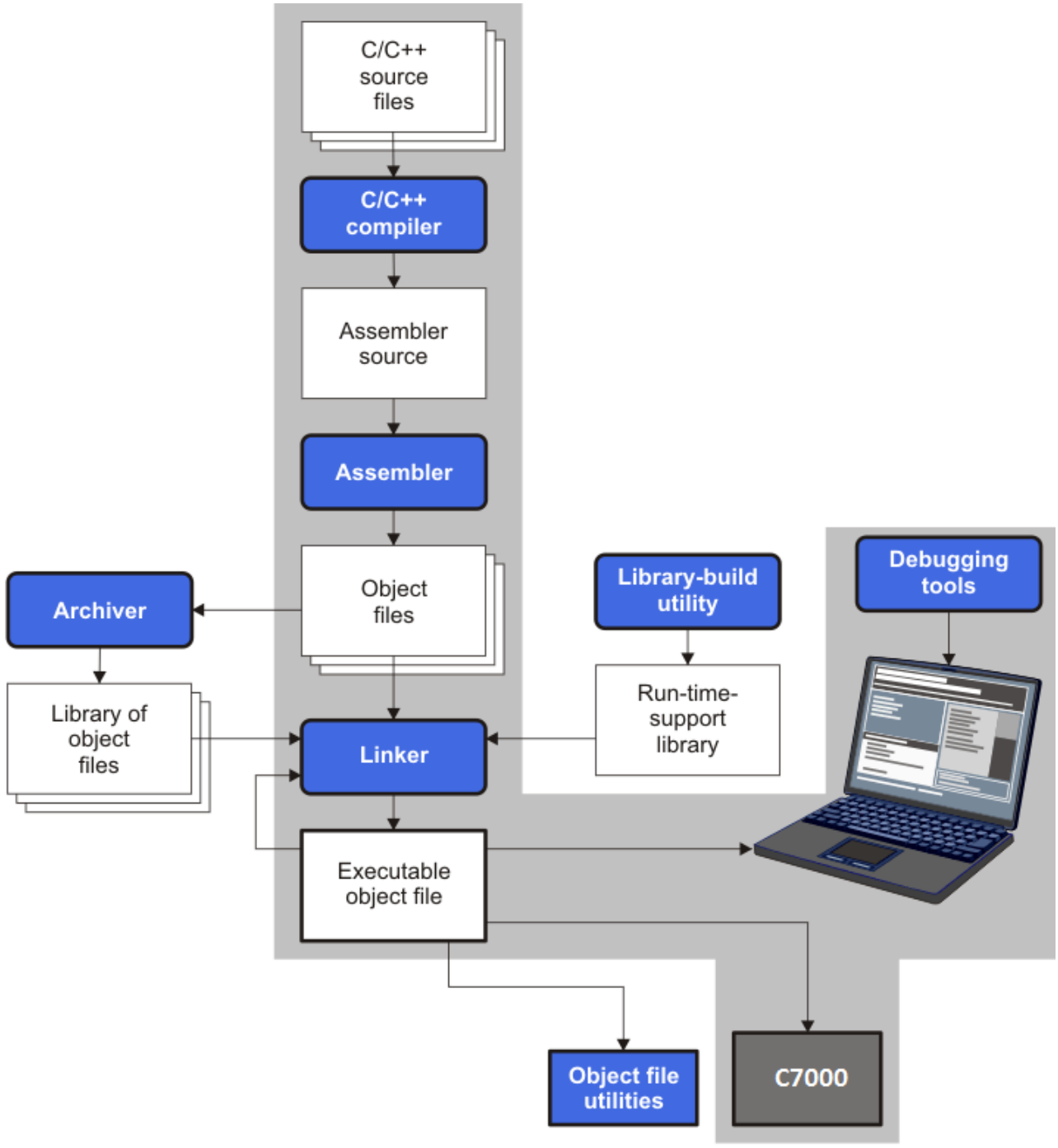


图 1-1. C7000 软件开发流程

以下列表描述了图 1-1 中显示的工具：

- **编译器**接受 C/C++ 源代码，生成 C7000 汇编语言源代码，并自动将其转换为机器语言可重定位的目标文件。请参阅章节 3。

- **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 11。
- **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。
- **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 7。

  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 7.4。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。
- **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 14。
- 此开发流程的主要产品是可执行的目标文件，其可以在 C7000 CPU（属于更大的器件）器件上执行。

此外，还提供了以下实用程序来帮助检查或管理给定目标文件的内容：

- **目标文件显示实用程序**以可读的或 XML 格式打印目标文件和目标库的内容。请参阅节 13.1。
- **反汇编器**解码目标模块中的机器代码以显示其所表示的汇编指令。请参阅节 13.2。
- **名称实用程序**打印目标文件或目标存档中定义或引用的目标和函数的符号名称列表。请参阅节 13.3。
- **符号去除实用程序**从目标文件和目标库中删除符号表和调试信息。请参阅节 13.4。

1.2 编译器接口

编译器是名为 cl7x 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 C 函数。更多有关调用约定的信息，请参阅章节 6。

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。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 5.2。
- **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 7。

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 5.12 。

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 C7000 的 ELF 格式由 C7000 嵌入式应用程序二进制接口 (EABI) 扩展，相关信息请参阅 SPRUIG4 文档。

章节 2

开始使用代码生成工具

本章概述了创建一个使用 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。	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方法。	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，C7000 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、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 > C7000 Compiler** 和 **Build > C7000 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 **cl7x.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，-mv7100、--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。

```
cl7x -mv7100 --c99 --opt_level=1 --define=c7000 --include_path="C:/ti/ti-cgt-c7000/include"
    hello.c objects.cpp
    --run_linker --library=lnk.cmd --heap_size=0x800 --output_file=myprogram.out
```

This page intentionally left blank.

章节 3

使用 C/C++ 编译器

编译器将您的源程序转换成 C7000 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。编译器会生成目标代码。

  可以在单命令中编译 C 和 C++ 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 11。

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 11.1.1。

3.2 调用 C/C++ 编译器

要调用编译器，请输入：

cl7x [*options*] [*filenames*] [**--run_linker** [*link_options*] *object files*]

cl7x	用于运行编译器的命令。
options	影响编译器对输入文件的处理方式的选项。表 3-6 到表 3-27 列出了这些选项。
filenames	一个或多个 C/C++ 源文件。
--run_linker (**-z**)	调用链接器的选项。--run_linker 选项的缩写形式为 -z。有关更多信息，请参阅章节 11。
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，则并通过链接创建一个名为 myprogram.out 的可执行程序，则需要输入：

```
cl7x symtab.c file.c --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 **cl7x**。

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 C7X_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 3-27 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 7100。	节 3.3.5
--silicon_errata_i2117		生成代码以处理 C7100 器件勘误表 i2117。	节 3.3.4
--silicon_errata_i2376={on|off}		生成代码以处理 C7504 器件勘误表 i2376。	节 3.3.4
--big_endian	-me	以大端字节序格式生成目标代码。默认格式为小端字节序。	节 3.3.4
--mma_version={ 1 | 2 | 2_256 | disallow }		指定要在目标器件上使用的矩阵乘法加速器 (MMA) 的版本。	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（如果未使用选项且 --vectypes=off，则为默认值）。	节 4.1
--opt_level=*n*	-O*n*	级别 0 (-O0) 仅优化寄存器使用情况（如果未使用选项且 --vectypes=on，则为默认值）。 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（如果使用选项但无设置，则为默认值）。 级别 3 (-O3) 使用级别 2 优化并对文件进行优化。 级别 4 (-O4) 使用级别 3 优化并执行链接时间优化。	节 4.1、节 4.3
--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
--assume_addresses_ok_for_stream={off|on}		如果启用，使用 --auto_stream 选项控制的自动流式传输将假定寻址模式合法地映射到流。 启用此假设可能会导致更多地使用流引擎 (SE) 或流地址生成器 (SA)。可能需要此类假设的情况包括但不限于：应用于向量指针的 64 位索引、无符号索引和数组索引。	节 4.15.9
--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
--auto_stream={off|saving|no_saving}		控制编译器是否在可能且有利的情况下尝试自动使用流引擎 (SE) 和流地址生成器 (SA)。 在“off”模式下，将禁止自动使用 SE 和 SA。在“saving”模式下，SE 和/或 SA 状态将被保存，以防它们已在使用中。在“no_saving”模式下，无需保存上下文即可自动使用 SE 和 SA。C7100 和 C7120 器件仅支持“off”或“no_saving”模式，“off”是默认值。较新的器件（例如 C7504）支持所有三种模式，并且 --auto_stream=saving 是默认设置。	节 4.15.9
--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 3.10
--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
--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) 会禁用优化信息文件。 级别 1 (-on2) 会生成优化信息文件。 级别 2 (-on2) 会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 4.12
--program_level_compile	-pm	组合源文件以执行程序级优化。	节 4.4
--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
--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 4.9.1

(1) **注意：**机器专用运行时选项（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 4.13
--symdebug:dwarf_version=3|4		指定 DWARF 格式版本。默认版本为 4。	节 3.3.6
--symdebug:none		禁用所有符号调试。	节 3.3.6 节 4.13

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 11.1.3
--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，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
--keep_asm	-k	保留汇编语言 (.asm) 文件。	

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 5.12
--c99		根据 ISO C99 标准处理 C 文件。	节 5.12
--c11		根据 ISO C11 标准处理 C 文件。	节 5.12
--c++14		根据 ISO C++14 标准处理 C++ 文件。	节 5.12
--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 5.6
--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
--pending_instantiations=#		指定在任何给定时间内可能正在进行的模板实例化的数量。使用 0 指定一个不受限制的数字。	节 3.3.4
--printf_support={nofloat|full| minimal}		支持更小、有限版本的 printf 函数系列（sprintf、fprintf 等）和 scanf 函数系列（sscanf、fscanf 等）运行时支持函数。	节 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 5.12.3
--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 5.12.3
--vectypes={on|off}		启用对 TI 矢量数据类型的支持。默认为 on。	节 5.3.2

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

表 3-9. 预定义宏选项（续）

选项	别名	效果	段
--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	将由 *num* 标识的诊断分类为错误。	节 3.7.1
--diag_remark=*num*	-pdsr	将由 *num* 标识的诊断分类为备注。	节 3.7.1
--diag_suppress=*num*	-pds	抑制由 *num* 标识的诊断。	节 3.7.1
--diag_warning=*num*	-pdsw	将由 *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*。在达到此错误数量后，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	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 3.9
--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
--common={on|off}		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 4.7.2
--disable_software_pipeline	-mu	关闭软件流水线。	节 4.7.1
--fp_not_associative	-mc	阻止对关联浮点运算进行重新排序。	节 4.10
--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 11.2.3
--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 11.2.2

表 3-12. 运行时模型选项（续）

选项	别名	效果	段
--ramfunc={on|off}		如果设置为 on，则表示每个函数都将从 RAM 运行。函数将被放置在 RAM 中，并针对 RAM 执行进行优化。等效于在转换单元的所有函数中指定 __attribute__((ram_func))。	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 3.13
--entry_parm={none|name|address}		将函数的参数指定给 --entry_hook 选项。	节 3.13
--exit_hook[=*name*]		启用出口挂钩。	节 3.13
--exit_parm={none|name|address}		将函数的参数指定给 --exit_hook 选项。	节 3.13
--remove_hooks_when_inlining		删除自动内联函数的入口/出口挂钩。	节 3.13

表 3-14. 汇编选项

选项	别名	效果	段
--c_src_interlist	-ss	交叉列出 C 源代码和汇编语句。	节 3.11 节 4.12

表 3-15. 文件类型说明符选项

选项	别名	效果	段
--asm_file=*filename*	-fa	不管其扩展名为何，都将 *filename* 标识为汇编源文件。默认情况下，编译器和汇编器将 .asm 文件视为汇编源文件。	节 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
--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 3-16. 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
--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 3-17. 默认文件扩展名选项

选项	别名	效果	段
--asm_extension=[.]*extension*	-ea	设置汇编源文件的默认扩展名。	节 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 3-18. 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 3-19. 性能顾问选项

选项	别名	效果	段
--advice:performance[={all|none}]		生成编译器优化建议。默认为 all。	节 4.11
--advice:performance_file={stdout|stderr|user_specified_filename}		指定将建议写入 stdout、stderr 或文件。	节 4.11
--advice:performance_dir={user_specified_directory_name}		指定在命名目录中创建建议文件。	节 4.11

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 11 以及 章节 12。

表 3-20. 链接器基本选项

选项	别名	说明
--run_linker	-z	启用链接。
--output_file=*file*	-o	为可执行输出文件命名。默认文件名为 .out file。
--map_file=*file*	-m	生成输入和输出段（包括空位）的映射或列表，并将列表放置在 *file* 中。
--stack_size=*size*	[-]-stack	将 C 系统栈大小设为 *size* 字节，并定义全局符号来指定栈大小。默认值 = 1K 字节。
--heap_size=*size*	[-]-heap	将堆大小（对于 C 中的动态内存分配）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 3-21. 文件搜索路径选项

选项	别名	说明
--library=*file*	-l	将存档库或链接命令 *file* 命名为链接器输入。
--disable_auto_rts		禁止自动选择运行时支持库。请参阅节 11.3.1.1。
--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 3-22. 命令文件预处理选项

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

表 3-23. 诊断消息选项

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

表 3-24. 链接器输出选项

选项	别名	说明
--absolute_exe	-a	生成绝对可执行目标文件。这是默认设置；如果 --absolute_exe 和 --relocatable 均未指定，链接器的行为就像指定了 --absolute_exe 一样。
--mapfile_contents=*attribute*		控制映射文件中包含的信息。
--relocatable	-r	成不可执行的、可重定位输出目标文件。

表 3-24. 链接器输出选项（续）

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

表 3-25. 符号管理选项

选项	别名	说明
--entry_point=*symbol*	-e	定义一个全局符号，用于指定可执行目标文件的主要入口点。
--globalize=*pattern*		将与 *pattern* 匹配的符号的符号链接更改为全局型。
--hide=*pattern*		隐藏与指定 *pattern* 匹配的符号。
--localize=*pattern*		将与指定 *pattern* 匹配的符号设为局部型。
--make_global=*symbol*	-g	将 *symbol* 设为全局型（覆盖 -h）。
--make_static	-h	将所有全局符号设为静态型。
--no_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 3-26. 运行时环境选项

选项	别名	说明
--arg_size=*size*	--args	为 argc/argv 存储器区域保存 *size* 个字节。
--cinit_compression[=type]		指定应用于 C 自动初始化数据的压缩类型。如果此选项没有指定 *type*，则默认为 lzss，表示 Lempel-Ziv-Storer-Szymanski 压缩。
--copy_compression[=type]		压缩由链接器复制表复制的数据。如果此选项没有指定 *type*，则默认为 lzss，表示 Lempel-Ziv-Storer-Szymanski 压缩。
--fill_value=*value*	-f	为输出段中的空穴设置默认填充值
--ram_model	-cr	在加载时初始化变量。详情请参见节 11.3.4。
--rom_model	-c	在运行时自动初始化变量。详情请参见节 11.3.4。
--trampolines[=off|on]		生成 far call trampolines。默认为 on。

表 3-27. 其他选项

选项	别名	说明
--compress_dwarf[=off|on]		大力减小输入目标文件中 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]		放置段以更大限度减少所需的 far trampolines 数量。默认值为 postorder。
--preferred_order=*function*		为函数放置设定优先级。
--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 4.12。--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：

```
cl7x --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 C7X_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 11.1.3。

--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 11.1。

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

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

--tool_version
打印编译器中每个工具的版本号。未发生编译。

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

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

3.3.3 其他有用的选项

以下是其他选项的详细说明：

--advice:performance 生成编译时优化建议。请参阅节 4.11。

--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，如果命令行指定 --float_operations_allowed=32，则编译器会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关联。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。

--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。

3.3.4 运行时模型选项

这些选项专用于 C7000 工具集。有关更多信息，请参阅参考的章节。

--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。设置为 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 4.7.2。
--disable_software_pipeline	关闭软件流水线。请参阅节 4.7.1。
--fp_not_associative	编译器不会对浮点运算进行重新排序。请参阅节 4.10。
--mma_version={ 1 | 2 | 2_256 | disallow }	指定待使用的器件上的矩阵乘法加速器 (MMA) 修订版。对于特定的 C7000 器件版本，--mma_version 的设置可能是与器件版本（默认）相对应的 MMA 版本或 `disallow`。禁止指定与器件版本不一致的 MMA 版本，否则会导致错误。 • 对于 --silicon_version=7100，默认为 --mma_version=1。 • 对于 --silicon_version=7120，默认为 --mma_version=2。 • 对于 --silicon_version=7504，默认为 --mma_version=2_256。 编译器在其生成的目标文件中放置适当的 MMA 构建属性。 目标文件指定：如果使用 --mma_version=disallow 选项禁用 MMA，或者应用程序中的代码不执行任何可以使用 MMA 的操作，则不使用 (UNUSED) MMA。 MMA 构建属性可确保使用不兼容 MMA 版本的目标文件不允许链接。MMA 的版本 1、2 和 2_256 彼此都不兼容。MMA 版本设置为 UNUSED 的目标文件可以与使用任何 MMA 版本（但最多使用一个版本）的文件链接。有关更多详细信息，请参阅 *C7000 EABI 技术参考指南* (SPRUIG4)。
--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。
--ramfunc={on|off}	如果设置为 on，则表示每个函数都将从 RAM 运行。函数将被放置在 RAM 中，并针对 RAM 执行进行优化。等效于在转换单元的所有函数中指定 __attribute__((ram_func))。如果设置为 off，则只有具有 ramfunc 函数属性的函数才会以此种方式被处理。请参阅节 5.13.2。
--silicon_version=*num*	选择目标 CPU 版本。请参阅节 3.3.5。
--silicon_errata_i2117	--silicon_errata_i2117 编译器选项生成的代码会在具有 C7100 CPU 内核的器件上自动处理器件勘误表 i2117。在边缘情况下使用此选项可能会对 MMA 性能产生负面影响。默认情况下，此选项处于禁用状态。
--silicon_errata_i2376={on|off}	--silicon_errata_i2376=on 编译器选项生成的代码会在具有 C7504 CPU 内核的器件上自动处理器件勘误表 i2376。性能应该不会受到此权变措施的严重影响。如果使用 --silicon_version=7504 编译器选项（或 -mv7504 别名），则此选项默认打开。要关闭此权变措施的使用，请使用 --silicon_errata_i2376=off。不建议关闭此权变措施，仅供高级用户在特定情况下关闭。

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 C7100 部件生成代码。目前，此选项的可用设置为：

- --silicon_version=7100（或其别名，-mv7100）
- --silicon_version=7120（或其别名，-mv7120）
- --silicon_version=7504（或其别名，-mv7504）

3.3.6 符号调试和分析选项

下述选项用于选择符号调试：

--symdebug:dwarf （默认）生成 C/C++ 源代码级调试器使用的指令。--symdebug:dwarf 选项的缩写形式为 -g。请参阅节 4.13。有关 DWARF 格式的详细信息，请参阅 *DWARF 调试标准*。

--symdebug:dwarf_version={3|4} 在指定 --symdebug:dwarf（默认值）时，指定待生成的 DWARF 调试格式版本（3 或 4）。默认情况下，编译器生成 DWARF 版本 4 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
.c	C 源文件
.C	取决于操作系统
.cpp、.cxx、.cc	C++ 源文件
.obj .c.obj .cpp.obj .o* .dll .so	对象

备注

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

有关如何更改编译器解释各个文件名的方式的信息，请参阅节 3.3.8。有关如何更改编译器解释和命名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：

```
cl7x *.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：

--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 objects.cp 的 C++ 文件，请使用 --cpp_file 和 --c_file 选项强制进行正确解释：

```
cl7x --c_file=file.s --cpp_file=objects.cp
```

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：

--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

以下示例将编译文件 fit.rrr ，并创建名为 fit.o 的目标文件：

```
cl7x --cpp_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：

```
cl7x --cpp_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：

```
cl7x --asm_directory=d:\assembly
```

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

```
cl7x --obj_directory=d:\object
```

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

```
cl7x --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：

```
cl7x --pp_directory=d:\preproc
```

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

```
cl7x --temp_directory=d:\temp
```

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 (C7X_C_OPTION)

您可能会发现，使用 C7X_C_OPTION 环境变量来设置编译器和链接器默认选项很有用。如果这样做，编译器和链接器将在每次运行时使用 C7X_C_OPTION 定义中的默认选项和/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 C7X_C_OPTION 环境变量并进行处理。下表展示了如何设置 C7X_C_OPTION 环境变量。为操作系统选择命令：

操作系统	输入
UNIX (Bourne shell)	**C7X_C_OPTION="** *option*$_1$ [*option*$_2$ ...]**"; export C7X_C_OPTION**
Windows	**set C7X_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 C7X_C_OPTION 环境变量，如下所示：

```
set C7X_C_OPTION=--quiet --src_interlist --run_linker
```

命令行或 C7X_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 C7X_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 C7X_C_OPTION 设置如上所示：

```
cl7x *c                          ; compiles and links
cl7x --compile_only *.c          ; only compiles
cl7x *.c --run_linker lnk.cmd    ; compiles and links using a command file
cl7x --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，请参阅节 12.4》中的链接器说明一章。

3.4.2 命名一个或多个备用目录 (C7X_C_DIR)

链接器使用 C7X_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)	**C7X_C_DIR="** *pathname*$_1$ **;** *pathname*$_2$ **;..."; export C7X_C_DIR**

操作系统	输入
Windows	**set C7X_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 C7X_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 C7X_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 C7X_C_DIR`
Windows	`set C7X_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 3-28 中列出的预定义宏名称。

表 3-28. 预定义 C7000 宏名称

宏名称	说明
__big_endian__	如果选择了大端模式（使用了 --endian=big 选项），则定义为 1；否则未定义。
__C7000__	对于所有 C7000 子目标，定义为 1。
__C7100__	对于 C7100 子目标，定义为 1。
__C7120__	对于 C7120 子目标，定义为 1。
__C7504__	对于 C7504 子目标，定义为 1。
__C7X_VEC_SIZE_BITS__	定义为 512 或 256，具体取决于 --silicon_version/-mv 选项。
__C7X_VEC_SIZE_BYTES__	定义为 64 或 32，具体取决于 --silicon_version/-mv 选项。
__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），则定义为 1；否则未定义。
__PTRDIFF_T_TYPE__	定义为 ptrdiff_t 类型，即 long
__SIZE_T_TYPE__	定义为 size_t 类型，即 unsigned long
__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 5.1。
__STDC_VERSION__	C 标准宏。

表 3-28. 预定义 C7000 宏名称（续）

宏名称	说明
__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_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。
__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。
__TIME__ (1)	以“*hh:mm:ss*”形式扩展到编译时间
__WCHAR_T_TYPE__	定义为 wchar_t 类型，即 unsigned int。

(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 3-28 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 C7X_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 C7X_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	`cl7x --include_path=/tools/files source.c`
Windows	`cl7x --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 C7X_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 C7X_C_DIR：

```
C7X_C_DIR "/usr/include;/usr/ucb"; export C7X_C_DIR
```

或使用以下命令调用编译器：

```
cl7x --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**诊断表明了改进代码的一种潜在方法。有关详细信息，请参阅节 4.11。

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，长消息会换行以便使用多行显示。

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、备注和建议诊断都是酌情处理的。

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。

--advice:performance 默认情况下，编译器会向 stdout 发出建议。使用 **--advice:performance=none** 进行编译可禁止生成性能建议。请参阅节 4.11，了解相关选项和示例。

--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。在达到此错误数量后，编译器放弃编译。（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 3-29 中列出的标识符之一开头。

表 3-29. 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 3-30 中定义的诊断标识符。

表 3-30. 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 3-30 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。

3.10 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 3.10.2）。如果函数体在同一模块中可见，或者使用了 -pm 且函数在正在编译的模块之一中可见，则可以在 --opt_level=0 或更高级别内联函数。如果包含定义信息的文件和调用点都使用了 --opt_level=4 进行编译，则可以在链接时内联函数。同时定义为静态和内联的函数更有可能被内联。
- 当使用 --opt_level=3 或更高级别时，编译器可能会自动内联符合条件的函数，即使这些函数没有被声明为内联函数也是如此。此过程会用到使用内联关键字显式定义的函数对应列出的相同决策因素列表。有关自动函数内联的更多信息，请参阅节 4.5。
- 除非 --opt_level=off，否则 pragma FUNC_ALWAYS_INLINE（节 5.8.12）和等效的 `always_inline` 属性（节 5.13.2）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 5.8.10）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 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。

3.10.1 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 5.15。除了所列出的这些之外，abs 和 memcpy 也是作为内在函数实现的。

3.10.2 内联限制

编译器会根据节 3.10 中提到的因素决定内联哪些函数。此外，还有一些限制可以取消函数被自动内联或基于关键字内联的资格。

如果函数符合以下条件，编译器将保留调用：

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

如果函数具有会给编译器带来困难情形的特性，编译器也不会内联调用：

- 具有可变长度的参数列表
- 从不返回
- 是超出深度限制的递归或非叶函数
- 未声明为内联且包含一条不是注释的 asm() 语句
- 是中断函数
- 是 main() 函数
- 未声明为内联，并且需要将过多的栈空间用于本地数组或结构体变量
- 包含易失性局部变量或参数
- 是包含 catch 的 C++ 函数
- 未在当前编译单元中定义且未使用 -O4 优化

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

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

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

3.10.3 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);
    ...
}
```

3.10.4 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;
}
```

3.11 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：

```
cl7x --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 4.12。使用 --c_src_interlist 选项会导致性能和/或代码大小下降。

foo.c 中 foo() 函数的 C 代码：

```
int foo(int a, int b, int
c)

{
    int d = a + b;
    int e = d - c;
    return e;
}
```

使用以下命令进行编译：

```
cl7x foo.c --c_src_interlist --symdebug:none
```

生成汇编文件 foo.asm，其中包含：

```
;----------------------------------------------------------------------
;   1 | int foo(int a, int b, int c)
;----------------------------------------------------------------------

;******************************************************************************
;* FUNCTION NAME: foo                                                         *
;*                                                                            *
;*   Regs Modified     : A4,A8,D0,SP                                          *
;*   Regs Used         : A4,A5,A6,A8,D0,SP                                    *
;*   Local Frame Size  : 0 Args + 0 Auto + 8 Save = 8 byte                    *
;******************************************************************************
||foo||:
;** --------------------------------------------------------------------------*

           MVC     .S1     RP,A8             ; [A_S1]
||         STD     .D1     A8,*SP(8)         ; [A_D1]

           ADDD    .D1     SP,0xfffffff8,SP  ; [A_D1]
;----------------------------------------------------------------------
;   3 | int d = a + b;
;----------------------------------------------------------------------
           ADDW    .D1     A5,A4,D0          ; [A_D1] |3|
;----------------------------------------------------------------------
;   4 | int e = d - c;
;----------------------------------------------------------------------
           SUBW    .D1     D0,A6,A4          ; [A_D1] |4|
;----------------------------------------------------------------------
;   5 | return e;
;----------------------------------------------------------------------
           MVC     .S1     A8,RP             ; [A_S1] BARRIER
           LDD     .D1     *SP(16),A8        ; [A_D1]

           RET     .B1     ; [A_B]
||         ADDD    .D1     SP,0x8,SP         ; [A_D1]

           ; RETURN OCCURS {RP}              ; []
```

3.12 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。

C7000 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。

3.13 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 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 5.8.27。

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**（如果未使用 --opt_level 选项且 --vectypes=off，则为默认值）
  - 不执行优化
- **--opt_level=0** 或 **-O0**（如果未使用 --opt_level 选项且 --vectypes=on，则为默认值）
  - 执行控制流图简化（节 4.14.3）
  - 将变量分配给寄存器（节 4.14.15）
  - 执行循环旋转（节 4.14.10）
  - 消除未使用的代码
  - 简化表达式和语句（节 4.14.5）
  - 扩展对声明的内联函数的调用（节 4.14.6）
- **--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 4.14.4）
  - 删除未使用的赋值（节 4.14.4）
  - 消除局部公用表达式 (节 4.14.4)
- **--opt_level=2** 或 **-O2**（如果在没有设置的情况下使用 --opt_level 选项，则为默认值）执行所有 --opt_level=1 (-O1) 优化，加上：
  - 执行软件流水线（节 4.7）
  - 执行循环优化（节 4.14.9、节 4.14.11 和节 4.14.13）
  - 消除全局公用子表达式（节 4.14.4）
  - 消除全局未使用的赋值（节 4.14.4）
  - 将循环中的数组引用转换为递增的指针形式（节 4.14.8）
  - 执行循环展开（节 5.8.34）
- **--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 3.10 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）
- **--opt_level=4** 或 **-O4**
  - 执行链接时优化。(节 4.6)

有关 --opt_level 和 --opt_for_speed 选项以及各种 pragma 如何影响内联的详细信息，请参阅节 3.10。

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 --opt_level 优化级别。相反，请使用 --opt_for_speed 选项来控制代码大小/性能权衡。更高的优化级别（--opt_level 或 -O）与偏低的 --opt_for_speed 级别（0、1 或 2）相结合会产生更小的代码大小。

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：

表 4-1. 优化级别

Level	说明
--opt_for_speed=0 (-mf0)	优化代码大小，性能恶化或受影响的风险为*高*。
--opt_for_speed=1 (-mf1)	优化代码大小，性能恶化或受影响的风险为*中*。
--opt_for_speed=2 (-mf2)	优化代码大小，性能恶化或受影响的风险为*低*。
--opt_for_speed=3 (-mf3)	优化代码性能/速度，代码大小恶化或受影响的风险为*低*。
--opt_for_speed=4 (-mf4)	优化代码性能/速度，代码大小恶化或受影响的风险为*中*。（默认值）
--opt_for_speed=5 (-mf5)	优化代码性能/速度，代码大小恶化或受影响的风险为*高*。

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。

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 4-2 中列出的选项与 --opt_level=3 一起使用以执行指定的优化：

表 4-2. 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

不要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，请将 --opt_for_speed 选项设置为 0、1 或 2 来减小代码大小。

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 4-3 选择相应的级别以附加到该选项。

表 4-3. 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。（如果使用 --opt_level=4 (-O4)，则不能使用 --program_level_compile 选项，因为链接时优化提供了与程序级优化相同的优化机会。）

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

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

--program_level_compile 选项要求使用 --opt_level=3，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 4-4 选择合适的级别以附加到 --call_assumptions 选项。

表 4-4. 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 4-5 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 4-5. 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。

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 3.10。

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 3.10.2 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 链接时优化（--opt_level=4 选项）

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

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

如果使用 --opt_level=4 (-O4)，则不能同时使用 --program_level_compile 选项，因为链接时优化提供了与程序级优化相同的优化机会（节 4.4）。链接时优化具有以下优点：

- 每个源文件都可以单独编译。程序级编译的一个问题是其要求所有源文件都要一次性传递给编译器。这通常需要对客户的构建过程进行重大修改。使用链接时优化，所有文件都可以单独编译。
- 第三方目标文件可以参与优化。如果第三方供应商提供了使用 --opt_level=4 选项编译的目标文件，这些文件将与用户生成的文件一起参与优化。这包括作为 TI 运行时支持的一部分提供的目标文件。未使用 –opt_level=4 编译的目标文件仍可在执行链接时优化的链接中使用。未使用 –opt_level=4 进行编译的那些文件则不参与优化。
- 可以使用不同的选项集编译源文件，但有一些例外（见下文）。对于程序级编译，必须使用相同的选项集编译所有源文件。借助链接时优化，可以使用不同的选项来编译文件。如果编译器确定两个选项不兼容，就会发出错误。

对 C7000 使用链接时优化 (--opt_level=4) 时，只链接使用相同 --silicon_version 和 --mma_version 选项编译的源文件和目标文件。链接使用不同 --silicon_version 或 --mma_version 选项编译的源文件和/或目标文件可能会导致链接失败。

4.6.1 选项处理

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

执行链接时优化时，有些选项是不兼容的。这些选项通常也会在命令行上产生冲突，但也可能是在链接时优化期间无法处理的选项。值得注意的是，当对 C7000 编译器使用链接时优化时，只使用通过相同 --silicon_version 和 --mma_version 选项编译的源文件和/或目标文件。

4.6.2 不兼容的类型

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

4.7 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_speed 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*流水线循环逻辑程序*，内核下方的区域称为*流水线循环收尾程序*。

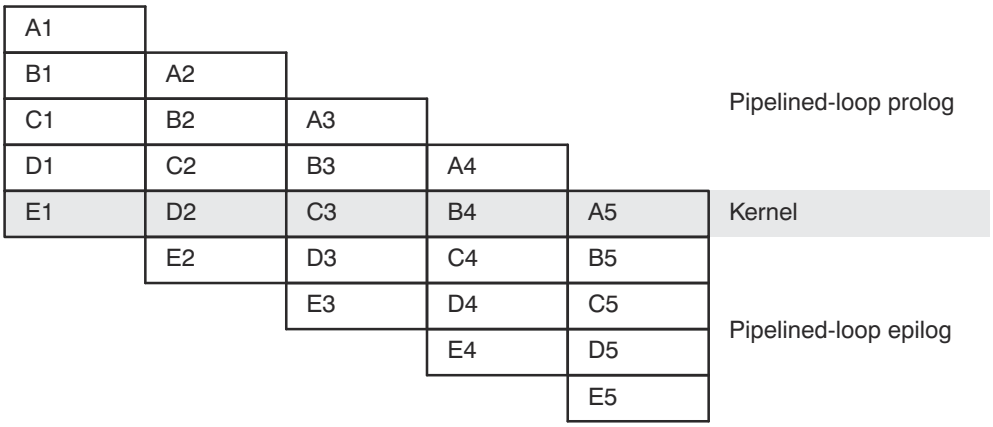


图 4-1. 进行了软件流水线处理的循环

4.7.1 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。

4.7.2 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。

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 选项会添加有关软件流水线循环特性的附加信息，包括循环内核每个周期的寄存器使用情况。此选项添加的信息可能仅对经验丰富的 C7000 用户有用。

以下示例显示了默认情况下（即未使用 --debug_software_pipeline option 选项时）发出的一些信息。当未使用 --debug_software_pipeline 选项时，仅显示最有用且可操作的信息。

示例 4-1. 软件流水线信息

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop found in file               : file_name.cpp
;*      Loop source line                 : 209
;*      Loop opening brace source line   : 210
;*      Loop closing brace source line   : 288
;*      Loop Unroll Multiple             : 2x
;*      Known Minimum Iteration Count    : 1
;*      Known Max Iteration Count Factor : 1
;*      Loop Carried Dependency Bound(^) : 3
;*      Partitioned Resource Bound       : 8 (pre-sched)
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 8  Schedule found with 3 iterations in parallel
;*
;*      Partitioned Resource Bound(*)    : 8 (post-sched)
;*
;*      Resource Partition (may include "post-sched" split/spill moves):
;*
;*                                               A-side   B-side
;*      Bound(.C2)                                  -        1
;*      Bound(.P2)                                  -        0
;*      Bound(.D)                                   8*       -
;*      Bound(.M .N .MN)                            0        3
;*      Bound(.L .S .LS)                            3        3
;*      Bound(.L .S .C .LS .LSC)                    3        2
;*      Bound(.L .S .C .M .LS .LSC .LSCM)           2        3
;*      Bound(.L .S .C .M .D .LS .LSC .LSCM .LSCMD)  7        5
;*
;*      Done
;*
;*      Redundant loop generated.
;*        See the C7000 C/C++ Optimization Guide (spruiv4)
;*        for more information.
;*
;*      Minimum safe iteration count    : 3
;*----------------------------------------------------------------------------*
```

4.7.2.1 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。请注意，这些项目中的许多项目仅在使用了 --debug_software_pipeline 选项时出现，或仅在发生特定条件时出现。

- **循环展开因子。**专为提高性能而展开的循环的次数。
- **已知最小迭代计数。**循环执行的最小次数。
- **已知最大迭代计数。**循环执行的最大次数。
- **已知最大迭代计数因子。**始终会平均划分循环迭代计数的因子。此信息可用于在可能的情况下展开循环。
- **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。编写线性汇编时，使用这些信息帮助分配功能单元。每个表条目都有对于 A 侧和 B 侧功能单元的值。星号标记条目将决定资源限制值。这些条目表示以下术语：
  - **.L 单元**是仅需要 .L 单元的指令总数。
  - **.S 单元**是仅需要 .S 单元的指令总数。
  - **.M 单元**是仅需要 .M 单元的指令总数。
  - **.N 单元**是仅需要 .N 单元的指令总数。
  - **.D 单元**是仅需要 .D 单元的指令总数。
  - **.C 单元**是仅需要 .C 单元的指令总数。
  - **.P 单元**是仅需要 .P 单元的指令总数。
  - **.M/.N 单元**是可以使用 .M 或 .M 单元的指令总数。
  - **.L/.S 单元**是可以使用 .L 或 .S 单元的指令总数。
  - **.L/.S/.C 单元**是可以使用 .L、.S 或 .C 单元的指令总数。
  - **.L/.S/.C/.M 单元**是可以使用 .L、.S、.C 或 .M 单元的指令总数。
  - **.L/.S/.C/.M/.D 单元**是可以使用 .L、.S、.C、.M 或 .D 单元的指令总数。
  - **.X 交叉路径**是所需交叉路径的总数。
- **Bound(.C2)。**由可以使用 .C2 单元的指令数确定的资源限制。
- **Bound(.D1 .D2 .D)。**由可以使用 .D1 和 .D2 单元的指令数确定的资源限制。
- **Bound(.M .N .MN)。**由可以使用 .M 和 .N 单元的指令数确定的资源限制。
- **Bound(.P2)。**由可以使用 .P2 单元的指令数确定的资源限制。
- **Bound(.L .S .LS)。**由可以使用 .L 和 .S 单元的指令数确定的资源限制。
- **Bound(.L .S .C .LS .LSC)。**由可以使用 .L、.S 和 .C 单元的指令数确定的资源限制。
- **Bound(.L .S .C .M .LS .LSC .LSCM)。**由可以使用 .L、.S、.C 和 .M 单元的指令数确定的资源限制。
- **Bound(.L .S .C .M .D .LS .LSC .LSCM .LSCMD)。**由可以使用 .L、.S、.C、.M 和 .D 单元的指令数确定的资源限制。
- **最大加载推测量。**推测出该循环中加载的最大字节数将超出数组末尾（后面或前面）。此处无需用户操作，因为编译器将使用可安全推测的加载指令。

4.7.2.2 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
  - 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_speed=0 或 --opt_for_speed=1 选项时。
- **未初始化的迭代计数器**。迭代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_speed=2 选项，软件流水线可能被抑制。当使用 --opt_for_speed=2 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_speed=4 或 --opt_for_speed=5。
- **无法识别迭代计数器**。循环迭代计数器无法被识别或在循环体中使用不当。

4.7.2.3 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 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。

  - **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 > 最小或最大迭代计数**。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并行迭代计数。必须启用冗余循环或传递并行迭代信息。
- 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。编译器试图插入移动指令来分割比 ii 周期长的寄存器寿命，但可能并不总是成功。
- 太多的谓词活跃在一侧上 C7000 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。
- **循环中使用的迭代变量 - 不能调整迭代计数**。循环迭代计数器在循环中除了作为循环迭代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。

4.7.2.4 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。

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。

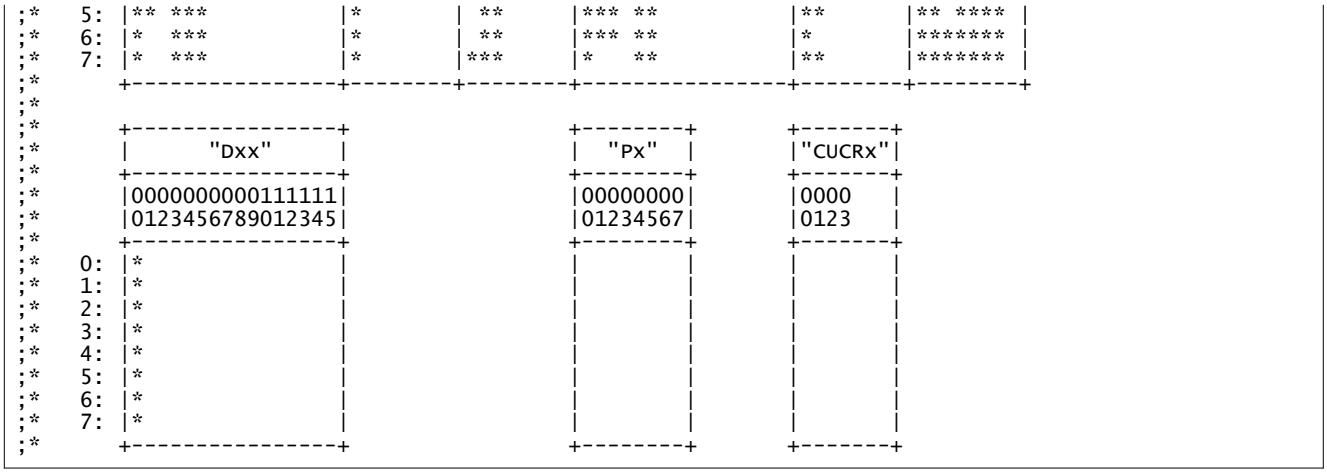
每行上的数字代表循环内核中的周期号。

每一列代表 C7000 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：

```
;*      Register Usage Tables:
;*
;*      +---------------------------------+---------------------------------+
;*      |               ASIDE             |               BSIDE             |
;*      +-----------------+--------+------+-----------------+--------+------+
;*      |      "Axx"      |  "ALx" | "AMx" |     "VBxx"      | "VBLx" | "VBMx" |
;*      +-----------------|--------|-------|-----------------|--------|--------+
;*      |0000000000111111|00000000|00000000|0000000000111111|00000000|00000000|
;*      |0123456789012345|01234567|01234567|0123456789012345|01234567|01234567|
;*      +-----------------|--------|-------|-----------------|--------|--------+
;*   0: |*  ***           |*       |***    |      *          |***     |******* |
;*   1: |*  ***           |**      | **    |     **          |***     |* ***** |
;*   2: |* ****           |*       |***    |   * **          |***     |*  **** |
;*   3: |******           |*       | **    |  ** **          |**      |*  **** |
;*   4: |** ***           |*       | **    |******           |***     |   **** |
```



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、A3、A4、A5、A6、AL0、AM0、AM1、AM2、VB5、VBL0、VBL1、VBL2、VBM0、VBM1、VBM2、VBM3、VBM4、VBM5、VBM6 和 D0 全都在这个周期内全都处于激活状态。

4.7.3 折叠逻辑程序和收尾程序以改善性能和代码大小

当循环为软件流水线时，通常需要逻辑程序和收尾程序。逻辑程序用于启动循环，收尾程序用于结束循环。

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。如果最小已知迭代计数太小，则会添加冗余循环或禁用软件流水线。折叠循环的逻辑程序和收尾程序可以减少安全执行流水线循环所需的最小迭代计数。

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。其余原因是由于逻辑程序和收尾程序。

软件流水线循环的逻辑程序和收尾程序最多由长度为 *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。在逻辑程序和收尾程序折叠期间，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_speed=2 或 --opt_for_speed=1 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。

4.7.3.1 推测执行

折叠逻辑程序和收尾程序通常可以减少最小安全迭代计数。如果最小已知迭代计数小于最小安全迭代计数，则需要一个冗余循环。否则，必须抑制流水线。当使用 --debug_software_pipeline 选项时，这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 Iteration Count: 1
....
;*Minimum safe iteration count: 7
```

4.8 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*迭代计数*。用于计算迭代次数的变量是*迭代计数器*。当迭代计数器达到等于迭代计数的限制时，循环终止。代码生成工具使用迭代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只需执行最少数量的循环迭代（最小迭代计数）即可装填流水线。

软件流水线循环的最小迭代计数由并行执行的迭代数量设置。在图 4-1 中，最小迭代计数为 5。在以下示例中，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迭代计数为 3。

```
A
B     A
C     B     A     ←三个并行迭代 = 最小迭代计数
      C     B
            C
```

当代码生成工具无法确定循环的迭代计数时，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，如果运行时迭代计数小于循环的最小安全迭代计数，则会执行该循环。第二个循环是软件流水线循环，如果运行时迭代计数大于或等于最小迭代计数，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 is the iteration count */
{
   for (I=0; I < N; I++) /* I is the iteration counter */
}
```

找到循环的软件流水线后，编译器会对 foo() 进行以下转换，并假定循环的最小迭代计数为 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++)   /* Unpipelined version */
   }
   else
   }
      for (I=0; I < N; I++)   /* Pipelined version   */
   }
}
foo(50); /* Execute software pipelined loop          */
foo(2);  /* Execute loop (unpipelined)*/
```

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 5.8.23）来帮助编译器避免产生冗余循环。

4.9 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 C7000 编译器接受；但是，它们会阻止优化器对代码进行全面优化。

4.9.1 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;
}
```

4.10 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。

4.11 使用性能建议优化代码

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，编译器可以执行更好的优化。在此类情况下，编译器可以通过发出“Advice”来提示采取某些措施提高性能。

默认情况下，编译器会发出诊断消息，建议您对源代码进行更改，以允许编译器执行更有效的优化。若要进一步控制性能建议的生成，请使用以下选项：

--advice:performance	指示编译器发送建议到 stdout 中（默认处于开启状态）。使用 **--advice:performance=none** 进行编译可完全禁止生成性能建议。
--advice:performance_file	指示编译器向文件发送建议。
--advice:performance_dir	指示编译器发送建议到特定目录中的文件中。
--diag_suppress	根据每次诊断禁用建议。此选项也可以在 FUNCTION_OPTIONS pragma 中使用（请参阅节 5.8.20）。

示例 1：下述命令行发送输出建议到名为 `myfile.adv` 的文件中：

```
cl7x --advice:performance_file=myfile.adv loop.c
```

示例 2：以下命令行显示了两种将输出建议发送到 `mydir` 子目录中名为 `myfile.adv` 的文件的方法：

```
cl7x --advice:performance_file=myfile.adv --advice:performance_dir=mydir loop.c
```

```
cl7x --advice:performance_file=mydir/myfile.adv loop.c
```

以下各小节介绍了支持的建议诊断。

4.11.1 建议 #35000：使用 restrict 提高循环性能

```
advice #35000-D: (Performance) Consider adding the restrict qualifier to the declaration of
"variable" to improve loop performance.
```

此建议确定了使用 `restrict` 限定函数参数的机会（如果这样做可能有助于提高循环性能）。确保通过 restrict 进行限定的指针不违反此类指针的任何限制；不能使用其他指针访问受限指针指向的对象。

若要查看有关使用 restrict 的更多信息，请参阅节 5.5.4

4.12 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。

4.13 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 4.7.1。

4.13.1 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）。

4.14 正在执行什么类型的优化？

C7000 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 4.14.1
别名消歧	节 4.14.2
分支优化和控制流简化	节 4.14.3
数据流优化 • 复制传播 • 通用子表达式消除 • 冗余分配消除	节 4.14.4
表达式简化	节 4.14.5
函数的内联扩展	节 4.14.6
函数符号别名	节 4.14.7
归纳变量和强度降低	节 4.14.8
循环不变量代码运动	节 4.14.9
循环旋转	节 4.14.10
循环折叠和循环合并	节 4.14.11
展开和阻塞	节 4.14.11
矢量化	节 4.14.13
指令调度	节 4.14.14
寄存器变量	节 4.14.15
寄存器跟踪/定位	节 4.14.16
软件流水线	节 4.14.17

4.14.1 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。

这种类型的优化始终处于启用状态。这对生成的应用的可调试性没有影响。

4.14.2 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。

4.14.3 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。

4.14.4 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。

4.14.5 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。

4.14.6 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 3.10。

这种类型的优化通过 --opt_level=0 和更高的优化设置来启用。

4.14.7 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 5.13.2。

4.14.8 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。

4.14.9 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。

这种类型的优化通过 --opt_level=2 和更高的优化设置来启用。这种优化可以提高性能，但会增加代码大小并降低可调试性。

4.14.10 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。这种优化可以提高性能，但会增加代码大小并降低可调试性。

4.14.11 循环折叠和循环合并

在某些情况下，当编译器认为组合循环嵌套（例如，“for”循环内的“for”循环）会提高性能时，则编译器会组合循环嵌套。

这些类型的优化通过 --opt_level=2 和更高的优化设置来启用。这些优化可以提高性能，但会增加代码大小并降低可调试性。

4.14.12 展开和阻塞

编译器可以展开包围最内部循环的外部循环。这种变换会对外部循环额外迭代一次，因此内部循环存在另一个副本。然后，第二个“内部循环”被“融合”回原始内部循环。因此，融合的内部循环对于内部循环的每次执行都会执行外部循环的两次迭代。这种转换称为“展开和阻塞”，可以提高可用的并行性和功能单元利用率。

如果编译器检测到内部循环中没有足够的可用并行性来有效利用 CPU 上的计算资源，则可以执行展开和阻塞。

如果 --opt_for_speed (-mf) 选项设置为级别 3 或更高（级别 4 是默认值）*并且* --opt_level (-o) 选项设置为除“off”之外的任何级别（如果 --vectypes=off，则 off 是默认值），则执行此类优化。这种优化可以提高性能，但会增加代码大小并降低可调试性。

4.14.13 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。

如果 --opt_for_speed (-mf) 选项设置为级别 2 或更高（级别 4 是默认值）*并且* --opt_level (-o) 选项设置为级别 2 或更高，则执行此类优化。这种优化可以提高性能，但会增加代码大小并降低可调试性。

4.14.14 指令排程

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

4.14.15 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。

4.14.16 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。

4.14.17 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 4.7。

这种类型的优化通过 --opt_level=2 和更高的优化设置来启用。

4.15 流引擎和流地址生成器

C7000 CPU 具有两个独特的功能：流引擎 (SE) 和流地址生成器 (SA)，这两项功能有助于将数据传送到 CPU 和计算寻址模式。

4.15.1 流引擎概述

流引擎 (SE) 在 CPU 功能单元和 L2 控制器之间提供高速连接。它们可以从 L2/L3/DDR 读取，但当用于从 L2 中读取时效率最高。

SE 是一种只读数据连接，可即时转发和格式化数据。SE 在保持寄存器中按每个 CPU 时钟的最高矢量宽度位显示数据。（C7000 的当前变体具有 512 位矢量宽度或 256 位矢量宽度。）此保持寄存器可与任何其他全局或本地寄存器一样使用。由于 SE 是只读引擎，因此流引擎寄存器可用作指令的输入，但指令无法写入寄存器。

每个 SE 都有一个矢量宽度保持寄存器。C7000 的当前变体有两个 SE 实例，分别名为 SE0 和 SE1。因此，有两个可用作指令输入的流引擎寄存器。

4.15.2 流引擎和流地址生成器的工作原理

在典型运行期间，流引擎 (SE) 和流地址生成器 (SA) 在概念上都建模为计算偏移量的七级循环嵌套。从以下模型可以观察到一些关键点：

- ICNT 值是 SE 或 SA 处理层级中的迭代次数。
- DIM 值是在每次迭代一个处理层级之间应用的偏移，以多个元素表示。这些值通常用于表示数据维度的长度。
- 处理下一个内部层级后，外部层级将复位偏移并添加 DIM 值。
- 在正常运行期间，所返回向量的元素必须相邻。此外，对于最内部的层级，对应于 ICNT0，处理是线性顺序的。元素在 VECLEN 元素的矢量大小块中返回。在最内部的层级处理每个向量后，SE 或 SA 会推进 VECLEN 元素。因此，在对 SE 或 SA 进行编程时，上述模型中最内部的两个循环共同处于最内部层级。

```
current_element_offset = 0;

/* ICNT5 programmed in the SE or SA template. */
for (i5 = 0; i5 < ICNT5; i5++)
{
    level5_offset = current_element_offset;
    /* ICNT4 programmed in the SE or SA template. */
    for (i4 = 0; i4 < ICNT4; i4++)
    {
        level4_offset = current_element_offset;
        /* ICNT3 programmed in the SE or SA template. */
        for (i3 = 0; i3 < ICNT3; i3++)
        {
            level3_offset = current_element_offset;
            /* ICNT2 programmed in the SE or SA template. */
            for (i2 = 0; i2 < ICNT2; i2++)
            {
                level2_offset = current_element_offset;
                /* ICNT1 programmed in the SE or SA template. */
                for (i1 = 0; i1 < ICNT1; i1++)
                {
                    level1_offset = current_element_offset;
                    /* ICNT0 programmed in the SE or SA template. */
                    /* VECLEN programmed in the SE or SA template. */
                    for (i0 = 0; i0 < ICNT0; i0 += VECLEN)
                    {
                        level0_offset = current_element_offset;
                        /* This inner loop demonstrates element processing of a vector.
                         * It is not considered a "level" of an SE or SA.
                         */
                        for (elem = 0; elem < VECLEN; elem++)
                        {
                            /* Element processing. For the SE, this builds a
                             * result vector element-by-element at the specified
                             * offset from the base address, including
                             * performing vector predication. For the SA, this
                             * builds a predicate.
                             */
```

```
                            process_element(current_element_offset);

                            /* DIM0 cannot be programmed. It is always 1. */
                            DIM0 = 1;
                            current_element_offset += DIM0;
                        }
                        /* Vector processing. For the SE, this yields the vector
                         * that was built in the SE register (e.g. SE0). For the
                         * SA, this yields the offset that the vector starts at
                         * in the SA register (e.g. SA0) and a vector predicate
                         * in the SA predicate register (e.g. PSA0).
                         */
                       process_return_vector_data(level0_offset);
                    }
                    /* DIM1 programmed in the SE or SA template. */
                    current_element_offset = level1_offset + DIM1;
                }
                /* DIM2 programmed in the SE or SA template. */
                current_element_offset = level2_offset + DIM2;
            }
            /* DIM3 programmed in the SE or SA template. */
            current_element_offset = level3_offset + DIM3;
        }
        /* DIM4 programmed in the SE or SA template. */
        current_element_offset = level4_offset + DIM4;
    }
    /* DIM5 programmed in the SE or SA template. */
    current_element_offset = level5_offset + DIM5;
}
```

4.15.3 流地址生成器概述

流地址生成器 (SA) 是一个多维偏移计算引擎。使用深度嵌套循环计算多维偏移需要花费 CPU 周期。当前的 C7000 型号最多提供四个地址生成器，分别名为 SA0、SA1、SA2 和 SA3，用于计算最多六个维度的偏移量。SA 可与加载/存储指令结合使用。

对 SA 进行编程类似于对 SE 进行编程。区别在于 SA 仅提供地址偏移，而 SE 提供实际数据。

4.15.4 使用流引擎和流地址生成器的优点

使用一个或多个流引擎和/或流地址生成器有几个优点：

- 这两种硬件特性都允许您为每个数据维度指定迭代计数 (ICNT) 和偏移量 (DIM)，最多可为六个维度。
- 这两种硬件特性都可以自动计算地址偏移量。编译器无需生成地址计算指令，这通常会降低循环性能。使用一个或多个流引擎可允许编译器对更多代码进行软件流水线处理，从而提高性能。
- 流引擎将数据从 L2 缓存上方预提取到靠近 CPU 的内存中。
- 流引擎提供了多种额外的数据格式化和模式访问功能。

4.15.5 用于流引擎和流地址生成器的接口

c7x_strm.h（作为 C7000 运行时支持的一部分包含在内，并由 c7x.h 自动包含）中定义了流引擎 (SE) 和流地址生成器 (SA) 内在函数接口。

如下所述，SE 和 SA 的基本使用依赖于在打开和使用 SE 或 SA 之前必须预先配置的参数模板。SE 和 SA 的模板都具有迭代计数 (ICNT)、偏移 (DIM) 和矢量长度 (VECLEN) 值。这些值用于在数据流被读取和推进时确定偏移量。

对于 SE，元素大小也直接配置到设置标志中，然后用在 SE 地址计算中。然而，SA 不是以这种方式工作的。对于 SA，偏移量根据使用 SA 的加载或存储指令的元素大小进行缩放。因此，允许具有不同元素大小的不同加载或存储指令有效地使用同一个 SA。

4.15.6 参数模板配置

参数模板是 __SE_TEMPLATE_v1 或 __SA_TEMPLATE_v1 类型的变量，用于指定计数器、大小和标志来控制流引擎 (SE) 和流地址生成器 (SA) 的行为。最多六层嵌套的迭代计数器 (ICNT) 和维度大小 (DIM) 适用于 SE 和 SA。功能标志专用于 SE 或 SA。参数模板实际上是 64 位值的矢量。

请注意，__SE_TEMPLATE_v1 和 __SA_TEMPLATE_v1 被定义为单独的类型。

要初始化 __SE_TEMPLATE_v1 类型的参数，请调用 __gen_SE_TEMPLATE_v1()。

```
__SE_TEMPLATE_v1 se_params = __gen_SE_TEMPLATE_v1();
```

要初始化 __SA_TEMPLATE_v1 类型的参数，请调用 __gen_SA_TEMPLATE_v1()。

```
__SA_TEMPLATE_v1 sa_params = __gen_SA_TEMPLATE_v1();
```

初始化后，使用赋值配置参数模板，如下所示：

- 为 0 级设置迭代计数器（维度总是暗指元素宽度）

```
    p.ICNT0 = <value>;
```

- 为 1-5 级设置迭代计数器和维度大小

```
    p.ICNT<n> = <value>;
    p.DIM<n> = <value>;
```

- 设置 DECDIM 宽度

```
    p.DECDIM<n>.WIDTH = <value>;
```

- 设置 LEZR 计数

```
    p.LEZR_CN = <value>;
```

例如，为具有四个维度的 SE 配置参数矢量：

```
// Setup Template Vector Based on Settings and Open the Stream
// Based on Iteration Counters and Dimensions (in Terms of # of Elems)
__SE_TEMPLATE_v1 params = __gen_SE_TEMPLATE_v1();
params.ICNT0 = 4;
params.ICNT1 = 2;
params.DIM1  = 4;
params.ICNT2 = 2;
params.DIM2  = 8;
params.ICNT3 = 4;
params.DIM3  = -16;
```

只要将 DIMFMT 标志设置为正确的格式标志，就不需要为未使用的维度配置计数器和维度大小。该标志通知硬件忽略这些未使用的字段中未初始化的数据。

对于流引擎，可以在使用 __SE_TEMPLATE_v1 类型定义的变量中配置以下附加标志。c7x_strm.h 文件中记录了这些标志。

```
ELETYPE
TRANSPOSE
PROMOTE
GRPDUP
VECLEN
ELDUP
DECIM
DIR
DIMFMT
DECDIM1
DECDIM2
LEZR
```

例如，以下语句使用 c7x_strm.h 中提供的常量来配置 SE 参数。

```
se_params.DIMFMT    = __SE_DIMFMT_4D;
se_params.DIR       = __SE_DIR_INC;
se_params.TRANSPOSE = __SE_TRANSPOSE_OFF;
se_params.DECIM     = __SE_DECIM_OFF;
se_params.VECLEN    = __SE_VECLEN_4ELEMS; // 4 ELEMENTS
se_params.ELETYPE   = __SE_ELETYPE_32BIT;
se_params.PROMOTE   = __SE_PROMOTE_OFF;
se_params.GRPDUP    = __SE_GRPDUP_OFF;
se_params.ELDUP     = __SE_ELEDUP_OFF;
se_params.DECDIM1   = __SE_DECDIM0;
se_params.DECDIM2   = __SE_DECDIM0;
se_params.LEZR      = __SE_LEZR_OFF;
```

对于流地址生成器，可以在使用 __SA_TEMPLATE_v1 类型定义的变量中配置以下附加标志。

```
VECLEN
DIMFMT
DECDIM1
DECDIM1SD
DECDIM2
DECDIM2SD
```

例如，以下语句使用 c7x_strm.h 中提供的常量来配置 SA 参数。

```
sa_params.VECLEN    = __SA_VECLEN_4ELEMS; // 4 ELEMENTS
sa_params.DIMFMT    = __SA_DIMFMT_4D;
sa_params.DECDIM1   = __SA_DECDIM0;
sa_params.DECDIM2   = __SA_DECDIM0;
```

4.15.7 使用流引擎

一旦配置了流引擎，就可以打开和使用相应的流。提供以下 API 方便此操作，它采用参数模板以及硬件应该从中获取流数据的起始内存地址：

- 打开流引擎 0： `__SE0_OPEN(void *addr, __SE_TEMPLATE_v1 param);`
- 打开流引擎 1： `__SE1_OPEN(void *addr, __SE_TEMPLATE_v1 param);`

使用以下 API 访问流引擎，该 API 将根据给定类型返回数据向量，用于转换流获取的数据类型：

- 读取流引擎 0（并推进）： `__SE0(type), __SE0ADV(type)`
- 读取流引擎 1（并推进）： `__SE1(type), __SE1ADV(type)`

备注

传递给流引擎访问 API 的类型必须精确指定为标量的 *<base_type>* 和向量的 *<base_type><length>*。*<base_type>* 必须是“char”、“uchar”、“cchar”、“short”、“ushort”、“cshort”、“int”、“uint”、“cint”、“float”、“cfloat”、“long”、“ulong”或“double”中的一个。例如，__SE0(int16)。指定的类型不能大于向量寄存器的大小。不允许使用 typedef 类型。有关从流引擎读取时 C++ 中的其他类型支持和模板支持，请参见节 5.16 中的“strm_eng”。

使用相应的关闭 API 来关闭流引擎：

- 关闭流引擎 0： `__SE0_CLOSE();`
- 关闭流引擎 1： `__SE1_CLOSE();`

具有四个维度的流的示例：

```
// OPEN STREAMING ENGINE 1 AT startaddr WITH PARAMETER TEMPLATE params
__SE1_OPEN((void*)startaddr, params);
// READ THE STREAM AND ADVANCE THE COUNTERS
for (I0 = 0; I0 < 8; I0++)
{
    uint8 Vout;
    Vout.lo = __SE1ADV(uint4);
    Vout.hi = __SE1ADV(uint4);
    Vresult += Vout;
}
// CLOSE THE STREAM
__SE1_CLOSE();
```

备注

从提供的存储器地址进行流式传输意味着一种契约：程序承诺始终不会修改流引擎在其生命周期内可以访问的任何存储器区域中的数据。

4.15.7.1 使用流引擎时进行硬编码的内在函数操作数

少数 C7000 指令（特别是一些 VFIR 和 VMATMPY 指令）*只能* 接受流引擎操作数，因为它们是硬连接到 SE。这包括针对单个 SE（例如 SE0）的访问以及同时针对一对 SE 流（例如 SE1:SE0）的访问。这些指令有一个特殊的内在函数接口，该接口是使用称为 SE_REG 或 SE_REG_PAIR 的枚举（必须与内在函数一起使用）定义的。由于这个接口，只能使用直接映射的低级别内在函数来访问这些指令。

- SE_REG

```
// Use the following for SE_REG operands.
enum SE_REG
{
    SE_REG_0     = 0, // READ SE0
    SE_REG_0_ADV = 1, // READ SE0 AND ADVANCE
    SE_REG_1     = 2, // READ SE1
    SE_REG_1_ADV = 3  // READ SE1 AND ADVANCE
};
```

- SE_REG_PAIR

```
// Use the following for SE_REG_PAIR operands.
enum SE_REG_PAIR
{
    SE_REG_PAIR_0     = 0, // READ SE0 AND SE1
    SE_REG_PAIR_0_ADV = 1  // READ SE0 AND SE1 AND ADVANCE BOTH STREAMS
};
```

4.15.8 使用流地址生成器

与 SE 类似，一旦配置了流地址生成器 (SA)，就可以打开和使用相应的 SA。提供了以下 API 来方便此操作，其将参数模板作为输入：

- 打开流地址生成器 0 (SA0)： `__SA0_OPEN(__SA_TEMPLATE_v1 param);`
- 打开流地址生成器 1 (SA1)： `__SA1_OPEN(__SA_TEMPLATE_v1 param);`
- 打开流地址生成器 2 (SA2)： `__SA2_OPEN(__SA_TEMPLATE_v1 param);`
- 打开流地址生成器 3 (SA3)： `__SA3_OPEN(__SA_TEMPLATE_v1 param);`

使用以下 API 访问流地址生成器。请注意，对于流地址生成器，基地址作为读取操作的输入给出，返回值是指向位置的指针，该位置随后被内存加载或被内存存储操作使用。这是因为，与 SE 不同，SA 只是提供一个添加到给定基地址的偏移量。给定类型用于转换通过 SA 加载或存储的数据的类型。

- 读取 SA0（并推进）： `__SA0(type, baseptr)`, `__SA0ADV(type, baseptr)`
- 读取 SA1（并推进）： `__SA1(type, baseptr)`, `__SA1ADV(type, baseptr)`
- 读取 SA2（并推进）： `__SA2(type, baseptr)`, `__SA2ADV(type, baseptr)`
- 读取 SA3（并推进）： `__SA3(type, baseptr)`, `__SA3ADV(type, baseptr)`

备注

为流地址生成器访问 API 指定的类型不能大于向量寄存器的大小。此外，对于非复数向量，元素类型不能大于 64 位。对于复数标量和复数向量，复数分量类型不能大于 64 位。（例如，cshort 的复数分量类型为 short。）允许使用 typedef 类型和限定类型。例如，以下条件都是合法的：

```
__SA0(int2, baseptr)
__SA0(__int2, baseptr)
__SA0(const __int2, baseptr)
```

有关从流地址生成器读取时 C++ 中支持的其他模板，请参见节 5.16 中的“strm_agen”。

使用相应的关闭 API 来关闭流地址生成器：

- 关闭 SA0： `__SA0_CLOSE();`
- 关闭 SA1： `__SA1_CLOSE();`
- 关闭 SA2： `__SA2_CLOSE();`
- 关闭 SA3： `__SA3_CLOSE();`

当上面显示的访问器 API 被自身使用时，编译器将为它们匹配基本的加载或存储操作，具体取决于它们是在赋值运算符的左侧 (LHS) 还是右侧 (RHS)。

以下是 SA 返回指针的一个示例，在 LHS 和 RHS 上对该示例进行解引用，以便将数据从一个位置复制到另一个位置。因为 SA 返回一个指针，所以在 RHS 上对其解引用会生成加载操作，而在 LHS 上对其解引用会生成存储操作。

```
// OPEN THE STREAMS, SA0 AND SA1
__SA0_OPEN(params);
__SA1_OPEN(params);
// COPY DATA FROM *(src_addr+SA1) TO *(dst_addr+SA0)
for (I0 = 0; I0 < 8; I0++)
{
    // COMPILER MATCHES VECTOR LOAD AND STORE FOR FOUR WORD
DATA
    *__SA0ADV(uint8, dst_addr) = *__SA1ADV(uint8, src_addr);
}
// CLOSE THE STREAMS
__SA0_CLOSE();
__SA1_CLOSE();
```

但是，SA 访问器 API 也可以作为加载或存储内在函数的输入，如下示例所示。在此示例中，由于 SA 只是返回一个未被解引用的指针，因此编译器不会试图匹配相应的基本向量加载或存储操作，而是使用内在函数指示的加载或存储操作。

```
// OPEN THE STREAMS, SA0 AND SA1
__SA0_OPEN(params);
__SA1_OPEN(params);
// COPY DATA (WITH UNPACK + PACK) FROM *(src_addr+SA1) TO *(dst_addr+SA0)
for (I0 = 0; I0 < 8; I0++)
{
   ulong8 data = __vload_unpack_long(__SA1ADV(uint8, src_addr));
   __vstore_packl(__SA0ADV(uint8, dst_addr), data);
}
// CLOSE THE STREAMS
__SA0_CLOSE();
__SA1_CLOSE();
```

备注

如果流地址生成器与标记为 `restrict` 的基址指针一起使用，则由 SA 计算的基于该指针的推导结果（基址 + 偏移量）也被认为是 `restrict`。这意味着 SA 生成的指针以及使用这些指针的所有加载和存储操作都不会被认为是其他存储器指针的别名。

因此，使用基于 `restrict` 基址指针的 SA 则约定：程序承诺始终不会修改流地址生成器在其生命周期内可以访问的任何存储器区域中的数据。

4.15.8.1 流地址生成器的矢量谓词

每个流地址生成器 (SA) 都有一个隐式谓词，该谓词是在存储到内存时根据有效矢量元素的数量进行设置的。然而，无法使用 C 语义表达这种行为，原因是编译器不能发现或跟踪硬件如何或何时进行断言，因为预测断言基于 SA 配置产生，并且可能因每次调用存储时而有所不同。

备注

所有 C7000 ISA 上的 SA 都可以配置为允许断言的 SA 存储。C7100 ISA 不支持断言的 SA 加载。然而，对于 C7120 和更高版本的 ISA，可以将 SA 配置为允许断言的 SA 加载。

为了在 C 中表达此功能，编译器提供了一个可供利用的显式语义框架。该框架使用从相应 SA 中提取的谓词值显式地断言存储操作。如果编译器检测到提取的谓词类型与 SA 访问器（以及相应的存储器操作）匹配，则编译器尝试将自己折叠起来以便利用 SA 的隐式断言功能。

将谓词值提取为 `__vpred` 类型值的 API 包括以下内容：

- 从 SA0 提取 VPRED 并根据指定类型进行缩放：`__SA0_VPRED(type)`
- 从 SA1 提取 VPRED 并根据指定类型进行缩放：`__SA1_VPRED(type)`
- 从 SA2 提取 VPRED 并根据指定类型进行缩放：`__SA2_VPRED(type)`
- 从 SA3 提取 VPRED 并根据指定类型进行缩放：`__SA3_VPRED(type)`

下述是这种用法的一个示例：

```
__SA0_OPEN(params);
for (I = 0; I < ((ROW * COL) / SIMD); I++)
{
    vpred sa0_vp = __SA0_VPRED(int16);             // EXTRACT SA0 PREDICATE
    int16 *addr  = __SA0ADV(int16, (int16 *)data); // EXTRACT SA0 ADDRESS
    __vstore_pred(sa0_vp, addr, data);             // USE PREDICATED INTRINSIC
}
__SA0_CLOSE();
```

备注

如此示例中所示，与特定流地址生成器关联的矢量谓词的提取必须在调用存储内在函数之外进行。如果不是这种情况，根据 C 规则，调用参数的评估顺序将是不明的。因此，执行以下语句：

```
__vstore_pred(__SA0_VPRED(int16), __SA0ADV(int16, (int16 *)data);
```

会导致未定义的行为，因为 SA0ADV 修改了 SA0_VPRED() 返回的值。

备注

无论在谓词提取 API 或 SA 访问中使用哪种类型，SA 推进量将始终根据其预配置的状态进行递增。给定的类型名并不影响如何进行 SA 推进。

4.15.8.2 断言与非断言的流地址存储和加载

可以将流地址生成器 (SA) 配置为导致生成包含 0 的 SA 矢量断言，此类断言可用于屏蔽加载或存储的组成部分。当以这种方式配置 SA 并且加载或存储基于 SA 时，除非在程序源代码中明确指定了矢量断言，否则不会指定矢量断言是否应用于操作。

以下定义适用于本节中 SA 和存储器操作的代码示例：

- *已断言：*操作的一部分*可能*会被屏蔽掉，这取决于 SA 配置。
- *未断言：*操作的一部分*将不会*被屏蔽掉，这取决于 SA 配置。

根据这些定义，断言的操作实际上是未断言操作的超集，因为所有未断言操作都可以表示为断言的操作。

如果指针 P1 的值源自另一个指针 P2，则 P1 定义为*基于* P2。例如，如果 `ptr = __SA0()`，则 `ptr` 基于 SA0。类似地，如果 `ptr1 = __SA0()` 且 `ptr2 = (ptr1 + 0)`，则 `ptr1` 和 `ptr2` 均基于 SA0。

下述 C 示例显示了各种存储和加载操作。对于每种操作类型，都会为需要未断言的操作或允许断言的操作的 SA 配置显示生成定义明确的行为或不明确行为的示例。

存储操作

- SA 配置为需要未断言的存储。（操作的某些部分将*不会*被屏蔽掉。）
  - 定义明确的示例：
    - ```
      *__SA0ADV(int16, baseptr) = data;              // Normal store
      ```
    - ```
      __vstore_packl(__SA0ADV(int8, baseptr), data); // Specialized store
      ```
    - ```
      int16 *ptr = __SA0ADV(int16, baseptr);
      *ptr = data;
      ```
  - 导致不明确行为的示例：

    无，因为 SA 配置为仅执行未断言的操作。下面所示的断言的存储操作在这一配置中也定义明确，尽管它们可能会更慢一些。
- SA 配置为允许断言的存储。（操作的某些部分可能会被屏蔽掉。）
  - 定义明确的示例：
    - ```
      __vpred vp = __SA0_VPRED(int16);
      int16 *ptr = __SA0ADV(int16, baseptr);
      __vstore_pred(vp, ptr, data);           // Normal store with explicit predication
      ```
    - ```
      __vpred vp = __SA0_VPRED(int8);
      int8 *ptr = __SA0ADV(int8, baseptr);
      __vstore_pred_packl(vp, ptr, data);     // Specialized store with explicit predication
      ```
  - 调用不明确行为的示例：
    - ```
      *__SA0ADV(int16, baseptr) = data;              // May be predicated
      ```
    - ```
      __vstore_packl(__SA0ADV(int8, baseptr), data); // May be predicated
      ```
    - ```
      int16 *ptr = __SA0ADV(int16, baseptr);
      *ptr = data;                                   // May be predicated
      ```

加载操作

- SA 配置为需要未断言的加载。（操作的某些部分将*不会*被屏蔽掉。）
  - 定义明确的示例：
    - ```
      int16 x = *__SA0ADV(int16, baseptr);                              // Normal load
      ```
    - ```
      ushort32 x = __vload_unpack_short(__SA0ADV(uchar32, baseptr); // Specialized load
      ```
    - ```
      int16 *ptr = __SA0ADV(int16, baseptr);
      int16 x    = *ptr;
      ```
  - 调用不明确行为的示例：

    无，因为 SA 配置为执行未断言的操作。下面所示的断言的加载操作在这一配置中也定义明确，尽管它们可能会更慢一些。
- SA 配置为允许断言的加载。（操作的某些部分可能会被屏蔽掉。）
  - 定义明确的示例：
    - ```
      __vpred vp = __SA0_VPRED(int16);
      int16 x = *__SA0ADV(int16, baseptr);                              // Normal load
      x = select(vp, x, (int16)(0));                                    // Apply predication
      ```
    - ```
      __vpred vp = __SA0_VPRED(int16);
      int16 *ptr = __SA0ADV(int16, baseptr);
      int16 x    = __vload_pred(vp, ptr);              // Normal load, explicit predication
      ```
    - ```
      __vpred vp = __SA0_VPRED(uchar32);
      uchar32 *ptr = __SA0ADV(uchar32, baseptr);
      ushort32 x = __vload_pred_unpack_short(vp, ptr); // Specialized load, explicit predication
      ```
  - 调用不明确行为的示例：
    - ```
      int16 x = *__SA0ADV(int16, baseptr);                              // May be predicated
      ```
    - ```
      ushort32 x = __vload_unpack_short(__SA0ADV(uchar32, baseptr); // May be predicated
      ```
    - ```
      int16 *ptr = __SA0ADV(int16, baseptr);
      int16 x = *ptr // May be predicated
      ```

4.15.9 自动使用流引擎和流地址生成器（--auto_stream 选项）

可以使用 --auto_stream 选项指示编译器自动使用流引擎 (SE) 和/或流地址生成器 (SA)。这种行为可以通过 --auto_stream 选项的以下设置来控制：

- **--auto_stream=off：**禁止自动使用 SE 和 SA。（C7100 和 C7120 的默认设置）
- **--auto_stream=saving：**允许自动使用 SE 和 SA 并保存上下文。如果进行函数调用时 SE 或 SA 可以打开，则使用此选项。使用此选项是安全的，但结果可能会比使用 --auto_stream=no_saving 时略慢，并且栈大小会增加。（C7504 和更高版本器件的默认设置）
- **--auto_stream=no_saving：**无需保存上下文即可自动使用 SE 和 SA。如果进行函数调用时 SE 或 SA 从未打开，则使用此选项。使用此选项的安全性不及使用 --auto_stream=saving，但结果会稍快，并且可以减小栈大小。

C7100 和 C7120 器件仅支持“off”和“no_saving”模式。使用 --auto_stream=no_saving 选项手动启用优化。这些器件不支持 SE 或 SA 上下文切换，因此默认情况下 --auto_stream=off。

较新的器件（例如 C7504）支持所有三种模式，并且 --auto_stream=saving 是默认设置。
      ```

启用 --auto_stream 后，内存访问在嵌套循环中转换，这些循环具有适合 SE 或 SA 配置模板的寻址模式。例如，假设您有以下代码：

```
void example1(char *in, char *restrict out, int len1, int len2)
{
    for (int i = 0; i < len1; i++)
        for (int j = 0; j < len2; j++)
            out[i*len1 + j] = in[i*len1 + j];
}
```

启用 --auto_stream 后，此代码在矢量化后转换为相当于 C7504 器件上的以下 SE 配置：

```
__SE_TEMPLATE_v1 tmplt = __gen_SE_TEMPLATE_v1();
tmplt.ICNT0 = 32;
tmplt.ICNT1 = (len2>>5)+((len2&0x1f) != 0);
tmplt.DIM1 = 32;
tmplt.ICNT2 = len1;
tmplt.DIM2 = len1;
tmplt.VECLEN = __SE_VECLEN_32ELEMS;
tmplt.DIMFMT = __SE_DIMFMT_3D;
```

有关如何对流编程的详细信息，请参阅节 4.15.6。

4.15.9.1 自动使用 SE 和 SA 的正确性

与上一节中的示例不同，以下示例未进行转换，因为 `len1` 和 `len2` 可能会超过 SE 的 32 位字段的大小，并且循环计数器可能会超过 32 位值：

```
void example2(char *in, char *restrict out, long len1, long len2)
{
    for (long i = 0; i < len1; i++)
        for (long j = 0; j < len2; j++)
            out[i*len1 + j] = in[i*len1 + j];
}
```

实际上，这种寻址模式几乎总是映射到流，尽管仍有可能出现边缘情况。此类情况包括但不限于：

- ICNT 值超出无符号 32 位范围。
- DIM 值超出有符号 32 位范围。
- 寻址中的加法或乘法超出有符号 32 位范围。
- 寻址超出 INT_MIN 到 INT_MAX 个元素的范围。

有关如何对流编程的详细信息，请参阅节 4.15.6。

使用 --assume_addresses_ok_for_stream 选项可让编译器忽略诸如此类边缘情况。使用此选项可使 `example2` 以与 `example1` 相同的方式进行转换。

如果在 SE 或 SA 打开并进行函数调用时使用了 --auto_stream=no_saving 选项，则可能生成不正确的代码。在这种情况下，如果编译器自动使用 SE 或 SA，则已打开的 SE 或 SA 的状态将丢失。

如果使用了 --auto_stream 且配置了 L1D 并将其用作 SRAM，也可能生成不正确的代码。在这种情况下，尝试使用 SE 访问 L1D 会失败。

4.15.9.2 对 SE 和 SA 的自动使用进行调优

只有当编译器认为在循环或循环嵌套中进行变换有益（这主要与循环迭代计数相关）时，才会自动使用 SE 和 SA。因此，使用 PROB_ITERATE 和 MUST_ITERATE pragma 有助于指导此转换。有关这些 pragma 的详细信息，请参阅节 5.8.23 和节 5.8.30。

此外，如果函数中已使用 SE 或 SA，则编译器不会使用 SE 或 SA。

要逐个函数控制自动 SE 和 SA 的行为，请使用 FUNCTION_OPTIONS pragma。以下 C++ 示例控制 C7100 器件上 pragma 之后函数的自动 SE 和 SA 行为。对于 C 代码，请将函数名称添加为 pragma 的第一个参数。有关 FUNCTION_OPTIONS pragma 的更多信息，请参阅节 5.8.20。

```
#pragma FUNCTION_OPTIONS("--auto_stream=no_saving --assume_addresses_ok_for_stream")
```

4.16 嵌套循环控制器 (NLC)

为了更好地促进嵌套循环 *合并*，即包含指令的外循环级别合并到内循环级别，C7000 架构提供了一个被称为嵌套循环控制器 (NLC) 的功能。此功能实现了不只一层循环嵌套的硬件循环控制。这允许编译器合并循环级别，同时预测内循环中的外循环指令执行情况。这样减少了循环控制开销，并让软件流水线循环更好地利用函数单元资源。循环合并会使编译器在合法且有利可图的情况尝试自动执行该操作。有利性由启发算法决定。

有两个 pragma 允许您显式启用或禁用专用嵌套循环的合并。在 C/C++语言中，这些 pragma 只能应用于循环，并且必须出现在循环结构的前面：

```
#pragma COALESCE_LOOP
#pragma NO_COALESCE LOOP
```

4.16.1 可能限制使用 NLC 的障碍

如上所述，编译器在尝试合并循环之前必须检查其合法性。如果编译器不能保证内循环至少执行一次，即使您禁用了有利性启发法，也不会合并循环。

为了通知编译器某个循环总是会被执行，请在循环体之前使用 MUST_ITERATE pragma：

```
#pragma MUST_ITERATE(1,65535,)
```

此 pragma 告知编译器该循环至少执行一次且不超过 65,535 次。

章节 5

C/C++ 语言实现

受 C7000 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 C7000 支持的 C++ 语言由 ANSI/ISO/IEC 14882:2014 标准定义，但有一些例外。

5.1 C7000 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 5.13）。

在支持 C89 的默认宽松 ANSI 模式下，编译器支持 C99 的某些功能。它支持 C99 模式下 C99 的所有语言功能。请参阅节 5.12。

*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 int（32 位）。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 5.4，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 规范中的某些运行时函数和功能。请参阅节 5.12。

5.1.1 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 5.4 (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 5.4。(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 5.3。(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 5.4。（6.4.4.4，5.1.1.2）
- 编译器目前仅支持一个区域设置“C”。(6.4.4.4)
- 编译器目前仅支持一个区域设置“C”。(6.4.5)

J.3.5 整数

- 未提供扩展整数类型。(6.2.5)
- 有符号整数类型的负值用 2 的补码表示，没有陷阱表示。(6.2.6.2)
- 未提供扩展整数类型，因此整数秩没有变化。(6.3.1.1)
- 当整数转换为不能代表该值的有符号整数类型时，通过丢弃不能存储在目标类型中的位，该值会被截断（不发出信号）；最低位不会被修改。(6.3.1.3)
- 有符号整数值的右移执行算术（有符号）移位。除右移以外的按位操作对位的操作方式与对无符号值的操作方式完全相同。即，在通常的算术转换之后，执行按位运算而不考虑整数类型的格式，尤其是符号位。(6.5)

J.3.6 浮点

- 浮点运算 (+ - * /) 的精度是精确到位的。未指定返回浮点结果的库函数的准确性。(5.2.4.2.2)
- 编译器不会为 FLT_ROUNDS 提供非标准值。(5.2.4.2.2)
- 编译器不会为 FLT_EVAL_METHOD 提供非标准负值。(5.2.4.2.2)
- 整数转换为浮点数时的舍入方向是 IEEE-754“舍入到最近”。(6.3.1.4)
- 浮点数转换为更窄浮点数时的舍入方向是 IEEE-754“舍入到偶数”。(6.3.1.5)
- 对于不能准确表示的浮点常量，实现方案使用最接近的可表示值。(6.4.4.2)
- 编译器不会收缩浮点表达式。(6.5)

- FENV_ACCESS pragma 的默认状态为“关闭”。(7.6.1)
- TI 编译器不会定义任何额外的浮点异常。（7.6，7.12）
- FP_CONTRACT pragma 的默认状态为“关闭”。(7.12.2)
- 如果舍入结果等于数学结果，则不会产生“不精确”的浮点异常。(F.9)
- 如果结果很小但不是不精确，则不会产生“下溢”和“不精确”的浮点异常。(F.9)

J.3.7 数组和指针

- 当将 64 位指针转换为 32 位整数时，指针将被截断。在将指针转换为 long 时，指针被视为相同大小的 unsigned long，并且适用普通整数转换规则。
- 在将指针转换为长整型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 5.3 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 3.10.2。(6.7.4)

J.3.9 结构体、联合体、枚举和位字段

- “plain”整数位字段会被视为有符号整数位字段。（6.7.2，6.7.2.1）
- 除了 _Bool、signed int 和 unsigned int，编译器还允许将 char、signed char、unsigned char、signed short、unsigned short、signed long、unsigned long、 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 6.2.2。(6.7.2.1)
- 结构体的非位字段成员按照节 6.2.1 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 5.3.1 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 5.8。(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 5.12.1 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 5.3 中进行了介绍。（5.2.4.2，7.18.2，7.18.3）
- 节 6.2.1 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 6.2.1。(6.5.3.4)

5.2 C7000 C++ 的特性

根据 ANSI/ISO/IEC 14882:2014 标准 (C++14) 中的定义，C7000 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
- 例外情况，通过 --exceptions 选项启用；请参阅节 5.6。
- 运行时类型信息 (RTTI) 无法禁用。虚拟类的元数据引用类的 type_info 对象，以便在运行时确定类的实际类型。

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）。

5.3 数据类型

表 5-1 列出了 C7000 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 6.2.1 中介绍了数据类型的存储和对齐。

表 5-1. C7000 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	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、wchar_t	32 位	二进制	0	4 294 967 295
long、signed long	64 位	二进制	-9 223 372 036 854 775 808	9 223 372 036 854 775 808
unsigned long	64 位	二进制	0	18 446 744 073 709 551 615
long long、signed long long	64 位	二进制	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 位	二进制	0	18 446 744 073 709 551 615
enum (1)	不尽相同	二进制	不尽相同	不尽相同
float	32 位	IEEE 32 位	1.175 494e-38(2)	3.40 282 346e+38
float complex	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(2)	1.79 769 313e+308
double complex	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(2)	1.79 769 313e+308
long double complex	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	64 位	二进制	0	0xFFFFFFFFFFFFFFFF

(1) 有关枚举类型大小的详细信息，请参阅节 5.3.1。
(2) 数字是最低精度。

有符号类型的负值用 2 的补码表示。

备注

当不同设备和编译器之间实现可移植性需要特定的数据类型大小时，建议代码使用 C 标准整数类型 `int64_t` 和 `int32_t`（等）。这些标准整数类型是在 `stdint.h` 中定义的，其作为 C 标准库支持的一部分被包含在运行时支持库中。

C7000 采用 LP64 代表性惯例。也就是说，类型 `int` 是 32 位，而 `long` 和 `pointer` 类型是 64 位。

C、C99 和 C++ 的这些附加类型被定义为标准类型的同义词：

表 5-2. C 语言标准类型

类型	定义
size_t	unsigned long
ptrdiff_t	long
wchar_t	unsigned int

表 5-2. C 语言标准类型（续）

类型	定义
wint_t	int
va_list	char *

5.3.1 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) 均必须具有适合“int”类型的值；不过，作为扩展，即使在 C 模式下，也可以使用大于“整数”的枚举常量。

对于枚举类型，编译器选择下述列表中第一个足够大且符号正确的类型来表示所有枚举常量值：

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

会跳过“long long”类型，因为其与“long”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。

5.3.2 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、16、32 或 64。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。

对向量数据类型的支持默认启用。如需禁用前缀*没有*双下划线的向量数量类型，可以使用 --vectypes=off 编译器选项。 例如，`__int4` 类型始终可用，但 --vectypes=off 选项会禁用 `int4` 类型。请注意，以下各表和示例使用的是没有双下划线前缀的类型名称。

C7000 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“c7x.h”头文件中。

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 C7000 实现与 OpenCL C 语言规范有所相似。C7000 编程模型提供下述内置向量数据类型：

表 5-3. 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。	64
uchar*n*	由 *n* 个 8 位无符号整数值组成的矢量。	64
short*n*	由 *n* 个 16 位有符号整数值组成的矢量。	32
ushort*n*	由 *n* 个 16 位无符号整数值组成的矢量。	32
int*n*	由 *n* 个 32 位有符号整数值组成的矢量。	16
uint*n*	由 *n* 个 32 位无符号整数值组成的矢量。	16
long*n*	由 *n* 个 64 位有符号整数值组成的矢量。	8
ulong*n*	由 *n* 个 64 位无符号整数值组成的矢量。	8
float*n*	由 *n* 个 32 位单精度浮点值组成的矢量。	16
double*n*	由 *n* 个 64 位双精度浮点值组成的矢量。	8
bool*n*	由 *n* 个 8 位无符号整数布尔值组成的向量。	64

n 为矢量长度 2、3、4、8、16、32 或 64。

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。long2 总大小为 128 位，与 8 字节边界对齐。

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 5-4. 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。	32
cshort*n*	由 *n* 对 16 位有符号整数值组成的矢量。	16
cint*n*	由 *n* 对 32 位有符号整数值组成的矢量。	8
clong*n*	由 *n* 对 64 位有符号整数值组成的矢量。	4
cfloat*n*	由 *n* 对 32 位浮点值组成的矢量。	8
cdouble*n*	由 *n* 对 64 位浮点值组成的矢量。	4

n 为矢量长度 1、2、4、8、16 或 32。请注意，64 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。

备注

矢量无法传递到 variadic 函数 (stdarg.h)，也无法传递到 printf()。

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 5.14。

5.4 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。

5.5 关键字

C7000 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 __cregister。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

5.5.1 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;
```

5.5.2 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 const 限定符应用于任何变量或数组的定义，以确保其值不被更改。

限定为 常量的全局对象放置在 .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。（不允许程序修改常量 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常量关键字时相同。

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

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

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

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

5.5.3 __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 5-5）。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。

以下控制寄存器在 c7x.h 头文件中声明。此外，大量扩展控制寄存器 (ECR) 在 c7x_ecr.h 中声明。

表 5-5. C7000 的控制寄存器

类别	寄存器	说明
常规控制寄存器	CPUID	CPU ID 寄存器
	PMR	电源管理寄存器
	DNUM	DSP 内核数寄存器
	TSC	时间戳计数器寄存器
	TSR	任务状态寄存器
	RP	返回指针寄存器
	BPCR	分支预测器控制寄存器
	STSC	影子时间戳计数器寄存器
计算控制寄存器	FPCR	浮点控制寄存器
	FSR	标志状态寄存器
	GPLY	伽罗瓦多项式寄存器
	GFPGFR	伽罗瓦域多项式发生器函数寄存器
事件控制寄存器	DEPR	调试事件优先级寄存器
	IESET	内部异常事件集寄存器
	ESTP_SS	事件服务表指针寄存器，安全监控器
	ESTP_S	事件服务表指针寄存器，监控器
	ESTP_GS	事件服务表指针寄存器，来宾监控器
	ECSP_SS	事件上下文保存指针寄存器，安全监控器
	ECSP_S	事件上下文保存指针寄存器，监控器
	ECSP_GS	事件上下文保存指针寄存器，来宾监控器
	TCSP	任务上下文保存指针
	RXMR_SS	返回执行模式寄存器，安全监控器
	RXMR_S	返回执行模式寄存器，监控器
	AHPEE	启用最高优先级的事件寄存器，当前正在使用
	PHPEE	启用最高优先级的事件寄存器，挂起
	IEBER	内部事件广播启用寄存器
	IERR	内部异常报告寄存器
	IEAR	内部异常地址寄存器
	IESR	内部异常状态寄存器
	IEDR	内部异常数据寄存器
	TCR	测试计数寄存器
	TCCR	测试计数配置寄存器
	GMER	来宾模式启用寄存器
	UMER	用户掩码启用寄存器
	SPBR	栈指针边界寄存器
	UFCMR	用户标志清除掩码寄存器
	IPE	处理器间事件寄存器
查找表和直方图控制寄存器	LTBR0 至 LTBR3	查找表基址寄存器

表 5-5. C7000 的控制寄存器（续）

类别	寄存器	说明
	LTCR0 至 LTCR3	查找表配置寄存器
	LTER	查找表启用寄存器
调试控制寄存器	DBGCTXT	调试上下文（覆盖）寄存器
	ILCNT	内环计数器寄存器
	OLCNT	外环计数器初始值寄存器
	LCNTFLG	16 位谓词标志寄存器
	SCRB	记分板位寄存器

__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 5-5 中的控制寄存器，必须按如下方式声明每个寄存器。c7x.h 头文件通过以下语法定义所有控制寄存器：

extern __cregister volatile unsigned int *register* **;**

完成对寄存器的声明后，就可以直接使用寄存器名称。

示例 5-1. 定义和使用控制寄存器

```
extern __cregister volatile unsigned long __CPUID;
int main()
{
    printf("__CPUID = 0x%lx\n", __CPUID);
}
```

5.5.3.1 在进行浮点运算后评估标志状态寄存器 (FSR) 中的标志

标志状态寄存器 (FSR) 包含表示浮点状态的位，可以使用 `__get_FSR(type)` API（在 `c7x.h` 中定义）来访问。该 API 采用 *type* 参数，后者是指会与浮点运算一同使用的有效标量或矢量浮点类型（“float3”除外）。

对于所有相关矢量通道，该 API 会返回“OR”形式的数据位。结果是一个包含以下字段的 8 位值：

- 位 7：SAT - C7000 运算不受支持
- 位 6：UNORD - 比较结果是无序的浮点标志
- 位 5：DEN - 源是反常浮点标志
- 位 4：INEX - 结果是不精确的浮点标志
- 位 3：UNDER - 结果是下溢浮点标志
- 位 2：OVER - 结果是溢出浮点标志
- 位 1：DIV0 - 被零除浮点标志
- 位 0：INVAL - 无效运算浮点标志

例如：

```
float4 a = ... ;
float4 b = ... ;
float4 c = a * b;
uint8_t fsr_val = __get_FSR(float4);
```

提供 `__get_FSR(type)` API 是为了方便访问 FSR。实际硬件寄存器是一个 64 位的值，分为八个 8 位块。每个 8 位块对应于输入或输出数据中的 64 位数据矢量切片，具体取决于正在执行的操作。一个 64 位切片可能由一个 64 位双精度值或两个 32 位浮点值组成，这些值由硬件进行“OR”运算。

但是，对于矢量运算，虽然对每个 64 位切片都执行此“OR”运算，但硬件不会对所有 64 位切片的结果进行“OR”运算。这样做的原因是当使用部分矢量（小于 512 位）时，矢量的上部通道被视为无效并被忽略，因此不应反映在最终的 FSR 结果中。为确保仅反映与有效矢量通道相关的信息，API 允许用户指定他们正在处理的数据的标量或矢量类型。然后，API 将确保只有有效的 64 位矢量切片通过一系列指令进行“OR”运算，以产生最终的 8 位结果。因此，所有无效通道都会被忽略。

备注

使用 `__get_FSR(type)` API 会导致性能降低。这是因为，该 API 会插入一系列指令以确保只有有效的矢量通道会反映在最终结果中。该 API 还可防止在使用它的整个函数中进行循环矢量化，因为矢量化会以用户无法跟踪的方式更改有效矢量通道的数量。

5.5.4 restrict 关键字

为了帮助编译器确定内存依赖关系，可以使用 restrict 关键字来限定指针、引用或数组。restrict 关键字是一个类型限定符，可以应用于指针、引用和数组。使用它表示用户保证，在指针声明的范围内，指向的对象只能由该指针访问。任何违反此保证的行为都会导致程序未定义。这种做法可以帮助编译器优化某些代码段，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 5.12。

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;
  }
}
```

5.5.5 volatile 关键字

C/C++ 编译器在所有模式下都支持 *volatile* 关键字，但 。此外，在 C89、C99 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、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;
        }
    }
}
```

5.6 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 11.3.1.1。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 7.1。

5.7 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 6.3。

5.8 pragma 指令

以下 pragma 指令会告知编译器如何处理某个函数、对象或代码段。

- CALLS (请参阅节 5.8.1)
- CLINK (请参阅节 5.8.2)
- CODE_ALIGN (请参阅节 5.8.4)
- CODE_SECTION (请参阅节 5.8.5)
- COALESCE_LOOP (请参阅节 5.8.3)
- DATA_ALIGN (请参阅节 5.8.6)
- DATA_MEM_BANK (请参阅节 5.8.7)
- DATA_SECTION (请参阅节 5.8.8)
- Diag_suppress、diag_remark、diag_warning、diag_error、diag_default、diag_push、diag_pop (请参阅节 5.8.9)
- FORCEINLINE (请参阅节 5.8.10)
- FORCEINLINE_RECURSIVE (请参阅节 5.8.11)
- FUNC_ALWAYS_INLINE (请参阅节 5.8.12)
- FUNC_CANNOT_INLINE (请参阅节 5.8.13)
- FUNC_EXT_CALLED (请参阅节 5.8.14)
- FUNC_IS_PURE (请参阅节 5.8.15)
- FUNC_IS_SYSTEM (请参阅节 5.8.16)
- FUNC_NEVER_RETURNS (请参阅节 5.8.17)
- FUNC_NO_GLOBAL_ASG (请参阅节 5.8.18)
- FUNC_NO_IND_ASG (请参阅节 5.8.19)
- FUNCTION_OPTIONS (请参阅节 5.8.20)
- INTERRUPT (请参阅节 5.8.21)
- LOCATION (请参阅节 5.8.22)
- MUST_ITERATE (请参阅节 5.8.23)
- NOINIT (请参阅节 5.8.24)
- NOINLINE (请参阅节 5.8.25)
- NO_COALESCE_LOOP (请参阅节 5.8.26)
- NO_HOOKS (请参阅节 5.8.27)
- once (请参阅节 5.8.28)
- pack (请参阅节 5.8.29)
- PERSISTENT (请参阅节 5.8.24)
- PROB_ITERATE (请参阅节 5.8.30)
- RETAIN (请参阅节 5.8.31)
- SET_CODE_SECTION (请参阅节 5.8.32)
- SET_DATA_SECTION (请参阅节 5.8.32)
- STRUCT_ALIGN (请参阅节 5.8.33)
- UNROLL (请参阅节 5.8.34)
- WEAK (请参阅节 5.8.35)

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。

5.8.1 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《*汇编语言工具用户指南*》中的节 13.1。

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 5.13.2）具有与 CALLS pragma 相同的效果，语法如下：

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

5.8.2 CLINK Pragma

备注

CLINK pragma 没有效果，因为默认情况下启用了条件链接。此处之所以对此进行记录，是因为使用此 pragma 会生成 .clink 指令。有关条件链接的更多信息，请参阅节 11.2.1。

CLINK pragma 可以应用于代码或数据符号。它会导致在包含符号定义的段中生成 .clink 指令。.clink 指令告诉链接器在条件链接过程中段有资格被删除。因此，如果段未被正在编译和链接的应用中的任何其他段引用，它将不会包含在生成的输出文件中。

C 中 pragma 的语法为：

#pragma CLINK (*symbol* **)**

C++ 中 pragma 的语法为：

#pragma CLINK

RETAIN pragma 与 CLINK pragma 的效果相反。有关更多详细信息，请参阅节 5.8.31。

5.8.3 COALESCE_LOOP Pragma

为了促进嵌套循环合并（包含指令的外部循环级别合并到内部循环级别），C7000 架构提供了一种被称为嵌套循环控制器 (NLC) 的功能。此功能实现了不超过一层循环嵌套的硬件循环控制。这允许编译器合并循环级别，同时预测内循环中的外循环指令执行情况。这样可以减少循环控制开销，并让软件流水线循环更好地利用函数单元资源。如果循环合并的操作合规且有利，则编译器会尝试自动执行该操作。有利性由启发法确定。

COALESCE_LOOP pragma 会明确为该 pragma 之后的嵌套循环结构启用合并。此 pragma 只能应用于循环，并且必须紧接在 C/C++ 中的循环结构之前出现：

该 pragma 在 C 和 C++ 中的语法为：

```
#pragma COALESCE_LOOP
```

如果编译器不能保证至少执行一次内循环，即使使用了 COALESCE_LOOP pragma，也不会合并循环。若要告知编译器将始终执行某个循环，请在循环体之前使用 MUST_ITERATE pragma。例如，以下 pragma 告知编译器该循环至少执行一次且不超过 65,535 次。

```
#pragma MUST_ITERATE(1,65535,)
```

5.8.4 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 5.13.2。

C 中 pragma 的语法为：

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

C++ 中 pragma 的语法为：

```
#pragma CODE_ALIGN ( constant )
```

5.8.5 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 5.13.2。

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;
}
```

5.8.6 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 5.13.4。

C 中 pragma 的语法为：

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

C++ 中 pragma 的语法为：

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

5.8.7 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。C7000 器件包含 16 个 64 位存储器组。*constant* 的值可以为 0、1、2、3、4、5、6、7、8、9、10、11、12、13、14 或 15。

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，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“n”会导致起始地址的最后七位的值等于“n*8”。例如，对于值 1，起始地址的最后七位将为 0x08。对于值 14，起始地址的最后七位将为 0x70。

示例 5-2 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 5-2. 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;
   for (i = 0; i < 100; i++)
   {
      w[i] = my_count++;
      x[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  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:  0x0000cba0
address of x:  0x0000c610
address of z:  0x0000cc80
address of zz: 0x0000cab0
```

5.8.8 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 或 .data 段的区域，此 pragma 很有用。

使用 DATA_SECTION pragma 与使用 GCC 样式的 `section` 变量属性的效果相同。请参阅节 5.13.4。

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 5-3. 使用 DATA_SECTION Pragma C 源文件

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

示例 5-4. 使用 DATA_SECTION Pragma C++ 源文件

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

5.8.9 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。

5.8.10 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 3.10。

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。

5.8.11 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 3.10。

5.8.12 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 3.10。

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 5.13.2。

C 中 pragma 的语法为：

#pragma FUNC_ALWAYS_INLINE (*func*)

C++ 中 pragma 的语法为：

#pragma FUNC_ALWAYS_INLINE

下述示例使用此 pragma：

```
#pragma FUNC_ALWAYS_INLINE(functionThatMustGetInlined)
static inline void functionThatMustGetInlined(void) {
    P1OUT |= 0x01;
    P1OUT &= ~0x01;
}
```

备注

使用 FUNC_ALWAYS_INLINE Pragma 时要小心：FUNC_ALWAYS_INLINE pragma 会覆盖编译器的内联决策。过度使用此 pragma 会导致编译时间或内存使用量增加，可能会耗尽所有可用存储器，并导致编译工具失效。

5.8.13 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 3.10。

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 5.13.2。

C 中 pragma 的语法为：

#pragma FUNC_CANNOT_INLINE (*func*)

C++ 中 pragma 的语法为：

#pragma FUNC_CANNOT_INLINE

5.8.14 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++ 函数可能从外部而不是通过 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。

5.8.15 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
```

5.8.16 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
```

5.8.17 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
```

5.8.18 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
```

5.8.19 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
```

5.8.20 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、--auto_stream、--assume_addresses_ok_for_stream、--diag_suppress、--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。

以下示例使用此 pragma 为使用 C 的单个函数设置优化级别：

```
#pragma FUNCTION_OPTIONS(myfunc, "--opt_level=3 --opt_for_speed=4")
```

以下示例使用此 pragma 为使用 C++ 的单个函数设置优化级别：

```
#pragma FUNCTION_OPTIONS("--opt_level=3 --opt_for_speed=4")
myfunc {
    ...
}
```

5.8.21 INTERRUPT Pragma

借助 INTERRUPT pragma，您可以使用 C 代码来直接处理中断。

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。

5.8.22 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 5.13.2。

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 5.13.4）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 5.8.24。

5.8.23 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。

5.8.23.1 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迭代计数。迭代计数是指循环的迭代次数。循环的迭代计数必须能够被 *multiple* 整除。所有参数都是可选的。例如，如果迭代计数可以为 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，如果迭代计数可以为 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，如果迭代计数不能被 multiple 整除，程序的结果会为 undefined。另外，如果迭代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];
    }
    ...
}
```

5.8.23.2 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 < iteration_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 可用于指定迭代计数的范围，以及迭代计数的系数。下述示例会告知编译器，循环执行 8 次到 48 次，并且 iteration_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 < iteration_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。

5.8.24 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 5.13.4。

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;
```

5.8.25 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 3.10。

5.8.26 NO_COALESCE_LOOP Pragma

为了促进嵌套循环合并（将包含指令的外部循环级别合并到内部循环级别），C7000 架构提供了一种被称为嵌套循环控制器 (NLC) 的功能。此功能实现了针对不超过一层的循环嵌套的硬件循环控制。这使得编译器能够合并循环级别，同时预测内循环中的外循环指令执行情况。这样可以减少循环控制开销，并让软件流水线循环更好地利用函数单元资源。如果循环合并的操作合规且有利，则编译器会尝试自动执行该操作。有利性由启发法确定。

NO_COALESCE_LOOP pragma 会明确为该 pragma 之后的嵌套循环结构禁用合并。此 pragma 只能应用于循环，并且必须紧接在 C/C++ 中的循环结构之前出现：

该 pragma 在 C 和 C++ 中的语法为：

```
#pragma NO_COALESCE_LOOP
```

5.8.27 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 3.13。

5.8.28 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。

5.8.29 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 5.13.5。

5.8.30 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。没有可用的 C 属性语法。有关使用类似语法的示例，请参阅节 5.8.23.1。

[[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迭代计数。迭代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迭代计数（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迭代计数（例如 10 次），则该 pragma 与以下所示类似：

```
#pragma PROB_ITERATE(, 10)  /* Note the blank field for min */
```

5.8.31 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 5.13.2 和节 5.13.4。有关条件链接的更多信息，请参阅节 11.2.1。

C 中 pragma 的语法为：

#pragma RETAIN (*symbol* **)**

C++ 中 pragma 的语法为：

#pragma RETAIN

5.8.32 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。

5.8.33 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 5.13.5 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。

5.8.34 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 5.8.23.1。

[[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。

5.8.35 WEAK Pragma

WEAK pragma 用于针对符号提供弱绑定。

C 中 pragma 的语法为：

```
#pragma WEAK ( symbol )
```

C++ 中 pragma 的语法为：

```
#pragma WEAK
```

如果 *symbol* 为引用，WEAK pragma 会使它成为弱引用；如果为定义，则会使它成为弱定义。符号可以是数据或函数变量。实际上，未解析的弱 *引用* 不会导致链接器错误，在运行时也没有任何效果。以下适用于弱引用：

- 不会搜索库来解析弱引用。弱引用保持未解析状态并不是错误。
- 在链接期间，未定义弱引用的值为：
  - 零（如果重定位类型为绝对地址）
  - 位置地址（如果重定位类型为 PC 相对地址）
  - 标称基地址的地址（如果重定位类型为基址相对地址）

弱 *定义* 不会更改从库中选择目标文件所遵循的规则。不过，如果链接集同时包含弱定义和非弱定义，则始终会使用非弱定义。

WEAK pragma 具有与使用 `weak` 函数或变量属性相同的效果。请分别参阅节 5.13.2 和节 5.13.4。

5.9 _Pragma 运算符

C7000 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。

5.10 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。

C7000 编译器和链接器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。

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 6.7.2 中描述了有关 EABI 模式的 TI 特定信息。

有关 C7000 EABI 的详细信息，请参阅《*C7000 嵌入式应用程序二进制接口 (EABI)*》参考指南 (SPRUIG4)。

5.11 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 14，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

5.12 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 5.12.1。）
- ANSI/ISO C11（--c11，请参阅节 5.12.2）

- ISO C++14 (--c++14 ，用于所有 C++ 文件 ，请参阅节 5.2。)

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 5.12.3。)

5.12.1 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

5.12.2 C11 支持 (--c11)

编译器支持 ISO 标准化的 2011 年标准 C。但是，除了节 5.12.1 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

5.12.3 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 inline 关键字语言异常，然后，inline 可用作代码中的标识符。

```
int main()
{
    int inline = 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 int16 类型扩展，并接受此代码。

```
extern int16 myFunc(void);
int main()
{
    return 0;
}
```

以下代码在所有模式下均被接受。__int16 类型与 ANSI/ISO C 标准不冲突，因此始终可以作为一种语言扩展。

```
extern __int16 myFunc(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 5.13 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。

5.13 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 5.13.6）
- __has_attribute

5.13.1 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 5-6 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 5-6. 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
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

表 5-6. GCC 语言扩展（续）

扩展	说明
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__ 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 5.13.6）
矢量扩展	使用矢量运算（请参阅节 5.14）
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

5.13.2 函数属性

支持以下 GCC 函数属性：

- alias
- aligned
- always_inline
- calls
- const
- constructor
- deprecated
- format
- format_arg
- malloc
- noinline
- noreturn
- pure
- section
- unused
- used
- visibility
- warn_unused_result
- weak

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 5.8.4。

always_inline 函数属性与 FUNC_ALWAYS_INLINE pragma 具有相同的效果。请参阅 节 5.8.12

calls 属性与 CALLS pragma 具有相同的效果，相关描述请参阅 节 5.8.1。

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。

malloc 属性应用于 stdlib.h 中 malloc、calloc、realloc 和 memalign 的声明。

noinline 函数属性与 FUNC_CANNOT_INLINE pragma 具有相同的效果。请参阅 节 5.8.13

retain 属性与 RETAIN pragma (节 5.8.31) 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 5.8.5

weak 属性与 WEAK pragma (节 5.8.35) 具有相同的效果。

5.13.3 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 5.8.23.1。

5.13.4 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
- weak

在变量上使用的 **aligned** 属性与 DATA_ALIGN pragma 的效果相同。请参阅 节 5.8.6

location 属性与 LOCATION pragma 的效果相同。请参阅节 5.8.22。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《

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 5.8.24。

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 5.8.31) 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 5.8.8

used 属性在 GCC 4.2 中定义（请参阅 http://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/Variable-Attributes.html#Variable-Attributes ）。

weak 属性与 WEAK pragma (节 5.8.35) 的效果相同。

5.13.5 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。

“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*。

5.13.6 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，。

__builtin_return_address() 函数始终返回零。

5.14 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 5.3.2 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。默认情况下，向量数据类型处于启用状态。您可以使用 --vectypes 编译器选项来禁用不以双下划线开头的向量数据类型名称。

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 5.14.1)、一元和二元运算符 (节 5.14.2)、用于组件访问的混合运算符 (节 5.14.4)。

备注

向量无法传递到 variadic 函数 (stdarg.h)，也无法传递到 printf()。

5.14.1 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。在初始化每个元素不具有相同初始值的向量时，应使用构造函数初始化 idiom，如下例所示。

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。在这种情况下，可以是“constructor”习语或“cast/scalar-widening”习语。

```
ushort4 myushort4a = ushort4(1); // constructor syntax (preferred)
ushort4 myushort4b = (ushort4)1; // cast/scalar-widening syntax
```

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);
}
```

5.14.2 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。

备注

目前，布尔向量仅支持赋值运算符 (=)。

表 5-7. 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 5-8. 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;
    ...
}
```

5.14.3 向量的三态运算符 (?:)

C7000 编译器支持向量类型的三态运算符，但是生成的代码将不是最优的，结果将逐通道进行评估。如果 src1 和 src2 是向量，请考虑以下示例：

```
int16 ex_ternary(int16 src1, int16 src2)
{
    return (src1 > src2) ? (src1 - src2) : (src2 - src1);
}
```

为了生成更优化的代码，您应该改用 c7x.h 运行时支持头文件中列出的向量比较内在函数来构造向量谓词，以及向量选择内在函数 __select(vpred, ...)。节 5.14.8 中对向量谓语进行了介绍。

或者，使用条件运算，例如 _add (vpred, ...) 和 _sub_cond (vpred, ...)，来实现与以下示例中所示相同的行为。

```
int16 ex_ternary_supported(int16 src1, int16 src2)
{
    vpred condition =__cmp_gt_pred(src1, src2);
    int16 result1 = src1 - src2; // if-clause
    int16 result2 = src2 - src1; // else-clause
    return __select(condition, result1, result2); // Lane-dependent select operation
}
```

5.14.4 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;   */
```

.s[0], .s[1], ..., .s[63] 访问向量中多达 64 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.s[10] * ucvec8.s[7];
/* ".s[10]" is 11th element of ucvec16;
 * ".s[7]" 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;
```

5.14.5 不受支持的矢量比较运算符

C7000 编译器中不支持将矢量作为 `if` 语句中的条件来进行比较。例如，如果 `src1` 和 `src2` 是矢量，那么以下用法将不受支持：

```
int ex_compare_unsupported(short16 src1, short16 src2)
{
    if (src1 > src2) { return 1; } // Unsupported, will not work properly
    else             { return 2; }
}
```

相反，应当使用 `c7x.h` 运行时支持头文件中列出的矢量比较内在函数来构造矢量谓词。节 5.14.8 中描述了矢量谓词。

5.14.6 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));
    ...
}
```

5.14.7 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。

针对不会导致每个元素恰好包含 0x0 或 0x1 的布尔向量，未定义重新解释。下面显示了已定义或未定义结果的示例：

```
ushort2 myshort2_0 = ushort2(0,1);
bool4 mybool4_0 = as_bool4(myshort2_0); // Defined

ushort2 myshort2_1 = ushort2(2,3);
bool4 mybool4_1 = as_bool4(myshort2_1); // Undefined

bool8 mybool8_0 = bool8(0,1,0,1,0,1,0,1);
float2 myfloat2_0 = as_float2(mybool8_0); // Defined

float2 myfloat2_1 = float2(1.0,2.0);
bool8 mybool8_1 = as_bool8(myfloat2_1); // Undefined
```

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 long 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）。mylong 的最低有效 32 位放在 fltvec2.s0 中，而 mylong 的最高有效 32 位放在 fltvec2.s1 中。不执行算术转换。

```
extern long mylong;
float2 fltvec2 = as_float2(mylong);
```

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);
```

5.14.8 矢量谓词类型

矢量谓词功能通过名为 `__vpred` 的特殊不透明类型管理。向量谓词值的每个位均对应 TI 向量类型中的一个字节通道。由于矢量大小上限为 64 字节，矢量谓词值大小为 64 位。

5.14.8.1 构造向量谓词类型

类型为 `__vpred` 的值只能使用生成向量谓词的内在函数来构造，这些内在函数列在 `c7x_vpred.h` 运行时支持头文件中，包括：

- 向量比较内在函数：`__cmp_{eq, ge, gt, le, lt}_pred(…)`
  注意：部分向量（大小小于 512 位）之间的比较将产生一个向量谓词值，在该值中，上层无效通道被屏蔽为 false。
- 向量谓词掩码内在函数：`__mask_{char, short, int, long}`
- 对其他向量谓词进行操作的向量谓词指令：`__negate(__vpred src)`

5.14.8.2 使用向量谓词类型

构建的 __vpred 类型向量谓词值可作为操作数，用于所示的任何内在函数。依赖向量谓词类型的内在函数列于 c7x_vpred.h 中。可执行有关向量谓词的其他运算 — 谓词加、谓词减、谓词转移和谓词存储。可使用向量选择 __select(vpred, ...) 内在函数将大多数操作变为可根据向量判断的操作：

```
int16 ex_compare_and_select(int16 src1, int16 src2)
{
    vpred condition = __cmp_gt_pred(src1, src2);
    int16 result1 = src1 - src2; // if-clause
    int16 result2 = src2 - src1; // else-clause
    return __select(condition, result1, result2);
}
```

备注

部分元素谓词是未定义的行为：如果向量谓词用于控制向量运算，则该运算的每个元素的谓词位将被编译器假定为全 1 或全 0。部分谓词运算元素，使该运算元素的谓词位并非全部为 1 或全部为 0，被视为未定义的行为。

5.14.8.3 布尔向量类型

布尔向量是 C/C++ 源文件中的一种向量数据类型。节 5.3.2 中对向量数据类型进行了介绍。目前，布尔向量允许字面量和串联运算符 (节 5.14.1)、赋值运算符 (节 5.14.2)、向量的混合运算符 (节 5.14.4)、向量的转换函数 (节 5.14.6) 以及向量的重新解释函数 (节 5.14.7)。

与其他整型向量类型不同，布尔向量不能用作向量三元运算符的条件。布尔向量类型当前不支持标准布尔运算，例如 &&、||、&、&=、|、|=、^、^=、!、~、==、!=、<=、<、>=、>、<=、<、>= 和 >。

布尔向量可用作 C7000 上大多数谓词内在函数低级别向量谓词类型 (节 5.14.8) 的抽象替代。建议使用布尔向量作为向量谓词。但是，布尔向量类型不能与低级别向量谓词类型完全互换。

使用向量谓词类型时，应对谓词执行适当的缩放。通过内在函数 __expand_vpred(__vpred, k) 和 __pack_vpred(__vpred, k)，可以将向量谓词按系数 *k*={0-63} 放大或缩小。

以下示例中的两个函数实现了相同的结果，一个具有布尔向量，另一个具有低级别向量谓词类型。这显示了向量谓词与布尔向量的功能差异。布尔向量谓词按通道输入数据，与元素类型无关。

```
// Boolean vector example
void foo(int4 *ptr, int4 data, char4 *ptr2, char4 data2)
{
    bool4 pred = bool4(0,1,1,0);
    __vstore_pred(pred, ptr, data);   // Word-based store
    __vstore_pred(pred, ptr2, data2); // Byte-based store
}

// Vector predicate example
void bar(int4 *ptr, int4 data, char4 *ptr2, char4 data2)
{
    __vpred pred = _mvrp(0x0000000000000ff0); // Word-scaled predicate
    __vstore_pred(pred, ptr, data);

    pred = __pack_vpred(pred, 2); // Byte-scaled predicate
    __vstore_pred(pred, ptr2, data2);
}
```

5.15 C7000 内在函数

C7000 编译器提供了内在函数，为 ISA 指令和例程提供了简单 C/C++ 运算符无法利用的功能。

为 C7000 ISA 定义的大多数指令都可以通过内在函数获得，这些内在函数列在 `c7x.h` 运行时支持头文件中。这些内在函数细分为以下使用类别：

- 高级别过载内在函数 (节 5.15.1)
- 为特殊加载或存储指令定义的内在函数 (节 5.15.2)
- 低级别、直接映射的内在函数 (节 5.15.3)
- 用于执行查找表和直方图操作的内在函数 (节 5.15.4)
- 用于迁移为 C6000 编译器编写的代码的传统内在函数 (节 5.15.6)
- 用于控制流引擎和流地址生成器的内在函数（请参阅节 4.15）

5.15.1 高级别过载内在函数

C7000 编译器提供的第一组内在函数与应用多种标量和矢量类型的运算有关。因此，对于它们应用的每个操作，这些内在函数具有相同的名称并根据输入类型进行重载。这些内在函数属于重载函数，因此在支持的所有内在函数中是最抽象也是级别最高的。

例如，C7000 ISA 为字节矢量（最多 64 个元素的 VABSB）、半字矢量（最多 32 个元素的 VABSH）、字矢量（最多 16 个元素的 VABSW）、双字矢量（最多 8 个元素的 VABSD）、单精度浮点矢量（最多 16 个元素的 VABSSP）和双精度浮点矢量（最多 8 个元素的 VABSDP）定义了一组绝对值指令。尽管存在这种可变性，但在进行所有运算时都使用相同的内在函数名称，并且仅根据输入操作数类型来区分运算。这包含在 `c7x.h` 中，如下所示：

```
/*---------------------------------------------------------------------------
 * ID: __abs
 *---------------------------------------------------------------------------*/
VABSB
char = __abs(char);
char2 = __abs(char2);
char3 = __abs(char3);
char4 = __abs(char4);
char8 = __abs(char8);
char16 = __abs(char16);
char32 = __abs(char32);
char64 = __abs(char64);
VABSH
short = __abs(short);
short2 = __abs(short2);
short3 = __abs(short3);
short4 = __abs(short4);
short8 = __abs(short8);
short16 = __abs(short16);
short32 = __abs(short32);
VABSW
int = __abs(int);
int2 = __abs(int2);
int3 = __abs(int3);
int4 = __abs(int4);
int8 = __abs(int8);
int16 = __abs(int16);
VABSD
long = __abs(long);
long2 = __abs(long2);
long3 = __abs(long3);
long4 = __abs(long4);
long8 = __abs(long8);
VABSSP
float = __abs(float);
float2 = __abs(float2);
float3 = __abs(float3);
float4 = __abs(float4);
float8 = __abs(float8);
float16 = __abs(float16);
VABSDP
```

```
double = __abs(double);
double2 = __abs(double2);
double3 = __abs(double3);
double4 = __abs(double4);
double8 = __abs(double8);
```

只要使用内在函数名称（在本例中为 __abs(…)），编译器就会为所使用的输入类型生成正确的对应指令。如需完整列表，请参阅 c7x.h。

5.15.2 为特殊加载和存储指令定义的内在函数

C7000 ISA 支持使用简单的 C/C++ 运算符无法利用的多种加载和存储运算。相反，为那些对加载和存储分别使用前缀“__vload_”和“__vstore_”的运算提供了一个过载内在函数。为以下加载和存储运算提供了重载内在函数：

- 向量加载和重复：__vload_dup(…)
- 向量加载和重复组：__vload_dup_vec(…)
- 向量加载和解包：__vload_unpack_{short, int, long}(…)
- 向量加载和取消交错：__vload_deinterleave_{int, long}(...)
- 向量交错和存储：__vstore_interleave(…)
- 向量打包存储：__vstore_{packl, packh, packhs1, pack_byte}(…)
- 向量反向位存储：__vstore_reverse_bit(…)
- 向量谓词存储：__vstore_pred(vpred, …)
- 向量交错，谓词存储：__vstore_pred_interleave(vpred, …)
- 向量谓词打包存储：
  __vstore_pred_{packl, packh, packhs1, pack_byte}(vpred, ...)
- 向量谓词反向位存储：__vstore_pred_reverse_bit(vpred, …)
- 向量常数存储：__vstore_const_{2word, 4word, 8word, 16word}(…)
- 存储向量谓词：__store_predicate_{char, short, int, long}(…)
- 原子交换：__atomic_swap(…)
- 原子比较和交换：__atomic_compare_swap(…)

5.15.3 直接映射的内在函数

还提供了内在函数，以促进内在函数与相应 C7000 指令之间的直接映射。因此，这些内在函数*不会过载*；它们是通过从硬件中进行少量抽象化而得出的，并且被认为是*低级别*。这些低级别内在函数的主要目的是确保不会生成除程序员所需指令以外的其他指令。这对于需要在输入时操作数*交错*或在输出时操作数*去交错*的操作特别有用。

所有直接映射的内在函数都列在顶层 `c7x.h` 文件所含的 `c7x_direct.h` 中。

例如，C7000 指令 VCMATMPYHW（向量复数矩阵乘法）要求其第二个源操作数沿 64 位边界交错。它还要求其输出也沿 64 位边界去交错。如 `c7x.h` 和 `c7x_direct.h` 所列，C7000 编译器为此指令提供了两个接口：

- 高级别内在函数：

```
/*----------------------------------------------------------------------------
 * ID: __cmatmpy_ext
 *----------------------------------------------------------------------------*/
/*
VCMATMPYHW
cint2 = __cmatmpy_ext(cshort2, cshort4);
cint4 = __cmatmpy_ext(cshort4, cshort8);
cint8 = __cmatmpy_ext(cshort8, cshort16);
*/
```

- 低级别内在函数：

```
/*----------------------------------------------------------------------------
 * ID: __vcmatmpyhw_vww
 *----------------------------------------------------------------------------*/
/*
VCMATMPYHW
__vcmatmpyhw_vww(cshort16, cshort16, cshort16, cint8&, cint8&);
*/
```

如果选择使用过载的“__cmatmpy_ext(…)”内在函数，编译器将假定输入和输出数据都是*非交错的*，并将尝试对此进行抽象处理。因此，编译器将在指令执行之前插入特殊指令以交错输入，并在指令执行之后插入特殊指令以去交错输出。*这种方法便于程序员使用，但却以牺牲指令周期为代价。*

更高级的编程器可能会选择使用直接映射的低级别“__vcmatmpyhw_vww(…)”内在函数，并自行管理交错和去交错。在这种情况下，交错输入显示为一对 `cshort16` 向量，输出显示为一对 `cint8` 向量，每个向量由 VCMATMPYHW 指令支持的最大宽度和基本类型确定。

5.15.4 查找表和直方图内在函数

`c7x_luthist.h` 中列出了用于配置和使用 C7000 查找表和直方图特征的内在函数，该文件包含在 C7000 运行时支持中。

5.15.5 矩阵乘法加速器 (MMA) 内在函数

`c7x_mma.h` 中列出了用于配置和使用矩阵乘法加速器 (MMA) 的内在函数，该文件包含在 C7000 运行时支持中。

5.15.6 传统内在函数

C7000 运行时支持定义了可用于使用 C7000 编译器编译 C6000 源代码的 C6000 传统内在函数。应根据《*C6000 到 C7000 迁移用户指南*》(SPRUIG5) 以及 `c6x_migration.h` 运行时支持头文件中提供的信息，完成包含这些内在函数的源代码编译。

5.16 C7000 可扩展矢量编程

编译器库中提供了一组实用程序，用于为 C7000 编写独立于矢量宽度的代码。要使用这些实用程序，请在源代码中 #include `c7x_scalable.h`。

这些实用程序仅可在 C++ 代码中使用，因为它们的实现中使用了 C++ 语言功能。

使用 TI C7000 编译器或使用 TI C7000 主机仿真进行编译时，可使用这些实用程序。

以下 API 可用，c7x_scalable.h 文件中对所有这些 API 进行了更详细的描述：

- 矢量类型查询和构造

```
c7x::max_simd<T>::value
c7x::element_count_of<T>::value
c7x::element_type_of<T>::type
c7x::component_type_of<T>::type
c7x::make_vector<T,N>::type
c7x::make_full_vector<T>::type
c7x::is_target_vector<T>::value
```

- 完整矢量类型

```
c7x::char_vec
 c7x::short_vec
 etc
```

- 半向量类型

```
c7x::char_hvec
c7x::short_hvec
etc
```

- 四分之一向量类型

```
c7x::char_qvec
c7x::short_qvec
etc
```

- 指针的主机仿真兼容类型

```
c7x::char_vec_ptr
c7x::const_short_vec_ptr
etc
```

- 模板化矢量重新解释和转换

```
c7x::reinterpret<T>(v)
c7x::convert<T>(v)
```

- 矢量重新解释和转换

```
c7x::as_char_vec(v)
c7x::convert_short_vec(v)etc
```

- 流引擎和流地址生成器帮助器

```
c7x::se_veclen<T>::value
c7x::se_eletype<T>::value
c7x::sa_veclen<T>::value
c7x::strm_eng<I,T>::get()
c7x::strm_eng<I,T>::get_adv()
c7x::strm_agen<I,T>::get(p)
c7x::strm_agen<I,T>::get_adv(p)
c7x::strm_agen<I,T>::get_vpred()
```

以下宏由 c7x_mma.h 定义，可用于确定有关 MMA 与可扩展矢量编程模型配合使用的信息：

表 5-9. 与 MMA 和可扩展矢量编程配合使用的宏

宏语法	说明
__MMA_A_MAT_BYTES__	A 矩阵的大小（以字节为单位）。目前，每个 A 矩阵包含一行。
__MMA_A_ROW_WIDTH_BYTES__	A 矩阵中一行的大小（以字节为单位）。
__MMA_A_ROWS__	A 矩阵中的行数。
__MMA_A_COLS(ebytes)	给定 A 矩阵各元素中字节数时该矩阵中的列数。通常对 sizeof() 有用。例如，__MMA_A_COLS(sizeof(short))。

表 5-9. 与 MMA 和可扩展矢量编程配合使用的宏 （续）

宏语法	说明
__MMA_A_ENTRIES__	可以包含在 A 存储中的 A 条目数。
__MMA_B_MAT_BYTES__	B 矩阵的大小（以字节为单位）。
__MMA_B_ROW_WIDTH_BYTES__	B 矩阵中一行的大小（以字节为单位）。
__MMA_B_ROWS(ebytes)	给定 B 矩阵各元素中的字节数时该矩阵中的行数。通常对 sizeof() 有用。例如，`__MMA_B_ROWS(sizeof(short))`。
__MMA_B_COLS(ebytes)	给定 B 矩阵各元素中的字节数时该矩阵中的列数。通常对 sizeof() 有用。例如，`__MMA_B_COLS(sizeof(short))`。
__MMA_C_MAT_BYTES__	C 矩阵的大小。目前，每个 C 矩阵包含一行。目前，对于较大的累加器，C 矩阵比 A 矩阵宽 4 倍。
__MMA_C_ROW_WIDTH_BYTES__	C 矩阵中行的大小。
__MMA_C_ROWS__	C 矩阵中的行数。
__MMA_C_COLS(ebytes)	给定 C 矩阵各元素中的字节数时该矩阵中的列数。通常对 sizeof() 有用。例如，`__MMA_C_COLS(sizeof(short))`。
__MMA_C_ENTRIES__	可以包含在 C 存储中的 C 条目数。

作为一个中等复杂度的示例，以下是使用输入类型作为模板的 memcpy 的 C++ 函数模板的实现。此示例使用流引擎和流地址生成器（请参阅节 4.15）。

```
#include <c7x_scalable.h>

using namespace c7x;

/* memcpy_scalable_strm<typename S>(const S*in, S *out, int len)
 *
 * S   - A basic data type such as short or float.
 * in  - The input buffer.
 * out - The output buffer.
 * len - The number of elements to copy.
 *
 * Defaulted template arguments:
 * V   - A full vector type of S
 */
template<typename S,
         typename V  = typename make_full_vector<S>::type>
void memcpy_scalable_strm(const S *restrict in, S *restrict out, int len)
{
    /*
     * Find the maximum number of vector loads/stores needed to copy the buffer,
     * including any remainder.
     */
    int cnt = len / element_count_of<V>::value;
    cnt += (len % element_count_of<V>::value > 0);

    /* Initialize the SE for a linear read in and the SA for a linear write out. */
    __SE_TEMPLATE_v1 in_tmplt = __gen_SE_TEMPLATE_v1();
    __SA_TEMPLATE_v1 out_tmplt = __gen_SA_TEMPLATE_v1();

    in_tmplt.VECLEN = se_veclen<V>::value;
    in_tmplt.ELETYPE = se_eletype<V>::value;
    in_tmplt.ICNT0 = len;

    out_tmplt.VECLEN = sa_veclen<V>::value;
    out_tmplt.ICNT0 = len;

    __SE0_OPEN(in, in_tmplt);
    __SA0_OPEN(out_tmplt);

    /* Perform the copy. If there is remainder, the last store will be predicated. */
    int i;
    for (i = 0; i < cnt; i++)
    {
        V tmp = strm_eng<0, V>::get_adv();
        __vpred pred = strm_agen<0, V>::get_vpred();
        V *addr = strm_agen<0, V>::get_adv(out);
        __vstore_pred(pred, addr, tmp);
    }

    __SE0_CLOSE();
    __SA0_CLOSE();
}
```

章节 6

运行时环境

本章介绍 C7000 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。

6.1 存储器

C7000 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 48 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）。

C7000 编译器要求所有代码和数据都必须位于 2GB 的虚拟地址空间内。使用链接器命令文件将代码和数据放置在此 2GB 虚拟地址区域内。C7000 编译器生成的代码使用位置无关寻址来获取函数的地址（在某些情况下）并访问存储器中静态分配的数据。使用与位置无关的寻址的指令范围有限。

有关链接器命令文件的信息可以在节 12.5 中找到。有关更多信息，请参阅 *C7000 嵌入式应用二进制接口 (EABI) 参考指南* (SPRUIG4)，特别是关于“计算代码地址”和“数据分配和寻址”的章节。

6.1.1 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 章节 8 中介绍的目标文件信息。有关 C7000 段名称的详细信息，请参阅 *C7000 嵌入式应用程序二进制接口 (EABI) 参考指南* (SPRUIG4)。

段有两种基本的类型：

- **未初始化的段**会在存储器中保留空间（通常为 RAM）。程序可以在运行时使用此空间来创建和存储变量。编译器会创建以下未初始化的段：
  - **.bss 段**为未初始化的全局变量和静态变量保留空间。这些变量由汇编器分配。
  - **.common 段**为未初始化的全局变量和静态变量保留空间。这些变量由链接器分配。未使用且未初始化的变量通常创建为通用符号（除非指定了 --common=off），以便将该变量从生成的应用中排除。
  - **.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，编译器则不会创建 .sysmem 段。
- **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 节 12.8.4.2。
  - 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。
  - **.got 段**包含全局偏移量表。此段是可写的。
  - **.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。
  - **.data 段**为初始化的非常量全局变量和静态变量保留空间。此段是可写的。
  - **.c7xabi.exidx 段**包含用于异常处理的索引表。**.c7xabi.extab 段**包含用于异常处理的堆栈展开指令。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，请参阅节 12.8。
  - **.const 段**包含字符串文字、浮点常量和使用 C/C++ 限定符 *const* 定义的数据（前提是该常量未定义为 *volatile* 或 节 5.5.2 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。
  - **.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。

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、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 5.8.5 和节 5.8.8）。

6.1.2 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 D15 寄存器来管理此堆栈。D15 是指向堆栈下一个未使用位置的*堆栈指针* (SP)。

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 0x2000 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，请参阅节 12.4。

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，即 16 个字节。SP 是 8 字节对齐的，因此大多数 64 位和更小的对象不会跨越大小为 64 位的内存库边界，并且按照惯例，SP 将始终指向一个空闲的 16 字节位置以便优化函数调用中的堆栈使用。

当所需的空间大于为每个栈帧保留的 16 个字节时，C/C++ 环境在函数入口自动递减 SP，以保留执行该函数所需的所有空间。

更多有关堆栈和堆栈指针的信息，请参阅节 6.4。

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 3.13。

6.1.3 动态存储器分配

C7000 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 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 节 12.4。

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。

6.2 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。

6.2.1 数据类型存储

对于基本（标量）类型的对象，其最小对齐方式取决于其所属类型的大小。对于具有数组类型的对象，其最小对齐方式是由其元素类型指定的。

有关数据类型的一般信息，请参阅节 5.3。表 6-1 列出了各种数据类型的寄存器存储空间和存储器存储空间：

表 6-1. 寄存器和存储器中的数据表示

数据类型	寄存器存储	存储器存储
char	寄存器的位 0-7	8 位，与 8 位边界对齐
unsigned char	寄存器的位 0-7	8 位，与 8 位边界对齐
short	寄存器的位 0-15	16 位，与 16 位边界对齐
unsigned short	寄存器的位 0-15	16 位，与 16 位边界对齐
int	寄存器的位 0-31	32 位，与 32 位边界对齐
unsigned int	寄存器的位 0-31	32 位，与 32 位边界对齐
long	整个标量寄存器或向量寄存器的位 0-63	64 位，与 64 位边界对齐
unsigned long	整个标量寄存器或向量寄存器的位 0-63。	64 位，与 64 位边界对齐
enum (1)	寄存器的位 0-31 或整个寄存器	32 位（与 32 位边界对齐）或 64 位（与 64 位边界对齐）
float	寄存器的位 0-31	32 位，与 32 位边界对齐
double	整个标量寄存器或向量寄存器的位 0-63。	64 位，与 64 位边界对齐
long double	整个标量寄存器或向量寄存器的位 0-63。	64 位，与 64 位边界对齐
结构体	成员按其各自类型的需要存储。	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要求进行存储和对齐。
数组	成员按其各自类型的需要存储。	成员按其各自类型的需要存储。结构中的所有数组都根据数组中每个元素的类型对齐。
指向数据成员的指针	整个标量寄存器或向量寄存器的位 0-63。	64 位，与 64 位边界对齐
指向成员函数的指针	组件按其各自类型的要求存储	64 位，与 64 位边界对齐
cchar	寄存器的位 0-15	8 位，与 8 位边界对齐
cshort	寄存器的位 0-31	16 位，与 16 位边界对齐
cint	整个标量寄存器或向量寄存器的位 0-63	64 位，与 32 位边界对齐
cfloat	整个标量寄存器或向量寄存器的位 0-63	64 位，与 32 位边界对齐
clong	向量寄存器的位 0-127	128 位，与 64 位边界对齐
cdouble	向量寄存器的位 0-127	128 位，与 64 位边界对齐

(1) 有关枚举类型大小的详细信息，请参阅节 5.3.1。

对于向量数据类型，对象的最小对齐方式由其元素的类型指定。

6.2.1.1 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 6-1）。默认情况下，char 类型是有符号的。

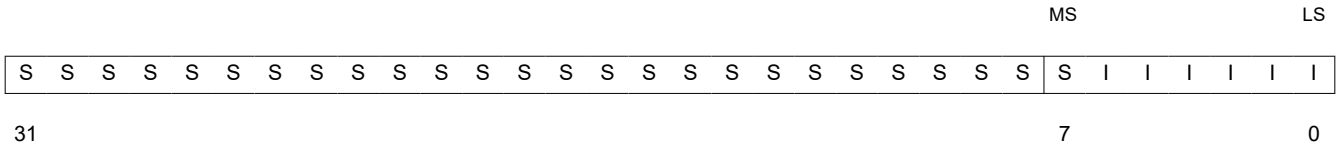
bool 类型也作为 8 位类型存储在寄存器的 0-7 位中。

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 6-1）。

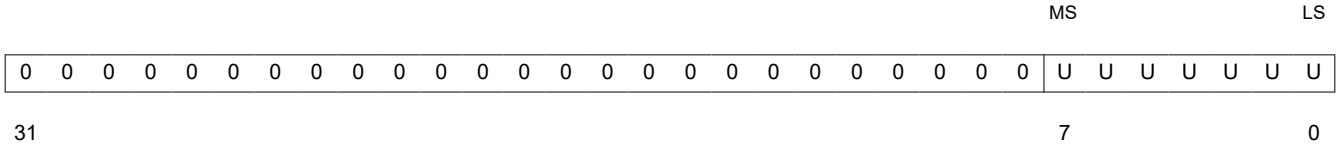
在大端模式下，将内存的第一个字节（即较低地址）移动到寄存器的 8-15 位上，并将内存的第二个字节移动到 0-7 位上以使 2 字节对象加载到寄存器中。在小端模式下，将内存的第一个字节（即较低地址）移动到寄存器的 0-7 位上，并将内存的第二个字节移动到 8-15 位上以使 2 字节对象加载到寄存器中。

图 6-1. Char 和 Short 数据存储格式

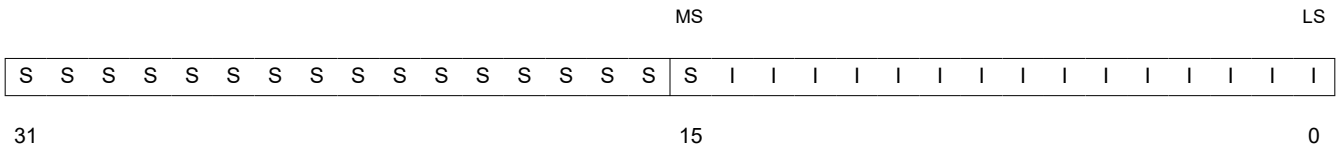
有符号 8 位 char



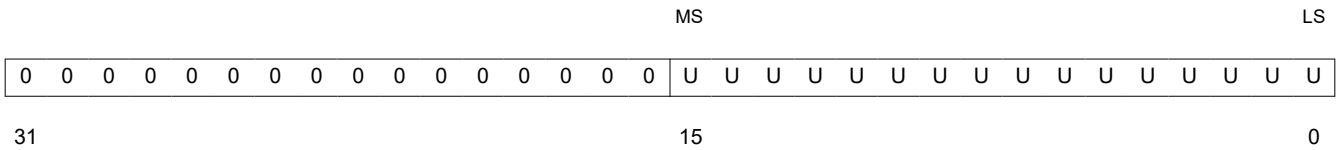
无符号 8 位 char



有符号 16 位 short



无符号 16 位 short



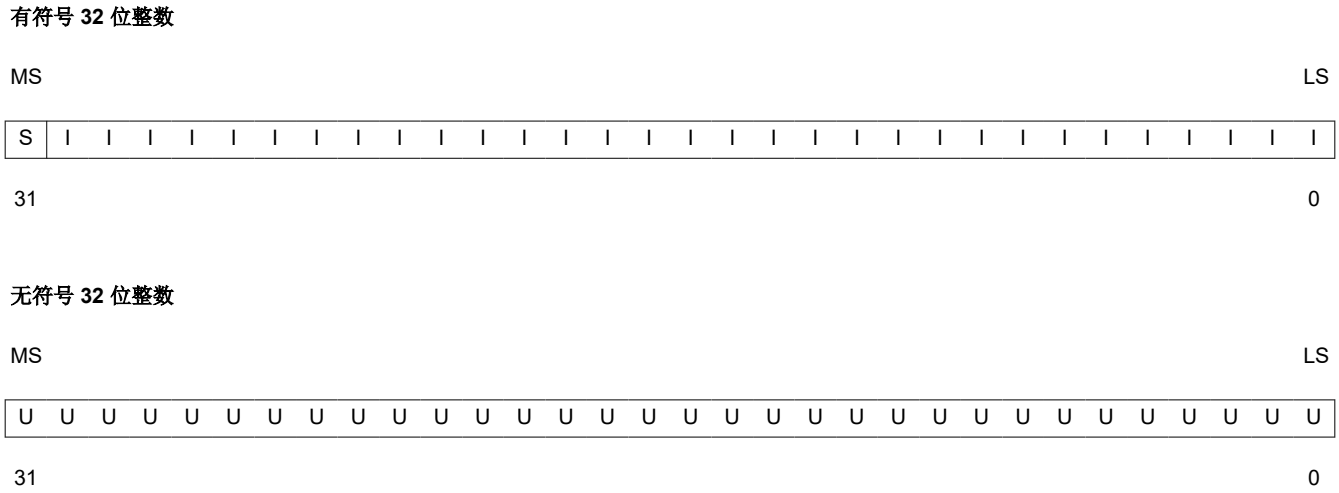
图例: S = 符号，I = 有符号整数，U = 无符号整数，MS = 最高有效，LS = 最低有效

6.2.1.2 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 6-2）。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 5.3.1。

图 6-2. 32 位数据存储格式

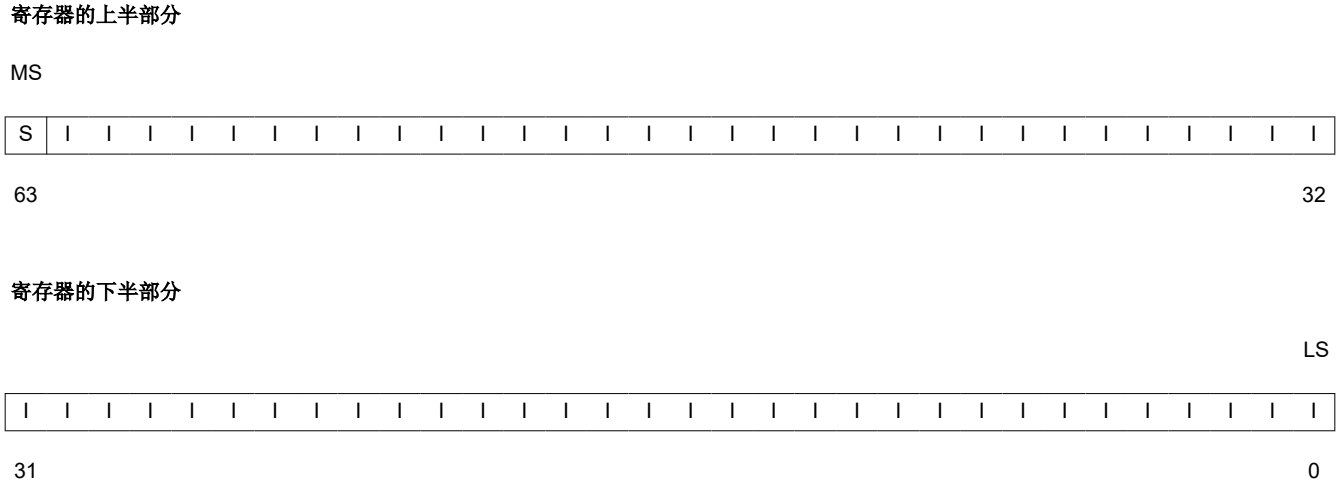


图例: S = 符号，U = 无符号整数，I = 有符号整数，MS = 最高有效，LS = 最低有效

6.2.1.3 长数据类型（有符号和无符号）

长数据和无符号长 数据类型存储在一个完整的 64 位寄存器中（请参阅图 6-3）。在大端字节序模式下，低位地址加载到寄存器的第 32-63 位中，高位地址加载到寄存器的第 0-31 位中。在小端字节序模式下，低位地址加载到寄存器的第 0-31 位中，高位地址加载到寄存器的第 32-63 位中。

图 6-3. 64 位数据存储格式 - 有符号 64 位长数据



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

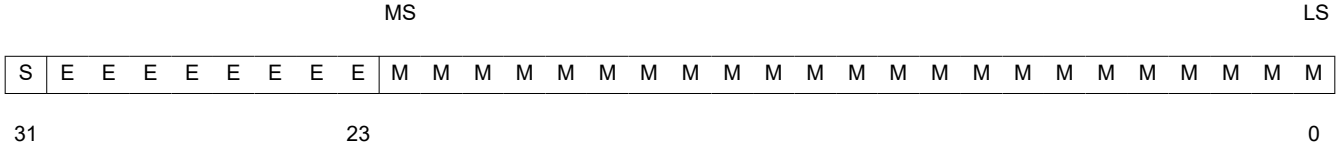
图 6-4. 无符号 64 位长数据

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

6.2.1.4 浮点数据类型

浮点数据类型作为 32 位对象存储在内存中。该值以 IEEE 754 标准定义的单精度浮点格式 (binary32) 存储。

图 6-5. 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

6.2.1.5 double 和 long double 数据类型

double 和 long double 数据类型存储在一个完整的 64 位寄存器中。该值以 IEEE 754 标准定义的双精度浮点格式 (binary64) 存储。即，指数长 11 位，尾数长 52 位。

6.2.1.6 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（64 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。

6.2.1.7 指向成员函数类型的指针

指向成员函数对象的指针具有以下相同的布局：

```
struct __mptr {
   __vptp f;
   ptrdiff_t d;
};
```

其中，_vptp 是以下联合体：

```
union {
   void (*f) ();
   int 0;
}
```

如果成员函数是非虚函数，参数 f 表示指向成员函数的指针。0 是类对象中虚函数指针的偏移。参数 d 是要添加到此指针的类对象开头之处的偏移。

6.2.1.8 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），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大小的倍数。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。

6.2.2 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 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 6-6 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 6-6. 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

6.2.3 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。有关初始化的更多信息，请参阅节 6.7。
- 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。

  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，以使标签唯一。该数字从 0 开始，每定义一个字符串就增加 1。

  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 */
```

6.3 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。

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 6-2 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。

C7000 有两个数据路径：一个带有 64 位“标量”寄存器的 A 端数据路径和一个带有 512 位“矢量”寄存器的 B 端数据路径。通过从名称中删除“V”，任何 B 端矢量寄存器的低 64 位也可以作为标量寄存器来访问。标量寄存器不限于存储标量值；如果合适，矢量可以存储在标量寄存器中。

D15 是栈指针 (SP)。栈指针必须始终在 2 字（8 字节）边界上保持对齐。SP 指向低于（小于）当前分配栈的第一个对齐地址。

RP、D15 (SP)、A8-A15、B14/VB14 和 B15/VB15 是*被调用者保存*寄存器。也就是说，需要由被调用的函数保留上述寄存器，确保其在从函数返回时具有与调用时相同的值。

所有其他寄存器都是*调用者保存*寄存器；也就是说，它们不会在调用中保留，因此如果调用后需要它们的值，调用者负责保存和恢复它们的内容。

表 6-2. 寄存器的使用

寄存器	文件	由被调用者保留？	在调用惯例中的作用
A0	A 端标量	否	
A1		否	指向按引用返回值的指针
A2		否	
A3		否	
A4		否	第 1 个标量参数
A5		否	第 2 个标量参数
A6		否	第 3 个标量参数
A7		否	第 4 个标量参数
A8		是	第 5 个标量参数
A9		是	第 6 个标量参数
A10		是	第 7 个标量参数
A11		是	第 8 个标量参数
A12		是	第 9 个标量参数
A13		是	
A14		是	
A15		是	
AL0-AL7	A 端局部 L	否	
AM0-AM7	A 端局部 M	否	

表 6-2. 寄存器的使用（续）

寄存器	文件	由被调用者保留？	在调用惯例中的作用
VB0	B 端矢量	否	第 1 个矢量参数
VB1		否	第 2 个矢量参数
VB2		否	第 3 个矢量参数
VB3		否	第 4 个矢量参数
VB4		否	第 5 个矢量参数
VB5		否	第 6 个矢量参数
VB6		否	第 7 个矢量参数
VB7		否	第 8 个矢量参数
VB8		否	第 9 个矢量参数
VB9		否	第 10 个矢量参数
VB10		否	第 11 个矢量参数
VB11		否	第 12 个矢量参数
VB12		否	第 13 个矢量参数
VB13		否	第 14 个矢量参数
VB14		是	第 15 个矢量参数
VB15		是	第 16 个矢量参数
VBL0-VBL7	B 端局部 L	否	
VBM-VBM7	B 端局部 M	否	
D0-D14	D 单元局部	否	
D15		是	栈指针
RP	控制	是	返回指针
P0	矢量谓词	否	第 1 个矢量谓词参数
P1		否	第 2 个矢量谓词参数
P2		否	第 3 个矢量谓词参数
P3		否	第 4 个矢量谓词参数
P4		否	第 5 个矢量谓词参数
P5		否	第 6 个矢量谓词参数
P6		否	第 7 个矢量谓词参数
P7		否	第 8 个矢量谓词参数
CUCR0-CUCR3	C 单元控制寄存器	否	

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 6-2 中未列出的控制寄存器会影响具有默认值的编译代码。

6.4 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 *C7000 嵌入式应用二进制接口 (EABI) 参考指南* (SPRUIG4)。

6.4.1 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。

C7000 有专门的指令和寄存器来管理调用和返回操作。CALL 指令将返回地址保存在返回指针 (RP) 寄存器中，并将控制权转移给被调用函数。RET 指令从 RP 恢复 PC，从而将控制权返回给被调用方。

C7000 CPU 具有流水线模型，可在受保护或不受保护状态下运行。当进行调用或返回时，CPU 必须处于受保护状态，而不是无保护状态。

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：

- 传递给函数的参数被放置在寄存器或堆栈上。
  - 如果将参数传递给函数，则将尽可能多的参数放入可用于参数传递的九个“标量”寄存器和十六个“矢量”寄存器中。
  - 声明类型为 64 位或更小的参数被分配给标量寄存器 A4 到 A12。
  - 声明在 64 位到 512 位之间的参数在矢量寄存器 VB0-VB15 中传递
  - 声明为矢量谓词的参数在矢量谓词寄存器 P0-P7 中传递。
  - 大于 512 位的参数通过引用传递。请参阅《*C7000 嵌入式应用二进制接口 (EABI) 参考指南*》(SPRUIG4) 中的“通过引用传递和返回的值”部分。
  - 所有其余的参数都被放置在堆栈中递增的地址处，这样第一个参数在进入被调用方时将位于地址 SP+16 处。
  - 每个具有标量或矢量类型的参数都放置在下一个与其类型正确对齐的可用地址处。
  - 结构体与大于或等于其大小的下一个 2 的幂对齐，最多 8 个字节。
  - 每个参数都保留一定数量的堆栈空间，其大小等于其四舍五入为其对齐的下一个倍数。
  - 对于可变参数的 C 函数（用省略号声明，表明其是使用不同数量的参数调用的），最后一个显式声明的参数和所有剩余的参数都在堆栈上传递，以便其栈地址可以作为访问未声明参数的引用。
  - 根据 C 语言，未在原型中声明且大小小于 int 大小的参数作为 int 传递。
- 调用函数必须保存寄存器 RP，如此调用函数不会被 CALL 指令覆盖。调用函数还必须保存未被调用函数保留的任何活跃的本地或全局寄存器（作为入口保存寄存器集的一部分）。入口保存寄存器组包括 A8-A15、B14/VB14 和 B15/VB15。
- 调用方（父级）会调用函数（子级）。

更多相关信息，请参阅《*C7000 嵌入式应用二进制接口 (EABI) 参考指南*》(SPRUIG4) 的“调用惯例”一章。

6.4.2 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）已经有 16 字节的预留堆栈帧空间可用，但如果存储其本地变量、临时存储区和函数参数所需的空间大于 16 字节，则被调用函数会在堆栈上分配额外空间。
2. 如果被调用函数调用任何其他函数，则必须确保 16 字节的未使用空间也被分配到堆栈上，并为其被调用函数保留。被调用函数也必须将其返回地址保存在堆栈中。否则，返回地址将留在返回寄存器 (RP) 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入口保存寄存器（A8-A15、VB14-VB15），则必须将这些寄存器保存在其他寄存器或堆栈中。被调用函数可以修改任何其他寄存器而无需保存寄存器，因为如果使用了寄存器，寄存器将在调用之前被调用函数保存。
4. 如果被调用函数需要一个结构体参数，并且其值可以放入 64 位或 512 位参数寄存器中，则按该值传递结构体。否则，被调用函数会接收一个指向该结构体的指针。如果从被调用函数内部写入结构体，则必须在堆栈上为结构体的本地副本分配空间，并且必须从传递给结构体的指针复制本地结构体。如果未写入结构体，则可以通过指针参数在被调用函数中间接引用该结构体。

   无论是在调用函数时（以便结构体参数作为地址传递）还是在声明函数时（以便函数知道将结构体复制到本地副本），都必须注意正确地声明接受结构体参数的函数。
5. 被调用函数会执行函数的代码。
6. 返回值按如下方式进行处理：
   - 如果被调用函数返回任何小于或等于 64 位的整数、指针、浮点数、双精度型、长双精度型、长整型或矢量数据类型，则返回值被放置在寄存器 A4 中。
   - 如果被调用函数返回大小大于 64 位的矢量数据类型，则返回值被放置在寄存器 VB0 中。
   - 如果被调用函数返回矢量谓词类型，则返回值被放置在寄存器 P0 中。
   - 如果被调用函数返回一个结构体，并且其足够小，可以放入寄存器 A4 或寄存器 VB0 中，则按该值返回。否则，调用方为该结构体分配空间并将返回空间的地址传递给寄存器 A1 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指针。

   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 中保存的任何入口保存寄存器（A8-A15、B14/VB14、B15/VB15）。
8. 如果保存，返回寄存器 (RP) 的值也会恢复。
9. 在此调用序列期间分配的任何空间都将被回收。
10. 该函数通过调用 RET 指令返回，该指令返回到包含在返回寄存器 (RP) 中的位置。

6.4.3 访问参数和局部变量

函数通过指向栈顶的寄存器 D15 (SP) 间接访问其栈参数和本地非寄存器变量。栈向较小的地址增长，因此通过 SP 的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 SP 的偏移量来访问的。

如需了解更多信息，请参阅节 6.4.2。更多有关 C/C++ 系统栈的信息，请参阅节 6.1.2。

6.5 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 节 12.6。

6.6 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 C7000 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 6-3 总结了用于算术运算的运行时支持函数。

表 6-3. C7000 运行时支持算术函数

返回类型	C 函数	说明
void	__c7xabi_abort_msg (const char *)	报告断言失败。（请参阅以下注意事项。）
double	__c7xabi_divd (double, double)	将两个双精度浮点数相除。
float	__c7xabi_divf (float, float)	将两个单精度浮点数相除。
long long	__c7xabi_divlli (long long, long long)	64 位带符号整数除法。
unsigned long long	__c7xabi_divull (unsigned long long, unsigned long long)	64 位无符号整数除法。
long long	__c7xabi_fixdlli (double)	将双精度浮点型转换为 64 位整数型。
unsigned	__c7xabi_fixdu (double)	将双精度浮点型转换为 32 位无符号整数型。
unsigned long long	__c7xabi_fixdull (double)	将双精度浮点型转换为 64 位无符号整数型。
ong long	__c7xabi_fixflli (float)	将单精度浮点型转换为 64 位整数型。
unsigned	__c7xabi_fixfu (float)	将单精度浮点型转换为 32 位无符号整数型。
unsigned long long	__c7xabi_fixfull (float)	将单精度浮点型转换为 64 位无符号整数型。
double	__c7xabi_fltllid (long long)	将 64 位整数型转换为双精度浮点型。
float	__c7xabi_fltllif (long long)	将 64 位整数型转换为单精度浮点型。
double	__c7xabi_fltulld (unsigned long long)	将 64 位无符号整数型转换为双精度浮点型。
float	__c7xabi_fltullf (unsigned long long)	将 64 位无符号整数型转换为单精度浮点型。
long long	__c7xabi_remlli (long long, long long)	64 位整数取模。
unsigned long long	__c7xabi_remull (unsigned long long, unsigned long long)	64 位无符号整数取模。
void	__c7xabi_strasg (int*, const int*, unsigned)	块复制。（请参阅以下注意事项。）
	__c7xabi_unwind_cpp_pr0	短帧展开，16 位范围。
	__c7xabi_unwind_cpp_pr1	长帧展开，16 位范围。
	__c7xabi_unwind_cpp_pr2	长帧展开，32 位范围。
	__c7xabi_unwind_cpp_pr3	展开，24 位编码，16 位范围。

__c7xabi_abort_msg() 函数：

```
void __c7xabi_abort_msg(const char *msg)
```

生成该函数 `__c7xabi_abort_msg()` 是为了在运行时断言（例如 C 断言宏）失败时输出诊断信息。该函数不得返回。也就是说，该函数必须调用中止或通过其他方式终止程序。

__c7xabi_strasg() 函数：

```
void __c7xabi_strasg(int* dst, const int* src, unsigned  cnt)
```

该函数 `__c7xabi_strasg()` 由编译器为高效行外结构或数组复制操作生成。`cnt` 参数表示大小，单位为字节。

6.7 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。

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 6.7.2。
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 6.7.2.6。
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。

6.7.1 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。

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。

6.7.2 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。

6.7.2.1 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。

6.7.2.2 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 6.7.2.5。

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 6.7.2.3。

6.7.2.3 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。若要使用此方法，请使用 --rom_model 选项调用链接器。

使用此方法时，链接器将从编译模块中直接被初始化的段中创建压缩初始化表和初始化数据。C/C++ 引导例程使用该表和数据以在 ROM 中初始化 RAM 中的变量。

图 6-7 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。

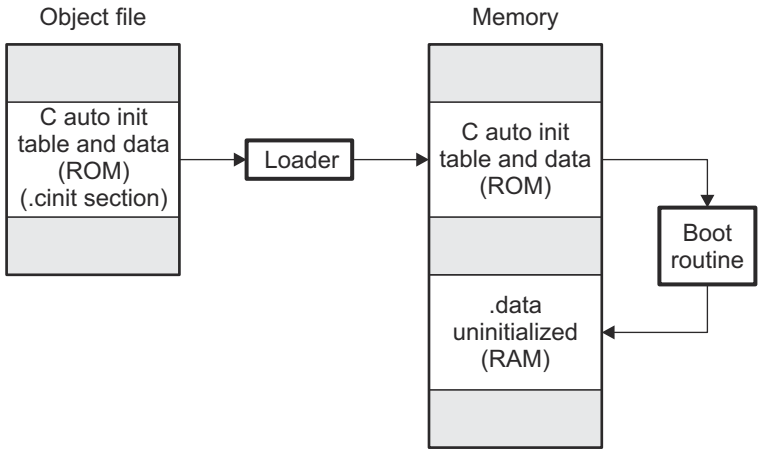
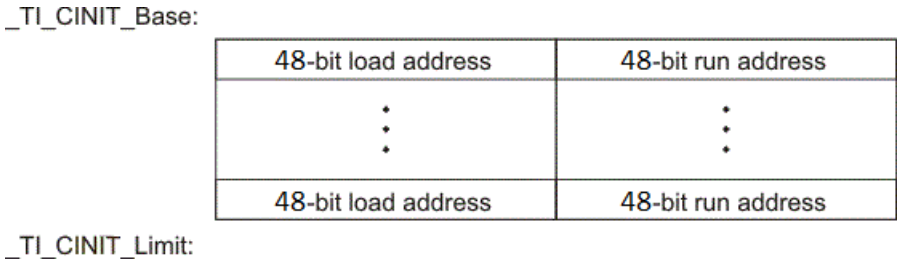


图 6-7. 运行时自动初始化

6.7.2.4 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：



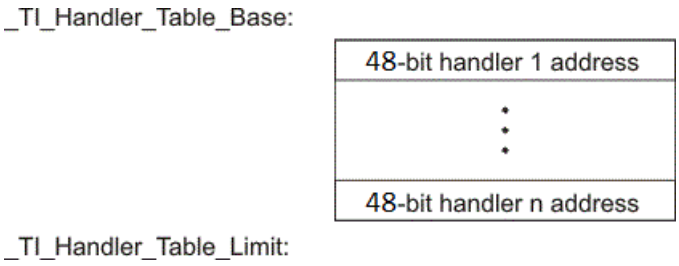
链接器定义的符号 __TI_CINIT_Base 和 __TI_CINIT_Limit 分别指向表的开头和结尾。此表中的每个条目对应一个需要初始化的输出段。可以使用不同的编码对每个输出段的初始化数据进行编码。

C 自动初始化记录中的加载地址指向以下格式的初始化数据：

8 位索引	编码数据

初始化数据的前 8 位是处理程序索引。它将索引到处理程序表中，以获取知道如何解码以下数据的处理程序函数的地址。

处理程序表是 32 位函数指针的列表。



8 位索引后面的*编码数据* 可以是以下格式类型之一。为清晰起见，还为每种格式介绍了 8 位索引。

6.7.2.4.1 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。

6.7.2.4.2 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。

6.7.2.4.3 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。

6.7.2.4.4 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。

6.7.2.5 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 6-8 演示了加载时变量的初始化。

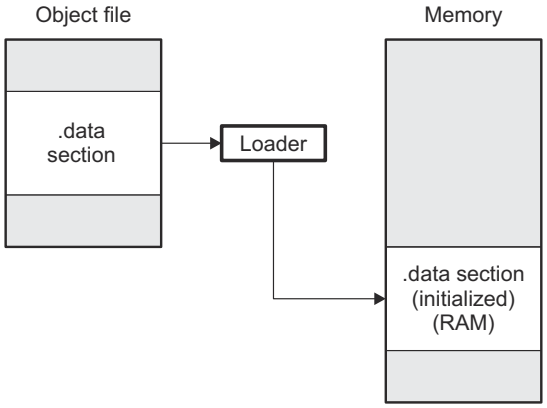


图 6-8. 加载时初始化

6.7.2.6 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。

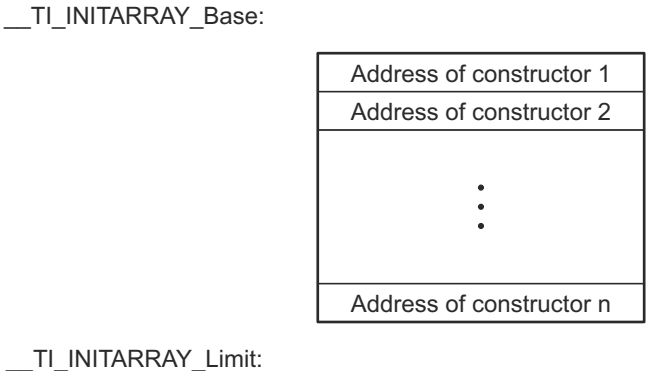


图 6-9. 构造函数表

章节 7

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 7.1 和节 7.2 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 7.4 中对此过程进行了介绍。

7.1 C 和 C++ 运行时支持库

C7000 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 7.1.8。

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 5.1 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

7.1.1 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 11.3.1。

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，请参阅章节 12。

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。

7.1.2 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 C7X_C_DIR 环境变量设为安装相关工具的 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：

- c7x.h -- 提供内在函数定义。
- c7x_direct.h -- 提供低级别“直接映射”内在函数定义。由 c7x.h 自动包含。
- c7x_vpred.h -- 提供低级别向量谓词内在函数定义。由 c7x.h 自动包含。
- c7x_mma.h -- 提供 MMA 内在函数定义。由 c7x.h 自动包含。
- c6x_migration.h -- 提供传统 C6000 内在函数的定义。
- c7x_luthist.h -- 定义查找表和直方图特征的内在函数。由 c7x.h 自动包含。
- c7x_scalable.h -- 独立于矢量宽度的代码的实用程序。不自动包括在内。请参阅节 5.16。
- c7x_strm.h -- 定义流引擎 (SE) 和流地址生成器 (SA) 的内在函数。（请参阅节 4.15。）由 c7x.h 自动包含。
- 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。

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、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

7.1.3 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\c7000_#.#.#\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 rts7100_le.lib、rts7100_le_eh.lib、rts7100_be.lib 或 rts7100_be_eh.lib 库，或者构建新库。有关库命名的详细信息，请参阅节 7.1.8；有关构建的详细信息，请参阅节 7.4

7.1.4 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

7.1.5 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 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 5.4，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。

7.1.6 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);`

  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。
- `time_t time(time_t *timer);`

  time() 函数会返回挂钟时间。

编译器安装程序的 `/lib/src` 目录中提供了时间相关和时钟相关的 RTS 库源文件。

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某些器件上的寄存器中，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 C7000 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。

TI RTS 库不提供时区查询表。本地时区由类型为结构体 tz 的全局变量 `_tz` 指定，该变量在 RTS 库源代码的 `tmzone.c` 中定义。此结构体由 mktime()和 strftime() 函数使用。要更改本地时区，请为此结构体中的以下字段分配新值：

- 将 `_tz.daylight` 字段设置为 mktime() 函数的精确值，以分配给结构体 `tm` 成员 `tm_isdst`。
- 将 `_tz.timezone` 字段设置为有符号整数，表示 UTC 以西秒数。例如，中部标准时间 (UTC+6) 在 UTC 以西 21600 秒。为 UTC 以东的时区使用负值。
- 将 `_tz.tzname` 字段设置为本地时区的缩写名称。此字符串由 strftime() 函数逐字用于 %Z 转换。例如，如果本地时区是中部标准时间，请为此字段使用“CST”。
- `_tz.dstname` 字段当前未被 RTS 库使用。

7.1.7 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。

7.1.8 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 11.3.1.1）。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：

rts7100_[*endian*][_eh].lib

endian	指示字节序：	
	le	小端字节序库
	be	大端字节序库
eh	指示库支持 (_eh) 还是不支持（空）异常处理。	

7.2 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，可能就是这个原因。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 选项节 12.4。

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）。

7.2.1 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。

便携式应用只能使用高级别 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：

```
cl7x main.c -z --heap_size=1000 --output_file=main.out
```

执行 main.out 会将

```
Hello, world
```

输出到文件，将

```
Hello again, world
```

输出到主机的 stdout 窗口。

7.2.1.1 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。

7.2.2 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 7.2.3。

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 7.2.5）。
- *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 7.2.3。

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 7.2.5。）

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。

7.2.3 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 7.2.5。）
- *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 7.2.3。

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。

7.2.4 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 7-1 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 7-1 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 7-1. 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"); }
```

7.2.5 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 7.2.2 部分中指定的接口来声明这些函数。用于运行 C7000 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 7-2 将执行以下操作：

add_device（续）

向器件表添加器件

- 将器件 *mydevice* 添加至器件表
- 打开该器件上名为 *test* 的文件并将其与 FILE 指针 *fid* 关联
- 将字符串 *Hello, world* 写入该文件
- 关闭该文件

示例 7-2 显示了为 C I/O 添加和使用器件：

示例 7-2. 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);
}
```

7.3 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，；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。

7.4 库构建流程

使用 C/C++ 编译器时，您可使用 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 rts7100_le.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。

7.4.1 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。

7.4.2 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 7.4.2.2，以了解可能需要您这样做的情形。

7.4.2.1 通过链接器自动重建标准库

链接器主要通过 C7X_C_DIR 环境变量查找运行时支持库。通常， C7X_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 C7X_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=rts7100_le.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/rts7100_le.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 章节 10。

现在链接器已经决定了要使用的库，接下来它会检查 C7X_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 C7X_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 7.4.2.2。
- 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。

7.4.2.2 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。）

7.4.2.2.1 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=rts7100_le.lib
```

7.4.2.2.2 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。

7.4.2.2.3 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 rts7100_le.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=rts7100_le.lib --name=rts7100_le_debug.lib --install_to=$Project/Debug
  --extra_options=<options_list>
```

7.4.2.2.4 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 7-1 中所述。

```
mklib --help
```

表 7-1. 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=rts7100_le.lib --index=$C_DIR/lib
```

构建类似 rts7100_le.lib 的自定义库，但启用符号调试支持：

```
mklib --pattern=rts7100_le.lib --extra_options="-g" --index=$C_DIR/lib --install_to=$Project/Debug
--name=rts7100_le_debug.lib
```

7.4.3 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。

7.4.3.1 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。

7.4.3.2 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 C7X_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 7-1 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 8

目标模块简介

链接器根据目标模块创建可执行目标文件。这些可执行目标文件可以通过 C7000 CPU 执行。

建议从代码*块*和数据块的角度去考虑目标模块，这样便可以更轻松地进行模块化编程。这些块被称为段。链接器提供了可用于创建和操作段的指令。

本章着重介绍段的概念和使用。

8.1 目标文件格式规范

由代码生成工具创建的目标文件符合 ELF（可执行链接格式）二进制格式规范，该格式由嵌入式应用二进制接口 (EABI) 使用。有关 EABI ABI 的信息，请参阅 *C7000 嵌入式应用二进制接口 (EABI) 参考指南* (SPRUIG4)。

C7000 具有一个适用于代码和数据的 48 位地址。相应地，目标文件使用 ELF64 格式。

8.2 可执行目标文件

链接器可用于生成静态可执行目标模块。可执行目标模块与用作链接器输入的目标文件具有相同的格式。然而，可执行目标模块中的各段均已合并，并放置在目标存储器中，并且重定位问题均已解决。

若要运行程序，必须将可执行目标模块中的数据传输或加载到目标系统存储器中。有关加载和运行程序的详细信息，请参阅章节 9。

8.3 段简介

目标文件的最小单元是*段*。段是占据存储器映射中连续空间的代码或数据块。目标文件的每个段都是独立且不同的。

ELF 格式可执行目标文件包含*程序段*。ELF 程序段是元段。它表示目标存储器的一个连续区域。它是具有相同属性（例如，可写或可读）的*段*的集合。ELF 加载程序需要程序段信息，但不需要段信息。ELF 标准允许链接器完全从可执行目标文件中省略 ELF 段信息。

目标文件通常包含三个默认段：

.text 段	包含可执行代码 [1]
.data 段	通常包含已初始化的数据
.bss	通常为未初始化的变量预留空间

您可以使用链接器创建、命名和链接其他类型的段。.text、.data 和 .bss 段是范例，说明了如何处理段。

有两种基本类型的段：

已初始化的段	包含数据或代码。.text 和 .data 段已被初始化。
未初始化的段	在存储器映射中为未初始化的数据预留空间。.bss 段未被初始化。

链接器的其中一项功能是将段重新定位到目标系统的存储器映射中；此功能被称为*放置*。大多数系统都包含几种类型的存储器，因此使用段可帮助您更高效地使用目标存储器。所有段均可单独重定位；您可以将任何段放入目标存储器的任何已分配块中。例如，您可以定义一个包含初始化例程的段，然后将该例程分配到包含 ROM 的存储器映射的一个部分。有关段放置的更多信息，请参阅 节 11.3.5。

[1] 某些目标允许在 .text 段中使用除文本以外的内容，例如变量。

图 8-1 展示了目标文件中的段与虚构目标存储器之间的关系。ROM 可以是 EEPROM、FLASH 或实际系统中的一些其他类型的物理存储器。

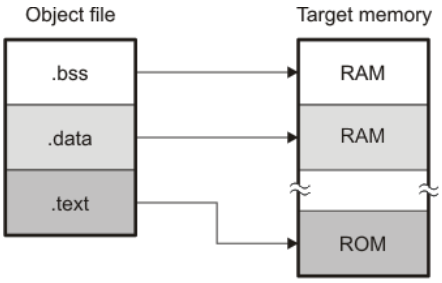


图 8-1. 存储器信息逻辑块分区

8.3.1 特殊段名

您可使用 .sect 和 .usect 指令创建所需的任何段名，但链接器和编译器的运行时支持库会以特殊方式处理某些段。如果您创建的段与特殊段重名，应注意遵守该特殊段的规则。

一些常见的特殊段有：

- .text -- 用于程序代码。
- .data -- 用于初始化的非常量对象（全局变量）。
- .bss -- 用于未初始化的对象（全局变量）。
- .const -- 用于已初始化的常量对象（字符串常量、变量声明的常量）。
- .cinit -- 用于在启动时初始化 C 全局变量。
- .stack -- 用于函数调用堆栈。
- .sysmem - 用于动态存储器分配池。

有关段的更多信息，请参阅 节 11.3.5。

8.4 链接器如何处理段

链接器有两个与段相关的主要功能。第一，链接器使用目标文件中的段作为构建块；它将输入段组合在一起，以在可执行输出模块中创建输出段。第二，链接器为输出段选择存储器地址；这称为*放置*。两个链接器指令支持以下功能：

- *MEMORY* 指令使您能够定义目标系统的存储器映射。您可以命名存储器的几个部分，并指定它们的起始地址和长度。
- *SECTIONS* 指令告诉链接器如何将输入段组合成输出段，以及将这些输出段放置在存储器中的哪个位置。

使用子段可更精确地控制段的放置。您可以使用链接器的 SECTIONS 指令指定每个子段的位置。如果不指定子段，则该子段将与具有相同基础段名的其他段组合在一起。请参阅节 12.5.5.1。

并不总是需要使用链接器指令。如果不使用此类指令，链接器将使用节 12.7 中所述的目标处理器的默认放置算法。当您*确实*使用链接器指令时，必须在链接器命令文件中指定它们。

有关链接器命令文件和链接器指令的更多信息，请参阅以下部分：

- 节 12.5，*链接器命令文件*
- 节 12.5.4，*MEMORY 指令*
- 节 12.5.5，*SECTIONS 指令*
- 节 12.7，*默认放置算法*

8.4.1 合并输入段

图 8-2 通过一个简化示例介绍了将两个文件链接在一起的过程。

请注意，这是一个简化示例，因此它不会显示待创建的所有段或这些段的实际顺序。请参阅节 12.7，了解 C7000 的实际默认存储器放置映射。

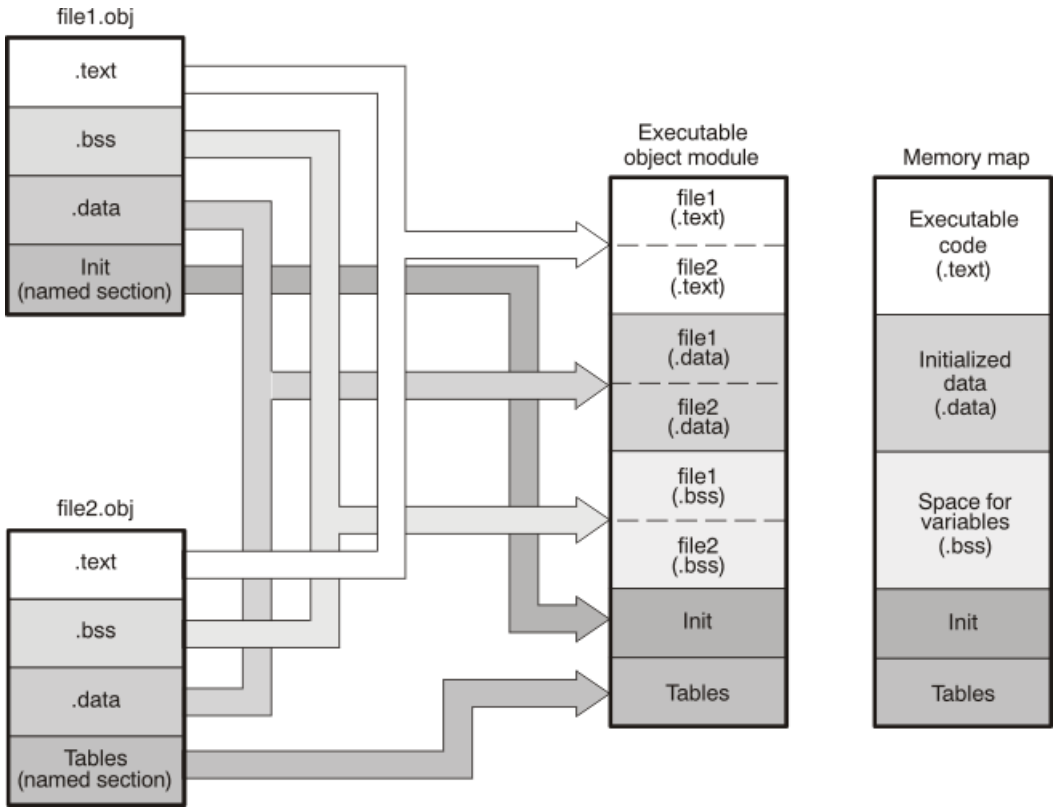


图 8-2. 组合输入段以构成可执行对象模块

在图 8-2 中，已将 file1.obj 和 file2.obj 组合在一起用作链接器输入。每个都包含 .text、.data 和 默认段；此外，每个都包含一个用户命名的段。可执行对象模块显示了各组合段。链接器将 file1.obj 的 .text 段和 file2.obj 的.text 段组合成一个 .text 段，然后将两个 .data 段和两个 段组合在一起，最后将用户命名的段放在末尾。存储器映射显示了待置于存储器中的组合段。

8.4.2 放置段

图 8-2 展示了链接器合并段的默认方法。有时，您可能不想使用默认设置。例如，您可能不想将所有 .text 段合并到一个 .text 段中。或者，您可能想要将用户命名的段放置在 .data 段通常所在的位置。大多数存储器映射都包含多种不同类型的存储器（RAM、ROM、EEPROM、FLASH 等），对应的空间量各不相同；您可能想要将某个段放在特定类型的存储器中。

如需存储器映射中段放置的进一步说明，请参阅 节 12.5.4 和 节 12.5.5 中的讨论。请参阅节 12.7，了解 C7000 的实际默认存储器分配映射。

8.5 符号

目标文件包含一个符号表，存储着目标文件中相关*符号*的信息。链接器在执行重定位时会使用此表。

目标文件符号是一个指定的 48 位整数值，通常表示一个地址。符号可以表示诸如函数、变量、段的起始地址，或绝对整数（如栈大小）之类的东西。

*绝对符号*是具有数字值的符号。它们可能是常量。对于链接器而言，这样的符号是无符号的，但整数会根据其使用方式被视为有符号或无符号类型。若处理为无符号类型，则绝对整数合法值的范围是 0 到 2^48-1，若处理为有符号类型，则为 -2^47 到 2^47-1。

8.5.1 局部符号

局部符号在单个目标文件中可见。每个目标文件对于特定符号可能有属于自己的局部定义。在一个目标文件中引用局部符号与另一个目标文件中的局部符号完全无关。

默认情况下符号是局部符号。

8.5.2 弱符号

弱符号是可能定义，也可能不定义的符号。

对于定义了“弱”绑定的符号与定义了全局绑定的符号，链接器的处理方式不同。如果某弱符号需要解析某未以其他方式解析的引用，链接器不会在目标文件的符号表中包含该弱符号（就像全局符号那样），而只会在“最终”链接的输出中包含该弱符号。

这样可使链接器省略解析引用时不需要的符号，最大限度地减少输出文件的符号表中包含的符号数量。减小输出文件符号表的大小，可减少链接所需的时间，特别是在有大量预先加载的符号要予以链接的情况下。

- **使用链接器命令文件：**若要在链接器命令文件中定义弱符号，请在赋值表达式中使用“弱”运算符来指定可以从输出文件的符号表中删除该符号（如果它未被引用）。在链接器命令文件中，可使用 MEMORY 或 SECTIONS 指令之外的赋值表达式来定义由链接器定义的弱符号。例如，可将“ext_addr_sym”定义如下：

  ```
  weak(ext_addr_sym) = 0x12345678;
  ```

  如果链接器命令文件执行最终链接，“ext_addr_sym”会作为弱符号提供给链接器；如果未引用该符号，它将不会被包含在生成的输出文件中。请参阅节 12.6.6。
- **使用 C/C++ 代码：**请参阅 节 5.8.35 中的 WEAK pragma 和 weak GCC 样式变量属性信息。

如果同一符号有多个定义，链接器会根据特定规则确定优先使用哪个定义。一些定义可能有弱绑定，另一些定义可能有强绑定。“强”在此上下文中意为该符号*并未*在链接器命令文件的赋值语句中进行弱绑定。链接器根据以下指导原则确定将引用解析为符号时使用哪个定义：

- 强绑定符号始终优先于弱绑定符号。
- 如果两个符号都是强绑定或都是弱绑定，则链接器命令文件中定义的符号优先于输入目标文件中定义的符号。
- 如果两个符号都是强绑定，并且都是在输入目标文件中定义的，链接器会发出符号重复定义错误，并暂停链接过程。

8.6 加载程序

链接器会创建一个可执行的目标文件，根据您的执行环境，可以通过多种方式加载该文件。这些方法包括使用 Code Composer Studio 或十六进制转换实用程序。相关详细信息，请参阅节 9.1。

This page intentionally left blank.

章节 9

程序加载和运行

即便在程序编写、编译并链接至可执行的目标文件后，仍需要完成很多任务，程序才能开始工作。程序必须加载到目标上，存储器和寄存器必须进行初始化，并且必须将程序设置为运行。

上述一些任务需要构建到程序本身中。很多必要任务都由编译器和链接器替用户完成，但如果需要更多地控制这些任务，那么了解各段代码如何配合工作会有所帮助。

本章将介绍程序加载、初始化和启动中涉及的各个概念。

请参阅器件文档，以了解自举的各种器件特定方面。

9.1 负载

程序需要先放入目标器件的存储器中，然后才能执行。*加载* 通过使用程序的代码和数据来初始化器件存储器，从而准备程序以供执行的过程。*加载程序* 可能是器件上的另一个程序、外部代理（例如，调试器）或可能在上电后初始化自身的器件，这被称为 *引导加载* 或 *自举加载*。

加载程序负责在程序启动之前在存储器中构建 *加载映像*。加载映像是执行前程序在存储器中的代码和数据。加载的具体内容取决于环境，例如是否存在操作系统。此节介绍了裸机器件的几种加载方案。此部分并不是详尽无遗。

可以通过以下方式加载程序：

- **在已连接的主机工作站上运行的调试器。**在典型的嵌入式开发设置中，器件隶属于运行 Code Composer Studio (CCS) 等调试器的主机。器件与 JTAG 接口等通信通道连接。CCS 读取程序，并通过通信接口将加载映像直接写入到目标存储器。
- **在另一个 CPU 上运行的加载器。**使用 ELF 加载器，器件上的另一个 CPU（通常为 ARM 核心）将可执行文件加载到存储器。
- **器件上运行的另一个程序。**运行的程序可以创建加载映像并将控制权移交给加载的程序。如果有操作系统，则它可能具有加载和运行程序的能力。

9.2 入口点

入口点是开始执行程序的地址。这是启动例程的地址。启动例程负责初始化和调用程序的其余部分。对于 C/C++ 程序，启动例程通常命名为 _c_int00（请参阅节 9.3.1）。加载程序后，入口点的值将放入 PC 寄存器中，并允许 CPU 运行。

目标文件有一个入口点字段。对于 C/C++ 程序，链接器默认会填入 _c_int00。您可以选择自定义入口点；请参阅节 12.4.11。器件本身无法从目标文件中读取入口点字段，因此必须在程序中的某处对其进行编码。

- 如果您使用引导加载程序，则引导表包含一个入口点字段。当它完成运行时，引导加载程序将跳转到入口点。
- 如果您使用托管的调试器（如 CCS），该调试器可以将程序计数器 (PC) 显式设置为入口点的值。

9.3 运行时初始化

在加载映像就位后，程序便可以运行。后续的小节描述了 C/C++ 程序的自举初始化。

9.3.1 _c_int00 函数

_c_int00 函数是 C/C++ 程序的*启动例程*（也被称为*引导例程*）。它执行 C/C++ 程序自身初始化的所有必要步骤。

_c_int00 名称表示这是中断号 0，即 RESET 的中断处理程序，可设置 C 环境。名称不需要完全是 _c_int00，但链接器会默认将 _c_int00 设为 C 程序的进入点。编译器的运行时支持库默认执行 _c_int00。

启动例程负责执行以下操作：

1. 通过初始化 SP 来设置栈
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. 使用相应的参数调用 main 函数
7. 当 main 返回时调用 exit

9.3.2 RAM 模型与 ROM 模型

根据应用需求选择启动模型。ROM 模型会在引导例程期间完成更多工作。RAM 模型会在加载应用程序期间完成更多工作。

如果您的应用程序可能需要频繁 RESET（重置）或者是独立的应用程序，那么 ROM 模型可能是更好的选择，因为引导例程拥有初始化 RAM 变量所需的所有数据。不过，对于具有操作系统的系统，那么可能最好使用 RAM 模型。

C 引导例程会将数据从 .cinit 段复制到要初始化的变量对应的运行时位置。

9.3.2.1 在运行时自动初始化变量 (--rom_model)

在运行时自动初始化变量是默认的自动初始化方法。若要使用此方法，请使用 --rom_model 选项调用链接器。

ROM 模型允许将初始化数据存储在慢速非易失性存储器中，并在每次程序复位时复制到快速存储器中。如果您的应用程序从刻录到慢速存储器的代码运行或需要在复位后继续运行，请使用此方法。

对于 的 ROM 模型，.cinit 段与所有其他已初始化段一同加载到存储器中。链接器定义了一个名为 __TI_CINIT_Base 的特殊符号，该符号指向存储器中初始化表的开头。程序开始运行时，C 引导例程会将表中的数据（由 .cinit 指向）复制到变量的运行时位置。

图 9-1 演示了使用 ROM 模型时的运行时自动初始化 ( )。

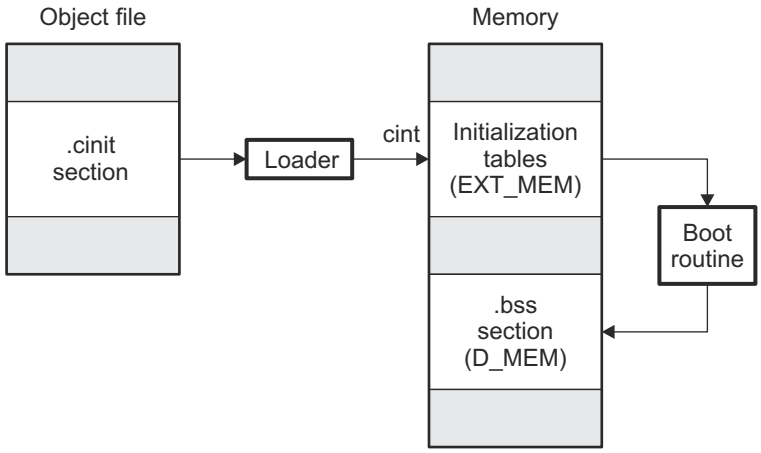


图 9-1. 运行时自动初始化

9.3.2.2 在加载时初始化变量 (--ram_model)

RAM 模型在加载时初始化变量。若要使用此方法，请使用 --ram_model 选项调用链接器。

此模型可以减少启动时间并节省初始化表使用的存储器空间。

在使用 --ram_model 链接器选项时，链接器会在 .cinit 段的标头中设置 STYP_COPY 位。这会告诉加载器不要将 .cinit 段加载到存储器中。 (.cinit 段不占用存储器映射中的任何空间。)

链接器会将 __TI_CINIT_Base 设置为等于 __TI_CINIT_Limit 以指示没有 .cinit 记录。

加载器会将值直接从 .data 段复制到存储器中。

图 9-2 演示了加载时变量的初始化。

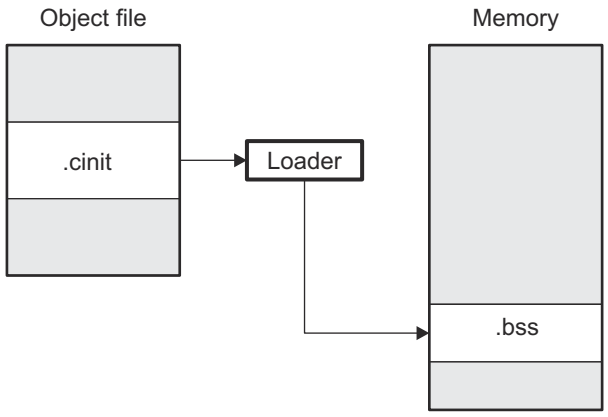


图 9-2. 加载时初始化

9.3.2.3 --rom_model 和 --ram_model 链接器选项

以下列表展示了调用链接器时使用 --ram_model 或 --rom_model 选项的结果。

- 符号 _c_int00 定义为程序进入点。_c_int00 符号是 boot.c.obj 中 C 引导例程的起点。引用 _c_int00 可确保自动从适当的运行时支持库将 boot.c.obj 链接进来。
- 如果使用 ROM 模型在运行时自动初始化（--rom_model 选项）：
  - 链接器定义了一个名为 __TI_CINIT_Base 的特殊符号，该符号指向存储器中初始化表的开头。当程序开始运行时，C 引导例程会将表中的数据（由 .cinit 指向）复制到变量的运行时位置。
- 如果使用 RAM 模型在加载时初始化（--ram_model 选项）：
  - 链接器会将 __TI_CINIT_Base 设置为等于 __TI_CINIT_Limit 以指示没有 .cinit 记录。

9.3.3 关于链接器生成的复制表

RTS 函数 copy_in 可在运行时用来移动代码和数据，通常是从加载地址移动到运行地址。此函数从复制表中读取大小和位置信息。链接器会自动生成几种复制表。请参阅节 12.8。

使用复制表可以创建和控制代码叠加。请参阅节 12.8.4，了解详细信息和示例。

链接器可以使用复制表来实现运行时重定位（如节 9.5 所述），但是复制表需要特定的表格式。

9.3.3.1 BINIT

BINIT（启动时初始化）复制表的特殊之处在于目标将在自动初始化时自动执行复制。更多有关 BINIT 复制表名称的信息，请参阅节 12.8.4.2。在复制 BINIT 复制表之后需要进行 .cinit 处理。

9.3.3.2 CINIT

EABI .cinit 表是特殊类型的复制表。 有关将 .cinit 段用于 ROM 模型的更多信息，请参阅节 9.3.2.1；有关将该段用于 RAM 模型的更多信息，请参阅节 9.3.2.2。

9.4 main 的参数

一些程序要求 main (argc, argv) 的参数有效。通常这对于嵌入式程序是不可能的，但 TI 运行时提供了一种方法来做到这一点。用户必须使用 --args 链接器选项分配合理大小的 .args 段。加载器负责填充 .args 段。无法明确加载器如何确定将哪些参数传递给目标。参数的格式与目标上的 char 指针数组相同。

有关分配存储器以进行参数传递的信息，请参阅节 12.4.4。

9.5 运行时重定位

有时您可能希望将代码加载到存储器的一个区域，并在运行前将它移至另一个区域。例如，基于外部存储器的系统中可能有对性能至关重要的代码。代码必须加载至外部存储器，但在内部存储器中能够以更快的速度运行。由于内部存储器空间有限，可以在不同时间交换不同的速度关键型功能。

链接器提供了处理此任务的一种方式。使用 SECTIONS 指令，您可以选择使链接器分配一个段两次：首先设置其加载地址，接着设置其运行地址。加载地址使用 *load* 关键字，运行地址使用 *run* 关键字。如果段在链接时被分配了两个地址，该段中定义的所有标签会重新定位为引用运行时地址，以便在代码运行时正确引用该段（例如，分支）。

如果您只为某个段提供了一个分配（加载或运行），该段只会分配一次，并会在相同的地址加载和运行。如果您提供了两个分配，该段实际上会被视作两个单独的段来进行分配。如果加载段未压缩，那么这两个段大小相同。

未初始化的段（例如 ）不会被加载，因此唯一重要的地址是运行地址。链接器只分配一次未初始化的段；如果您同时指定运行地址和加载地址，链接器会向您发出警告并忽略加载地址。

有关运行时重定位的完整说明，请参阅节 12.5.6。

9.6 其他信息

请参阅以下章节和文档了解详情：

- 节 6.7“系统初始化”
- 节 11.3.2“运行时初始化”
- 节 12.4.4“分配存储器供加载器使用以传递参数（--arg_size 选项）”
- 节 12.4.11“定义入口点（--entry_point 选项）”
- 节 12.5.6.1“指定加载和运行地址”
- 节 12.8“由链接器生成的复制表”
- 节 12.10.1“运行时初始化”

章节 10

归档器说明

借助 C7000 归档器，您可以将多个单独的文件合并为一个存档文件。您可以使用归档器将一组目标文件收集到一个对象库中。在链接阶段，链接器仅会在库中包含会解析外部引用的成员。归档器允许通过删除、替换、提取或添加成员来修改库。

10.1 归档器概述

您可以采用任何类型的文件来构建库。链接器接受归档库作为输入。它可以使用包含单个目标文件的库。

归档器很有用的应用之一是构建目标模块库。例如，您可以编写多个算术例程，将它们组合起来，并使用归档器将目标文件收集到一个逻辑组中。然后，您可以将目标库指定为链接器输入。链接器会搜索库并包含会解析外部引用的成员。

10.2 归档器在软件开发流程中的作用

图 10-1 展示了归档器在软件开发流程中的作用。阴影部分突出显示了最常用的归档器开发路径。

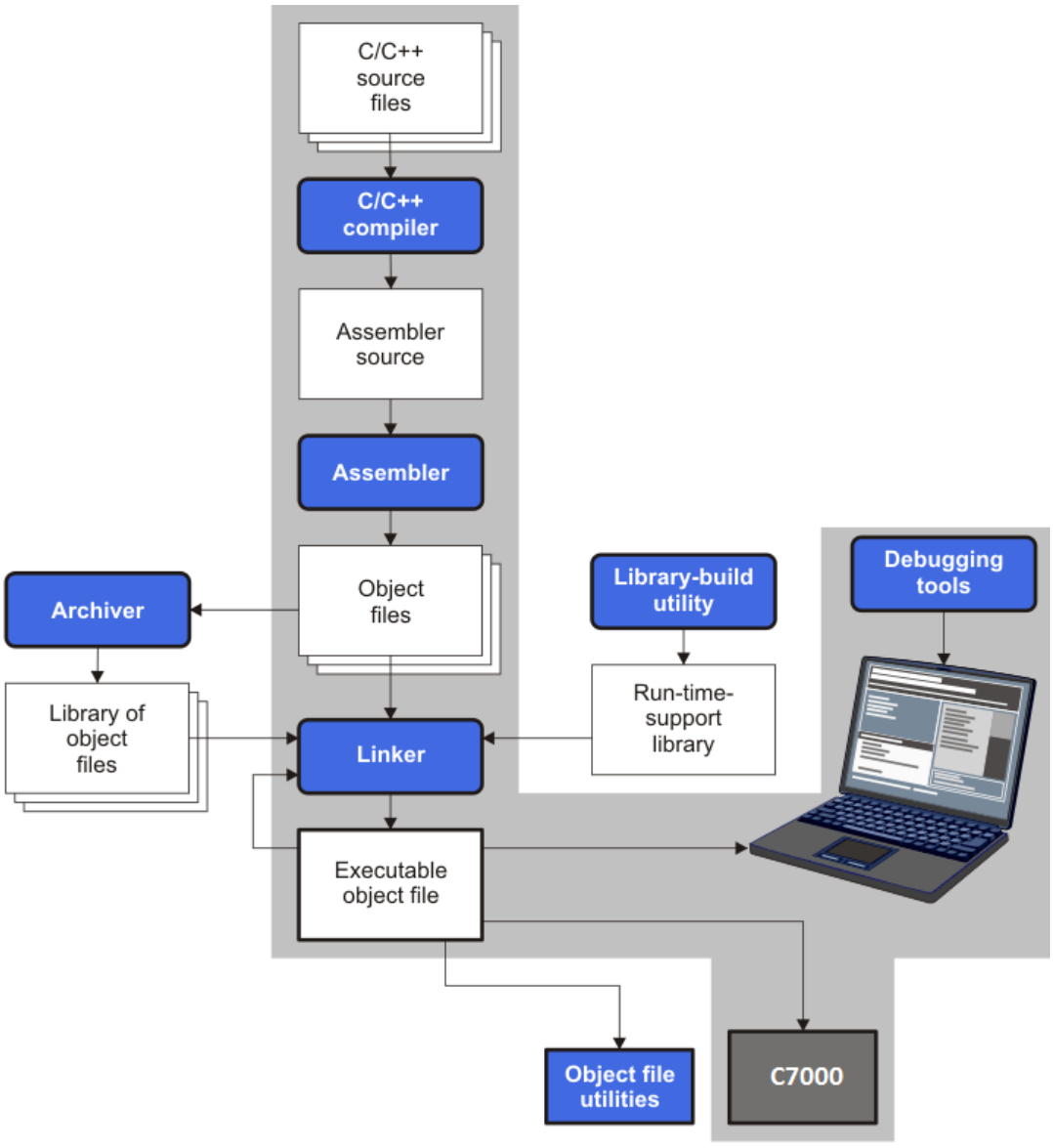


图 10-1. C7000 软件开发流程中的归档器

10.3 调用归档器

若要调用归档器，请输入：

ar7x[-]*command* [*options*] *libname* [*filename*$_1$ ... *filename*$_n$]

ar7x 是用于调用归档器的命令。

[-]*command* 命令归档器如何操作现有库成员和任何指定成员。可以在命令前面添加可选的连字符。调用归档器时，必须使用以下命令之一，但每次调用时只能使用一个命令。归档器命令如下所示：

- **@** 使用指定文件的内容而非命令行条目。用户可以使用此命令来避免主机操作系统对命令行长度的限制。在命令文件的行首使用；以包括注释。（有关使用归档器命令文件的示例，请参阅归档器命令文件。）
- **a** 将指定的文件添加到库。此命令不会替换与添加的文件同名的现有成员；而只是在归档末尾*附加*新成员。
- **d** 从库中删除指定的成员。
- **r** 替换库中的指定成员。如果不指定文件名，则归档器会用当前目录中的同名文件替换库成员。如果在库中找不到指定的文件，则归档器会添加而非替换它。
- **t** 输出库的目录。如果指定文件名，则仅列出这些文件。如果不指定任何文件名，则归档器会列出指定库中的所有成员。
- **x** 提取指定文件。如果不指定成员名称，则归档器会提取所有库成员。当归档器提取成员时，它只是将成员复制到当前目录；而*不会*将其从库中删除。

options 除了其中一个*命令*，用户还可以指定选项。若要使用选项，请将它们与命令结合在一起；例如，若要使用命令和 s 选项，请输入 -as 或 as。连字符仅对归档器选项是可选的。这些是归档器选项：

- **-h** 提供命令行帮助。
- **-q** （静默）不显示横幅和状态消息。
- **-s** 输出库中定义的全局符号列表。（此选项仅对 a、r 和 d 命令有效。）
- **-u** 仅当替换项具有更晚的修改日期时替换库成员。必须使用 r 命令和 -u 选项来指定要替换哪些成员。
- **-v** （详细）提供根据旧库及其成员创建新库的逐个文件说明。

libname 为要构建或修改的归档库命名。如果不为 *libname* 指定扩展名，则归档器使用默认扩展名 *.lib*。

filenames 指定要操作的单个文件的名称。这些文件可以是现有库成员或要添加到库中的新文件。输入文件名时，必须输入完整文件名，包括扩展名（如适用）。

备注

指定库成员的名称：一个库可以（但不可取）包含多个名称相同的成员。如果用户尝试删除、替换或提取的成员的名称与另一个库成员的名称相同，则归档器会删除、替换或提取使用该名称的第一个库成员。

10.4 归档器示例

以下是典型的归档器操作示例：

- 如果您要创建一个名为 function.lib 的库，其中包含文件 sine.obj、cos.obj 和 flt.obj，请输入：

```
ar7x -a function sine.obj cos.obj flt.obj
```

归档器的响应方式如下：

```
==> new archive 'function.lib'
==> building new archive 'function.lib'
```

- 您可以使用 -t 命令列印 function.lib 的目录，输入：

```
ar7x -t function
```

归档器的响应方式如下：

```
    SIZE   DATE                        FILE NAME
--------   ------------------------    -----------------
    4260   Thu Mar 28 15:38:18 2019    sine.obj
    4260   Thu Mar 28 15:38:18 2019    cos.obj
    4260   Thu Mar 28 15:38:18 2019    flt.obj
```

- 如果您要向库中添加新成员，请输入：

```
ar7x -as function atan.obj
```

归档器的响应方式如下：

```
    ==> symbol defined: '_sin'
    ==> symbol defined: '_cos'
    ==> symbol defined: '_tan'
    ==> symbol defined: '_atan
    ==> building archive 'function.lib'
```

因为此示例没有指定 libname 的扩展名，所以归档器会向名为 function.lib 的库中添加文件。如果 function.lib 不存在，归档器会创建它。 (-s 选项会告知归档器列出库中定义的全局符号。)

- 如果您要修改库成员，可以提取、编辑和替换它。在此示例中，假设有一个名为 sine.obj 的库，其中包含成员 push.asm、pop.asm 和 swap.asm。

```
ar7x -x function sine.obj
```

归档器会制作一份 sine.obj 的副本并将其放置在当前目录中；它不会从库中删除 sine.obj。现在，您可以检查或编辑提取出的文件。若要用编辑后的副本替换库中的 sine.obj 副本，请输入：

```
ar7x -r function sine.obj
```

- 如果您要使用命令文件，请在 -@ 命令后指定命令文件名。例如：

```
ar7x -@modules.cmd
```

归档器的响应方式如下：

```
    ==>  building archive 'modules.lib'
```

归档器命令文件 是 modules.cmd 命令文件。r 命令用于指定命令文件中给出的文件名会替换 modules.lib 库中的同名文件。-u 选项用于指定仅当当前文件的修订日期比库中的文件更新时才替换这些文件。

归档器命令文件

```
; Command file to replace members of the
;     modules library with updated files
; Use r command and u option:
ru
; Specify library name:
modules.lib
; List filenames to be replaced if updated:
align.obj
bss.obj
data.obj
text.obj
sect.obj
clink.obj
copy.obj
double.obj
drnolist.obj
emsg.obj
end.obj
```

10.5 库信息归档器说明

节 10.1 至节 10.4 说明了如何使用归档器创建在一个或多个应用的链接器中使用的目标文件的库。您可以有相同目标文件库的多个版本，每个版本都使用不同的构建选项集构建。例如，您可为大端字节序和小端字节序、不同架构版本或不同 ABI 使用不同的目标文件库版本，具体取决于客户端应用程序的典型构建环境。但是，如果您有几个库版本，则在了解特定应用程序需要链接哪个版本的库时可能会很麻烦。

当单个库的多个版本可用时，库信息归档器可用于创建所有目标文件库版本的索引库。此索引库在链接器中用于代替目标文件库的特定版本。链接器查看链接应用程序的构建选项，并使用指定的索引库确定在链接器中包括目标文件库的哪个版本。如果在索引库中找到一个或多个兼容的库，则为应用程序链接更合适的兼容库。

10.5.1 调用库信息归档器

若要调用库信息归档器，请输入以下命令：

libinfo7x [*options*] **--output**=*libname libname*$_1$ [*libname*$_2$ ... *libname*$_n$]

libinfo7x	是用于调用库信息归档器的命令。	
options	用于更改库信息归档器的默认行为。这些选项包括：	
	--output *libname*	指定要创建或更新的索引库的名称。此选项为必备项。
	--update	更新使用 --output 选项指定的索引库中的任何现有信息，而不是创建新索引。
libnames	指定要操作的单个目标文件库的名称。输入库名时，必须输入完整文件名，包括扩展名（如适用）。	

10.5.2 库信息归档器示例

我们来看看这些具有相同成员但使用不同构建选项构建的目标文件库：

目标文件库名称	构建选项
mylib_7100_le.lib	--silicon_version=7100
mylib_7100_be.lib	--silicon_version=7100 --big_endian

使用库信息归档器，用户可以根据上面的库创建名为 mylib.lib 的索引库：

```
libinfo7x --output mylib.lib mylib_7100_be.lib mylib_7100_le.lib
```

现在，用户可以指定 mylib.lib 作为应用链接器的库。链接器使用索引库来选择要使用的库的适当版本。如果在 --run_linker 选项之前指定了 --issue_remarks 选项，链接器会报告选择了哪个库。

- **示例 1**（小端字节序）：

```
cl7x -mv7100 --endian=little --issue_remarks main.c -z -l lnk.cmd ./mylib.lib
<Linking>
remark: linking in "mylib_7100_le.lib" in place of "mylib.lib"
```

- **示例 2**（大端字节序）：

```
cl7x -mv7100 --endian=big --issue_remarks main.c -z -l lnk.cmd ./mylib.lib
<Linking>
remark: linking in "mylib_7100_be.lib" in place of "mylib.lib"
```

10.5.3 列出索引库的内容

可以对索引库使用归档器的 -t 选项，以便列出索引库已索引的归档：

```
ar7x t mylib.lib
      SIZE   DATE                        FILE NAME
  --------   ------------------------    -----------------
       119   Wed Feb 03 12:45:22 2018    mylib_7100_be.lib
       119   Wed Feb 03 12:45:22 2018    mylib_7100_le.lib
         0   Wed Sep 30 12:45:22 2018    __TI_$$LIBINFO
```

已索引的目标文件库在归档器列表中具有附加的 .libinfo 扩展名。__TI_$$LIBINFO 成员非常特殊，其指定 *mylib.lib* 作为索引库，而不是常规库。

如果对索引库使用了归档器的 -d 命令来删除 .libinfo 成员，则在指定索引库时链接器将不再选择相应的库。

对索引库使用任何其他归档器选项，或使用 -d 来删除 __TI_$$LIBINFO 成员，会导致出现未定义的行为，且不受支持。

10.5.4 要求

您必须遵循以下要求，才能使用二进制索引文件：

- 链接器命令行上必须至少有一个应用目标文件出现在索引库之前。
- 指定为库信息归档器输入的每个目标文件库都只能包含使用相同构建选项构建的目标文件成员。
- 链接器需要索引库及其索引的所有库都位于同一目录中。

章节 11

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，请参阅章节 12。

11.1 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。

11.1.1 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：

cl7x --run_linker {**--rom_model | --ram_model**} *filenames*
[*options*] [**--output_file=** *name.out*] **--library=** *library* [*lnk.cmd*]

cl7x --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 cl7x --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 11.3.4。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。（章节 12 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，请参阅章节 12。

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：

```
cl7x --run_linker --ram_model prog1 prog2 prog3 --output_file=prog.out
     --library=rts7100_le.lib
```

11.1.2 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：

cl7x *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 11.1.1 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：

```
cl7x prog1.c prog2.c prog3.c --run_linker --ram_model --output_file=prog.out --
library=rts7100_le.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 cl7x --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 11.3.4。

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. C7X_C_OPTION 环境变量中 --run_linker 选项后面的参数

11.1.3 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 C7X_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。

11.2 链接器代码优化

这些技术用于进一步优化您的代码。

11.2.1 条件链接

默认情况下会执行条件链接。即，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 5.8.31）强制将包含特定符号的段包含在链接中。

可以使用 CODE_SECTION（节 5.8.5）和 DATA_SECTION（节 5.8.8）pragma 强制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，才能强制将该段包含在链接中。

CLINK pragma（节 5.8.2）指示包含此符号定义的段可以在条件链接期间予以删除。这是默认行为，因此 CLINK pragma 通常不起作用。

11.2.2 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，即使它们没有被使用。

假设有一个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 64 字节边界对齐以支持分支机制。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。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。

11.2.3 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。

11.3 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，请参阅章节 12 中的链接器说明。

11.3.1 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。

11.3.1.1 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 11.3.4。

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 11-1. 使用 --issue_remarks 选项

```
cl7x --silicon_version=7100 --issue_remarks main.c --run_linker --rom_model
<Linking>
remark: linking in "libc.a"
remark: linking in "rts7100_le.lib" in place of "libc.a"
```

11.3.1.2 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 C7X_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。

cl7x --run_linker {**--rom_model** | **--ram_model**} *filenames* **--library=** *libraryname*

11.3.1.3 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。

11.3.2 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 11.3.4。

11.3.3 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 6.7.2.6 讨论了全局构造函数表的格式。

11.3.4 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 6.7.2.4 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 6.7.2.3）。
- 在*加载时*初始化全局变量。使用 --ram_model 链接器选项（请参阅节 6.7.2.5）。

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 11.1）。

有关 EABI 使用 --rom_model 和 --ram_model 的链接约定的信息，请分别参阅节 6.7.2.3 和节 6.7.2.5。

11.3.5 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 6.1.1。

编译器创建两种基本类型的段：初始化段和未初始化段。表 11-1 总结了初始化段。表 11-2 总结了未初始化段。

表 11-1. 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 节 12.8.4.2）
.c7xabi.exidx	用于异常处理的索引表；只读（请参阅 --exceptions 选项）。
.c7xabi.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。
.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。
.got	全局偏移表。
.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 节 12.8。）
.ovly	复制除引导时间 (.binit) 复制表以外的表。只读数据。
.TI.crctab	生成的 CRC 校验表。只读数据。

表 11-2. 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
.stack	
.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。

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 节 12.5。

11.3.6 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 rts7100_le.lib 进行输入。

要链接该程序，请使用以下语法：

cl7x --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《节 12.5。

链接器命令文件

```
--rom_model
--heap_size=0x2000
--stack_size=0x0100
--library=rts7100_le.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
}
```

章节 12

链接器说明

C7000 链接器通过组合目标模块来创建静态可执行模块。本章介绍了用于创建静态可执行模块的链接器选项、指令和语句。此外，还讨论了目标库、命令文件和其他重要概念。

段的概念是链接器操作的基础；章节 8 包含关于段的详细讨论。

12.1 链接器概述

C7000 链接器使您能够在存储器映射中高效地分配输出段。链接器在组合目标文件时会执行以下任务：

- 将段分配到目标系统配置的存储器中
- 重定位符号和段以将它们分配给最终地址
- 解析输入文件之间未定义的外部引用

链接器命令语言可控制存储器配置、输出段定义和地址绑定。该语言支持表达式赋值和求值。需通过定义和创建相关设计的存储器模型来配置系统存储器。MEMORY 和 SECTIONS 这两个强大的指令可用于：

- 将段分配到特定的存储器区域中
- 组合目标文件段
- 在链接时定义或重新定义全局符号

12.2 链接器在软件开发流程中的作用

图 12-1 展示了链接器在软件开发流程中的作用。链接器可接受若干种类型的文件作为输入，包括目标文件、命令文件、库和部分链接的文件。链接器会创建一个可执行目标模块，可下载到若干种开发工具中，或由 C7000 器件执行。

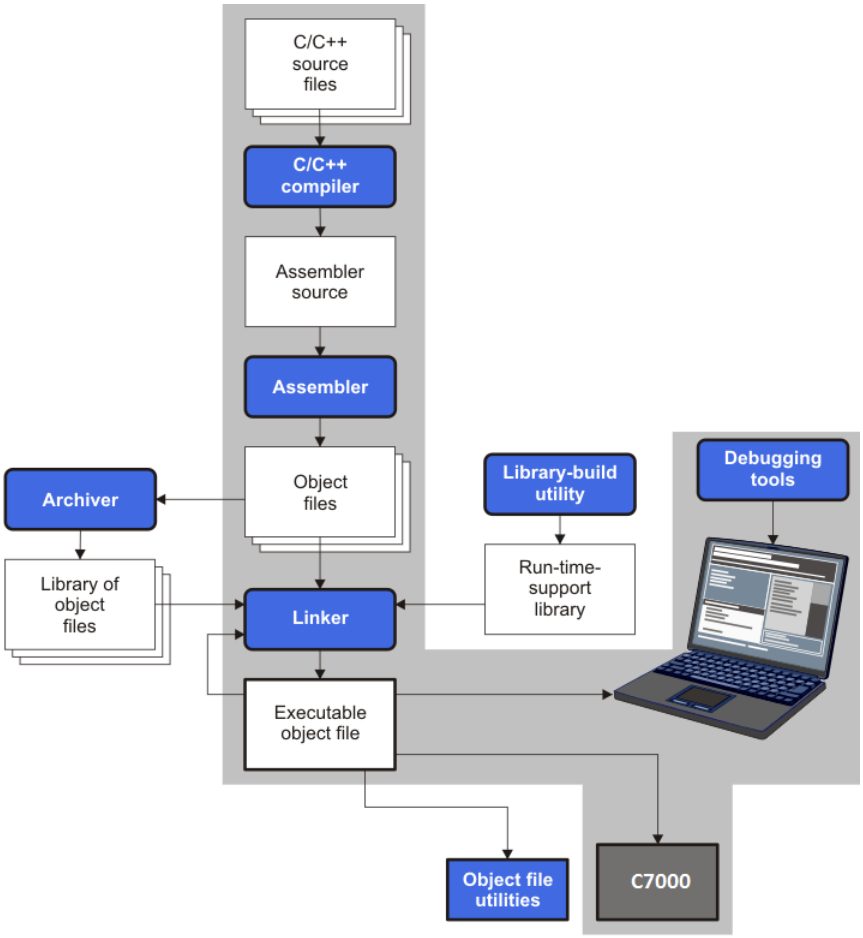


图 12-1. C7000 软件开发流程中的链接器

12.3 调用链接器

调用链接器的一般语法如下：

cl7x --run_linker [*options*] *filename*$_1$ .... *filename*$_n$

cl7x --run_linker	是用于调用链接器的命令。--run_linker 选项的短形式为 -z。
options	可出现在命令行或命令文件中的任意位置。（节 12.4 讨论了相关选项。）
filename $_1$, *filename* $_n$	可以是目标文件、链接器命令文件或存档库。输入文件的默认扩展名为 .c.obj（对于 C 源文件）和 .cpp.obj（对于 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 选项来指定输出文件的名称，否则默认输出文件名为 *a.out*。

备注

由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。从汇编源文件生成的目标文件仍然具有 .obj 扩展名。

有两种调用链接器的方法：

- 在命令行中指定选项和文件名。此示例将链接 file1.c.obj 和 file2.c.obj 两个文件，并创建一个名为 link.out 的输出模块。

```
cl7x --run_linker file1.c.obj file2.c.obj --output_file=link.out
```

- 将文件名和选项放在链接器命令文件中。在链接器命令文件中指定的文件名必须以字母开头。例如，假设文件 linker.cmd 包含以下命令行：

```
--output_file=link.out file1.c.obj file2.c.obj
```

  现在可以从命令行调用链接器；将命令文件名指定为输入文件：

```
cl7x --run_linker linker.cmd
```

  使用命令文件时，还可以在命令行中指定其他选项和文件。例如，可输入：

```
cl7x --run_linker --map_file=link.map linker.cmd file3.c.obj
```

  链接器在命令行中遇到文件名时立即读取并处理命令文件，因此会按以下顺序链接文件：file1.c.obj、file2.c.obj 和 file3.c.obj。此示例会创建一个名为 link.out 的输出文件和一个名为 link.map 的映射文件。

有关为 C/C++ 文件调用链接器的信息，请参阅节 12.10。

12.4 链接器选项

链接器选项可控制链接操作。可以在命令行上或命令文件中使用这些选项。链接器选项必须在前面加连字符 (-)。各选项可以用可选的空格与参数（如果选项带有参数）隔开。

表 12-1. 基本选项汇总

选项	别名	说明	段
--run_linker	-z	启用链接	节 12.3
--output_file	-o	为可执行输出模块命名。默认文件名为 a.out。	节 12.4.22
--map_file	-m	生成输入和输出段（包括空洞）的映射或列表，并将列表放置在 *文件名* 中。	节 12.4.17
--stack_size	-stack	将 C 系统栈大小设置为 *大小* 字节，并定义全局符号来指定栈大小。默认值 = 1K 字节	节 12.4.27
--heap_size	-heap	将堆大小（对于 C 中的动态存储器分配）设为 *大小* 字节，并定义全局符号来指定栈大小。默认值 = 1K 字节	节 12.4.13

表 12-2. 文件搜索路径选项汇总

选项	别名	说明	段
--library	-l	将归档库或链接命令 *文件名*命名为链接器输入	节 12.4.15
--disable_auto_rts		禁止自动选择运行时支持库	节 12.4.8
--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	节 12.4.15.3
--reread_libs	-x	强制重新读取库，以解析反向引用	节 12.4.15.3
--search_path	-i	在查找默认位置之前，更改库搜索算法以查找用 *路径名*命名的目录。此选项必须出现在 --library 选项之前。	节 12.4.15.1

表 12-3. 命令文件预处理选项汇总

选项	别名	说明	段
--define		将 *名称*预定义为预处理器宏。	节 12.4.10
--undefine		删除预处理器宏 *名称*。	节 12.4.10
--disable_pp		禁用命令文件预处理	节 12.4.10

表 12-4. 诊断选项汇总

选项	别名	说明	段
--diag_error		将由 *num* 标识的诊断分类为错误	节 12.4.7
--diag_remark		将由 *num* 标识的诊断分类为备注	节 12.4.7
--diag_suppress		抑制由 *num* 标识的诊断	节 12.4.7
--diag_warning		将由 *num* 标识的诊断分类为警告	节 12.4.7
--display_error_number		显示诊断的标识符及其文本	节 12.4.7
--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。	节 12.4.7
--emit_warnings_as_errors	-pdew	将警告视为错误	节 12.4.7
--issue_remarks		发出备注（非严重警告）	节 12.4.7
--no_demangle		禁止还原诊断中的符号名称	节 12.4.19
--no_warnings		抑制警告诊断（仍会发出错误）	节 12.4.7
--set_error_limit		将错误限值设置为 *num*。在达到此错误数量后，链接器将放弃链接。（默认为 100。）	节 12.4.7
--verbose_diagnostics		提供详细的诊断，以换行方式显示原始源代码	节 12.4.7
--warn_sections	-w	创建未定义的输出段时显示一条消息	节 12.4.31

表 12-5. 链接器输出选项汇总

选项	别名	说明	段
--absolute_exe	-a	生成绝对可执行模块。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。	节 12.4.3.1
--mapfile_contents		控制映射文件中包含的信息。	节 12.4.18
--relocatable	-r	生成不可执行、可重定位的输出模块	节 12.4.3.2
--xml_link_info		生成结构良好的 XML *文件*，其中包含有关链接结果的详细信息	节 12.4.32

表 12-6. 符号管理选项汇总

选项	别名	说明	段
--entry_point	-e	定义一个全局符号，用于指定输出模块的主要入口点	节 12.4.11
--globalize		将与*模式*匹配的符号的符号链接更改为全局型	节 12.4.16
--hide		隐藏与*模式*匹配的全局符号	节 12.4.14
--localize		将与*模式*匹配的符号的符号链接更改为局部型	节 12.4.16
--make_global	-g	将*符号*设为全局型（覆盖 -h）	节 12.4.16.1
--make_static	-h	将所有全局符号设为静态型	节 12.4.16.1
--no_sym_merge	-b	禁止合并符号调试信息	节 12.4.20
--no_symtable	-s	从输出模块中去除符号表信息和行号条目	节 12.4.21
--retain		保留原本应丢弃的段列表	节 12.4.25
--scan_libraries	-scanlibs	扫描所有库中的重复符号定义	节 12.4.26
--symbol_map		将符号引用映射到不同名称的符号定义	节 12.4.28
--undef_sym	-u	将未解析的外部*符号*放入输出模块的符号表中	节 12.4.30
--unhide		显示（取消隐藏）与 *模式* 匹配的全局符号	节 12.4.14

表 12-7. 运行时环境选项汇总

选项	别名	说明	段
--arg_size	--args	分配可供加载程序传递参数之用的存储器	节 12.4.4
--fill_value	-f	为输出段内的空洞设置默认填充值；*fill_value* 是 64 位常数	节 12.4.12
--ram_model	-cr	在加载时初始化变量	节 12.4.24
--rom_model	-c	在运行时自动初始化变量	节 12.4.24
--trampolines		生成 far call trampolines；默认开启	节 12.4.29

表 12-8. 其他选项汇总

选项	别名	说明	段
--linker_help	-help	显示有关语法和可用选项的信息	-
--minimize_trampolines		选择 trampoline 最小化算法（参数可选；算法默认为 postorder）	节 12.4.29.2
--preferred_order		为函数放置设定优先级	节 12.4.23
--zero_init		控制对未初始化变量的预初始化。默认为 on。如果使用了 --ram_model，则始终为 off。	节 12.4.33

12.4.1 文件、段和符号模式中的通配符

链接器允许使用星号 (*) 和问号 (?) 通配符来指定文件、段和符号名。使用 * 可匹配任意数量的字符，使用 ? 可匹配单个字符。使用通配符可以更方便地处理遵循适用命名规则的相关对象。例如：

```
mp3*.obj    /* 匹配以 mp3 开头的任何 .obj     */
task?.o*    /* 匹配 task1.obj、task2.c.obj、taskX.o55 等 */
SECTIONS
{
   .fast_code: { *.obj(*fast*) }              > FAST_MEM
   .vectors  : { vectors.c.obj(.vector:part1:*)> 0xFFFFFF00
   .str_code : { rts*.lib<str*.c.obj>(.text) }  > S1ROM
}
```

12.4.2 通过链接器选项指定 C/C++ 符号

链接时符号与 C/C++ 标识符名称相同。编译器*不会*在 C/C++ 标识符的开头添加下划线。

有关引用符号名称的更多信息，请参阅 节 5.11。

有关 C++ 符号命名的具体信息，请参阅节 5.11。

有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 12.6。

12.4.3 重定位功能 (--absolute_exe 和 --relocatable 选项)

链接器会执行重新定址，该过程即在符号的地址发生改变 时调整对符号的所有引用。

链接器支持两个选项 (--absolute_exe 和 --relocatable) ，让用户可以生成绝对输出模块或可重定位的输出模块。--absolute_exe 和 --relocatable 选项可能无法一同使用。

遇到不包含重定位或符号表信息的文件时，链接器会发出警告消息 (但会继续执行) 。只有每个输入文件均不包含需要重定位的信息 (即，每个文件都没有未解析的引用，并且都绑定至链接器创建它时所绑定的同一虚拟地址) 时，重新链接绝对文件才会成功。

12.4.3.1 生成绝对输出模块 (--absolute_exe 选项)

如果使用不带 --relocatable 选项的 --absolute_exe，链接器会生成*可执行的绝对输出模块*。绝对文件*不*包含重定址信息。可执行文件包含以下内容：

- 由链接器定义的特殊符号 (请参阅节 12.5.9.4)
- 介绍程序入口点等信息的标头
- *无*未解析的引用

以下示例链接 file1.c.obj 和 file2.c.obj，并生成绝对输出模块，被称为 a.out：

```
cl7x --run_linker --absolute_exe file1.c.obj file2.c.obj
```

备注

--absolute_exe 和 --relocatable 选项

如果不使用 --absolute_exe 或 --relocatable 选项，链接器的行为与指定 --absolute_exe 相同。

12.4.3.2 生成可重定位输出模块 (--relocatable 选项)

当您使用 --relocatable 选项时，链接器会在输出模块中保留重定位条目。如果输出模块会进行重定位 (加载时) 或重新链接 (由另一个链接器执行) ，请使用 --relocatable 来保留重定位条目。

当您使用不带 --absolute_exe 选项的 --relocatable 选项时，链接器会生成不可执行的文件。不可执行的文件不包含特殊链接器符号或可选文件头。该文件可以包含未解析的引用，但这些引用不会妨碍输出模块的创建。

此示例链接 file1.c.obj 和 file2.c.obj，并生成被称为 a.out 的可重定位输出模块：

```
cl7x --run_linker --relocatable file1.c.obj file2.c.obj
```

输出文件 a.out 可以在加载时与其他目标文件重新链接或进行重定位。（链接将与其他文件重新链接的文件被称为部分链接。如需更多信息，请参阅节 12.9。）

12.4.4 分配存储器供加载器使用以传递参数（--arg_size 选项）

--arg_size 选项会指示链接器分配存储器，这样加载器便能够从加载器的命令行向程序传递参数。--arg_size 选项的语法为：

--arg_size= *size*

size 是要在目标存储器中为命令行参数分配的字节数。

默认情况下，链接器会创建 __c_args__ 符号并将其设置为 -1。指定 --arg_size=*size* 时，会出现以下情况：

- 链接器会创建一个名为 .args 的 *size* 字节的未初始化段。
- __c_args__ 符号包含 .args 段的地址。

加载器和目标启动代码会使用 .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。有关加载器的信息，请参阅节 3.6

12.4.5 压缩（--cinit_compression 和 --copy_compression 选项）

默认情况下，链接器不压缩复制表（节 9.3.3 和节 12.8）源数据段。--cinit_compression 和 --copy_compression 选项指定通过链接器进行压缩。

--cinit_compression 选项指定链接器应用于 C 自动初始化复制表源数据段的压缩类型。默认为 lzss。

可使用链接器生成的复制表来管理重叠。为了节省 ROM 空间，链接器可压缩由复制表复制的数据。压缩数据在复制过程中被解压。--copy_compression 选项控制复制数据表的压缩。

这些选项的语法为：

--cinit_compression[=*compression_kind*]

--copy_compression[=*compression_kind*]

compression_kind 可以是以下类型之一：

- **off**。不要压缩数据。
- **rle**。使用行程编码格式压缩数据。
- **lzss**。使用 Lempel-Ziv-Storer-Szymanski 压缩格式压缩数据（如果未指定 *compression_kind*，这将是默认值）。

更多有关压缩的信息，请参阅节 12.8.5。

12.4.6 压缩 DWARF 信息（--compress_dwarf 选项）

--compress_dwarf 选项通过消除输入目标文件中的重复信息，以激进的方式减小 DWARF 信息的大小。

对于与 EABI 搭配使用的 ELF 目标文件，--compress_dwarf 选项可以消除无法通过使用 ELF COMDAT 组进行删除的重复信息。（有关 COMDAT 组的信息，请参阅 ELF 规范。）

12.4.7 控制链接器诊断

链接器按照某些 C/C++ 编译器选项来控制由链接器生成的诊断。必须在 --run_linker 选项之前指定诊断选项。

--diag_error=*num*

将 *num* 标识的诊断分类为错误。若要查找诊断消息的数字标识符，请在单独的链接中首先使用 --display_error_number 选项。然后使用 --diag_error=*num* 将诊断重新归类为错误。只能更改任意诊断的严重性。

--diag_remark=*num*	将 *num* 标识的诊断分类为备注。若要查找诊断消息的数字标识符，请在单独的链接中首先使用 --display_error_number 选项。然后使用 --diag_remark=*num* 将诊断重新归类为备注。只能更改任意诊断的严重性。
--diag_suppress=*num*	抑制 *num* 标识的诊断。若要查找诊断消息的数字标识符，请在单独的链接中首先使用 --display_error_number 选项。然后使用 --diag_suppress=*num* 抑制诊断。只能抑制任意诊断。
--diag_warning=*num*	将 *num* 标识的诊断分类为警告。若要查找诊断消息的数字标识符，请在单独的链接中首先使用 --display_error_number 选项。然后使用 --diag_warning=*num* 将诊断重新分类为警告。只能更改任意诊断的严重性。
--display_error_number	显示诊断的数字标识符及其文本。使用此选项确定需要向诊断抑制选项（--diag_suppress、--diag_error、--diag_remark 和 --diag_warning）提供哪些参数。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。有关如何理解诊断消息的更多信息，请参阅 节 3.7。
--emit_references:file [=*filename*]	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。此信息可用于确定为什么要将每个段包含在链接的应用中。输出文件是一个简单的 ASCII 文本文件。*filename* 用作创建的文件的基本名称。例如，--emit_references:file=myfile 在当前目录中生成一个名为 myfile.txt 的文件。
--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	抑制警告诊断（仍会发出错误）。
--set_error_limit=*num*	将错误限制设置为 *num*，可以是任何十进制值。在出现此数量的错误后，链接器将放弃链接。（默认为 100。）
--verbose_diagnostics	提供详细的诊断，以换行方式显示原始源代码，并指示错误在源代码行中的位置

12.4.8 自动选择库（--disable_auto_rts 选项）

--disable_auto_rts 选项会禁用运行时支持 (RTS) 库的自动选择。有关自动选择过程的详细信息，请参阅节 11.3.1.1

12.4.9 不要删除未使用的段（--unused_section_elimination 选项）

为了尽量减小占用空间，ELF 链接器不包含对于解析最终可执行文件中的任何引用而言不需要的段。使用 --unused_section_elimination=off 可禁用此优化。链接器默认行为等效于 --unused_section_elimination=on。

12.4.10 链接器命令文件预处理（--disable_pp、--define 和 --undefine 选项）

链接器使用标准 C 预处理器来预处理链接器命令文件。因此，命令文件可以包含众所周知的预处理指令，例如 #define、#include 和 #if / #endif。

三个链接器选项控制着预处理器：

--disable_pp	禁用命令文件预处理
--define=*name*[=*val*]	将 *name* 预定义为预处理器宏
--undefine=*name*	删除宏 *name*

编译器具有含义相同的 --define 和 --undefine 选项。但是，链接器选项不同；只有在 --run_linker 之后指定的 --define 和 --undefine 选项会传递给链接器。例如：

```
cl7x --define=FOO=1 main.c --run_linker --define=BAR=2 lnk.cmd
```

链接器只能看到 BAR 的 --define；编译器只能看到 FOO 的 --define。

当一个命令文件包含 (#include) 另一个命令文件时，预处理上下文以常规方式从父级传递到子级（即，在父级定义的宏在子级中可见）。但是，当不是通过 #include 来调用命令文件时，无论是在命令行中还是通过在另一个命

令文件中进行指定的常规方式，预处理上下文都**不会**传递到嵌套的文件中。 例外情况是 --define 和 --undefine 选项，这些选项是从遇到的位置开始全局应用。例如：

```
--define GLOBAL
#define LOCAL
#include "incfile.cmd"     /*看到 GLOBAL 和 LOCAL */
nestfile.cmd               /* 仅看到 GLOBAL       */
```

在命令文件中使用 --define 和 --undefine 时有两点注意事项。首先，它们具有如上所述的全局影响。其次，它们本身实际上并不是预处理指令，因此它们会被宏替换，进而可能会产生意想不到的后果。为了消除这种影响，可以在符号名称外添加引号。例如：

```
--define MYSYM=123
--undefine MYSYM     /* 扩展到 --undefine 123 (!)*/
--undefine "MYSYM"   /* 啊，这样更好             */
```

链接器按照以下顺序搜索 #include 文件，直到找到该文件：

1. 如果 #include 文件名位于引号中（而不是 <尖括号> 中），则在包含当前文件的目录中进行搜索。
2. 如果使用了 --include_path 编译器选项（在 --run_linker 或 -z 选项之前），请搜索使用该选项指定的路径。
3. 如果定义了 环境变量，则搜索该定义指向的目录。请参阅节 12.4.15.2。

有两个例外：相对路径名（如“../name”）总是相对于当前目录进行搜索；绝对路径名（例如“/usr/tools/name”）完全绕过搜索路径。

链接器提供了表 12-9 中列出的内置宏定义。链接器中这些宏的可用性取决于所使用的命令行选项，而不是所链接文件的构建属性。如果这些宏未按预期设置，请确认工程命令行使用了正确的编译器选项设置。

表 12-9. 预定义的 C7000 宏名称

宏名称	说明
__DATE__	扩展到 *mmm dd yyyy* 形式的编译日期
__FILE__	扩展到当前源文件名
__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__	如果启用了 EABI，则已定义为 1；否则未定义。
__TIME__	扩展到“*hh:mm:ss*”形式的编译时间
__C7000__	始终已定义
__C7100__	如果 --silicon_version=7100，则已定义为 1。
__C7120__	如果 --silicon_version=7120，则已定义为 1。
__C7504__	如果 --silicon_version=7504，则已定义为 1。

12.4.11 定义入口点（--entry_point 选项）

开始执行程序的存储器地址被称为*入口点*。当加载器将程序加载到目标存储器中时，必须将程序计数器 (PC) 初始化为入口点；然后，PC 指向程序的开头。

链接器可为入口点分配四个值之一。下面按照链接器尝试使用值的顺序列出了这些值。如果使用前三个值之一，该值必须是符号表中的一个外部符号。

- 由 --entry_point 选项指定的值。语法为：

  --entry_point= *global_symbol*

  其中，*global_symbol* 定义入口点，并且必须定义为输入文件的外部符号。C 或 C++ 对象的外部符号名称可能与源语言中声明的名称不同。
- 符号 _c_int00 的值（如果存在）。如果要链接由 C 编译器生成的代码，则 _c_int00 符号*必须*作为入口点。
- 符号 main 的值（如果存在）

- 0（默认值）

以下示例会链接 file1.c.obj 和 file2.c.obj。符号 begin 是入口点；begin 必须在 file1 或 file2 中定义为 external。

```
cl7x --run_linker --entry_point=begin file1.c.obj file2.c.obj
```

有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 12.6。

12.4.12 设置默认填充值（--fill_value 选项）

--fill_value 选项用于填充输出段中的孔洞。此选项的语法为：

--fill_value= *value*

参数 *value* 为 64 位常数（最多 16 个十六进制数字）。如果不使用 --fill_value，链接器会使用 0 作为默认填充值。

下面的示例使用十六进制值 ABCDABCD 来填充孔洞：

```
cl7x --run_linker --fill_value=0xABCDABCDABCDABCD file1.c.obj file2.c.obj
```

12.4.13 定义堆大小（--heap_size 选项）

对于 malloc() 使用的 C 运行时存储器池，C/C++ 编译器使用一个名为 的未初始化段。可在链接时使用 --heap_size 选项来设置此存储器池的大小。--heap_size 选项的语法为：

--heap_size= *size*

size 必须是一个常量。以下示例定义了一个 4K 字节的堆：

```
cl7x --run_linker --heap_size=0x1000 /* defines a 4k heap (.sysmem section)*/
```

链接器创建 段的前提是输入文件中存在 段。

链接器还会创建全局符号 ，并为其分配一个等于堆大小的值。默认大小为 1K 字节。有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 12.6。

12.4.14 隐藏符号

“符号隐藏”可阻止符号在输出文件的符号表中被列出。局部化用于防止链接单元中出现名称空间冲突（请参阅节 12.4.16），而“符号隐藏”用于隐藏不应在链接单元外可见的符号。此类符号的名称在目标文件阅读器中仅显示为空字符串或“no name”。链接器通过 --hide 和 --unhide 选项支持符号隐藏。

这些选项的语法为：

--hide=' *pattern* **'**

--unhide=' *pattern* **'**

pattern 是一个“glob”（带有可选 ? 或 * 通配符的字符串）。? 用于匹配单个字符。* 用于匹配零个或多个字符。

--hide 选项会隐藏链接名称与 *模式*相匹配的全局符号。它通过将名称更改为空字符串来隐藏与模式匹配的符号。隐藏的全局符号也会局部化。

--unhide 选项显示（取消隐藏）与 --hide 选项隐藏的*模式*相匹配的全局符号。--unhide 选项从符号隐藏中排除与模式相匹配的符号，前提是由 --unhide 定义的模式比由 --hide 定义的模式具有更严格的限制。

这些选项具有以下属性：

- 可在命令行上多次指定 --hide 和 --unhide 选项。
- --hide 和 --unhide 的顺序不重要。
- 一个符号只与一个由 --hide 或 --unhide 定义的模式相匹配。
- 一个符号与最严格的模式相匹配。如果模式 A 与比模式 B 更窄的集相匹配，则认为模式 A 比模式 B 更严格。

- 如果一个符号与来自 --hide 和 --unhide 的模式相匹配，但两个模式之间互不取代，那么这属于错误。如果模式 A 可以匹配模式 B 能够匹配的所有内容甚至更多内容，则认为 模式 A 取代模式 B。如果模式 A 取代模式 B，则认为模式 B 比模式 A 具有更严格的限制。
- 这些选项会影响最终链接和部分链接。

在映射文件中，这些符号列在“Hidden Symbols”（隐藏符号）标题下。

12.4.15 改变库搜索算法（--library、--search_path 和 C7X_C_DIR）

通常，需要指定一个文件作为链接器输入时，只需输入文件名，链接器便会在当前目录中查找文件。例如，假设当前目录包含 object.lib 库。如果此库定义了 file1.c.obj 文件中引用的符号，则文件的链接方式如下：

```
cl7x --run_linker file1.c.obj object.lib
```

若要使用不在当前目录中的文件，请使用 --library 链接器选项。--library 选项的缩写形式为 -l。 此选项的语法为：

--library=[*pathname*] *filename*

filename 是存档、目标文件或链接器命令文件的名称。最多可以指定 128 个搜索路径。

当对象库的一个或多个成员被指定用作输出段的输入时，不需要 --library 选项。有关分配存档成员的更多信息，请参阅节 12.5.5.5。

可使用 --search_path 链接器选项或 C7X_C_DIR 环境变量来调整链接器的目录搜索算法。链接器按以下顺序搜索对象库和命令文件：

1. 搜索使用 --search_path 链接器选项指定的目录。--search_path 选项必须出现在命令行上或命令文件中的 --library 选项之前。
2. 搜索使用 C7X_C_DIR 指定的目录。
3. 如果未设置 C7X_C_DIR，请搜索使用 C7X_A_DIR 环境变量指定的目录。
4. 搜索当前目录。

12.4.15.1 指定备用库目录（--search_path 选项）

--search_path 选项指定一个包含输入文件的备用目录。--search_path 选项的缩写形式为 `-I`。此选项的语法为：

--search_path= *pathname*

pathname 指定一个包含输入文件的目录。

链接器在搜索使用 --library 选项命名的文件时，首先会搜索使用 --search_path 指定的目录。每个 --search_path 选项仅指定一个目录，但用户可以在每次调用时使用多个 --search_path 选项。如果使用 --search_path 选项指定备用目录，其必须位于命令行或命令文件中的任何 --library 选项之前。

例如，假设有两个名为 r.lib 和 lib2.lib 的存档库位于 ld 和 ld2 目录中。下表显示了 r.lib 和 lib2.lib 所在的目录、如何设置环境变量，以及如何在链接期间使用这两个库。请根据操作系统选择相应的行：

操作系统	输入
UNIX (Bourne shell)	`cl7x --run_linker f1.c.obj f2.c.obj --search_path=/ld --search_path=/ld2 --library=r.lib --library=lib2.lib`
Windows	`cl7x --run_linker f1.c.obj f2.c.obj --search_path=\ld --search_path=\ld2 --library=r.lib --library=lib2.lib`

12.4.15.2 指定备用库目录（C7X_C_DIR 环境变量）

环境变量是由您定义并为其分配字符串的系统符号。链接器使用名为 C7X_C_DIR 的环境变量来指定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)	**C7X_C_DIR ="** *pathname*$_1$ **;** *pathname*$_2$ **; ... "; export C7X_C_DIR**

操作系统	输入
Windows	**set C7X_C_DIR =** *pathname*$_1$; *pathname*$_2$; ...

pathnames 是包含输入文件的目录。在命令行或命令文件中使用 --library 链接器选项可告知链接器要搜索哪个库或链接器命令文件。路径名必须遵循以下约束条件：

- 路径名必须用分号分隔。
- 路径开头或结尾的空格或制表符将被忽略。例如，下面的分号前后的空格被忽略：

```
set C7X_C_DIR= c:\path\one\to\tools ; c:\path\two\to\tools
```

- 路径中允许使用空格和制表符来适应包含空格的 Windows 目录。例如，下述路径名是有效的：

```
set C7X_C_DIR=c:\first path\to\tools;d:\second path\to\tools
```

在以下示例中，假设两个名为 r.lib 和 lib2.lib 的存档库位于 ld 和 ld2 目录中。下表显示了如何设置环境变量，以及如何在链接期间使用这两个库。请根据操作系统选择相应的行：

操作系统	调用命令
UNIX (Bourne shell)	`C7X_C_DIR="/ld ;/ld2"; export C7X_C_DIR;` `cl7x --run_linker f1.c.obj f2.c.obj --library=r.lib --library=lib2.lib`
Windows	`C7X_C_DIR=\ld;\ld2` `cl7x --run_linker f1.c.obj f2.c.obj --library=r.lib --library=lib2.lib`

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 C7X_C_DIR`
Windows	`set C7X_C_DIR=`

汇编器使用名为 C7X_A_DIR 的环境变量来指定包含复制文件/头文件的备用目录。如果未设置 C7X_C_DIR，则链接器会在使用 C7X_A_DIR 指定的目录中搜索对象库。有关对象库的更多信息，请参阅节 12.6.7。

12.4.15.3 详尽读取和搜索库 (--reread_libs 和 --priority 选项)

有两种方法可以详尽搜索未解析的符号：

- 在无法解析符号引用的情况下重新读取库 (--reread_libs)。
- 按照指定库的顺序来搜索库 (--priority)。

链接器通常仅在命令行或命令文件中遇到输入文件 (包括存档库) 时读取一次。读取存档时，任何对未定义符号的引用进行解析的成员均包含在链接中。如果某个输入文件稍后引用先前读取的存档库中定义的符号，则该引用不会被解析。

使用 --reread_libs 选项可以强制链接器重新读取所有库。链接器会重新读取库，直到没有更多引用可供解析为止。使用 --reread_libs 进行链接的速度可能会更慢，因此应该仅在需要时使用。例如，如果 a.lib 包含对 b.lib 中所定义符号的引用，而 b.lib 包含对 a.lib 中所定义符号的引用，则可以通过两次列出这两个库的其中之一来解析这些相互依赖关系，如下所示：

```
cl7x --run_linker --library=a.lib --library=b.lib --library=a.lib
```

或者也可以强制链接器为您执行该操作：

```
cl7x --run_linker --reread_libs --library=a.lib --library=b.lib
```

--priority 选项为库提供了另一种搜索机制。使用 --priority 会使包含相应符号定义的第一个库满足每个未被解析的引用。例如：

```
objfile  references A
lib1     defines B
lib2     defines A, B; obj defining A references B
```

在现有模型下，objfile 解析其对 lib2 中 A 的引用，拉入对 B 的引用，进而解析为 lib2 中的 B。

在 --priority 下，objfile 解析其对 lib2 中 A 的引用，拉入对 B 的引用，但现在会通过按顺序搜索库来解析 B，并将 B 解析为找到的第一个定义，即 lib1 中的定义。

如果库为其他库中的相关函数集提供覆盖定义而无需提供整个库的完整版本，则适合使用 --priority 选项。

例如，假设您想覆盖 rts7100_le.lib 中定义的 malloc 和 free 版本，而不提供对 rts7100_le.lib 的完全替代。 在 rts7100_le.lib 之前使用 --priority 并链接您的新库可确保所有对 malloc 和 free 的引用都解析为新库。

--priority 选项的作用是在发生上述情况时支持使用 SYS/BIOS 来链接程序。

12.4.16 更改符号局部化

符号局部化会将符号链接从全局更改为局部（静态）。此功能用于隐藏不应该广泛可见但必须是全局符号（因为它们被库中的多个模块访问）的符号。链接器通过 --localize 和 --globalize 链接器选项支持符号局部化。

这些选项的语法为：

--localize=' *pattern* **'**

--globalize=' *pattern* **'**

pattern 是一个“glob”（带有可选 ? 或 * 通配符的字符串）。? 用于匹配单个字符。* 用于匹配零个或多个字符。

--localize 选项将与 *pattern* 匹配的符号的符号链接更改为局部。

--globalize 选项将与 *pattern* 匹配的符号的符号链接更改为全局。--globalize 选项仅影响由 --localize 选项局部化的符号。--globalize 选项从符号局部化中排除与模式匹配的符号，前提是 --globalize 定义的模式比 --localize 定义的模式具有更严格的限制。

有关在链接器选项（如 --localize 和 --globalize）中使用 C/C++ 标识符的信息，请参阅节 12.4.2。

这些选项具有以下属性：

- 可在命令行上多次指定 --localize 和 --globalize 选项。
- --localize 和 --globalize 选项的顺序不重要。
- 一个符号只与一个由 --localize 或 --globalize 定义的模式匹配。
- 一个符号与最严格的模式匹配。如果模式 A 比模式 B 匹配更窄的集合，则模式 A 被认为比模式 B 更严格。
- 如果一个符号与来自 --localize 和 --globalize 的模式匹配，但两个模式之间互不取代，那么这是错误的。如果模式 A 可以匹配模式 B 能够匹配的所有内容甚至更多内容，则认为 A 取代 B。如果模式 A 取代模式 B，则认为模式 B 比模式 A 具有更严格的限制。
- 这些选项会影响最终和部分链接。

在映射文件中，这些符号列在“Localized Symbols”（局部化符号）标题下。

12.4.16.1 将所有全局符号设为静态（--make_static 选项）

--make_static 选项会将所有全局符号都设为静态。静态符号对外部链接的模块不可见。通过将全局符号设为静态，全局符号本质上将是隐藏状态。这样一来，同名（在不同文件中）的外部符号将被视为具有唯一性。

--make_static 选项实际上会使所有 .global 汇编器指令无效。所有符号都成为了定义它们的模块中的局部符号，因此不能有外部引用。例如，假设 file1.c.obj 和 file2.c.obj 都定义了名为 EXT 的全局符号。使用 --make_static 选项可以确保在链接这些文件时不发生冲突。将分开处理 file1.c.obj 中定义的符号 EXT 与 file2.c.obj 中定义的符号 EXT。

```
cl7x --run_linker --make_static file1.c.obj file2.c.obj
```

--make_static 选项会将所有全局符号都设为静态。如果您有一个想要保持全局属性的符号，并且您使用 --make_static 选项，则可以使用 --make_global 选项将该符号声明为全局。--make_global 选项会使您为符号指定的 --make_static 选项无效。--make_global 选项的语法为：

--make_global= *global_symbol*

12.4.17 创建映射文件（--map_file 选项）

--map_file 选项的语法为：

--map_file= *filename*

链接器映射会描述：

- 存储器配置
- 输入和输出段分配
- 由链接器生成的复制表
- Trampoline
- 外部符号重定位后的地址
- 隐藏的和局部化的符号

映射文件包含输出模块的名称以及入口点，并且最多还可以包含三个表：

- 一个表显示当 MEMORY 指令指定任何非默认配置时的新存储器配置。该表具有从链接器命令文件中的 MEMORY 指令生成的以下列。有关 MEMORY 指令的信息，请参阅节 12.5.4。
  - **Name。**这是使用 MEMORY 指令指定的存储器范围的名称。
  - **Origin。**这是存储器范围的起始地址。
  - **Length。**这是存储器范围的长度。
  - **Unused。**这是该存储器区域中未使用（可用）的存储器总量。
  - **Attributes。**这是与指定范围相关联的一到四个属性：

    R 指定可以读取存储器。

    W 指定可以写入存储器。

    X 指定存储器可包含可执行代码。

    I 指示可以初始化存储器。

- 一个表显示每个输出段以及构成输出段的输入段的链接地址（段放置映射）。该表具有以下列；此信息是根据链接器命令文件内 SECTIONS 指令中的信息生成的：
  - **Output section。**这是使用 SECTIONS 指令指定的输出段的名称。
  - **Origin。**为每个输出段列出的第一个原点是该输出段的起始地址。以缩进格式列出的原点值是输出段中该部分的起始地址。
  - **Length。**为每个输出段列出的第一个长度是该输出段的长度。以缩进格式列出的长度值是输出段中该部分的长度。
  - **Attributes/input sections。**这列出了与输出段相关联的输入文件或值。如果无法分配输入段，映射文件将用“FAILED TO ALLOCATE”指示这一情况。

  有关 SECTIONS 指令的更多信息，请参阅节 12.5.5。
- 一个表显示每个外部符号及其地址（按符号名称排序）。
- 一个表显示每个外部符号及其地址（按符号地址排序）。

以下示例会链接 file1.c.obj 和 file2.c.obj，并创建一个名为 map.out 的映射文件：

```
cl7x --run_linker file1.c.obj file2.c.obj --map_file=map.out
```

输出映射文件，demo.map 展示了一个映射文件的示例。

12.4.18 管理映射文件内容（--mapfile_contents 选项）

--mapfile_contents 选项可帮助管理由链接器生成的映射文件的内容。--mapfile_contents 选项的语法为：

--mapfile_contents= *filter*[, *filter*]

指定 --map_file 选项后，链接器会生成一个映射文件，其中包含有关存储器使用情况的信息、链接期间创建的段的放置信息、有关链接器生成的复制表的详细信息，以及符号值。

--mapfile_contents 选项提供了一种机制来控制在映射文件中包含或排除哪些信息。从命令行指定 --mapfile_contents=help 时，系统将显示一个帮助屏幕，其中列出了可用的过滤器选项。提供了以下过滤器选项：

属性	说明	默认状态
copytables	复制表	打开
entry	入口点	打开
load_addr	显示加载地址	关闭
memory	存储器范围	打开
modules	模块视图	打开
sections	段	打开
sym_defs	每个文件定义的符号	关闭
sym_dp	按数据页排序的符号	打开
sym_name	按名称排序的符号	打开
sym_runaddr	按运行地址排序的符号	打开
all	启用所有属性	
none	禁用所有属性	

--mapfile_contents 选项通过指定以逗号分隔的显示属性列表来控制显示过滤器设置。如果前缀为字 no，则会禁用而不是启用属性。例如：

```
--mapfile_contents=copytables,noentry
--mapfile_contents=all,nocopytables
--mapfile_contents=none,entry
```

默认情况下会包含在指定 --map_file 选项时当前包含在映射文件中的那些段。--mapfile_contents 选项中指定的过滤器按照它们在命令行中出现的顺序进行处理。在上面的第三个示例中，第一个过滤器 none 会清除所有映射文件内容。然后，第二个过滤器 entry 使有关入口点的信息能够包含在生成的映射文件中。也就是说，当指定 --mapfile_contents=none,entry 时，映射文件*仅*包含有关入口点的信息。

load_addr 和 sym_defs 属性默认都是禁用的。

如果启用 load_addr 过滤器，则映射文件除了运行地址外还包括符号列表中包含的符号的加载地址（如果加载地址与运行地址不同）。

可使用 sym_defs 过滤器来包含按逐个文件排序的信息。您可能会发现，指定以下 --mapfile_contents 选项将映射文件的 sym_name、sym_dp 和 sym_runaddr 段替换为 sym_defs 段会很有用：

```
--mapfile_contents=nosym_name,nosym_dp,nosym_runaddr,sym_defs
```

默认情况下，有关应用中定义的全局符号的信息会包含在按名称、数据页和运行地址排序的表中。 如果使用 --mapfile_contents=sym_defs 选项，还会列出静态变量。

12.4.19 禁用名称还原 (--no_demangle)

默认情况下，链接器在诊断中使用已还原的符号名称。例如：

未定义的符号 首次在文件中引用

ANewClass::getValue()	test.cpp.obj

--no_demangle 选项改为在诊断中显示符号的链接名称。例如：

未定义的符号	首次在文件中引用
_ZN9ANewClass8getValueEv	test.cpp.obj

有关引用目标文件符号名称的信息，请参阅节 5.11。有关链接器符号命名的具体信息，请参阅节 12.6。

12.4.20 合并符号调试信息

默认情况下，链接器会消除符号调试信息的重复条目。在编译 C 程序以进行调试时，通常会生成此类重复信息。例如：

```
-[ header.h ]-
typedef struct
{
   <define some structure members>
} XYZ;
-[ f1.c ]-
#include "header.h"
...
-[ f2.c ]-
#include "header.h"
...
```

当编译这些文件以进行调试时，f1.c.obj 和 f2.c.obj 都有符号调试条目用于描述类型 XYZ。对于最终的输出文件，只需要一组这样的条目。链接器会自动消除重复条目。

如果希望链接器在目标文件中保留此类重复条目，请使用 --no_sym_merge 选项。使用 --no_sym_merge 选项可以使链接器运行得更快，并且在链接期间使用更少的主机存储器，但由于具有重复的调试信息，生成的可执行文件可能非常大。

12.4.21 去除符号信息（--no_symtable 选项）

使用 --no_symtable 选项可以忽略符号表信息和行号条目，从而可以创建较小的输出模块。对于生产应用程序，当您不想向消费者披露符号信息时，--no_sym_table 选项会非常有用。

此示例会链接 file1.c.obj 和 file2.c.obj 并创建一个输出模块，其中的行号和符号表信息会被去除并命名为 nosym.out：

```
cl7x --run_linker --output_file=nosym.out --no_symtable file1.c.obj file2.c.obj
```

使用 --no_symtable 选项会限制之后对符号调试器的使用。

备注

去除符号信息：--no_symtable option 现已被弃用。若要去除符号表信息，请按照节 13.4 中所述，使用 strip7x 实用程序。

12.4.22 指定输出模块（--output_file 选项）

当没有遇到错误时，链接器会创建一个输出模块。如果您没有为输出模块指定文件名，则链接器会为其提供默认名称 a.out。如果要将输出模块写入其他文件，请使用 --output_file 选项。--output_file 选项的语法为：

--output_file= *filename*

filename 是新输出模块的名称。

以下示例将链接 file1.c.obj 和 file2.c.obj，并创建一个名为 run.out 的输出模块：

```
cl7x --run_linker --output_file=run.out file1.c.obj file2.c.obj
```

12.4.23 确定函数放置优先级（--preferred_order 选项）

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*

12.4.24 C 语言选项（--ram_model 和 --rom_model 选项）

--ram_model 和 --rom_model 选项促使链接器使用 C 编译器所需的链接惯例。这两个选项都通知链接器该程序是一个 C 程序并且需要一个启动例程。

- --ram_model 选项指示链接器在加载时初始化变量。
- --rom_model 选项指示链接器在运行时自动初始化变量。

如果您使用不编译任何 C/C++ 文件的链接器命令行，则必须使用 --rom_model 或 --ram_model 选项。如果命令行在需要时未能包含这些选项之一，则您将看到消息“warning: no suitable entry-point found; setting to 0”（警告： 没有找到合适的入口点；设置为 0）。

如果您使用单个命令行进行编译和链接，则 --rom_model 是默认选项。如果使用了 --rom_model 或 --ram_model 选项，该该选项必须跟在 --run_linker 选项之后。

如需更多信息，请参阅节 12.10、节 9.3.2.1 和节 9.3.2.2。

12.4.25 保留丢弃的段（--retain 选项）

当 --unused_section_elimination 为 on 时，ELF 链接器不会在最终链接中包含可执行文件解析引用时不需要用到的段。--retain 选项会让链接器保留原本不会保留的段的列表。此选项支持通配符“*”和“?”。使用通配符时，参数两边应加上引号。此选项的语法为：

--retain=*sym_or_scn_spec*

--retain 选项采取以下形式之一：

- **--retain=** *symbol_spec*

  指定该符号格式会保留定义 *symbol_spec* 的段。例如，以下代码会保留用于定义以 init 开头的符号的段：

  ```
  --retain='init*'
  ```

  您无法指定 --retain='*'。

- **--retain=** *file_spec*(*scn_spec*[, *scn_spec*, ...]

  指定该文件格式会保留与 *file_spec* 匹配的文件中的一个或多个 *scn_spec* 匹配的段。例如，以下代码会保留所有输入文件中的 .intvec 段：

  ```
  --retain='*(.int*)'
  ```

  您可以指定 --retain='*(*)' 来保留所有输入文件中的所有段。不过，这不能阻止库成员中的段被优化掉。
- **--retain=** *ar_spec*<*mem_spec*, [*mem_spec*, ...>(*scn_spec*[, *scn_spec*, ...]

  指定该归档格式会保留以下位置与一个或多个 *scn_spec* 匹配的段：与 *ar_spec* 匹配的归档文件中与一个或多个 *mem_spec* 匹配的成员内。例如，以下代码会保留 rts7100_le.lib 库中 printf.c.obj 内的 .text 段：

  ```
  --retain=rts7100_le.lib<printf.c.obj>(.text)
  ```

  如果使用 --library 选项指定了库 (--library=rts7100_le.lib)，则会使用库搜索路径来搜索该库。您无法指定 '*<*>(*)'。

12.4.26 扫描所有库中的重复符号定义（--scan_libraries）

--scan_libraries 选项用于在链接期间扫描所有库，以查找链接中实际所含那些符号的重复符号定义。该扫描不会考虑绝对符号和 COMDAT 段中定义的符号。--scan_libraries 选项有助于确定链接器实际所选的那些符号与库中相同符号的其他现有定义。

库扫描功能可用于在库中存在多个定义时，根据定义检查符号引用的意外解析。

12.4.27 定义栈大小（--stack_size 选项）

C7000 C/C++ 编译器使用未初始化的段 .stack 来为运行时栈分配空间。可在链接时使用 --stack_size 选项设置此段的大小（以字节为单位）。--stack_size 选项的语法为：

--stack_size= *size*

size 必须是常量并以字节为单位。以下示例定义了一个 4K 字节的栈：

```
cl7x --run_linker --stack_size=0x1000 /* defines a 4K heap (.stack section)*/
```

如果在输入段中指定了其他的栈大小，则输入段的栈大小将被忽略。输入段中定义的任何符号仍然有效；只是栈大小不同。

链接器在定义 .stack 段时，还会定义全局符号 ，并为其分配一个等于该段大小的值。默认的软件栈大小为 1K 字节。有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 12.6。

12.4.28 符号映射（--symbol_map 选项）

符号映射允许由其他名称的符号来解析符号引用，这样便可使用替代定义来覆盖函数。这可用于在替代实现中打补丁以提供补丁（错误修复）或替代功能。--symbol_map 选项的语法为：

--symbol_map= *refname=defname*

例如，以下代码使链接器通过 foo_patch 定义来解析对 foo 的任何引用：

```
--symbol_map=foo=foo_patch
```

使用 --symbol_map 选项传递的字符串不应包含空格，也不应在外侧使用引号。这样，可以在命令行、链接器命令文件和选项文件中使用相同的链接器选项语法。

12.4.29 生成 Far 调用 Trampoline（--trampolines 选项）

C7000 器件具有 PC 相关的调用指令和 PC 相关的分支指令，其范围小于整个地址空间。使用这些指令时，目标地址必须足够靠近指令，以便调用和目标之间的差异适应可用的编码位。如果被调用函数离调用函数太远，链接器会生成错误或生成 trampoline，具体取决于 --trampolines 选项的设置（on 还是 off）。

PC 相关调用的替代项是绝对调用，通常作为间接调用来实现：将被调用地址加载到寄存器中，然后调用该寄存器。这通常不是想要的结果，因为它需要更多的指令（依速度和大小）并且需要一个额外的寄存器来保存地址。

默认情况下，如果目标太远，编译器会生成可能需要 trampoline 的调用。在某些架构中，这种类型的调用被称为“near 调用”。

--trampolines 选项使用户能够控制 trampoline 的生成。当设置为“on”时，此选项会使链接器为其调用目标范围外链接的每个调用生成一个 trampoline 代码段。trampoline 代码段包含一个指令序列，该序列会执行到达原始调用地址的透明长分支。每个超出被调用函数范围的调用指令都会重定向到 trampoline。

此选项的语法为：

--trampolines[=on|off]

默认设置为 on。对于 C7000，默认启用 trampoline。

例如，在一段 C 代码中，bar 函数调用 foo 函数。编译器为函数生成以下代码：

```
bar:
      ...
      CALL .B1    foo     ; call the function "foo"
      ...
```

如果 foo 函数置于 bar 内部对 foo 的调用范围之外，那么使用 --trampolines 时，链接器会将原来对 foo 的调用更改为对 foo_trampoline 的调用，如下所示：

```
bar:
      ...
      CALL .B1 $Tramp$L$PI$$myfunc ; call a trampoline for foo
      ...
```

上面的代码会生成一个名为 foo_trampoline 的 trampoline 代码段，其中包含执行到达原始调用函数 foo 的长分支的代码。例如：

```
$Tramp$L$PI$$myfunc:
    BE .B1 foo ; long branch to foo (with constant extension)
```

可在对同一被调用函数的调用之间共享 trampoline。唯一的要求是对被调用函数的所有调用都在被调用函数的 trampoline 附近进行链接。

当链接器生成一个映射文件（--map_file 选项）并且已生成一个或多个 trampoline 时，该映射文件将包含有关已生成的 trampoline 以及所到达的函数的统计信息。映射文件中还会提供每个 trampoline 的调用列表。

备注

链接器假定 D15 包含栈指针

汇编语言程序员必须知道链接器假定 D15 包含栈指针。链接器必须在由其生成的 trampoline 代码中保存和恢复栈上的值。如果用户不使用 D15 作为栈指针，则应使用禁用 trampoline 的链接器选项 --trampolines=off 。否则，trampoline 可能会破坏存储器并覆盖寄存器值。

12.4.29.1 使用 Trampoline 的优缺点

使用 trampoline 的优点是可以将所有的调用都当成 near 调用，这种调用的速度更快且更高效。您只需修改无法到达的调用。此外，几乎不需要考虑相互调用的函数的相对位置。调用必须通过 trampoline 的情况比 near 调用少见。

虽然生成 far 调用 trampoline 提供了更直接的解决方案，但 trampoline 的缺点是比直接调用函数要慢一些。trampoline 需要调用和分支。此外，虽然内联代码可以根据调用环境进行定制，但 trampoline 是以更通用的方式生成的，其效率可能略低于内联代码。

如果某个调用无法到达其被调用函数，为该调用创建 trampoline 代码段的另一种方法是实际修改该调用的源代码。在某些情况下，可在不影响代码大小的情况下完成此过程。但是，一般来说，这种方法是极其困难的，尤其是当代码的大小受变换影响时。

12.4.29.2 尽量减少所需的 Trampoline 数量（--minimize_trampolines 选项）

--minimize_trampolines 选项的目标是在尝试放置段时尽量减少所需的 far 调用 trampoline 数量，但可能的代价是存储器打包不再处于理想状态。语法为：

--minimize_trampolines=postorder

该参数选择要使用的启发法。postorder 启发法尝试将函数放置在函数调用方之前，以便在放置调用方时知道被调用方的 PC 相对偏移。首先放置被调用方，当放置调用方时被调用方的地址是已知的，因此链接器可以明确地知道是否需要 trampoline。

12.4.29.3 将 Trampoline 从加载空间传送到运行空间

在存储器中的一个位置加载代码而在另一个位置运行代码的做法有时会很有用。链接器提供了为一个段指定不同的加载分配和运行分配的功能。实际将代码从加载空间复制到运行空间的重任将留给您完成。

必须先执行复制函数，然后才能在运行空间中执行实际函数。为了方便执行这个复制函数，汇编器提供了 .label 指令，允许您定义加载时地址。然后可以使用这些加载时地址来确定所要复制代码的起始地址和大小。但是，如果代码包含的某个调用需要 trampoline 才能到达其被调用函数，则此机制将*不*起作用。这是因为 trampoline 代码是在链接时生成的，即在定义与 .label 指令关联的加载时地址之后。如果链接器在包含 trampoline 调用的输入段中检测到 .label 符号的定义，则会生成警告。

若要解决此问题，可使用 START()、END() 和 SIZE() 运算符（请参阅节 12.5.9.7）。通过这些运算符可以定义一些符号来表示链接器命令文件中的加载时起始地址和大小。这些符号可由复制代码引用，并且在分配 trampoline 段之后直到链接时才会解析符号的值。

以下示例说明了如何使用与输出段相关联的 START() 和 SIZE() 运算符来复制 trampoline 代码段以及含有需要 trampoline 的调用的代码：

```
SECTIONS
{   .foo : load = ROM, run = RAM, start(foo_start), size(foo_size)
           { x.obj(.text) }
    .text: {} > ROM
    .far : { --library=rts.lib(.text) } > FAR_MEM
}
```

x.c.obj 中的一个函数包含运行时支持调用。运行时支持库位于 far 存储器中，因此该调用超出范围。链接器会将一个 trampoline 段添加到 .foo 输出段。复制代码可以引用符号 foo_start 和 foo_size 作为整个 .foo 输出段的加载起始地址和大小的参数。因此，复制代码可以将 trampoline 段以及 .text 中的原始 x.c.obj 代码从其加载空间复制到其运行空间。

有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 12.6。

12.4.30 引入未解析的符号 (--undef_sym 选项)

--undef_sym 选项将未解析符号的链接名称引入链接器的符号表。这会强制链接器搜索库并包含定义该符号的成员。链接器必须在链接定义符号的成员*之前* 遇到 --undef_sym 选项。--undef_sym 选项的语法为：

--undef_sym= *symbol*

例如，假设名为 rts7100_le.lib 的库包含一个定义符号 symtab 的成员；没有任何目标文件链接引用 symtab。但是，假设您计划重新链接输出模块，并希望在此链接中包含定义 symtab 的库成员。使用如下所示的 --undef_sym 选项会强制链接器在 rts7100_le.lib 中搜索定义 symtab 的成员，并链接该成员。

```
cl7x --run_linker --undef_sym=symtab file1.c.obj file2.c.obj rts7100_le.lib
```

如果不使用--undef_sym，则不会包括该成员，因为在 file1.c.obj 和 file2.c.obj 中没有对该成员的显式引用。

12.4.31 创建未定义的输出段时显示一条消息 (--warn_sections)

在链接器命令文件中，您可以设置 SECTIONS 指令来描述如何将输入段组合成输出段。但是，如果链接器遇到一个或多个没有在 SECTIONS 指令中定义相应输出段的输入段，则链接器会将同名的输入段组合到该名称的输出段中。默认情况下，链接器不会显示一条消息来告知您发生了这种情况。

使用 --warn_sections 选项可以使链接器在创建新的输出段时显示消息。

有关 SECTIONS 指令的更多信息，请参阅节 12.5.5。有关链接器默认操作的更多信息，请参阅节 12.7。

12.4.32 生成 XML 链接信息文件 (--xml_link_info 选项)

链接器支持通过 --xml_link_info=*file* 选项生成 XML 链接信息文件。此选项会使链接器生成一个格式良好的 XML 文件，其中包含有关链接结果的详细信息。这个文件中包含的信息包括由链接器生成的映射文件中当前生成的所有信息。有关生成的 XML 文件内容的具体信息，请参阅附录 A。

12.4.33 零初始化（--zero_init 选项）

C 和 C++ 标准要求，未显式初始化的全局变量和静态变量必须先设置为 0，然后才能执行程序。C/C++ 编译器默认支持对未初始化的变量执行预初始化。若要将其关闭，请指定链接器选项 --zero_init=off。

--zero_init 选项的语法为：

--zero_init[={*on*|*off*}]

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 链接选项时，链接器不会生成初始化记录，加载程序必须处理数据和零初始化。

备注

不建议禁用零初始化的情况：通常不建议禁用零初始化。如果关闭零初始化，将不会自动将未初始化的全局目标和静态目标初始化为零。然后您需要通过其他方式将这些变量初始化为零。

12.5 链接器命令文件

借助链接器命令文件，您可以将链接器选项和指令放入一个文件中；在经常使用相同的选项和指令调用链接器时，这会很有用。在使用 MEMORY 和 SECTIONS 指令来自定义应用程序时，链接器命令文件也很有用。您必须在命令文件中使用这些指令；不能在命令行中使用它们。

链接器命令文件是包含以下一项或多项的 ASCII 文件：

- 输入文件名，指定了目标文件、归档库或其他命令文件。（如果一个命令文件调用另一个命令文件作为输入，则此语句必须是进行调用的命令文件中的*最后一个*语句。链接器不会从调用的命令文件返回。）
- 链接器选项在命令文件中的使用方式与在命令行中使用时相同
- MEMORY 和 SECTIONS 链接器指令。MEMORY 指令定义了目标内存配置（请参阅节 12.5.4）。SECTIONS 指令控制着如何构建和分配段（请参阅节 12.5.5）。
- 赋值语句为全局符号定义值和赋值

若要使用命令文件来调用链接器，请输入 cl7x --run_linker 命令，后跟命令文件的名称：

cl7x --run_linker *command_filename*

链接器按照遇到输入文件的顺序处理它们。如果链接器将一个文件识别为目标文件，则它会链接该文件。否则，它假定文件是命令文件，并开始从中读取和处理命令。不管使用何种系统，命令文件名都区分大小写。

连接器命令文件 展示了名为 link.cmd 的示例链接器命令文件。

连接器命令文件

```
a.c.obj                    /*  第一个输入文件名          */
b.c.obj                    /*  第二个输入文件名          */
--output_file=prog.out   /*  指定输出文件的选项 */
--map_file=prog.map      /*  指定映射文件的选项    */
```

连接器命令文件 中的示例文件仅包含文件名和选项。（您可以在命令文件中添加注释，使用 /* 和 */ 来分隔。）若要使用这个命令文件来调用链接器，请输入以下命令：

```
cl7x --run_linker link.cmd
```

使用命令文件时，可以在命令行中放入其他参数：

```
cl7x --run_linker --relocatable link.cmd x.c.obj y.c.obj
```

链接器在遇到文件名时立即处理命令文件，因此 a.c.obj 和 b.c.obj 在 x.c.obj 和 y.c.obj 之前链接至输出模块。

您可以指定多个命令文件。例如，如果您有一个包含文件名的文件 names.lst，还有另一个包含链接器指令的文件 dir.cmd，则您可以输入：

```
cl7x --run_linker names.lst dir.cmd
```

一个命令文件可以调用另一个命令文件；这种类型的嵌套限制为 16 级。如果一个命令文件调用另一个命令文件作为输入，则此语句必须是进行调用的命令文件中的*最后一个*语句。

除了作为分隔符，空格和空白行在命令文件中没有其他意义。这也适用于命令文件中的链接器指令的格式。带有链接器指令的命令文件 展示了包含链接器指令的命令文件示例。

带有链接器指令的命令文件

```
a.obj b.obj c.obj          /* 输入文件名        */
--output_file=prog.out     /* 选项                */
--map_file=prog.map
MEMORY                     /* MEMORY 指令       */
{
```

```
  FAST_MEM:  origin = 0x0100    length = 0x0100
  SLOW_MEM:  origin = 0x7000    length = 0x1000
}
SECTIONS                     /* SECTIONS 指令    */
{
  .text:  > SLOW_MEM
  .data:  > SLOW_MEM
  .bss:   > FAST_MEM
}
```

有关 MEMORY 指令的更多信息，请参阅节 12.5.4，有关 SECTIONS 指令的更多信息，请参阅节 12.5.5。

12.5.1 链接器命令文件中的保留名称

以下名称（大小写都有）保留为链接器指令的关键字。请勿在命令文件中将其用作符号名称或段名。

ADDRESS_MASK	f	LENGTH	ORG	SIZE
ALGORITHM	FILL	LOAD	ORIGIN	START
ALIGN	GROUP	LOAD_END	PAGE	TABLE
ATTR	HAMMING_MASK	LOAD_SIZE	PALIGN	TYPE
BLOCK	HIGH	LOAD_START	PARITY_MASK	UNION
COMPRESSION	INPUT_PAGE	MEMORY	RUN	UNORDERED
COPY	INPUT_RANGE	MIRRORING	RUN_END	VFILL
DSECT	l（小写 L）	NOINIT	RUN_SIZE	
ECC	LAST	NOLOAD	RUN_START	
END	LEN	o	SECTIONS	

此外，TI 工具使用的任何段名都是保留名称，不能用作其他名称的前缀，除非该段是 TI 工具所用段名的子段。例如，段名不能以 .debug 开头。

12.5.2 链接器命令文件中的常量

您可以使用两种语法方案中的任何一种来指定常量：一种方案用于指定汇编器中使用的十进制、八进制或十六进制常量（但不是二进制常量），另一种方案用于指定 C 语法中的整数常量。例如：

格式	十进制	八进制	十六进制
汇编器格式	32	40q	020h
C 格式	32	040	0x20

12.5.3 从链接器命令文件访问文件和库

许多应用程序使用自定义链接器命令文件（简称 LCF）来控制代码和数据在目标存储器中的放置。例如，您可能希望将特定文件中的特定数据对象放入目标存储器中的特定位置。这很容易实现，只需使用可用的 LCF 语法来引用所需的目标文件或库。但是，许多开发人员在尝试执行此操作时会遇到一个问题：从 LCF 内部访问先前在链接器的命令行调用中指定的目标文件或库时，链接器会生成“file not found”（文件未找到）错误。大多数情况下，发生此错误的原因是用于访问链接器命令行上文件的语法与用于访问 LCF 中相同文件的语法不一致。

让我们考虑一个简单的示例。假设一个应用程序需要在名为“DDR”的存储器区域中定义名为“app_coeffs”的常量表。此外，假设在位于目标文件 app_coeffs.c.obj 的 .data 段中定义“app_coeffs”数据对象。然后，app_coeffs.c.obj 文件包含在目标文件库 app_data.lib 中。在 LCF 中，可按如下方式控制“app_coeffs”数据对象的放置：

```
SECTIONS
{
   ...
   .coeffs: { app_data.lib<app_coeffs.c.obj>(.data) } > DDR
   ...
}
```

现在假设 app_data.lib 对象库位于一个名为“lib”的子目录中（相对于构建应用程序的位置）。为了从构建命令行访问 app_data.lib，可组合使用 –i 和 –l 选项来设置链接器可用于查找 app_data.lib 库的目录搜索路径：

```
%> cl7x <compile options/files> -z -i ./lib -l app_data.lib mylnk.cmd <link options/files>
```

-i 选项将 lib 子目录添加到目录搜索路径，-l 选项指示链接器查看目录搜索路径中的目录以查找 app_data.lib 库。但是，如果不更新 mylnk.cmd 中对 app_data.lib 的引用，则链接器将无法找到 app_data.lib 库并会生成“file not found”（文件未找到）错误。原因是当链接器在 SECTIONS 指令中遇到对 app_data.lib 的引用时，引用前没有 -l 选项。因此，链接器会尝试打开当前工作目录中的 app_data.lib。

本质上，链接器有几种不同的打开文件的方式：

- 如果指定了路径，链接器将在指定位置查找文件。对于绝对路径，链接器将尝试打开指定目录中的文件。对于相对路径，链接器将进入从当前工作目录开始的指定路径，并尝试在该位置打开文件。
- 如果没有指定路径，链接器将尝试打开当前工作目录中的文件。
- 如果 -l 选项位于文件引用之前，则链接器将尝试在目录搜索路径下的目录之一中查找并打开所引用的文件。通过 –i 选项和环境变量（如 C_DIR 和 C7X_C_DIR）设置目录搜索路径。

只要在命令行上和所有适用的 LCF 中以一致的方式引用某个文件，链接器就能够找到并打开您的目标文件和库。

回到前面的示例，可在 mylnk.cmd 中对 app_data.lib 的引用前面插入一个 -l 选项，从而确保链接器在构建应用程序时会找到并打开 app_data.lib 库：

```
SECTIONS
{
   ...
   .coeffs: { -l app_data.lib<app_coeffs.c.obj>(.data) } > DDR
   ...
}
```

从 LCF 中引用文件时使用 –l 选项的另一个好处是，如果所引用文件的位置发生变化，则可修改目录搜索路径以合并文件的新位置（例如，在命令行中使用 –i 选项），而无需修改 LCF。

12.5.4 MEMORY 指令

链接器确定输出段在存储器中的分配位置；它必须具有目标存储器的模型才能完成此任务。MEMORY 指令用于指定目标存储器的模型，使用户能够定义系统包含的存储器类型以及它们占用的地址范围。链接器在分配输出段时会保留此模型，并用它来确定目标代码可以使用的存储器位置。

C7000 系统的存储器配置因应用而异。MEMORY 指令用于指定各种配置。在使用 MEMORY 定义存储器模型后，用户可以使用 SECTIONS 指令将输出段分配到已定义的存储器中。如需了解更多信息，请参阅 节 8.4。

C7000 编译器要求所有代码和数据均位于 2GB 的虚拟地址空间内。使用链接器命令文件将代码和数据放置在此 2GB 虚拟地址区域内。C7000 编译器生成的代码使用位置无关寻址来获取函数的地址（在某些情况下）并访问存储器中静态分配的数据。使用位置无关寻址的指令具有有限的范围。如需了解更多信息，请参阅 *C7000 嵌入式应用二进制接口（EABI）参考指南*（SPRUIG4），特别是关于“计算代码地址”和“数据分配和寻址”的章节。

12.5.4.1 默认存储器型号

如果您不使用 MEMORY 指令，链接器将使用默认存储器型号，该模型可能适用于在通用模拟器上运行可执行文件。请查阅器件文档以确定可用的存储器范围。该型号会假设系统中存在完整的 48 位地址空间（2^{48} 个位置）并且可供使用。有关默认存储器型号的更多信息，请参阅节 12.7。

12.5.4.2 MEMORY 指令语法

MEMORY 指令可识别实际存在于目标系统中并可被程序使用的存储器范围。每个范围都有若干特性：

- 名称
- 起始地址
- 长度
- 可选属性集
- 可选的填充规格

使用 MEMORY 指令时，请务必确定程序在运行时可以访问的所有存储器范围。MEMORY 指令定义的存储器会被配置；而未使用 MEMORY 指令显式指定的任何存储器则不会被配置。链接器不会将程序的任何部分放入未配置的存储器中。用户可通过不在 MEMORY 指令语句中添加相应的地址范围来表示不存在的存储器空间。

在命令文件中指定 MEMORY 指令的方法是使用 MEMORY（大写）一词后跟用大括号括起来的内存范围规格列表。下面的示例中的 MEMORY 指令定义了一个系统，该系统在地址 0x0000 0000 处具有 4K 字节的快速外部存储器，在地址 0x0000 1000 处具有 2K 字节的慢速外部存储器，在地址 0x1000 0000 处具有 4K 字节的慢速外部存储器。它还演示了存储器范围表达式以及起始/结束/大小地址运算符的使用方式（请参阅节 12.5.4.3）。

MEMORY 指令

```
/*********************************************************/
/*      Sample command file with MEMORY directive        */
/*********************************************************/
file1.c.obj file2.c.obj              /*    Input files     */
--output_file=prog.out            /*    Options         */
#define BUFFER 0
MEMORY
{
   FAST_MEM (RX): origin = 0x00000000    length = 0x00001000 + BUFFER
   SLOW_MEM (RW): origin = end(FAST_MEM) length = 0x00001800 - size(FAST_MEM)
   EXT_MEM  (RX): origin = 0x10000000    length = size(FAST_MEM)
```

MEMORY 指令的一般语法如下：

MEMORY

{

name 1 [**(** *attr* **)**] **: origin =** *expr* **, length =** *expr* [**, fill =** *constant*] [**LAST(** *sym* **)**]

.

.

name n [**(** *attr* **)**] **: origin =** *expr* **, length =** *expr* [**, fill =** *constant*] [**LAST(** *sym* **)**]

}

name 指定存储器范围的名称。存储器名称可包含 1 到 64 个字符；有效字符包括 A-Z、a-z、$、. 和 _。名称对链接器而言没有特殊意义，它们只用于标识存储器范围。存储器范围名称是链接器的内部名称，不会保留在输出文件或符号表中。所有存储器范围必须具有唯一名称且不得重叠。

attr 指定与命名的范围关联的一到四个属性。属性是可选的；使用时，必须将它们括在圆括号中。属性可以将输出段的分配限制在特定存储器范围内。如果不使用任何属性，则可以将任何输出段分配到任何范围内，不受限制。任何未指定属性的存储器（包括默认模型中的所有存储器）都具有所有这四个属性。有效属性为：

- **R** 指定可以读取存储器。
- **W** 指定可以写入存储器。
- **X** 指定存储器可包含可执行代码。
- **I** 指定可以初始化存储器。

origin 指定存储器范围的起始地址；输入形式为 *origin*、*org* 或 *o*。相应的值（以字节为单位指定）是一个 64 位的整数常量表达式，可以采用十进制、八进制或十六进制格式。

length 指定存储器范围的长度；输入形式为 *length*、*len* 或 *l*。相应的值（以字节为单位指定）是一个 64 位的整数常量表达式，可以采用十进制、八进制或十六进制格式。

fill 指定存储器范围的填充字符；输入形式为 *fill* 或 *f*。填充值是可选的。相应的值是一个 的整数常量，可以采用十进制、八进制或十六进制格式。填充值用于填充未分配给段的存储器范围区域。

LAST （可选）指定一个符号，可在运行时将其用于查找存储器范围中上次分配字节的地址。请参阅节 12.5.9.8。

备注

填充存储器范围：如果为大存储器范围指定填充值，则输出文件将非常大，因为填充存储器范围（即使填充值为 0s）会导致为该范围内所有未分配的存储器块生成原始数据。

以下示例指定了一个具有 R 和 W 属性以及填充常量 0FFFFFFFFh 的存储器范围：

```
MEMORY
{
RFILE (RW) : o = 0x00000020, l = 0x00001000, f = 0xFFFFFFFF
}
```

通常将 MEMORY 指令与 SECTIONS 指令结合使用以控制输出段的放置。有关 SECTIONS 指令的更多信息，请参阅节 12.5.5。

12.5.4.3 表达式和地址运算符

存储器范围原点和长度可以使用带有以下运算符的整数常量表达式：

二元运算符：	* / % + - << >> == = < <= > >= & | && ||
一元运算符：	- ~ !

表达式根据标准 C 运算符优先级规则进行求值。

不会检查上溢和下溢，但会使用更大的整数类型来计算表达式。

可使用预处理指令 #define 常量代替整数常量。不能在存储器指令表达式中使用全局符号。

三个地址运算符从先前的存储器范围条目中引用存储器范围属性：

START(MR)	返回先前定义的存储器范围 MR 的起始地址。
SIZE(MR)	返回先前定义的存储器范围 MR 的大小。
END(MR)	返回先前定义的存储器范围 MR 的结束地址。

表达式形式的原点和长度

```
/*********************************************************/
/*     带有 MEMORY 指令的示例命令文件          */
/*********************************************************/
file1.c.obj file2.c.obj                /*     输入文件      */
--output_file=prog.out              /*     选项          */
#define ORIGIN 0x00000000
#define BUFFER 0x00000200
#define CACHE  0x0001000
MEMORY
{
   FAST_MEM (RX): origin = ORIGIN + CACHE length = 0x00001000 + BUFFER
   SLOW_MEM (RW): origin = end(FAST_MEM)  length = 0x00001800 - size(FAST_MEM)
   EXT_MEM  (RX): origin = 0x10000000     length = size(FAST_MEM) - CACHE
```

12.5.5 SECTIONS 指令

在使用 MEMORY 指令指定目标系统的存储器型号后，用户可以使用 SECTIONS 指令将输出段置于具有特定名称的存储器范围中或者具有特定属性的存储器中。例如，用户可以将 .text 和 .data 段分配到名为 FAST_MEM 的区域中，将 .bss 段分配到名为 SLOW_MEM 的区域中。

SECTIONS 指令通过如下方式控制段：

- 描述输入段如何合并为输出段
- 在可执行程序中定义输出段
- 用户可以控制将输出段置于存储器中相对于其他段和相对于整个存储器空间的位置（请注意，存储器的放置顺序并*不是*段在 SECTIONS 指令中出现的顺序。）
- 允许对输出段重命名

如需更多信息，请参阅节 8.4。子段可使用户更精确地控制段。

如果用户不指定 SECTIONS 指令，链接器会使用默认算法组合并分配段。节 12.7 详细介绍了此算法。

12.5.5.1 SECTIONS 指令语法

SECTIONS 指令在命令文件中通过 SECTIONS（大写）一词后跟用大括号括起来的输出段规范列表来指定。

SECTIONS 指令的一般语法如下：

SECTIONS
{
name : [*property* [**, property**] [**, property**] ...]
name : [*property* [**, property**] [**, property**] ...]
name : [*property* [**, property**] [**, property**] ...]
}

每个段规范均以 *name* 开头，用于定义一个输出段。（输出段是输出文件中的一个段。）段名可以涉及段、子段或归档库成员。（有关多级子段的信息，请参阅节 12.5.5.4。）段名之后是一个属性列表，用于定义段的内容和段的分配方式。各个属性之间可以用可选逗号分隔。一个段的可能属性如下：

- **Load allocation** 定义段在存储器中的加载位置。请参阅节 9.5 和节 12.5.6。

  语法： **load =** *allocation* 或

> *allocation*

- **Run allocation** 定义段在存储器中的运行位置。

  语法： **run = *allocation*** 或

  run > *allocation*

- **Input sections** 定义构成输出段的各输入段（目标文件）。请参阅节 12.5.5.3。

  语法： **{ *input_sections* }**

- **Section type** 定义特殊段类型的标志。请参阅节 12.5.8。

  语法： **type = COPY** 或

  type = DSECT 或

- **Fill value** 定义用于填充未初始化空洞的值。请参阅节 12.5.10。

  语法： **fill = *value***

以下示例显示了示例链接器命令文件中的 SECTIONS 指令。

```
/*************************************************/
/*  Sample command file with SECTIONS directive   */
/*************************************************/
file1.c.obj    file2.c.obj           /* Input files   */
--output_file=prog.out           /* Options       */
SECTIONS
{
   .text:       load = EXT_MEM, run = 0x00000800
   .const:      load = FAST_MEM
   .bss:        load = SLOW_MEM
   .vectors:    load = 0x00000000
      {
         t1.c.obj(.intvec1)
         t2.c.obj(.intvec2)
         endvec = .;
      }
   .data:alpha: align = 16
   .data:beta:  align = 16
}
```

图 12-2 显示了上面的示例中由 SECTIONS 指令定义的输出段（.vectors、.text、.const、.bss、.data:alpha 和 .data:beta），并显示了如何使用节 12.5.4.2 中提供的 MEMORY 指令在存储器中分配这些段。

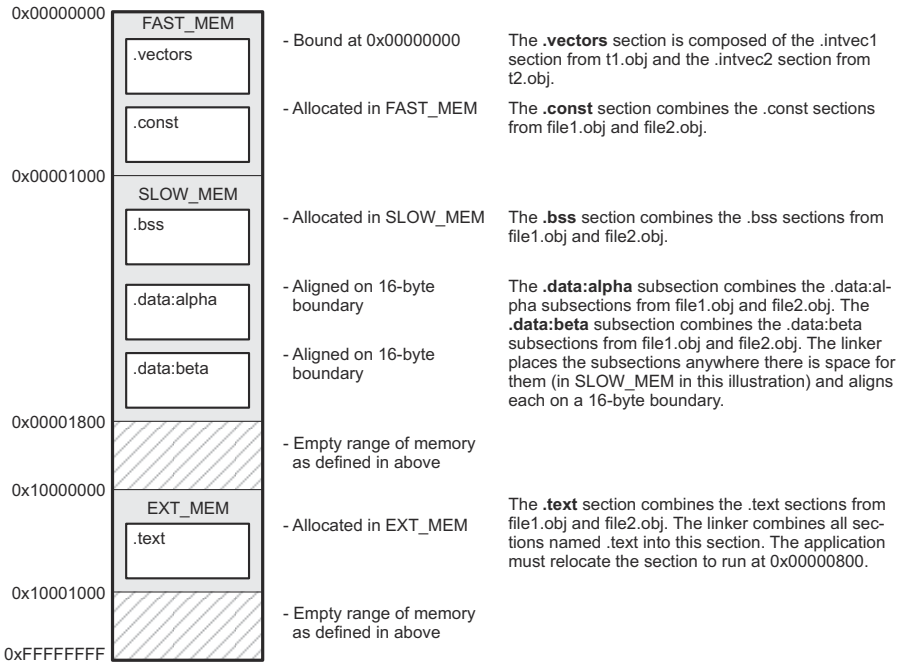


图 12-2. 由 SECTIONS 指令定义的段放置示例

12.5.5.2 段分配和放置

链接器在目标存储器中为每个输出段分配两个位置：即加载位置和运行位置。它们通常是相同的，用户也可以认为每个段只有一个地址。在目标存储器中放入输出段并向其分配地址的过程被称为放置。有关加载位置和运行位置不同的更多信息，请参阅节 12.5.6。

如果用户不指示链接器如何分配某个段，它会使用默认算法放置该段。通常，链接器会将段置于所配置存储器中的任何合适位置。用户可以在 SECTIONS 指令中进行定义，覆盖某个段的默认放置位置，并提供有关如何分配的指令。

用户可通过指定一个或多个分配参数来控制放置。每个参数均包含一个关键字、一个等号或大于号（可选）和一个值（可选择置于括号中）。如果加载位置和运行位置不同，则关键字 LOAD 之后的所有参数均适用于加载位置，关键字 RUN 之后的所有参数均适用于运行位置。分配参数包括：

绑定 为段分配一个具体地址。

```
.text: load = 0x1000
```

指定的存储器 将段分配到在具有指定名称（如 SLOW_MEM）或属性的 MEMORY 指令中定义的一个范围内。

```
.text: load > SLOW_MEM
```

对齐 使用 align 或 palign 关键字指定，该段必须始于一个地址边界。

```
.text: align = 0x100
```

分块 使用 block 关键字指定，该段必须置于符合分块系数的两个地址之间。如果段太大，则始于地址边界。

```
.text: block(0x100)
```

对于负载（通常是唯一的）分配，请使用大于号，并省略负载关键字：

```
    .text: > SLOW_MEM
    .text: {...}> SLOW_MEM
    .text: > 0x4000
```

如果使用多个参数，则可以按如下方式将它们串到一起：

```
    .text: > SLOW_MEM align 16
```

也可以选择使用圆括号来提高可读性：

```
    .text: load = (SLOW_MEM align(16))
```

用户也可以根据输入段规范，确定输入文件中可合并构成输出段的段。请参阅节 12.5.5.3。

12.5.5.2.1 绑定

用户可以在段名称后面加上地址来设置输出段的起始地址：

```
    .text: 0x00001000
```

此示例指定 .text 段必须从位置 0x1000 开始。绑定地址必须是 64 位常量。

输出段可以绑定到所配置的存储器中的任何位置（假设有足够的空间），但不能重叠。如果没有足够的空间将段绑定到指定地址，链接器会发出错误消息。

备注

绑定与对齐和已命名的存储器不兼容：如果用户使用对齐或已命名的存储器，则无法将段绑定到地址。如果用户尝试这样做，链接器会发出错误消息。

12.5.5.2.2 指定的存储器

可将一个段分配到由 MEMORY 指令定义的存储器范围中（请参阅节 12.5.4）。以下示例将指定范围并将段链接到这些范围中：

```
MEMORY
{
   SLOW_MEM (RIX)  : origin = 0x00000000,  length = 0x00001000
   FAST_MEM (RWIX) : origin = 0x03000000,  length = 0x00000300
}
SECTIONS
{
   .text  :   > SLOW_MEM
   .data  :   > FAST_MEM ALIGN(128)
   .bss   :   > FAST_MEM
}
```

在此示例中，链接器将 .text 放置到名为 SLOW_MEM 的区域中。.data 和 .bss 输出段将分配到 FAST_MEM 中。可在指定的存储器范围内对齐某个段；.data 段在 FAST_MEM 范围内的 128 字节边界上对齐。

同样，可将一个段链接到具有特定属性的存储器区域中。为此，请指定一组属性（括在圆括号中）而不是存储器名称。使用相同的 MEMORY 指令声明，可指定：

```
SECTIONS
{
   .text: > (X)   /* .text --> executable memory      */
   .data: > (RI)  /* .data --> read or init memory    */
   .bss : > (RW)  /* .bss --> read or write memory    */
}
```

在此示例中，.text 输出段可以链接到 SLOW_MEM 或 FAST_MEM 区域中，因为这两个区域都具有 X 属性。.data 段也可以进入 SLOW_MEM 或 FAST_MEM 中，因为这两个区域都具有 R 和 I 属性。但是，.bss 输出段必须进入 FAST_MEM 区域，因为只有 FAST_MEM 的声明中具有 W 属性。

尽管链接器首先使用较低的存储器地址并在可能的情况下避免碎片化，但您无法控制将段分配在指定存储器范围中的何处。在前面的示例中，假设不存在冲突性分配，.text 段将从地址 0 开始。如果某个段必须从特定地址开始，请使用绑定而不是指定的存储器。

12.5.5.2.3 使用 HIGH 位置说明符来控制放置

默认情况下，链接器在指定的存储器范围内从低地址到高地址分配输出段。或者，用户可以通过使用 SECTION 指令声明中的 HIGH 位置说明符，使链接器在存储器范围内从高地址到低地址分配某个段。用户可使用 HIGH 位置说明符将 RTS 代码与应用程序代码分开，这样一来应用程序中的微小变化就不会导致存储器映射发生较大变化。

例如，给定以下 MEMORY 指令：

```
MEMORY
{
    RAM             : origin = 0x0200, length = 0x0800
    FLASH           : origin = 0x1100, length = 0xEEE0
    VECTORS         : origin = 0xFFE0, length = 0x001E
    RESET           : origin = 0xFFFE, length = 0x0002
}
```

以及附带的 SECTIONS 指令：

```
SECTIONS
{
    .bss     : {} > RAM
    .sysmem  : {} > RAM
    .stack   : {} > RAM (HIGH)
}
```

用于放置 .stack 段的 HIGH 说明符会使链接器尝试将 .stack 分配到 RAM 存储器范围内的更高地址。.bss 和 段将分配到 RAM 中的较低地址。示例 12-1 所示为映射文件的一部分，其中展示了在 RAM 中为典型程序分配给定段的位置。

示例 12-1. 使用 HIGH 说明符进行链接器放置

```
.bss        0    00000200    00000270     UNINITIALIZED
                   00000200    0000011a     rtsxxx.lib  : defs.c.obj (.bss)
                   0000031a    00000088                 : trgdrv.c.obj (.bss)
                   000003a2    00000078                 : lowlev.c.obj (.bss)
                   0000041a    00000046                 : exit.c.obj (.bss)
                   00000460    00000008                 : memory.c.obj (.bss)
                   00000468    00000004                 : _lock.c.obj (.bss)
                   0000046c    00000002                 : fopen.c.obj (.bss)
                   0000046e    00000002     hello.c.obj (.bss)
.sysmem     0    00000470    00000120     UNINITIALIZED
                   00000470    00000004     rtsxxx .lib : memory.c.obj (.sysmem)
.stack      0    000008c0    00000140     UNINITIALIZED
                   000008c0    00000002     rtsxxx .lib : boot.c.obj (.stack)
```

如示例 12-1 所示，.bss 和 .sysmem 段被分配至 RAM 的较低地址 (0x0200 - 0x0590)，而 .stack 段被分配至地址 0x08c0（虽然较低的地址也可用）。

如果不使用 HIGH 说明符，链接器分配将产生示例 12-2 中所示的代码

如果 HIGH 说明符与特定地址绑定操作或自动段拆分操作（>> 运算符）搭配使用，则会忽略该说明符。

示例 12-2. 在没有 HIGH 说明符的情况下进行链接器放置

```
.bss        0    00000200    00000270     UNINITIALIZED
                   00000200    0000011a     rtsxxx.lib  : defs.c.obj (.bss)
                   0000031a    00000088                 : trgdrv.c.obj (.bss)
                   000003a2    00000078                 : lowlev.c.obj (.bss)
                   0000041a    00000046                 : exit.c.obj (.bss)
                   00000460    00000008                 : memory.c.obj (.bss)
                   00000468    00000004                 : _lock.c.obj (.bss)
                   0000046c    00000002                 : fopen.c.obj (.bss)
                   0000046e    00000002     hello.c.obj (.bss)
.stack      0    00000470    00000140     UNINITIALIZED
                   00000470    00000002     rtsxxx.lib  : boot.c.obj (.stack)
.sysmem     0    000005b0    00000120     UNINITIALIZED
                   000005b0    00000004     rtsxxx.lib  : memory.c.obj (.sysmem)
```

12.5.5.2.4 对齐和分块

使用 align 关键字可以告诉链接器将输出段放置在位于 n 字节边界上的地址处，其中 n 是 2 的幂。例如，以下代码分配 .text，使其落在 32 字节边界上：

```
     .text: load = align(32)
```

分块是一种较弱的对齐形式，它将一个段分配到大小为 n 的块*内*的任何位置。指定的块大小必须是 2 的幂。例如，以下代码分配 .bss，使整个段包含在单个 128 字节的中，或从该边界开始：

对齐或分块可单独使用，也可与存储器区域结合使用，但对齐和分块不能同时使用。

12.5.5.2.5 对齐和填充

与 align 关键字一样，使用 palign 关键字可以告诉链接器将输出段放置在位于 n 字节边界上的地址处，其中 n 是 2 的幂。此外，palign 确保段的大小是其放置对齐限制值的倍数，并会根据需要将段大小填充到这样的边界。

例如，以下代码行在 PMEM 区域内的 2 字节边界上分配 .text。.text 段的大小需保证是 2 字节的倍数。以下两条语句是等效的：

```
   .text: palign(2) {} > PMEM
   .text: palign = 2 {} > PMEM
```

如果链接器向初始化的输出段添加填充，则填充空间也会被初始化。默认情况下，填充空间的填充值为 0（零）。但是，如果为输出段指定一个填充值，则该段的任何填充也将使用该填充值进行填充。例如，考虑以下段规范：

```
   .mytext: palign(8), fill = 0xffffffff {} > PMEM
```

在此示例中，在应用 palign 运算符之前，.mytext 段的长度为 。.mytext 的内容如下：

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
```

应用 palign 运算符后，.mytext 的长度为 8 个字节，且其内容如下：

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
0006 0xffff
```

.mytext 的大小已提升到 8 字节的倍数，并且由链接器创建的填充已填充为 0xff。

链接器命令文件中指定的填充值被解释为 16 位常量。如果指定以下代码：

```
   .mytext: palign(8), fill = 0xff {} > PMEM
```

链接器假定的填充值为 0x00ff，然后 .mytext 将具有以下内容：

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
0006 0x00ff
```

如果 palign 运算符应用于未初始化的段，则该段的大小会根据需要提升到相应的边界，但不会初始化创建的任何填充。

palign 运算符也可以采用 *power2* 参数。此参数告诉链接器添加填充以将段的大小增加到下一个“2 的幂”边界。此外，该段也在这个“2 的幂”边界上对齐。例如，考虑以下段规范：

```
    .mytext: palign(power2) {} > PMEM
```

假设 .mytext 段的大小为 120 字节，PMEM 从地址 0x10020 开始。在应用 palign(power2) 运算符后，.mytext 输出段将具有以下属性：

```
  name     addr         size     align
-------    ----------   -----    -----
.mytext    0x00010080   0x80     128
```

12.5.5.3 指定输入段

输入段规范确定输入文件中组合在一起构成输出段的各段。一般而言，链接器会按照指定的顺序连接各输入段以将这些段组合在一起。不过，如果为输入段指定了对齐或阻塞，输出段内的所有输入段都会按如下方式排序：

- 所有对齐的段从大到小排列
- 所有阻塞的段从大到小排列
- 所有其他段从大到小排列

输出段的大小为所含各输入段的大小之和。

示例 12-3 显示了段规范的最常见类型；请注意，这里没有列出任何输入段。

示例 12-3. 指定段内容的最常用方法

```
SECTIONS
{
   .text:
   .data:
   .bss:
}
```

在示例 12-3 中，链接器从输入文件中获取所有 .text 段，并将它们组合到 .text 输出段中。链接器按照其在输入文件中遇到 .text 输入段的顺序连接这些输入段。链接器对 .data 和 .bss 段执行类似的操作。用户可以将这种类型的规范用于任何输出段。

用户可以显式指定构成输出段的输入段。每个输入段由其文件名和段名标识。如果文件名带有连字符 (或包含特殊字符) ，请将其放入引号中：

```
SECTIONS
{
  .text :              /* Build .text output section        */
  {
    f1.c.obj(.text)      /* Link .text section from f1.c.obj    */
    f2.c.obj(sec1)       /* Link sec1 section from f2.c.obj     */
    "f3-new.c.obj"       /* Link ALL sections from f3-new.c.obj */
    f4.c.obj"(.text,sec2) /* Link .text and sec2 from f4.c.obj   */
   }
}
```

输入段之间不必使用相同名称，也不必与它们所属的输出段具有相同名称。如果列出某个文件时未提供段，则其*所有*的段都将包含在输出段中。如果任何其他输入段与输出段同名，但未通过 SECTIONS 指令显式指定，则其将在输出段的末尾自动链接进来。例如，如果链接器在前面的示例中发现更多 .text 段，并且这些 .text 段*没有*在 SECTIONS 指令中的任何位置指定，则链接器将在 f4.c.obj(sec2) 之后连接这些额外的段。

示例 12-3 中的规范实际上是以下内容的简略表达：

```
SECTIONS
{
  .text: { *(.text) }
  .data: { *(.data) }
  .bss:  { *(.bss)  }
}
```

规范 *(.text) 表示*所有输入文件中未分配的 .text 段*。这种格式在以下情况下很有用：

- 用户希望输出段包含具有指定名称的所有输入段，但输出段名称与输入段的名称不同。
- 用户希望链接器在处理大括号内的其他输入段或命令之前分配输入段。

以下示例展示了上述两种用途：

```
SECTIONS
{
   .text   :  {
                    abc.c.obj(xqt)
                    *(.text)
              }
   .data   :  {
                    *(.data)
                    fil.c.obj(table)
              }
}
```

在此示例中，.text 输出段包含来自文件 abc.c.obj 的指定段 xqt，后跟所有 .text 输入段。.data 段包含所有 .data 输入段，后跟来自文件 fil.c.obj 的指定段表。此方法包括所有未分配的段。 例如，如果当链接器遇到 *(.text) 时其中一个 .text 输入段已包含在另一个输出段中，则链接器无法将第一个 .text 输入段添加到第二个输出段中。

每个输入段充当一个前缀，以收集名称更长的段。例如，模式 *(.data) 与 .dataspecial 匹配。因此，前缀启用子段（在下一节中将介绍这些子段）。

12.5.5.4 使用多级子段

基础段名与一个或多个子段名（由冒号分隔）可识别为子段。例如，名为 A:B 和 A:B:C 的子段为基础段 A 的子段。在链接器命令文件的特定位置会指定一个基础名称，例如 A，选择段 A 以及 A 的任何子段，例如 A:B 或 A:C:D。

名称（例如 A:B）可指定该名称的（子）段以及以该名称起始的（多级）子段，例如 A:B:C、A:B:OTHER 等。A:B 的所有子段同样也是 A 的子段。A 和 A:B 都是 A:B:C 的超段。在一个子段的一组超段中，最近的超段是名称最长的超段。因此，在 {A, A:B} 之间，A:B:C:D 最近的超段是 A:B。多级子段具有如下限制：

1. 在一个文件（或库单元）中指定**输入**段时，段名会选择同名输入段以及该名称的任何子段。
2. 未显式分配的输入段会在同名的现有**输出**段或这样的输出段的现有最近超段中分配。此规则的例外是，在部分链接（由 --relocatable 链接器选项指定）期间，子段只会分配到同名的现有输出段中。
3. 如果未定义 2) 中介绍的输出段，输入段将置于与输入段的基本名称同名的**新创建的输出**段中。

请考虑具有以下名称的链接输入段：

europe:north:norway	europe:central:france	europe:south:spain
europe:north:sweden	europe:central:germany	europe:south:italy
europe:north:finland	europe:central:denmark	europe:south:malta
europe:north:iceland		

此 SECTIONS 规范分配输入段的方式如注释中所示：

```
SECTIONS {
  nordic:  {*(europe:north)
            *(europe:central:denmark)} /* the nordic countries */
  central: {*(europe:central)}         /* france, germany      */
  therest: {*(europe)}                 /* spain, italy, malta  */
}
```

此 SECTIONS 规范分配输入段的 方式如注释中所示：

```
SECTIONS {
  islands: {*(europe:south:malta)
            *(europe:north:iceland)}  /* malta, iceland   */
  europe:north:finland : {}           /* finland          */
  europe:north         : {}           /* norway, sweden   */
  europe:central       : {}           /* germany, denmark */
  europe:central:france: {}           /* france           */
  /* (italy, spain) go into a linker-generated output section "europe" */
}
```

备注

多级子段的向上兼容性：使用现有单级子段功能的现有链接器命令，以及不包含具有多个冒号字符的段名的命令，其运行方式与原来一样。但如果链接器命令文件中的段名，或提供给链接器的输入段中的段名包含多个冒号字符，运行方式可能会有所变化。您应仔细考虑多级规则产生的影响，看它是否会影响特定系统链接。

12.5.5.5 指定库或存档成员作为输出段的输入

您可以指定对象库或存档的一个或多个成员作为输出段的输入。可考虑以下 SECTIONS 指令：

示例 12-4. 将成员存档至输出段

```
SECTIONS
{
boot>BOOT1
{
-l rtsXX.lib<boot.c.obj> (.text)
-l rtsXX.lib<exit.c.obj strcpy.c.obj> (.text)
}
.rts>BOOT2
{
-l rtsXX.lib (.text)
}
.text>RAM
{
* (.text)
}
}
```

在示例 12-4 中，boot.c.obj、exit.c.obj 和 strcpy.c.obj 的 .text 段是从运行时支持库中提取的，并放置在 .boot 输出段中。引用的运行时支持库对象的其余部分将分配给 .rts 输出段。最后，所有其他 .text 段的其余部分将放置在 .text 段中。

为了指定一个存档成员或成员列表，需要在库名称后用尖括号 < 和 > 将成员名称括起来。指定的存档文件中以逗号或空格分隔的任何目标文件在尖括号内都是合法的。

在 < > 中列出特定存档成员时，在示例 12-4 中的每个库之前列出的 --library 选项（通常意味着对选项后面的指定文件进行库路径搜索）是可选项。使用 < > 意味着要引用某个库。

若要在一个地方收集库中的一组输入段，请在 SECTIONS 指令中使用 --library 选项。例如，以下代码会将 rts7100_le.lib 中的所有 .text 段收集到 .rtstest 段中：

```
SECTIONS
{
   .rtstest { -l rts7100_le.lib(.text) } > RAM
}
```

备注

SECTIONS 指令对 --priority 的影响：在 SECTIONS 指令中指定某个库会导致这个库被输入到库列表中，该库列表由链接器进行搜索以解析引用。如果使用 --priority 选项，则会首先搜索在命令文件中指定的第一个库。

12.5.5.6 使用多个存储器范围进行分配

链接器可使用户指定一个显式的存储器范围列表，可以向其中分配输出段。考虑以下示例：

```
MEMORY
{
   P_MEM1 :  origin = 0x02000,  length = 0x01000
   P_MEM2 :  origin = 0x04000,  length = 0x01000
   P_MEM3 :  origin = 0x06000,  length = 0x01000
   P_MEM4 :  origin = 0x08000,  length = 0x01000
}
SECTIONS
{
   .text   : { } > P_MEM1 | P_MEM2 | P_MEM4
}
```

运算符 | 用于指定多个存储器范围。输出段 .text 作为一个整体分配给与之适应的第一个存储器范围。存储器范围按指定顺序访问。在此示例中，链接器首先尝试在 P_MEM1 中分配段。如果该尝试失败，链接器会尝试将该段置于 P_MEM2 中，依此类推。如果未在任何指定的存储器范围中成功分配输出段，则链接器会发出错误消息。

借助这种类型的 SECTIONS 指令规范，链接器可以无缝处理超出最初分配的存储器范围可用空间的输出段。用户可以让链接器将段移动到其他某个区域，而不是修改链接器命令文件。

12.5.5.7 在非连续存储器范围之间自动拆分输出段

链接器可以在多个存储器范围之间拆分输出段以提高分配效率。使用 >> 运算符来指示如有必要，可将输出段拆分成指定的存储器范围：

```
MEMORY
{
   P_MEM1 :  origin = 0x2000,  length = 0x1000
   P_MEM2 :  origin = 0x4000,  length = 0x1000
   P_MEM3 :  origin = 0x6000,  length = 0x1000
   P_MEM4 :  origin = 0x8000,  length = 0x1000
}
SECTIONS
{
  .text: { *(.text) } >> P_MEM1 | P_MEM2 | P_MEM3 | P_MEM4
}
```

在此示例中，>> 运算符指示可在任何列出的存储器区域之间拆分 .text 输出段。如果 .text 段增长到超出 P_MEM1 中的可用存储器，则会在输入段边界上拆分 .text 段，而输出段的其余部分将分配给 P_MEM2 | P_MEM3 | P_MEM4。

| 运算符用于指定多个存储器范围的列表。

还可以使用 >> 运算符来指示可在单个存储器范围内拆分输出段。当多个输出段必须分配到同一存储器范围，但一个输出段的限制会导致存储器范围被分区时，此功能会很有用。考虑以下示例：

```
MEMORY
{
   RAM :  origin = 0x1000,  length = 0x8000
}
SECTIONS
{
  .special: { f1.c.obj(.text) } load = 0x4000
  .text: { *(.text) } >> RAM
}
```

.special 输出段被分配在靠近 RAM 存储器范围中间的地方。这会在 RAM 中留下两个未使用的区域：从 0x1000 到 0x4000，以及从 f1.c.obj(.text) 的末尾到 0x8000。.text 段的规格允许链接器围绕 .special 段拆分 .text 段，并使用 RAM 中 .special 两侧的可用空间。

>> 运算符还可用于在匹配指定属性组合的所有存储器范围之间拆分输出段。例如：

```
MEMORY
{
  P_MEM1 (RWX) : origin = 0x1000,  length = 0x2000
  P_MEM2 (RWI) : origin = 0x4000,  length = 0x1000
}
SECTIONS
{
  .text: { *(.text) } >> (RW)
}
```

链接器会尝试将全部或部分输出段分配至属性与 SECTIONS 指令中指定的属性匹配的任何存储器范围。

此 SECTIONS 指令与以下指令具有相同的效果：

```
SECTIONS
{
.text: { *(.text) } >> P_MEM1 | P_MEM2}
}
```

某些段不应拆分：

- 编译器创建的某些段，包括
  - .cinit 段，其中包含 C/C++ 程序的自动初始化表

  - .pinit 段，其中包含 C++ 程序的全局构造函数列表
  - .bss 段，其中定义了全局变量
- 具有输入段规格（其中包含要计算的表达式）的输出段。该表达式可能会定义一个符号。该符号在程序中用于在运行时管理输出段。
- 应用了 START()、END() 或 SIZE() 运算符的输出段。这些运算符提供有关段的加载或运行地址以及大小的信息。拆分该段可能会损害运算的完整性。
- UNION 的运行分配。（允许拆分 UNION 的加载分配。）

如果对这些段中的任何一个段使用 >> 运算符，则链接器将发出警告并忽略该运算符。

12.5.6 在不同的加载和运行地址放置段

有时您可能希望将代码加载到存储器的一个区域，而在另一个区域运行。例如，慢速外部存储器中可能有对性能至关重要的代码。代码必须加载至慢速外部存储器，但在快速外部存储器中能够以更快的速度运行。

链接器提供了实现此目标的简单方式。您可以使用 SECTIONS 指令使链接器分配一个段两次：第一次设置其加载地址，第二次设置其运行地址。例如：

```
    .fir: load = SLOW_MEM, run = FAST_MEM
```

加载地址使用 *load* 关键字，运行地址使用 *run* 关键字。

请参阅节 9.5，了解运行时重定址的概述。

应用必须将段从其加载地址复制到其运行地址；如果您另外指定了运行地址，这 *不会* 自动发生。（TABLE 操作符指示链接器生成复制表；请参阅节 12.8.4.1。）

12.5.6.1 指定加载和运行地址

加载地址决定了加载程序将段的原始数据放在何处。对该段的任何引用（例如其中的标号）都会引用其运行地址。

如果您只为某个段提供了一个分配（加载或运行），该段只会分配一次，并会在相同的地址加载和运行。如果您提供了这两个分配，该段会被视作两个大小相同的段来进行分配。这意味着，这两个分配都会占用存储器映射中的空间，并且彼此不能叠加，也不能与其他段叠加。（UNION 提供了一种叠加段的方式；请参阅节 12.5.7.2。）

如果加载地址或运行地址包含额外的参数，例如对齐或阻塞，请在相应的关键字后列出。在遇到关键字 *load* 后，与分配相关的一切内容都会影响加载地址，直到遇到关键字 *run*，在此之后，一切内容都会转而影响运行地址。加载和运行分配完全独立，因此其中任何一个的资格（例如对齐）都不会对另一个产生影响。您还可以先指定运行地址，再指定加载地址。可以使用括号来提高可读性。

下面的示例指定了加载和运行地址。

本例中，对齐只会应用到加载：

```
    .data: load = SLOW_MEM, align = 32, run = FAST_MEM
```

下例中使用了括号，但作用与上例完全相同：

```
    .data: load = (SLOW_MEM align 32), run = FAST_MEM
```

下例会针对运行分配将 FAST_MEM 对齐到 32 位并将所有加载分配对齐到 16 位：

```
    .data: run  = FAST_MEM, align 32, load = align 16
```

有关运行时重定位的更多信息，请参阅节 9.5。

未初始化的段（例如 .bss）不会被加载，因此它们唯一重要的地址是运行地址。链接器只分配一次未初始化的段：如果您同时指定运行地址和加载地址，链接器会向您发出警告并忽略加载地址。另一方面，如果只指定一个地址，无论您称其为加载地址还是运行地址，链接器都会将其视为运行地址。

以下示例指定了未初始化的段的加载和运行地址：

```
    .bss: load = 0x1000, run = FAST_MEM
```

链接器会发出警告，忽略加载，并在 FAST_MEM 中分配空间。以下所有示例都具有相同的效果。.bss 段将分配到 FAST_MEM 中。

```
    .dbss: load = FAST_MEM
    .bss: run = FAST_MEM
    .bss: > FAST_MEM
```

12.5.7 使用 GROUP 和 UNION 语句

两个 SECTIONS 语句可用于组织或保留存储器：GROUP 和 UNION。将段分组可使链接器在存储器中连续分配这些段。将段合并可使链接器为它们分配相同的运行地址。

12.5.7.1 将输出段一同进行分组

除非使用 UNORDERED 运算符，否则 SECTIONS 指令的 GROUP 选项会强制按列出的顺序连续分配多个输出段。例如，假设一个名为 term_rec 的段包含 .data 段中某个表的终止记录。可强制链接器将 .data 和 term_rec 分配在一起：

将段分配在一起

```
SECTIONS
{
   .text           /* 正常的输出段              */
   .bss            /* 正常的输出段              */
   GROUP 0x00001000 : /* 指定一组段     */
   {
      .data        /* 组中的第一个段         */
      term_rec     /* 在 .data 之后立即分配 */
   }
}
```

可使用绑定、对齐或指定的存储器通过与单个输出段相同的方式分配 GROUP。在前面的示例中，GROUP 绑定到地址 0x1000。这意味着 .data 分配到存储器中的 0x1000 位置，而 term_rec 紧跟其后。

备注

不能为 GROUP 内的段指定地址：使用 GROUP 选项时，只能为组指定绑定、对齐或分配到指定的存储器中。不能对组内的段使用绑定、指定的存储器或对齐。

12.5.7.2 利用 UNION 语句叠加段

对于某些应用，用户可能希望在运行时期间分配占用同一地址的多个段。例如，用户可能希望在不同执行阶段使用快速外部存储器中的若干例程。用户也可能希望若干不同时处于活动状态的数据对象共享一个存储器块。SECTIONS 指令中的 UNION 语句可在同一运行时地址分配多个段。

在 UNION 语句 中（ ），file1.c.obj 和 file2.c.obj 的 .bss 段在 FAST_MEM 中的同一地址进行分配。在存储器映射中，union 占用的空间为其最大组件所占的空间。union 的组件仍是独立段；它们只是作为一个单元一同被分配。

UNION 语句

```
   SECTIONS
   {
      .text: load = SLOW_MEM
      UNION: run  = FAST_MEM
```

```
    {
       .bss:part1: { file1.c.obj(.bss) }
       .bss:part2: { file2.c.obj(.bss) }
    }
       .bss:part3: run = FAST_MEM { globals.c.obj(.bss) }
}
```

分配一个作为 union 一部分的段仅影响其*运行地址*。加载时的段*永远不会*叠加。如果经过初始化的段是 union 成员（经过初始化的段具有原始数据，例如 .text），则*必须*分别指定其负载分配。请参阅 UNION 段的单独加载地址。（在将经过初始化的段与未经初始化的段相结合时，此规则有一个例外；请参阅节 12.5.7.3。）

UNION 段的单独加载地址

```
UNION run = FAST_MEM
{
   .text:part1: load = SLOW_MEM, { file1.c.obj(.text) }
   .text:part2: load = SLOW_MEM, { file2.c.obj(.text) }
}
```

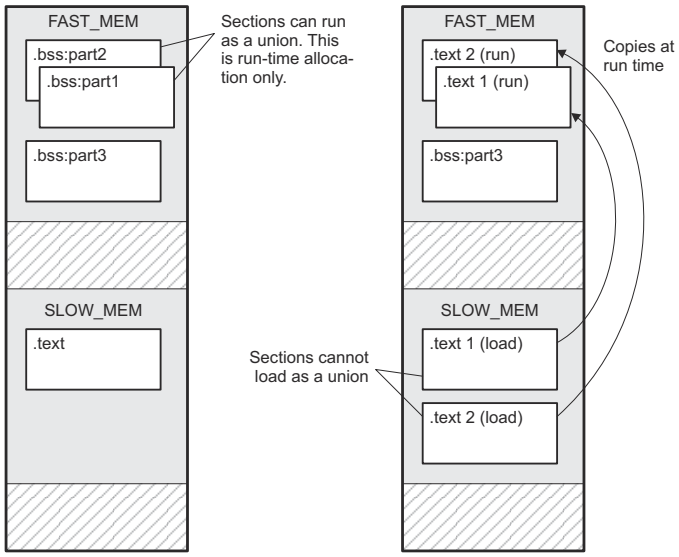


图 12-3. 存储器分配如 UNION 语句和 UNION 段的单独加载地址所示

.text 段包含原始数据，因此无法作为 union *加载*，但可以作为 union *运行*。因此，每个 .text 段都需要有自己的加载地址。如果用户未能为 UNION 中的初始化段提供加载分配，链接器会发出警告，并将所配置存储器中的任意位置作为加载空间分配。

未初始化的段未加载，不需要加载地址。

UNION 语句仅适用于运行地址的分配，因此无需为 union 本身指定加载地址。如用于分配，union 可被视为未初始化的段：指定的任何分配均被视为运行地址，如果同时指定了运行和加载地址，链接器会发出警告，并忽略加载地址。

12.5.7.3 将存储器用于多种用途

减少应用对存储器需求的一种方式是将存储器的一部分空间用于多种用途。用户可以首先使用存储器中的一部分空间进行系统初始化和启动。该阶段完成后，相同的存储器空间可改变用途，作为一组未初始化的数据变量或一个堆。若要实施此方案，请使用以下 UNION 语句的变化形式，它允许初始化一个段，而使其他段保留未初始化状态。

通常，一个 union 中经过初始化的段（具有原始数据，例如 .text）必须拥有其单独指定的负载分配。但一个 union 中有且只有一个初始化的段可分配至该 union 的运行地址。在 UNION 语句中列出，但没有负载分配，该段就会使用该 union 的运行地址作为它自己的负载地址。例如：

```
UNION run = FAST_MEM
{ .cinit .bss }
```

在本例中，.cinit 段是初始化的段。它会在 union 的运行地址被加载到 FAST_MEM 中。相反，.bss 是一个未初始化的段。它的运行地址也是该 union 的运行地址。

12.5.7.4 嵌套 UNION 和 GROUP

链接器允许使用 SECTIONS 指令对 GROUP 和 UNION 语句进行任意嵌套。通过嵌套 GROUP 和 UNION 语句，可以表示段的分层重叠和分组。嵌套 GROUP 和 UNION 语句显示了如何将两个重叠层组合在一起。

嵌套 GROUP 和 UNION 语句

```
SECTIONS
{
   GROUP 0x1000 : run = FAST_MEM
   {
      UNION:
      {
         mysect1: load = SLOW_MEM
         mysect2: load = SLOW_MEM
      }
      UNION:
      {
         mysect3: load = SLOW_MEM
         mysect4: load = SLOW_MEM
      }
   }
}
```

对于此示例，链接器执行以下分配：

- 四个段（mysect1、mysect2、mysect3、mysect4）被分配唯一且不重叠的加载地址。该分配取决于为每个段提供的特定加载分配。
- mysect1 和 mysect2 段会在 FAST_MEM 中被分配相同的运行地址。
- mysect3 和 mysect4 段会在 FAST_MEM 中被分配相同的运行地址。
- mysect1/mysect2 和 mysect3/mysect4 的运行地址按照 GROUP 语句的指示进行连续分配（受限于对齐和分块限制）。

为了引用组和联合体，链接器诊断消息使用以下表示法：

GROUP_*n* UNION_*n*

其中 *n* 是一个序号（从 1 开始），表示链接器控制文件中该组或联合体的词法顺序，不考虑嵌套。组和联合体都自带计数器。

12.5.7.5 检查分配器的一致性

链接器会检查为联合体、组和段指定的加载和运行分配的一致性。使用的规则如下：

- 仅允许对顶级段、组或联合体（即未嵌套在任何其他组或联合之下的段、组或联合体）运行分配。链接器使用顶级结构的运行地址来计算组和联合体内组件的运行地址。
- 对于联合体，链接器不接受加载分配。
- 对于未初始化的段，链接器不接受加载分配。
- 在大多数情况下，必须为已初始化的段提供加载分配。但是，如果一个已初始化的段位于已定义加载分配器的组内，对于该段，链接器不接受加载分配。
- 作为一种捷径，您可以为整个组指定加载分配，从而确定嵌套在该组中的每个已初始化段或子段的加载分配。但是，仅当以下所有条件都成立时，才接受整个组的加载分配：

  - 该组已初始化（即，它至少有一个已初始化的成员）。
  - 该组未嵌套在具有加载分配器的另一个组中。
  - 该组不包含具有已初始化段的联合体。
- 如果该组包含一个具有已初始化段的联合体，则需要为嵌套在该组中的每个已初始化段指定加载分配。考虑以下示例：

```
SECTIONS
{
  GROUP: load = SLOW_MEM, run = SLOW_MEM
  {
   .text1:
   UNION:
   {
    .text2:
    .text3:
   }
  }
}
```

为组提供的加载分配器不唯一指定联合体中元素（.text2 和 .text3）的加载分配。在这种情况下，链接器会发出诊断消息以请求显式指定这些加载分配。

12.5.7.6 为 UNION 和 GROUP 命名

可通过在位于声明之后的圆括号中输入名称来为 UNION 或 GROUP 命名。例如：

```
GROUP(BSS_SYSMEM_STACK_GROUP)
{
   .bss    :{}
   .sysmem :{}
   .stack  :{}
} load=D_MEM,  run=D_MEM
```

您定义的名称将用于在诊断中轻松识别问题 LCF 区域。例如：

```
warning: LOAD placement ignored for "BSS_SYSMEM_STACK_GROUP": object is uninitialized
   UNION(TEXT_CINIT_UNION)
   {
      .const :{}load=D_MEM, table(table1)
      .pinit :{}load=D_MEM, table(table1)
   }run=P_MEM
warning:table(table1) operator ignored: table(table1) has already been applied to a section
in the "UNION(TEXT_CINIT_UNION)" in which ".pinit" is a descendant
```

12.5.8 特殊段类型（DSECT、COPY、NOLOAD 和 NOINIT）

您可以为输出段分配以下特殊类型：DSECT、COPY、NOLOAD 和 NOINIT。这些类型会影响链接和加载程序时处理程序的方式。您可以通过在段定义后加上类型来为段分配类型。例如：

```
SECTIONS
   {
   sec1: load = 0x00002000, type =  DSECT  {f1.c.obj}
   sec2: load = 0x00004000, type =  COPY   {f2.c.obj}
   sec3: load = 0x00006000, type =  NOLOAD {f3.c.obj}
   sec4: load = 0x00008000, type =  NOINIT {f4.c.obj}
   }
```

- DSECT 类型会创建具有以下特性的虚拟段：
  - 它不会包含在输出段存储器分配中。它不会占用存储器，也不会包含在存储器映射列表中。
  - 它可以与其他输出段、其他 DSECT 和未配置的存储器重叠。
  - 虚拟段中定义的全局符号会正常进行重定位。它们会出现输出模块的符号表中，并且值与实际加载 DSECT 时原本该有的值相同。其他输入段可以引用这些符号。
  - 如果在 DSECT 中找到任何未定义的外部符号，则会搜索指定的归档库。
  - 该段的内容、重定位信息以及行号信息不会放到输出模块中。

在上述示例中，f1.c.obj 中的段都不会被分配，但所有符号都会进行重定位，就像这些段在地址 0x2000 处进行链接。其他段可以引用 sec1 中的任何全局符号。

- COPY 段与 DSECT 段相似，不同的是它的内容及关联信息会被写入输出模块。包含 C7000 C/C++ 编译器初始化表的 .cinit 段在运行时初始化模型下具有此属性。
- NOLOAD 段在一点上与普通输出段不同，那就是：该段的内容、重定位信息和行号信息不会放到输出模块中。链接器会为该段分配空间，并且它会出现在存储器映射列表中。
- NOINIT 段不会由链接器在 C 自动初始化过程中进行初始化。您需要自行根据需要对此段进行初始化。

12.5.9 在链接时分配符号

链接器赋值语句允许定义外部（全局）符号并在链接时为其赋值。此功能可用于初始化变量，或针对取决于赋值的值来初始化其指针。有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 12.6。

12.5.9.1 赋值语句的语法

链接器中赋值语句的语法与 C 语言中赋值语句的语法类似：

symbol	=	*expression;*	为符号分配表达式的值
symbol	+ =	*expression;*	将表达式的值加到符号
symbol	- =	*expression;*	从符号减去表达式的值
symbol	* =	*expression;*	将符号和表达式相乘
symbol	/ =	*expression;*	符号除以表达式

符号应是在外部定义的。如果不是，链接器会定义一个新符号，并将其加入符号表。表达式必须遵循节 12.5.9.3 中定义的规则。赋值语句*必须*以分号结束。

链接器在分配所有输出段*后*处理赋值语句。因此，如果表达式包含一个符号，该符号使用的地址会反映可执行输出文件中该符号的地址。

例如，假设程序从由两个外部符号标识的两个表（表 1 和表 2）中的一个读取数据。程序使用 cur_tab 符号作为当前表的地址。cur_tab 符号必须指向表 1 或表 2。用户可以使用链接器赋值语句在链接时分配 cur_tab：

```
prog.c.obj           /* 输入文件                              */
cur_tab = Table1;  /* 将 cur_tab 分配至其中一个表 */
```

12.5.9.2 向符号分配 SPC

一个由点 (.) 表示的特殊符号，表示分配期间段程序计数器 (SPC) 的当前值。SPC 会跟踪某个段中的当前位置。. 符号只能在 SECTIONS 指令的赋值语句中使用，因为 . 仅在分配期间有意义，并且 SECTIONS 会控制分配过程。（请参阅节 12.5.5。）

. 符号指段的当前运行地址，而不是当前加载地址。

一种特殊类型的赋值语句为 .（点）符号赋值。这会调整输出段内的 SPC 并在两个输入段之间创建一个空洞。分配给 . 以创建空洞的任何值都是相对于段的开头，而不是相对于 . 符号实际表示的地址。空洞和 . 的赋值语句详见节 12.5.10。

12.5.9.3 赋值表达式

链接器表达式适用以下规则：

- 表达式可以包含全局符号、常量和表 12-10 中列出的 C 语言运算符。
- 所有数字都被视为长（64 位）整数。
- 链接器使用与汇编器相同的方式来识别常量。也就是说，除非有后缀（H 或 h 表示十六进制，Q 或 q 表示八进制），否则数字将识别为十进制。还会识别 C 语言前缀（0 表示八进制，0x 表示十六进制）。十六进制常量必须以数字开头。不允许使用二进制常量。
- 表达式中的符号只有符号*地址*的值。不会执行类型检查。

- 链接器表达式可以是绝对的或可重定位的表达式。如果表达式包含*任何*可重定位符号（以及 0 个或多个常量或绝对符号），则该表达式是可重定位的表达式。否则，就是绝对表达式。如果一个符号被赋予了可重定位表达式的值，则该符号是可重定位的符号；如果被赋予了绝对表达式的值，则是绝对符号。

链接器支持表 12-10 中按优先顺序列出的 C 语言运算符。同一组中的运算符具有相同的优先级。除了表 12-10 中列出的运算符之外，链接器还有一个 align 运算符，该运算符允许符号在输出段内的 n 字节边界上对齐（n 是 2 的幂）。例如，以下表达式使当前段内的 SPC 在下一个 16 字节边界上对齐。align 运算符是当前 SPC 的函数，因此它只能在与 . 相同的上下文中使用，也就是说，在 SECTIONS 指令中使用。

```
.= align(16);
```

表 12-10. 表达式中使用的运算符组（优先级）

组 1（最高优先级）		组 6	
!	逻辑非	且	按位与
~	按位非		
-	否定		
组 2		**组 7**	
*	乘法	|	按位或
/	除法		
%	模数		
组 3		**组 8**	
+	加法	&&	逻辑与
-	减法		
组 4		**组 9**	
>>	算术右移	||	逻辑或
<<	算术左移		
组 5		**组 10（最低优先级）**	
==	等于	=	赋值
! =	不等于	+ =	A + = B 等效于 A = A + B
>	大于	- =	A - = B 等效于 A = A - B
<	小于	* =	A * = B 等效于 A = A * B
< =	小于等于	/ =	A / = B 等效于 A = A / B
> =	大于等于		

12.5.9.4 由链接器自动定义的符号

.text 指定 .text 输出段的第一个地址。（标志着可执行代码的*开始*。）

etext 指定 .text 输出段后的第一个地址。（标志着可执行代码的*结束*。）

.data 指定 .data 输出段的第一个地址。（标志着已初始化数据表的*开始*。）

edata 指定 .data 输出段后的第一个地址。（标志着已初始化数据表的*结束*。）

.bss 指定 .bss 输出段的第一个地址。（标志着未初始化数据的*开始*。）

end 指定 .bss 输出段后的第一个地址。（标志着未初始化数据的*结束*。）

如果使用 --ram_model 或 --rom_model 选项，链接器会自动定义以下符号，以支持 C/C++。

__TI_STACK_SIZE 指定 .stack 段的大小。

__TI_STACK_END 指定 .stack 段的结束。

__TI_SYSMEM_SIZE 指定 .sysmem 段的大小。

__TI_STATIC_BASE	指定值将在引导时加载到数据指针寄存器 (DP) 中。通常这是包含符号（通过 near-DP 寻址引用）定义的第一个段的开始位置。

有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 12.6。

12.5.9.5 向符号分配一个段的确切开始值、结束值和大小值

代码生成工具目前支持在存储器的一个（慢）区域加载程序代码而在另一个（更快）区域运行程序代码的功能。实现此目标的方法是为链接器命令文件中的输出段或组指定单独的加载和运行地址。然后执行一系列指令，从而预先将程序代码从其加载区域移动到其运行区域以满足需要。

在使用此功能设置系统时，程序员必须承担一些责任。其中一项责任是确定要移动的程序代码的大小和运行时地址。当前执行此过程的机制涉及在复制代码中使用 .label 指令。

这种指定程序代码大小和加载地址的方法存在局限性。虽然适用于完全包含在单个源文件中的单个输入段，但如果程序代码分布在多个源文件中，或者如果程序员想要将整个输出段从加载空间复制到运行空间，那么这种方法就会变得更加复杂。

这种方法存在的另一个问题是，它没有考虑到移动的段可能有一个关联的 far 调用 trampoline 段需要随之移动。

12.5.9.6 为什么点运算符有时不起作用

点运算符 (.) 用于在链接时用输出段中的特定地址定义符号。它的解释方式与 PC 类似。无论当前段中的当前偏移量是多少，都是与点关联的值。考虑在 SECTIONS 指令内使用输出段规范：

```
outsect:
{
        s1.c.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.c.obj(.text)
        end_of_s2 = .;
}
```

此语句创建了三个符号：

- end_of_s1 — s1.c.obj 中 .text 的结束地址
- start_of_s2 — s2.c.obj 中 .text 的开始地址
- end_of_s2 — s2.c.obj 中 .text 的结束地址

假设 s1.c.obj 和 s2.c.obj 之间由于对齐而产生了边界填充。那么 start_of_s2 并不是 .text 段真正的起始地址，但却是在 s2.c.obj 中对齐 .text 段所需的边界填充之前的地址。这是由于链接器将点运算符解释为当前 PC。这也是成立的，因为点运算符是独立于其周围的输入段进行评估的。

以上示例中的另一个潜在问题是 end_of_s2 可能不会考虑输出段结尾所需的任何边界填充。end_of_s2 不能可靠地作为输出段的结尾地址。避开这个问题的一种方式是，接在有问题的输出段之后创建一个虚拟段。例如：

```
GROUP
{
   outsect:
   {
      start_of_outsect = .;
      ...
   }
      dummy: { size_of_outsect = .- start_of_outsect; }
}
```

12.5.9.7 地址和维度运算符

使用以下六个运算符可以定义加载时和运行时地址和大小的符号：

LOAD_START(*sym* **)** **START(** *sym* **)**	用相关分配单元的加载时起始地址定义 *sym*

LOAD_END(*sym* **)** **END(** *sym* **)**	用相关分配单元的加载时结束地址定义 *sym*
LOAD_SIZE(*sym* **)** **SIZE(** *sym* **)**	用相关分配单元的加载时大小定义 *sym*
RUN_START(*sym* **)**	用相关分配单元的运行时起始地址定义 *sym*
RUN_END(*sym* **)**	用相关分配单元的运行时结束地址定义 *sym*
RUN_SIZE(*sym* **)**	用相关分配单元的运行时大小定义 *sym*
LAST(*sym* **)**	用相关存储器范围中上次分配的字节的运行时地址定义 *sym*。

备注

链接器命令文件运算符等效性：LOAD_START() 和 START() 是等效的，LOAD_END()/END() 和 LOAD_SIZE()/SIZE() 也是如此。为清楚起见，建议使用 LOAD 名称。

这些地址和维度运算符可与几种不同类型的分配单元（包括输入项、输出段、GROUP 和 UNION）相关联。以下几节提供了一些示例来说明如何在每种情况下使用运算符。

链接器定义的这些符号可在运行时通过 _symval 运算符予以访问，这本质上是一个强制转换操作。例如，假设链接器命令文件包含以下内容：

```
.text: RUN_START(text_run_start), RUN_SIZE(text_run_size) { *(.text) }
```

您的 C 程序可以按如下方式访问这些符号：

```
extern char text_run_start, text_run_size;
printf(".text load start is %lx\n", _symval(&text_run_start));
printf(".text load size is %lx\n",  _symval(&text_run_size));
```

更多有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 12.6。

12.5.9.7.1 输入项

考虑在 SECTIONS 指令内使用输出段规范：

```
outsect:
{
        s1.c.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.c.obj(.text)
        end_of_s2 = .;
}
```

这可以通过使用 START 和 END 运算符重写为以下格式：

```
outsect:
{
        s1.c.obj(.text) { END(end_of_s1) }
        s2.c.obj(.text) { START(start_of_s2), END(end_of_s2) }
}
```

end_of_s1 和 end_of_s2 的值将会相同，就像在原始示例中使用了点运算符一样，但 start_of_s2 将在两个 .text 段之间需要添加的任何必需填充之后进行定义。请记住，点运算符会导致 start_of_s2 在两个输入段之间插入的任何必需填充之前进行定义。

将这些运算符与输入段相关联的语法需要使用大括号 { } 将运算符列表括起来。列表中的运算符应用于紧接在列表之前出现的输入项。

12.5.9.7.2 输出段

START、END 和 SIZE 操作符也可与输出段关联。下面我们举例说明：

```
outsect: START(start_of_outsect), SIZE(size_of_outsect)
{
        <list of input items>
}
```

在此例中，SIZE 操作符定义 size_of_outsect 在输出段中加入必要的边界填充，以符合任何对齐要求。

为输出段指定操作符的语法不需要用括号括住操作符列表。操作符列表只需包含在输出段的分配规范中。

12.5.9.7.3 GROUP

以下是 START 和 SIZE 运算符在 GROUP 规范上下文中的另一种用法：

```
GROUP
{
        outsect1: { ...}
        outsect2: { ...}
} load = ROM, run = RAM, START(group_start), SIZE(group_size);
```

如果要在一个位置加载整个 GROUP 并在另一个位置运行，这会很有用。复制代码可以使用 group_start 和 group_size 作为复制起始地址和复制大小参数。

12.5.9.7.4 UNION

RUN_SIZE 和 LOAD_SIZE 操作符提供了一种机制，区别 UNION 负载空间的大小及其组成部分在运行前将要复制到的空间的大小。下面我们举例说明：

```
UNION: run = RAM, LOAD_START(union_load_addr),
       LOAD_SIZE(union_ld_sz), RUN_SIZE(union_run_sz)
{
        .text1: load = ROM, SIZE(text1_size) { f1.c.obj(.text) }
        .text2: load = ROM, SIZE(text2_size) { f2.c.obj(.text) }
}
```

此处 union_ld_sz 将等于 union 中放置的所有输出段的大小之和。union_run_sz 值等于 union 中最大的输出段。这些符号均会根据分块或对齐要求加入任意边界填充。

12.5.9.8 LAST 运算符

LAST 运算符类似于前面描述的 START 和 END 运算符。但是，LAST 适用于存储器范围而不是段。在 MEMORY 指令中使用该运算符可以定义一个符号，而这个符号可以在运行时用于了解链接程序时分配了多少存储器空间。有关语法的详细信息，请参阅节 12.5.4.2。

例如，一个存储器范围可能定义如下：

```
D_MEM : org = 0x20000020 len = 0x20000000 LAST(dmem_end)
```

然后，您的 C 程序可以在运行时使用 _symval 运算符访问此符号。例如：

```
extern char dmem_end;
printf("End of D_MEM memory is %lx\n", _symval(&dmem_end));
```

更多有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 12.6。

12.5.10 创建和填充孔洞

链接器支持*在输出段内*创建没有任何链接的区域。这些区域被称为*孔洞*。在特殊情况下，未初始化的段也可以被视为孔洞。本节介绍了链接器如何处理孔洞以及如何用值填充孔洞（和未初始化的段）。

12.5.10.1 已初始化和未初始化的段

关于输出段的内容，有两条规则需要注意。输出段符合以下条件之一：

- 包含*整个*段的原始数据
- *没有*原始数据

具有原始数据的段被称为*已初始化*的段。这意味着目标文件包含段的实际存储器映像内容。加载段时，此映像将加载到位于段的指定起始地址处的存储器中。如果有任何内容汇编到 .text 和 .data 段中，那么它们*始终*具有原始数据。

默认情况下，.bss 段和由 .usect 指令定义的段没有原始数据（它们*未初始化*）。它们在存储器映射中占据空间但没有实际内容。未初始化的段通常会在快速外部存储器中为变量保留空间。在目标文件中，一个未初始化的段具有一个正常的段标头并且可以在其中定义符号；但是，该段中没有存储任何存储器映像。

12.5.10.2 创建空洞

用户可以在已初始化的输出段中创建一个空洞。当用户强制链接器在输出段内的输入段之间留出额外空间时，就会创建空洞。创建此类空洞时，*链接器必须为该空洞提供原始数据*。

只能在输出段*内*创建空洞。输出段*之间*可以存在空间，但这种空间不是空洞。若要填充输出段之间的空间，请参阅节 12.5.4.2。

若要在输出段中创建空洞，必须在输出段定义中使用特殊类型的链接器赋值语句。赋值语句会修改 SPC（由 **.** 表示），修改方法是向其添加、为其分配更大的值或在地址边界上将其对齐。有关赋值语句的运算符、表达式和语法的说明，请参阅节 12.5.9。

以下示例使用了赋值语句在输出段中创建空洞：

```
SECTIONS
{
   outsect:
   {
      file1.c.obj(.text)
     . += 0x0100    /* Create a hole with size 0x0100 */
      file2.c.obj(.text)
    . = align(16);  /* Create a hole to align the SPC */
      file3.c.obj(.text)
    }
}
```

输出段 outsect 的构建方式如下：

1. 将 file1.c.obj 中的 .text 段链接进来。
2. 链接器创建一个 256 字节的空洞。
3. 在空洞之后将 file2.c.obj 中的 .text 段链接进来。
4. 链接器通过在 16 字节边界上对齐 SPC 来创建另一个空洞。
5. 最后，将 file3.c.obj 中的 .text 段链接进来。

在一个段内分配给 **.** 符号的所有值指代*该段内的相对地址*。链接器处理 **.** 符号的赋值语句时会认为该段从地址 0 开始（即使已指定绑定地址也是如此）。请考虑示例中的赋值语句 **.**= align(16)。该语句实际上会将 file3.c.obj .text 段与 outsect 内的 16 字节边界上的起点对齐。如果 outsect 最终分配至未对齐的地址起点，则 file3.c.obj .text 段也不会对齐。

. 符号指段的当前运行地址，而不是当前加载地址。

减小 **.** 符号的表达式是非法的。例如，在 **.** 符号的赋值语句中使用 -= 运算符是无效的。**.** 符号的赋值语句中最常用的运算符是 += 和 align。

如果输出段包含某种类型的所有输入段（例如 .text），则可使用以下语句在输出段的开头或结尾创建一个空洞。

```
    .text:   {  .+= 0x0100; }     /* Hole at the beginning */
    .data:   {  *(.data)
                . += 0x0100; }    /* Hole at the end       */
```

在输出段中创建空洞的另一种方法是，将未初始化的段与已初始化的段组合形成单个输出段。*在这种情况下，链接器会将未初始化的段视为空洞并为其提供数据。*以下示例说明了此方法：

```
SECTIONS
{
   outsect:
   {
      file1.c.obj(.text)
     file1.c.obj(.bss)           /* This becomes a hole */
   }
}
```

由于 .text 段有原始数据，整个 outsect 也必须包含原始数据。因此，未初始化的 .bss 段将成为空洞。

未初始化的段只有在与已初始化的段组合时才会成为空洞。如果将多个未初始化的段链接在一起，则生成的输出段也未经初始化。

12.5.10.3 填充孔洞

当已初始化的输出段中存在孔洞时，链接器必须提供原始数据来填充该孔洞。链接器会使用 64 位填充值来填充孔洞。该填充值在存储器中持续复制，直到填满孔洞。链接器按如下方式确定填充值：

1. 如果孔洞是由于将未初始化的段与已初始化的段组合在一起而形成的，则可以为未初始化的段指定填充值。在段名后添加一个 = sign（符号）和一个 64 位常量。例如：

   ```
   SECTIONS
   {  outsect:
      {
         file1.c.obj(.text)
         file2.c.obj(.bss)= 0xFF00FF00  /* 用 0xFF00FF00
   填充该孔洞 */
      }
   }
   ```

2. 还可以通过在段定义之后提供填充值来为输出段中的所有孔洞指定填充值：

   ```
   SECTIONS
   {  outsect:fill = 0xFF00FF00       /* 用 0xFF00FF00 * 填充该孔洞/
      {
         .+= 0x0010;                 /* 这会形成一个孔洞          */
         file1.c.obj(.text)
         file1.c.obj(.bss)              /* 这会形成另一个孔洞    */
      }
   }
   ```

3. 如果没有为某一个孔洞指定初始化值，则链接器将使用通过 --fill_value 选项指定的值来填充该孔洞（请参阅节 12.4.12）。例如，假设命令文件 link.cmd 包含以下 SECTIONS 指令：

   ```
   SECTIONS { .text: { .= 0x0100; } /* Create a 100 word hole */ }
   ```

   现在使用 --fill_value 选项来调用链接器：

   ```
   cl7x --run_linker --fill_value=0xFFFFFFFFFFFFFFFF link.cmd
   ```

   这会用 0xFFFFFFFF 填充孔洞。
4. 如果不使用 --fill_value 选项调用链接器，或以其他方式指定填充值，则链接器将用 0 填充孔洞。

每当在已初始化的输出段中创建并填充孔洞时，都会在链接映射中标识该孔洞以及链接器用来填充该孔洞的值。

12.5.10.4 对未初始化的段进行显式初始化

您可以强制链接器对未初始化的段进行初始化，只需在 SECTIONS 指令中为其指定显式填充值即可实现这一点。这种做法会使整个段具有原始数据（填充值）。例如：

```
SECTIONS
{
   .bss: fill = 0x1234123412341234 /* Fills .bss with 0x1234123412341234 */
}
```

备注

填充段：填充一个段（即使是填充 0）会导致在输出文件中为整个段生成原始数据，因此如果您为很大的段或孔洞指定填充值，则输出文件将非常大。

12.6 链接器符号

C/C++ 源代码可能需要引用由链接器定义的符号，而不是 C/C++ 源代码中的符号。对于充当函数或数组的由链接器定义的符号，您通常可以在 C/C++ 代码中直接引用此类符号。对于 C/C++ 代码中的其他用途，通常需要使用其他技术（例如 _symval 运算符）来访问由链接器定义的符号。后续各小节将介绍这些技术。

12.6.1 链接器定义的函数和数组

在大多数情况下，您可以像访问 C/C++ 函数一样访问链接器定义的函数。只需为这类函数提供外部声明（原型），然后正常访问这类函数即可：

```
extern int linker_defined_function(void);
printf("value is %d\n", linker_defined_function());
```

在大多数情况下，您可以像访问 C/C++ 数组一样访问链接器定义的数组。 只需为这类数组提供外部声明（可以省略第一个维度），然后正常访问数组即可：

```
extern int linker_defined_data[][10][10];
printf("value is %d\n", linker_defined_data[2][3][4]);
```

如果因为函数或数组超出正常地址范围而收到重定位错误，请按照 节 12.6.4 中所述使用 _symval 运算符。

12.6.2 链接器定义的整数值

要访问表示整数值的链接器符号，请使用 _symval 内置运算符，这本质上是一个强制转换操作。

例如，链接器符号 __TI_STACK_SIZE 表示普通整数。要在 C/C++ 代码中获取符号的值作为整数，请使用以下语法：

```
extern void __TI_STACK_SIZE;
size_t get_stack_size() { return _symval(&__TI_STACK_SIZE); }
```

此类外部声明中的类型无关紧要，因为只需要符号的地址。在严格的 ANSI 模式下，无法使用 void 类型来声明此变量，因此请改用 unsigned char。

备注

请勿尝试在 C/C++ 中单独使用 _TI_STACK_SIZE 符号。符号的值未定义，因此符号的地址可能是无效的存储器地址。

要了解为何需要使用 _symval 运算符来访问链接器定义的整数值，请参阅 节 12.6.4，了解链接器符号与 C 标识符的区别。

12.6.3 链接器定义的地址

要访问表示地址的由链接器定义的符号，请使用 _symval 内置运算符获取符号的值来作为指针值。

例如，链接器符号 _TI_STACK_END 表示一个地址。要在 C/C++ 代码中获取符号的值来作为指针值，请使用以下语法：

```
extern void __TI_STACK_END;
void *get_stack_end() { return (void*)_symval(&__TI_STACK_END); }
```

尽管链接器符号 __TI_STACK_END 是一个地址，因此看起来非常像 C/C++ 指针值，但您无法直接将 C/C++ 变量 __TI_STACK_END 定义为指针变量并省略获取符号的地址。有关详细信息，请参阅节 12.6.4。下面是一个*错误*示例。

```
extern void * __TI_STACK_END;
void *get_stack_end() { return __TI_STACK_END; }  // wrong, missing &
```

12.6.4 有关 _symval 运算符的更多信息

当您在 C/C++ 中声明 `int x = 1234;` 等变量时，系统将在地址 &x 处创建标识符（名称）为“x”且内容为 1234 的对象。此外，系统还会创建一个名为 x 的相关链接器符号。该链接器符号仅表示地址，而不是对象本身。引用链接器符号 x 会得到链接器符号的值，即地址。但是，对 C/C++ 标识符 x 的引用会得到内容 1234。如果您需要此 C/C++ 标识符的地址，则需要使用 &x。因此，C/C++ 表达式 &x 与链接器表达式 x 具有相同的值，单链接器符号没有关联类型。

假设链接器定义的符号表示整数而不是地址，例如 __TI_STACK_SIZE。无法在编译器中直接引用整数链接器符号，因此我们使用了一个技巧。首先，假设此链接器符号代表一个地址。在 C/C++ 代码中声明名称相同的假变量。现在，请参考 &__TI_STACK_SIZE，这是一个与链接器符号 __TI_STACK_SIZE 具有相同值的表达式。该值的类型不正确，您可以通过转换更改该类型，如下所示：

```
extern unsigned char __TI_STACK_SIZE;
size_t stack_size = (size_t) &__TI_STACK_SIZE;
```

如以上示例所示，省略 _symval 在大部分时间中都会起作用，但并非总是如此。在某些情况下，指针值不足以表示链接器符号值。例如，一些目标具有 16 位地址空间，因此为 16 位指针。TI 链接器符号为 64 位，因此链接器符号的值可大于可由目标指针表示的值。在这种情况下，表达式 &v 仅反映链接器符号“v”实际值的较低 16 位。要解决此问题，请使用内置的 _symval 运算符，它会复制链接器符号的所有值位：

```
extern void v;
unsigned long value = (unsigned long)_symval(&v);
```

对于每种链接器符号，请使用此模式：

```
extern void name;
desired_type name = (desired_type)_symval(&name);
```

例如，

```
extern void farfunc;

void foo()
{
    void (*func)(void) = (void (*)(void))_symval(&farfunc);
    func();
}
```

在 C7000 器件上，表示整数或出于其他某种原因而不表示有效地址的链接器符号不能始终用指针值表示。在此类情况下使用 _symval 以确保获得正确的值。

12.6.5 使用 _symval、PC 相对寻址和远数据

在 C7000 器件上，大部分符号引用都通过 PC 相对重定位来处理（与位置无关）。这种重定位与当前 PC 值只能相差 +/- 2GB。但是，C7000 器件的对象可能超出此 +/- 2GB 范围，在这种情况下，PC 相对重定位会溢出，并会发生链接器错误。

要访问此类远距离对象，请使用 _symval 运算符获取完整地址。使用 _symval 会强制链接器使用绝对重定位，这会使用完整地址。

12.6.6 弱符号

弱符号是可能定义，也可能不定义的符号。

有关链接器如何处理弱符号的详细信息，请参阅节 8.5.2。

12.6.6.1 弱符号引用

在执行最终链接后，弱符号引用可能具有定义，也可能没有定义。如果符号未定义，则其地址被视为 0。在尝试使用该变量的内容之前，C/C++ 代码必须检查弱引用的地址，以确保该值不为 0。

```
extern __attribute__((weak)) unsigned char * foo;
if (&foo != 0)
    *foo = 1;
```

如果对应于 foo 的链接器符号可能没有有效地址（例如，因为该符号包含整数值而非地址），或者可能超出 2GB 范围 PC 相对寻址，请使用 _symval 内置运算符，如下所示：

```
extern __attribute__((weak)) unsigned char * foo;
if (_symval(&foo) != 0)
    *foo = 1;
```

12.6.6.2 弱符号定义

弱符号定义是有效的定义，但如果在链接时找到此类定义，则会丢弃该定义，取而代之的是非弱定义。您可以在链接器命令文件中定义弱符号 C/C++。

在 C/C++ 中，按如下所示定义弱符号：

```
__attribute__((weak)) int bar;
```

在链接器命令文件中，可使用 MEMORY 或 SECTIONS 指令之外的赋值表达式来确定由链接器定义的符号。若要在链接器命令文件中定义弱符号，请在赋值表达式中使用“弱”运算符来指定可以从输出文件的符号表中删除该符号（如果该符号未被引用）。例如，可将“ext_addr_sym”定义如下：

```
weak(ext_addr_sym) = 0x12345678;
```

当使用链接器命令文件执行最终链接时，“ext_addr_sym”会作为弱绝对符号呈现给链接器。如果未引用此符号，它便不会包含在生成的输出文件中。

12.6.7 利用对象库解析符号

对象库属于分区存档文件，其中包含目标文件。通常，一组相关模块被组合起来构成库。将对象库指定为链接器输入时，链接器会包含定义现有未解析符号引用的所有库成员。您可以使用归档器来构建和维护库。节 10.1 介绍了关于归档器的更多信息。

使用对象库可以缩减可执行模块的链接时间和大小。通常，如果在链接时指定了包含函数的目标文件，则无论是否会使用对应函数，都会链接该文件；不过，如果存档库中存在相同的函数，那么只有引用了对应函数时，才会包含该文件。

指定库的顺序非常重要，因为链接器在搜索库时仅包含那些会解析未定义符号的成员。您可以根据需要多次指定同一个库；只要包含库，链接器就会搜索库。此外，您可以使用 --reread_libs 选项来重新读取库，直到没有其他引用可供解析（请参阅节 12.4.15.3）。库包含一个表格，其中列出了该库中定义的所有外部符号；链接器会通过该表格进行搜索，直到确定无法使用该库来解析任何其他引用。

以下示例会链接多个文件和库并做下列假设：

- 输入文件 f1.c.obj 和 f2.c.obj 都引用一个名为 *clrscr* 的外部函数。
- 输入文件 f1.c.obj 引用了符号 *origin*。
- 输入文件 f2.c.obj 引用了符号 *fillclr*。
- 库 libc.lib 的成员 0 包含 *origin* 的定义。
- 库 liba.lib 的成员 3 包含 *fillclr* 的定义。
- 这两个库的成员 1 都定义了 *clrscr*。

如果您输入：

```
cl7x --run_linker f1.c.obj f2.c.obj liba.lib libc.lib
```

那么：

- liba.lib 的成员 1 会满足 f1.c.obj 和 f2.c.obj 对 *clrscr* 的引用，因为链接器通过搜索该库可以找到 *clrscr* 的定义。
- libc.lib 的成员 0 会满足对 *origin* 的引用。
- liba.lib 的成员 3 会满足对 *fillclr* 的引用。

不过，如果您输入：

```
cl7x --run_linker f1.c.obj f2.c.obj libc.lib liba.lib
```

那么，对 *clrscr* 的引用会由 libc.lib 的成员 1 满足。

如果库中没有定义任何链接的文件引用符号，您可以使用 --undef_sym 选项来强制链接器包含库成员。（请参阅节 12.4.30。）下例会在链接器的全局符号表中创建一个未定义的符号 rout1：

```
cl7x --run_linker --undef_sym=rout1 libc.lib
```

如果 libc.lib 的任何成员定义了 rout1，链接器会包含该成员。

会根据 SECTIONS 指令的默认分配算法来分配库成员；请参阅节 12.5.5。

节 12.4.15 介绍了几种指定对象库所在目录的方法。

12.7 默认放置算法

MEMORY 和 SECTIONS 指令提供了灵活的方法来进行段的构建、组合和分配。但是，任何*未*指定的存储器位置或段仍必须由链接器处理。链接器使用相应的算法并根据您提供的任何规格来构建和分配段。

如果不使用 MEMORY 和 SECTIONS 指令，链接器将从接近零的存储器地址开始分配段。首先放置的是 .text 段，然后放置各种数据段。

有关默认存储器分配的信息，请参阅节 8.4.1。

在可执行输出文件中，所有 .text 输入段被连接起来形成一个 .text 输出段，所有 .data 输入段被组合起来形成一个 .data 输出段。

如果使用 SECTIONS 指令，则会根据 SECTIONS 指令指定的规则和接下来在节 12.7.1 中介绍的通用算法来执行分配。

12.7.1 分配算法如何创建输出段

可通过以下两种方式之一构建输出段：

方法 1	作为 SECTIONS 指令定义的结果
方法 2	通过将同名的输入段组合成一个未在 SECTIONS 指令中定义的输出段

如果作为 SECTIONS 指令的结果构成输出段，则段的内容完全由此定义确定。（如需查看如何定义输出段内容的示例，请参阅节 12.5.5。）

如果输出段是通过组合未由 SECTIONS 指令指定的输入段构成的，则链接器会将所有同名的此类输入段组合到该名称的输出段中。例如，假设文件 f1.c.obj 和 f2.c.obj 都包含名为 Vectors 的指定段，并且 SECTIONS 指令没有为它们定义输出段。链接器会将输入文件中的两个 Vectors 段组合成名为 Vectors 的单个输出段，将这个输出段分配到存储器中，并将其包含在输出文件中。

默认情况下，链接器在创建未在 SECTIONS 指令中定义的输出段时不显示消息。使用 --warn_sections 链接器选项（请参阅节 12.4.31）可以使链接器在创建新的输出段时显示消息。

链接器确定所有输出段的组成方式后，必须将它们分配到配置的存储器中。MEMORY 指令可以指定配置存储器的哪些部分。如果没有 MEMORY 指令，链接器将使用默认配置。（有关配置存储器的更多信息，请参阅节 12.5.4。）

12.7.2 减少存储器碎片

链接器的分配算法会尝试尽量减少存储器碎片。这样可以更高效地使用存储器并增加程序纳入存储器的概率。算法分以下几步：

1. 每个提供具体绑定地址的输出段将置于存储器中的该位置。
2. 将分配包含在具体的已命名存储器范围中的每个输出段，或具有存储器属性限制的每个输出段。每个输出段将置于已命名区域中的第一个可用空间，必要时考虑对齐。
3. 任何其余段的分配顺序与定义顺序相同。未在 SECTIONS 指令中定义的段的分配顺序与遇到段的顺序相同。每个输出段将置于第一个可用存储器空间中，必要时考虑对齐。

12.8 使用由链接器生成的复制表

链接器支持对链接器命令文件语法进行扩展，使您能够：

- 在引导时更轻松地将对象从加载空间复制到运行空间
- 在运行时更轻松地管理存储器叠加
- 拆分具有不同加载和运行地址的 GROUP 和输出段

有关复制表及其使用的介绍，请参阅节 9.3.3。

12.8.1 使用复制表进行引导加载

在某些嵌入式应用中，在引导时需要将代码和/或数据从一个位置复制或下载到另一个位置，然后应用才会开始其主执行线程。例如，应用的代码和/或数据可能位于闪存存储器中，需要将其复制到片上存储器后应用才能开始执行线程。

开发此类应用的一种方式是创建复制表，其中包含每个代码块或数据块的三个元素，需要在引导时将它们从闪存移至片上存储器：

- 加载地址
- 运行地址
- 大小

开发此类应用的流程应该与以下流程类似：

1. 编译应用以生成 .map 文件，其中包含每个段的加载和运行地址，在加载和运行时会分别放置。
2. 编辑复制表（由引导加载程序使用）更正需要在引导时移动的每个代码块或数据块的加载和运行地址以及大小。
3. 再次编译应用，加入更新的复制表。
4. 运行应用。

此流程会为您带来维护复制表的沉重负担（仍然需要手动完成）。应用每次添加或删除一些代码或数据，您都必须重复此流程，以保持复制表的内容是最新的。

12.8.2 在复制表中使用内置链接运算符

使用链接器命令文件句法中包含的 LOAD_START()、RUN_START() 和 SIZE() 操作符，可避免一些维护负担。例如，链接器命令文件可标注为以下形式，而不必编译应用以生成 .map 文件：

```
SECTIONS
{
   .flashcode: { app_tasks.c.obj(.text) }
      load = FLASH, run = PMEM,
      LOAD_START(_flash_code_ld_start),
      RUN_START(_flash_code_rn_start),
      SIZE(_flash_code_size)
   ...
}
```

在本例中，LOAD_START()、RUN_START() 和 SIZE() 运算符指示链接器创建三个符号：

符号	说明
_flash_code_ld_start	.flashcode 段的加载地址
_flash_code_rn_start	.flashcode 段的运行地址
_flash_code_size	.flashcode 段的大小

然后可在复制表中引用这些符号。每次链接应用时，复制表中的实际数据将自动更新。此方法省去了节 12.8.1 中所述流程的步骤 1。

虽然维护复制表的工作显著减少，但您还有责任使复制表内容与链接器命令文件中定义的符号保持同步。理想情况下，链接器会自动生成引导复制表。这样可避免两次编译应用，*而且* 无需再显式管理引导复制表的内容。

有关 LOAD_START()、RUN_START() 和 SIZE() 运算符的更多信息，请参阅节 12.5.9.7。

12.8.3 重叠管理示例

假设应用程序包含存储器重叠区，而必须在运行时管理重叠。存储器重叠区在链接器命令文件中使用 UNION 来定义，如使用 UNION 定义存储器重叠区 中所示：

使用 UNION 定义存储器重叠区

```
SECTIONS
{
   ...
   UNION
   {
      GROUP
      {
         .task1: { task1.c.obj(.text) }
         .task2: { task2.c.obj(.text) }
      } load = ROM, LOAD_START(_task12_load_start), SIZE(_task12_size)
      GROUP
      {
         .task3: { task3.c.obj(.text) }
         .task4: { task4.c.obj(.text) }
      } load = ROM, LOAD_START(_task34_load_start), SIZE(_task_34_size)
   } run = RAM, RUN_START(_task_run_start)
 ...
}
```

应用程序必须在运行时管理存储器重叠区的内容。也就是说，当需要来自 .task1 或 .task2 的任何服务时，应用程序必须首先确保 .task1 和 .task2 驻留在存储器重叠区中。.task3 和 .task4 与此类似。

若要在运行时影响 .task1 和 .task2 从 ROM 到 RAM 的复制，应用程序必须先获取各任务的加载地址 (_task12_load_start)，再获取运行地址 (_task_run_start) 和大小 (_task12_size)。然后使用这些信息来执行实际的代码复制。

12.8.4 使用 table() 运算符生成复制表

链接器支持链接器命令文件语法的扩展，使您能够进行以下操作：

- 确定在应用程序运行期间的某个时刻可能需要从加载空间复制到运行空间的任何对象组件
- 指示链接器自动生成一个复制表，其中（至少）包含需要复制的组件的加载地址、运行地址和大小
- 指示链接器生成您指定的符号，该符号提供由链接器生成的复制表的地址。例如，使用 UNION 定义存储器重叠区 可重新写成为由链接器生成的复制表生成地址所示的代码：

为由链接器生成的复制表生成地址

```
SECTIONS
{
   ...
   UNION
   {
      GROUP
      {
         .task1: { task1.c.obj(.text) }
         .task2: { task2.c.obj(.text) }
      } load = ROM, table(_task12_copy_table)
      GROUP
      {
         .task3: { task3.c.obj(.text) }
         .task4: { task4.c.obj(.text) }
      } load = ROM, table(_task34_copy_table)

   } run = RAM
   ...
}
```

使用链接器命令文件为由链接器生成的复制表生成地址中所示的 SECTIONS 指令，链接器可以生成两个名称如下的复制表：_task12_copy_table 和 _task34_copy_table。每个复制表提供与复制表关联的 GROUP 的加载地址、运行地址和大小。可使用链接器生成的符号 _task12_copy_table 和 _task34_copy_table（它们分别提供两个复制表的地址）从应用程序源代码访问此信息。

通过使用此方法，无需担心复制表的创建或维护。您可以在 C/C++ 中引用由链接器生成的任何复制表的地址，从而将该值传递给通用复制例程，然后该例程将处理复制表并影响实际复制。

12.8.4.1 table() 操作符

您可以使用 table() 操作符指示链接器生成复制表。table() 操作符可应用于输出段、GROUP 或 UNION 成员。为特定 table() 规范生成的复制表可通过一个符号访问，该符号由您指定，作为一个参数提供给 table() 操作符。链接器创建一个具有此名称的符号，为其指定复制表的地址，作为此符号的值。然后可从应用中使用由链接器生成的符号访问此复制表。

您针对已知 UNION 的成员应用的每个 table() 规范必须包含唯一名称。如果将 table() 操作符应用于 GROUP，该 GROUP 的成员不会标记 table() 规范。链接器会检测违反这些规则的情况，并作为警告报告，忽略每次违规使用 table() 规格的情况。链接器不会为错误的 table() 操作符规范生成复制表。

复制表可自动生成；请参阅节 12.8.4。表操作符可使用压缩功能；请参阅节 12.8.5。

12.8.4.2 启动时复制表

链接器支持特殊的复制表名称 BINIT（或 binit）；您可以使用该名称来创建启动时复制表。使用 .cinit 段在启动时初始化变量之前该表会被处理。例如，节 12.8.2 中所述的启动加载应用程序的链接器命令文件可以重写如下：

```
SECTIONS
{
   .flashcode: { app_tasks.c.obj(.text) }
   load = FLASH, run = PMEM,
      table(BINIT)
   ...
}
```

对于此示例，链接器会创建一个复制表，可通过链接器生成的一个特殊符号 __binit__ 来访问该表，其中包含需要在启动时从加载位置复制到运行位置的所有对象组件的列表。如果链接器命令文件未在任何情况下使用 table(BINIT)，则会向 __binit__ 符号赋值 -1 以指示某个特定应用程序不存在启动时复制表。

您可以将 table(BINIT) 规格应用于输出段、GROUP 或 UNION 成员。如果是在 UNION 的上下文中使用，则只能使用 table(BINIT) 指定 UNION 的一个成员。如果是应用于 GROUP，则该 GROUP 的任何成员都不能用 table(BINIT) 进行标记。链接器会检测违反这些规则的情况，并作为警告报告，但会忽略每次违规使用 table(BINIT) 规格的情况。

12.8.4.3 使用 table() 操作符管理目标组件

如果您需要共同管理多个代码段，可以针对多个不同的目标组件应用同一 table() 操作符。此外，如果您要通过多种方式管理特定的目标组件，则可以对其应用多个 table() 操作符。请考虑用于管理目标组件的链接器命令文件中的链接器命令文件片段：

用于管理目标组件的链接器命令文件

```
SECTIONS
{
   UNION
   {
      .first: { a1.c.obj(.text), b1.c.obj(.text), c1.c.obj(.text) }
            load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)
      .second: { a2.c.obj(.text), b2.c.obj(.text) }
            load = EMEM, run = PMEM, table(_second_ctbl)
   }
   .extra: load = EMEM, run = PMEM, table(BINIT)
   ...
}
```

在本例中，输出段 .first 和 .extra 在引导时从外部存储器 (EMEM) 复制到程序存储器 (PMEM)，同时处理 BINIT 复制表。应用开始执行其主线程后，可以使用两个叠加复制表管理叠加的内容，这两个复制表分别命名为：_first_ctbl 和 _second_ctbl。

12.8.4.4 由链接器生成的复制表段和符号

链接器会为由其生成的每个复制表创建并分配一个单独的输入段。每个复制表符号都用包含相应复制表的输入段的地址值进行定义。

链接器会为每个重叠复制表输入段生成一个唯一的名称。例如，table(_first_ctbl) 会将 .first 段的复制表放置在名为 .ovly:_first_ctbl 的输入段中。链接器会创建单个输入段 .binit 以包含整个启动时复制表。

控制由链接器生成的复制表段的放置说明了如何使用链接器命令文件中的输入段名称来控制由链接器生成的复制表段的放置。

控制由链接器生成的复制表段的放置

```
SECTIONS
{
   UNION
   {
      .first: { a1.c.obj(.text), b1.c.obj(.text), c1.c.obj(.text) }
             load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)
      .second: { a2.c.obj(.text), b2.c.obj(.text) }
             load = EMEM, run = PMEM, table(_second_ctbl)
   }
   .extra: load = EMEM, run = PMEM, table(BINIT)
   ...
   .ovly: { } > BMEM
   .binit: { } > BMEM
}
```

对于控制由链接器生成的复制表段的放置中的链接器命令文件，启动时复制表会生成到 .binit 输入段中，该输入段收集到 .binit 输出段中，该输出段映射到 BMEM 存储器区域中的某个地址。_first_ctbl 生成到 .ovly:_first_ctbl 输

入段中，_second_ctbl 生成到 .ovly:_second_ctbl 输入段中。这些输入段的基本名称与 .ovly 输出段的名称匹配，因此这些输入段会收集到 .ovly 输出段中，然后该输出段映射到 BMEM 存储器区域中的某个地址。

如果没有为由链接器生成的复制表段提供显式的放置指令，则会根据链接器的默认放置算法来分配它们。

链接器不允许将其他类型的输入段与同一输出段中的复制表输入段进行组合。链接器不允许将从部分链接会话创建的复制表段用作后续链接会话的输入。

12.8.4.5 拆分对象组件和重叠管理

可以拆分包含单独加载和运行放置指令的段。链接器可以访问拆分对象组件每个部分的加载地址和运行地址。使用 table() 操作符，用户可以指示链接器生成此信息并将其放入复制表。链接器会在复制表对象中为拆分对象组件的每个部分提供一个 COPY_RECORD 条目。

例如，设想一个具有七个任务的应用。任务 1 至 3 与任务 4 至 7 重叠（使用 UNION 指令）。所有这些任务的加载放置都会在四个不同的存储器区域（LMEM1、LMEM2、LMEM3 和 LMEM4）之间分配。重叠定义为存储器区域 PMEM 的一部分。用户必须在运行时将每组任务移入重叠区，才能使用该组中的服务。

用户可以将 table() 操作符与分拆操作符 >> 结合使用来创建包含所有所需信息的复制表，以将任一组任务移入存储器重叠区，如创建复制表来访问拆分对象组件中所示。

创建复制表来访问拆分对象组件

```
SECTIONS
{
   UNION
   {
      .task1to3: { *(.task1), *(.task2), *(.task3) }
              load >> LMEM1 | LMEM2 | LMEM4, table(_task13_ctbl)
      GROUP
      {
         .task4: { *(.task4) }
         .task5: { *(.task5) }
         .task6: { *(.task6) }
         .task7: { *(.task7) }
      } load >> LMEM1 | LMEM3 | LMEM4, table(_task47_ctbl)
   } run = PMEM
   ...
   .ovly: > LMEM4
}
```

拆分对象组件驱动程序显示了此类应用可能的驱动程序。

拆分对象组件驱动程序

```
#include <cpy_tbl.h>
extern COPY_TABLE task13_ctbl;
extern COPY_TABLE task47_ctbl;
extern void task1(void);
...
extern void task7(void);
main()
{
   ...
   copy_in(&task13_ctbl);
   task1();
   task2();
   task3();
   ...
   copy_in(&task47_ctbl);
   task4();
   task5();
   task6();
   task7();
   ...
}
```

.task1to3 段的内容会分拆到该段的加载空间并且在其运行空间中是连续的。由链接器生成的复制表 _task13_ctbl 会针对分拆段 .task1to3 的每个部分包含一个单独的 COPY_RECORD。当 _task13_ctbl 的地址被传递到 copy_in() 时，.task1to3 的每个部分都会从其加载位置复制到运行位置。

包含任务 4 至 7 的 GROUP 的内容也会拆分到加载空间中。链接器会按顺序为 GROUP 中每个成员应用分拆操作符，从而执行 GROUP 分拆。然后，该 GROUP 的复制表会针对该 GROUP 中每一个成员的每个部分包含一个 COPY_RECORD 条目。当 copy_in() 对 _task47_ctbl 进行处理时，这些部分都会复制到存储器重叠区。

分拆操作符可应用于输出段、GROUP、UNION 或 UNION 成员的加载放置。链接器不允许将分拆操作符应用于 UNION 或 UNION 成员的运行放置。链接器会检测此类违规情况，发出警告，并忽略违规使用分拆操作符的情况。

12.8.5 压缩

自动生成复制表时，链接器提供了一种压缩加载空间数据的方法。这可以减少只读存储器占用空间。在将压缩的数据从加载空间复制到运行空间时可以解压缩此数据。

可通过两种方式指定压缩：

- 可使用链接器命令行选项 --copy_compression=compression_kind 将指定的压缩应用于任何应用了 table() 运算符的输出段。
- table() 运算符接受一个可选的压缩参数。语法为：.

  table(*name* **, compression=** *compression_kind* **)**

  compression_kind 可以是以下类型之一：
  - **off**。不要压缩数据。
  - **rle**。使用行程编码格式压缩数据。
  - **lzss**。使用 Lempel-Ziv-Storer-Szymanski 压缩格式压缩数据。

不带 compression 关键字的 table() 运算符使用通过命令行选项 --copy_compression 指定的压缩类型。如果没有使用命令行选项指定 *compression_kind*，则默认为 LZSS 压缩。

选择压缩时，不能保证链接器将会压缩加载数据。仅当压缩会减小加载空间的整体大小时，链接器才会压缩加载数据。在某些情况下，即使压缩会使加载段大小变小，如果解压缩例程会抵消节省量，那么链接器也不会压缩数据。

例如，假设 RLE 压缩将 section1 的大小减少 30 个字节。此外，还假设 RLE 解压缩例程在加载空间中占用 40 字节。通过选择对 section1 进行压缩，加载空间将增加 10 个字节。因此，链接器不会压缩 section1。另一方面，如果有另一个段（比如 section2）可以从应用相同的压缩中受益超过 10 个字节，则可以压缩这两个段，使得整体加载空间减小。在此类情况下，链接器会压缩这两个段。

当这样做无法实现节省时，不能强制链接器压缩数据。

不能对解压缩例程或包含 .cinit 的 GROUP 中的任何成员进行压缩。

12.8.5.1 压缩的复制表格式

无论 *compression_kind* 如何，复制表格式都是相同的。复制记录的大小字段将被重载以支持压缩。图 12-4 显示了压缩的复制表布局。

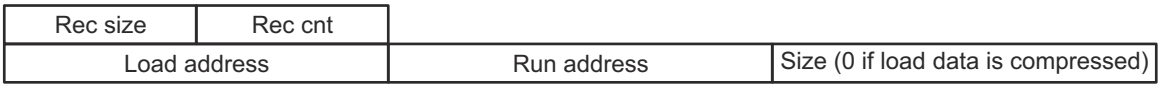


图 12-4. 压缩的复制表

在图 12-4 中，如果复制记录中的大小不为零，则此大小表示要复制的数据的大小，也意味着加载数据的大小与运行数据的大小相同。此大小为 0 时，表示加载数据被压缩。

12.8.5.2 目标文件中的压缩段表示

链接器会创建一个单独的输入段来保存压缩数据。请考虑链接器命令文件中的以下 table() 操作。

```
SECTIONS
{
    .task1: load = ROM, run = RAM, table(_task1_table)
}
```

输出目标文件有一个名为 .task1 的输出段，其中具有不同的加载地址和运行地址。这是可行的，因为当段未被压缩时，加载空间和运行空间具有相同的数据。

或者，请考虑以下代码：

```
SECTIONS
{
    .task1: load = ROM, run = RAM, table(_task1_table, compression=rle)
}
```

如果链接器对 .task1 段进行压缩，则加载空间数据和运行空间数据会不同。链接器会创建以下两个段：

- **.task1**：此段未初始化。此输出段表示 task1 段的运行空间映像。
- **.task1.load**：此段已初始化。此输出段表示 task1 段的加载空间映像。此段的大小通常比 .task1 输出段小得多。

在为 .task1 段的加载放置指定的存储器区域中，链接器为 .task1.load 输入段分配加载空间。只有一个加载段用于表示 .task1 的加载放置，也就是 .task1.load 段。如果尚未压缩 .task1 数据，则 .task1 输入段将有两个分配：一个用于该段的加载放置，另一个用于该段的运行放置。

12.8.5.3 压缩的数据布局

压缩的加载数据具有以下布局：

位索引	压缩的数据

加载数据的前 位是处理程序索引。此处理程序索引用于索引到处理程序表中，以获取知道如何解码后续数据的处理程序函数的地址。处理程序表是 48 位函数指针的列表，如图 12-5 所示。

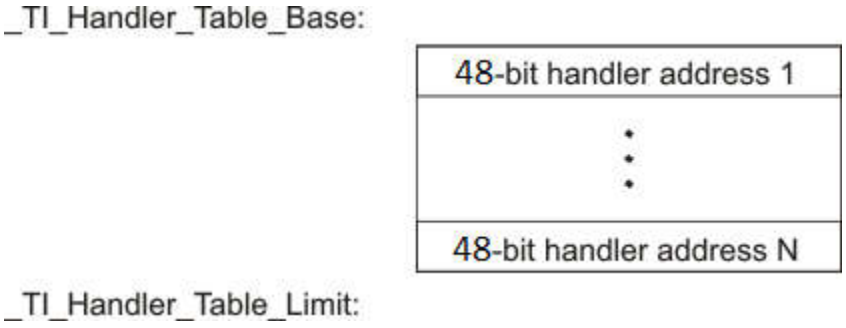


图 12-5. 处理程序表

链接器为加载和运行空间创建一个单独的输出段。例如，如果使用了 RLE 来压缩 .task1.load，则处理程序索引会指向处理程序表中具有运行时支持例程 __TI_decompress_rle() 地址的条目。

12.8.5.4 运行时解压缩

在运行时期间，用户可以调用运行时支持例程 copy_in() 来将数据从加载空间复制到运行空间。复制表的地址会传递给此例程。首先，该例程读取记录计数。然后，它会针对每条记录重复以下步骤：

1. 从记录中读取加载地址、运行地址和大小。
2. 如果大小为零，则转到第 5 步。
3. 调用 memcpy 来传递运行地址、加载地址和大小。
4. 如果要读取其他记录，则转到第 1 步。
5. 读取加载地址的。
6. 从 (&__TI_Handler_Base)[index] 读取处理程序地址。
7. 调用处理程序并传递加载地址 + 1 和运行地址。
8. 如果要读取其他记录，则转到第 1 步。

运行时支持库中提供了用于处理加载数据解压缩的例程。

12.8.5.5 压缩算法

以下几个小节提供了有关 RLE 和 LZSS 格式的解压缩算法的信息。若要查看解压缩算法的示例，请参阅运行时支持库中的以下函数：

- **RLE：**copy_decompress_rle.c 文件中的 __TI_decompress_rle() 函数。
- **LZSS：**copy_decompress_lzss.c 文件中的 __TI_decompress_lzss() 函数。

行程编码 (RLE)：

位索引	使用行程编码压缩的初始化数据

位索引之后的数据使用行程编码 (RLE) 格式进行压缩。C7000 使用一种可以使用以下算法解压缩的简单行程编码：请查看 copy_decompress_rle.c 以了解详细信息。

1. 读取第一个，即分隔符 (D)。
2. 读取下一个 (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 (L)。
   a. 如果 L == 0，则长度要么是 值，要么已经到达数据的末尾，读取下一个 (L)。
   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 位值 (L)。
5. 读取下一个 (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。

C7000 运行时支持库有一个例程 __TI_decompress_rle() 可以解压缩使用 RLE 压缩的数据。此函数的第一个参数是指向（位于 位索引后）的地址，第二个参数是 C 自动初始化记录的运行地址。

Lempel-Ziv-Storer-Szymanski 压缩 (LZSS)：

位索引	使用 LZSS 压缩的数据

8 位索引之后的数据使用 LZSS 压缩格式进行压缩。C7000 运行时支持库有一个例程 __TI_decompress_lzss() 可以解压缩使用 LZSS 压缩的数据。此函数的第一个参数是指向（位于 位索引后）的地址，而第二个参数是 C 自动初始化记录的运行地址。

请查看 copy_decompress_lzss.c 以了解 LZSS 算法的详细信息。

12.8.6 复制表内容

要使用由链接器生成的复制表，用户必须知道复制表的内容。此信息包含在一个运行时支持库头文件 cpy_tbl.h 中。该文件包含由链接器生成的复制表数据结构的 C 源代码表示。

C7000 cpy_tbl.h 文件

```
/*****************************************************************************/
/* cpy_tbl.h                                                                 */
/*                                                                           */
/* Copyright (c) 2003 Texas Instruments Incorporated     http://www.ti.com/  */
/*                                                                           */
/* Specification of copy table data structures which can be automatically    */
/* generated by the linker (using the table() operator in the LCF).          */
/*****************************************************************************/
#ifndef _CPY_TBL
#define _CPY_TBL
#include <stdint.h>
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
/*****************************************************************************/
/* 复制记录数据结构                                                          */
/*****************************************************************************/
typedef struct copy_record
{
   uint64_t load_addr;
   uint64_t run_addr;
   uint32_t size;
} __attribute((__packed__)) COPY_RECORD;
/*****************************************************************************/
/* 复制表数据结构                                                            */
/*****************************************************************************/
typedef struct copy_table
{
   uint16_t rec_size;
   uint16_t num_recs;
   COPY_RECORD    recs[1];
} COPY_TABLE;
/*****************************************************************************/
/* 通用复制例程原型。                                                        */
/*****************************************************************************/
extern void copy_in(COPY_TABLE *tp);
#ifdef __cplusplus
} /* extern "C" */
#endif /* __cplusplus */
#endif /* !_CPY_TBL */
```

对于已标记要进行复制的每个对象组件，链接器会为其创建一个 COPY_RECORD 对象。每个 COPY_RECORD 至少包含对象组件的以下信息：

- 加载地址
- 运行地址
- 大小

链接器会将与同一复制表关联的所有 COPY_RECORD 收集到一个 COPY_TABLE 对象中。COPY_TABLE 对象包含给定 COPY_RECORD 的大小、表中 COPY_RECORD 的数量以及表中 COPY_RECORD 的数组。例如，在节 12.8.4.2 的 BINIT 示例中，各输出段 .first 和 .extra 将在 BINIT 复制表中拥有各自的 COPY_RECORD 条目。BINIT 复制表将如下所示：

```
COPY_TABLE __binit__ = { 12, 2,
                         { <load address of .first>,
                           <run address of .first>,
                           <size of .first> },
                         { <load address of .extra>,
                           <run address of .extra>,
                           <size of .extra> } };
```

12.8.7 通用复制例程

C7000 cpy_tbl.h 文件中的 cpy_tbl.h 文件还包含运行时支持库中提供的通用复制例程 copy_in() 的原型。copy_in() 例程只接受一个参数：由链接器生成的复制表的地址。该例程随后会处理复制表数据对象，并执行复制表中指定的每个对象组件的复制。

运行时支持 cpy_tbl.c 文件显示的 cpy_tbl.c 运行时支持源文件中提供了 copy_in() 函数定义。

运行时支持 cpy_tbl.c 文件

```
/*****************************************************************************/
/* cpy_tbl.c                                                                 */
/*                                                                           */
/* 2011 德州仪器版权所有                                */
/*                                                                           */
/* 通用复制例程。给定由链接器生成的       */
/* COPY_TABLE 数据结构的地址，通过      */
/* 相应的 LCF table() 运算符复制指定用于复制的所有对象组件。*/
/*                                                                           */
/*****************************************************************************/
#include <cpy_tbl.h>
#include <string.h>
typedef void (*handler_fptr)(const unsigned char *in, unsigned char *out);
/*****************************************************************************/
/* COPY_IN()                                                                 */
/*****************************************************************************/
void copy_in(COPY_TABLE *tp)
{
   unsigned short I;
   for (I = 0; I < tp->num_recs; I++)
   {
      COPY_RECORD crp = tp->recs[i];
      unsigned char *ld_addr = (unsigned char *)crp.load_addr;
      unsigned char *rn_addr = (unsigned char *)crp.run_addr;
      if (crp.size)
      {
         /*-------------------------------------------------------------------*/
         /* 复制记录具有非零大小，因此数据不会被压缩。    */
         /* 仅复制数据。                                                      */
         /*-------------------------------------------------------------------*/
         memcpy(rn_addr, ld_addr, crp.size);
      }
#ifdef __TI_EABI__
      else if (HANDLER_TABLE)
      {
         /*-------------------------------------------------------------------*/
         /* 复制记录的大小为零，因此数据会被压缩。加载数据    */
         /* 的第一个字节具有处理程序索引。将此索引与 */
         /* 处理程序表一同使用以获取此数据的处理程序。然后通过     */
         /* 传递加载和运行地址来调用处理程序。                    */
         /*-------------------------------------------------------------------*/
         unsigned char index = *((unsigned char *)ld_addr++);
         handler_fptr hndl = (handler_fptr)(&HANDLER_TABLE)[index];
         (*hndl)((const unsigned char *)ld_addr, (unsigned char *)rn_addr);
      }
#endif
   }
}
```

12.9 部分（增量）链接

已链接的输出文件可以继续与其他模块链接。这称为*部分链接*或*增量链接*。通过部分链接，可对较大的应用程序进行分区，分别链接每个部分，然后将所有部分链接在一起，以创建最终的可执行程序。请遵循以下指导原则来生成要重新链接的文件：

- 由链接器生成的中间文件*必须*具有重定位信息。首次链接文件时使用 --relocatable 选项。（请参阅节 12.4.3.2。）
- 中间文件*必须*包含符号信息。默认情况下，链接器会在其输出中保留符号信息。如果用户打算重新链接文件，请勿使用 --no_sym_table 选项，因为 --no_sym_table 会从输出模块中去除符号信息。（请参阅节 12.4.21。）
- 只有构建输出段且未进行分配时，才应考虑使用中间链接操作。所有分配、绑定和 MEMORY 指令都应在最终链接中执行。由于 ELF 目标文件格式 搭配使用，部分链接期间不会将各输入段合并成输出段，除非链接步骤命令文件中指定了匹配的 SECTIONS 指令。
- 如果中间文件中的全局符号与其他文件中的全局符号具有相同的名称，并且用户想要将它们视为静态符号（仅在该中间文件中可见），则必须使用 --make_static 选项链接各文件（请参阅 节 12.4.16.1）。
- 如果是链接 C 代码，在最终链接器之前，请勿使用 --ram_model 或 --rom_model。每次使用 --ram_model 或 --rom_model 选项调用链接器时，链接器都会尝试创建一个入口点。（请参阅节 12.4.24 、节 9.3.2.1 和 节 9.3.2.2 。）

以下示例展示了如何使用部分链接：

步骤 1： 链接文件 file1.com；使用 --relocatable 选项在输出文件 out1.out 中保留重定位信息。

```
cl7x --run_linker --relocatable --output_file=out1 file1.com
```

file1.com 包含：

```
SECTIONS
{
   ss1:   {
          f1.c.obj
          f2.c.obj
            ...
          fn.c.obj
          }
}
```

步骤 2： 链接文件 file2.com；使用 --relocatable 选项以在输出文件 out2.out 中保留重定位信息。file2.com 包含：

```
SECTIONS
{
   ss2:   {
          g1.c.obj
          g2.c.obj
            ...
          gn.c.obj
          }
}
```

步骤 3： 链接 out1.out 和 out2.out。

```
cl7x --run_linker --map_file=final.map --output_file=final.out out1.out out2.out
```

12.10 链接 C/C++ 代码

C/C++ 编译器生成可进行汇编和链接的汇编语言源代码。例如，可以汇编包括模块 prog1、prog2 等的 C 程序，然后进行链接以生成名为 prog.out 的可执行文件：

```
cl7x --run_linker --rom_model --output_file prog.out prog1.c.obj prog2.c.obj ... rts7100_le.lib
```

--rom_model 选项告诉链接器使用由 C/C++ 环境定义的特殊约定。

TI 提供的归档库包含 C/C++ 运行时支持函数。

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。

12.10.1 运行时初始化

C/C++ 程序必须与用于初始化和执行程序的代码链接起来，这个代码被称为 *bootstrap* 例程（*boot.c.obj* 对象模块）。_c_int00 符号定义为程序入口点，是 boot.c.obj 中 C 引导例程的起点。引用 _c_int00 可确保自动从运行时支持库将 boot.c.obj 链接进来。程序在运行时会首先执行 boot.c.obj。boot.c.obj 符号包含用于初始化运行时环境的代码和数据；该符号执行以下任务：

- 设置系统堆栈和配置寄存器
- 处理运行时 *.cinit* 初始化表并自动初始化全局变量（如果使用了 --rom_model）
- 禁用中断和调用 _main

运行时支持对象库中包含 boot.c.obj。用户可以：

- 使用归档器以从库中提取 boot.c.obj，然后直接链接该模块。
- 添加合适的运行时支持库作为输入文件（当用户使用 --ram_model 或 --rom_model 选项时，链接器会自动提取 boot.c.obj）。

12.10.2 对象库和运行时支持

章节 7 介绍了 rts.src 中包含的其他运行时支持函数。如果用户的程序使用这些函数，则必须将适当的运行时支持库与目标文件链接。

用户还可以自行创建对象库并链接它们。链接器仅包含和链接那些会解析未定义引用的库成员。

12.10.3 设置堆栈段的大小

C/C++ 语言使用两个未初始化段，被称为 .sysmem 和 .stack 分别用于设置 malloc( ) 函数和运行时堆栈使用的存储器池。用户可以通过使用 --heap_size 或 --stack_size 选项并紧跟该选项后指定段大小作为 常量来设置这些存储器池的大小。如果不使用上述选项，堆的默认大小为 1K 字节，而栈的默认大小为 1K 字节。

请参阅节 12.4.13 来设置堆大小参阅节 12.4.27 来设置栈大小。

12.10.4 在运行时初始化和自动初始化变量

在运行时自动初始化变量是默认的自动初始化方法。若要使用此方法，请使用 --rom_model 选项调用链接器。详细信息请参阅 节 9.3.2.1 。

在加载时初始化变量可通过缩短启动时间和节省初始化表使用的存储器来提高性能。若要使用此方法，请使用 --ram_model 调用链接器。详细信息请参阅 节 9.3.2.2 。

请参阅节 9.3.2.3，了解使用 --ram_model 或 --rom_model 调用链接器时需执行的步骤。

12.10.5 CMMU 配置产生的约束

C7000 编译器可能会在矢量化过程中生成矢量谓词存储，这些矢量谓词存储最多可扩展到超出地址末尾 63 个字节。如果 MEMORY 指令定义的范围之后的 63 个字节可能由于 Corepac 存储器管理单元 (CMMU) 配置而在运行时无法访问，则该存储器范围的长度最多应减少 63 个字节，以防止在存储器中放置可能由于矢量谓词存储而在运行时导致页错误的段。

有关 MEMORY 指令的更多信息，请参阅节 12.5.4。有关矢量化的更多信息，请参阅《*C7000 C/C++ 优化指南*》(SPRUIV4)。

12.11 链接器示例

此示例链接了名为 demo.c.obj、filter.c.obj 和 tables.obj 的三个目标文件，并创建了名为 demo.out 的程序。

假定目标存储器具有以下程序存储器配置：

地址范围	内容
0x0080 至 0x7000	片上 RAM_PG
0xC000 至 0xFF80	片上 ROM

地址范围	内容
0x0080 至 0x0FFF	RAM 块 ONCHIP
0x0060 至 0xFFFF	映射的外部地址 EXT

地址范围	内容
0x00000020 至 0x00210000	PMEM
0x00400000 至 0x01400000	EXT0
0x01400000 至 0x01800000	EXT1
0x02000000 至 0x03000000	EXT2
0x03000000 至 0x04000000	EXT3
0x40000000 至 0x82000000	BMEM

输出段的构造方式如下：

- demo.c.obj、filters.c.obj 中 .text 段包含的可执行代码，以及来自 RTS 库的可执行代码链接到程序存储器 PMEM。
- tables.c.obj 中定义了两个数据对象。每个对象置于其自有输出段中：.tableA 和 .tableB。加载程序时，.tableA 和 .tableB 输出段链接到 BMEM 区域中的单独位置。但是，对这些表的运行时访问引用由符号“filter_matrix”指示的运行时位置，其定义为包含 .tableA 和 .tableB 的 UNION 的起始位置。此位置链接到 EXT1 存储器区域。在运行时，在尝试从复制的表中访问数据之前，应用程序负责将 .tableA 或 .tableB 从其在 BMEM 中的加载位置复制到在 EXT1 中的运行位置。链接器支持节 12.8 中所述的复制表机制，以为完成此操作提供便利。
- 对于支持并使用 DP 相对寻址的架构，使用 DP 相对寻址访问的所有数据对象均收集到一个包括 .bss 输出段的组中。该组链接到 BMEM 存储器区域。
- 由于 demo.out 程序使用在加载和运行 demo.out 时必须指定的命令行参数，应用程序必须预留空间，以便将命令行参数传递至 .args 段中的程序。为 .args 段分配的空间大小在靠近链接器命令文件顶部的‘--args 0x1000’选项中指示。然后将 .args 输出段链接到 BMEM 存储器区域。在 RTS 库中包含的引导例程中提供了对处理命令行参数的支持，该引导例程将链接到 demo.out 程序。
- 软件栈的大小通过靠近链接器命令文件顶部的“--stack 0x6000”选项来指示。同样，堆的大小（通过它可以在运行时动态地分配存储器）通过靠近链接器命令文件顶部的“--heap 0x3000”选项来指示。.stack 和 .sysmem（包含堆）输出段链接到 BMEM 存储器区域。

链接器命令文件 mylnk.cmd 展示了此示例的链接器命令文件。输出映射文件，demo.map 展示了映射文件。

链接器命令文件 mylnk.cmd

```
/****************************************************************************/
/*** 指定链接器选项                                                       ***/
/****************************************************************************/
-cr                                  /* --ram_model: 加载时初始化 */
--heap 0x3000
--stack 0x6000
--args 0x1000
--output_file=demo.out                    /* 命名输出文件       */
--map_file=demo.map                       /* 创建输出映射文件  */
--undefined_sym=filter_table_A            /* 引入未定义符号 */
```

```
--undefined_sym=filter_table_B            /* 引入未定义符号 */
/****************************************************************************/
/*** 指定输入文件                                                         ***/
/****************************************************************************/
demo.c.obj
tables.obj
filter.c.obj
/****************************************************************************/
/*** 指定要链接的运行时支持库                                             ***/
/****************************************************************************/
-l libc.a
/****************************************************************************/
/*** 指定存储器配置                                                       ***/
/****************************************************************************/
MEMORY
{
  PMEM: o = 00000020h l = 0020ffe0h
  EXT0: o = 00400000h l = 01000000h
  EXT1: o = 01400000h l = 00400000h
  EXT2: o = 02000000h l = 01000000h
  EXT3: o = 03000000h l = 01000000h
  BMEM: o = 40000000h l = 02000000h
}
/****************************************************************************/
/*** 指定输出段                                                           ***/
/****************************************************************************/
SECTIONS
{
  .text : > PMEM
  UNION
  {
    .tableA: { tables.obj(tableA) } load > BMEM, table(tableA_cpy)
    .tableB: { tables.obj(tableB) } load > BMEM, table(tableB_cpy)
  } RUN = EXT1, RUN_START(filter_matrix)
  GROUP
  {
    .rodata:
    .bss:
  } > EXT2
  .stack:  > BMEM
  .args :  > BMEM
  .cinit:  > BMEM
  .cio:    > BMEM
  .const:  > BMEM
  .data:   > BMEM
  .sysmem: > BMEM
}
/****************************************************************************/
/*** 命令文件结束                                                         ***/
/****************************************************************************/
```

输入以下命令来调用链接器：

```
cl7x --run_linker mylnk.cmd
```

这会创建一个如输出映射文件，demo.map 中所示的映射文件，以及一个名为 demo.out 的输出文件，可以在 C7000 上运行。

输出映射文件，demo.map

```
OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: "_c_int00"  address: 000000007ec0
MEMORY CONFIGURATION
         name            origin         length     used     unused   attr    fill
----------------------  -------------  --------  --------  --------  ----  --------
  PMEM                  000000000020   0020ffe0  00008170  00207e70  RWIX
  EXT0                  000000400000   01000000  00000000  01000000  RWIX
  EXT1                  000001400000   00400000  00000080  003fff80  RWIX
  EXT2                  000002000000   01000000  000000b0  00ffff50  RWIX
  EXT3                  000003000000   01000000  00000000  01000000  RWIX
  BMEM                  000040000000   02000000  0000a6e4  01ff591c  RWIX
SEGMENT ALLOCATION MAP
```

```
   run origin     load origin     length   init length attrs members
--------------  --------------  ---------- ----------- ----- -------
000000000040    000000000040    00008170   00008170    r-x
  000000000040    000000000040    00008140   00008140    r-x .text
  000000008180    000000008180    00000030   00000030    r-- .ovly
000001400000    00004000a5e4    00000080   00000080    rw-
  000001400000    00004000a5e4    00000080   00000080    rw- .tableA
000001400000    00004000a664    00000080   00000080    rw-
  000001400000    00004000a664    00000080   00000080    rw- .tableB
000002000000    000002000000    000000b0   00000000    rw-
  000002000000    000002000000    000000b0   00000000    rw- .bss
000040000000    000040000000    00009000   00000000    rw-
  000040000000    000040000000    00006000   00000000    rw- .stack
  000040006000    000040006000    00003000   00000000    rw- .sysmem
000040009000    000040009000    00001384   00001384    rw-
  000040009000    000040009000    00001000   00001000    rw- .args
  00004000a000    00004000a000    00000384   00000384    rw- .data
00004000a384    00004000a384    00000140   00000140    r--
  00004000a384    00004000a384    00000140   00000140    r-- .const
00004000a4c4    00004000a4c4    00000120   00000000    rw-
  00004000a4c4    00004000a4c4    00000120   00000000    rw- .cio
SECTION ALLOCATION MAP
 output                                  attributes/
section   page     origin        length       input sections
--------  ----  -------------  ----------   ----------------
.text       0    000000000040   00008140
                   000000000040   00002400     rts7100_le.lib : _printfi.c.obj (.text:__TI_printfi)
                   000000002440   00000900                    : _printfi.c.obj (.text:_pconv_a)
                   000000002d40   00000800                    : _printfi.c.obj (.text:_pconv_g)
                   000000003540   00000680                    : _printfi.c.obj (.text:_pconv_e)
                   000000003bc0   000003c0                    : memory.c.obj (.text:aligned_alloc)
                   000000003f80   00000380                    : frcdivd.c.obj (.text:__TI_frcdivd)
                   000000004300   00000340                    : _printfi.c.obj (.text:_pconv_f)
                   000000004640   00000300                    : _printfi.c.obj (.text:fcvt)
                   000000004940   00000300                    : fputs.c.obj (.text:fputs)
...
.rodata     0    000002000000   00000000     UNINITIALIZED
.bss        0    000002000000   000000b0     UNINITIALIZED
                   000002000000   000000a0       (.common:__TI_tmpnams)
                   0000020000a0   00000008       rts7100_le.lib : memory.c.obj (.bss)
                   0000020000a8   00000008       (.common:parmbuf)
.stack      0    000040000000   00006000     UNINITIALIZED
                   000040000000   00000010       rts7100_le.lib : boot.c.obj (.stack)
                   000040000010   00005ff0       --HOLE--
.sysmem     0    000040006000   00003000     UNINITIALIZED
                   000040006000   00000010       rts7100_le.lib : memory.c.obj (.sysmem)
                   000040006010   00002ff0       --HOLE--
.args       0    000040009000   00001000
                   000040009000   00001000       --HOLE-- [fill = 0]
.data       0    00004000a000   00000384
                   00004000a000   000001e0       rts7100_le.lib : defs.c.obj (.data:_ftable)
                   00004000a1e0   000000d8                      : host_device.c.obj (.data:_device)
                   00004000a2b8   000000a0                      : host_device.c.obj (.data:_stream)
                   00004000a358   00000010                      : exit.c.obj (.data)
                   00004000a368   00000008                      : _lock.c.obj (.data:_lock)
                   00004000a370   00000008                      : _lock.c.obj (.data:_unlock)
                   00004000a378   00000004                      : defs.c.obj (.data)
                   00004000a37c   00000004                      : errno.c.obj (.data)
                   00004000a380   00000004                      : memory.c.obj (.data)
.const      0    00004000a384   00000140
                   00004000a384   00000101       rts7100_le.lib : ctype.c.obj
(.const:.string:_ctypes_)
                   00004000a485   00000003       --HOLE-- [fill = 0]
                   00004000a488   00000024                      : _printfi.c.obj (.const:.string)
                   00004000a4ac   00000018       demo.c.obj (.const:.string)
.tableA     0    00004000a5e4   00000080     RUN ADDR = 000001400000
                   00004000a5e4   00000080       tables.obj (tableA)
.tableB     0    00004000a664   00000080     RUN ADDR = 000001400000
                   00004000a664   00000080       tables.obj (tableB)
.cinit      0    000040000000   00000000     UNINITIALIZED
.cio        0    00004000a4c4   00000120     UNINITIALIZED
                   00004000a4c4   00000120       rts7100_le.lib : trgmsg.c.obj (.cio)
.ovly       0    000000008180   00000030
                   000000008180   00000018       (.ovly:tableA_cpy)
                   000000008198   00000018       (.ovly:tableB_cpy)
...
```

```
LINKER GENERATED COPY TABLES
tableA_cpy @ 00008180 records: 1, size/record: 20, table size: 24
        .tableA: load addr=00004000a5e4, load size=00000080, run addr=000001400000,
         run size=00000080, compression=none
tableB_cpy @ 00008198 records: 1, size/record: 20, table size: 24
        .tableB: load addr=00004000a664, load size=00000080, run addr=000001400000,
         run size=00000080, compression=none
GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name
address       name
------------  ----
000000008040  C$$EXIT
000000006eb0  C$$IO$$
0000000069c0  HOSTclose
000000006100  HOSTlseek
000000005f80  HOSTopen
...
000000008180  tableA_cpy
000000008198  tableB_cpy
000000007cc0  unlink
000000007d40  wcslen
000000007dc0  write
...
[96 symbols]
```

This page intentionally left blank.

章节 13

目标文件实用程序

本章介绍了如何调用以下实用程序：

- **目标文件显示实用程序**以文本格式和 XML 格式打印目标文件、可执行文件和/或归档库的内容。
- **反汇编器**以目标文件和可执行文件作为输入并生成汇编列表作为输出。此列表展示了汇编指令、对应的操作码以及段程序计时器值。
- **名称使用程序**打印目标文件、可执行文件和/或归档库中定义和引用的名称列表。
- **符号去除实用程序**从目标文件和可执行文件中删除符号表和调试信息。

13.1 调用目标文件显示实用程序

目标文件显示实用程序 *ofd7x* 以文本和 XML 格式显示目标文件（.obj）、可执行文件（.out）和/或归档库（.lib）的内容。隐藏的符号列为*无名称*，而局部化的符号像任何其他局部符号一样列出。

若要调用目标文件显示实用程序，请输入以下命令：

ofd7x [*options*] *input filename* [*input filename*]

ofd7x 是调用目标文件显示实用程序的命令。

input filename 指定目标文件 (.obj)、可执行文件 (.out) 或归档库 (.lib) 源文件的名称。文件名必须包含扩展名。

options 标识要使用的目标文件显示实用程序选项。选项不区分大小写，可出现在命令行中的命令之后的任意位置。在每个选项前面加一个连字符。

--call_graph 以 XML 格式显示函数栈使用信息和被调用函数信息。虽然开发人员可以访问 XML 输出，但这个选项主要是为 Code Composer Studio 等工具而设计，用来显示应用程序在最坏情况下对堆栈的使用。

--dwarf_display=*attributes* 通过指定以逗号分隔的*属性*列表来控制 DWARF 显示过滤器设置。给属性加上前缀 no disables 而不是 enables。例如：

```
--dwarf_display=nodabbrev,nodline
--dwarf_display=all,nodabbrev
--dwarf_display=none,dinfo,types
```

属性的顺序很重要（请参阅 --obj_display）。可通过调用 ofd7x --dwarf_display=help 获取可用显示属性列表。

--dwarf 将 DWARF 调试信息附加到程序输出中。

--help 显示帮助。

--output=*filename* 将程序输出发送至名为 *filename* 的文件，而不是显示在屏幕上。

--obj_display *attributes* 通过指定以逗号分隔的*属性*列表来控制目标文件显示过滤器设置。给属性加上前缀 no disables 而不是 enables。例如：

```
--obj_display=rawdata,nostrings
--obj_display=all,norawdata
--obj_display=none,header
```

属性的顺序很重要。例如，在“--obj_display=none,header”中，ofd7x 禁用所有输出，然后重新启用文件头信息。如果以相反的顺序指定属性 (header,none)，则启用文件头，禁用所有输出，包括文件头。因此，屏幕上不会显示给定文件的任何内容。可通过调用 ofd7x --obj_display=help 获取可用显示属性列表。

--verbose 显示详细的文本输出。

--xml 以 XML 格式显示输出。

--xml_indent=*num* 设置嵌套 XML 标签的空格数。

如果将一个归档文件作为目标文件显示实用程序的输入，则就像在命令行上进行传递一样来处理归档的每个目标文件成员。按照目标文件成员在归档文件中出现的顺序来处理它们。

如果在不带任何选项的情况下调用目标文件显示实用程序，则在控制台屏幕上显示有关输入文件内容的信息。

备注

目标文件显示格式：目标文件显示实用程序默认以文本格式生成数据。此数据不打算用作进一步处理这些信息的程序的输入。应使用 XML 格式进行机械处理。

13.2 调用反汇编器

反汇编器 *dis7x* 用于检查汇编器或链接器的输出。这个实用程序接受目标文件或可执行文件作为输入，并将反汇编的目标代码写入标准输出或指定文件。

若要调用反汇编器，请输入以下命令：

dis7x [*options*] *input filename*[.] [*output filename*]

dis7x 是用于调用反汇编器的命令。

options 标识要使用的名称实用程序选项。选项不区分大小写，可出现在命令行中的调用之后的任意位置。在每个选项前面加一个连字符 (-)。命名实用程序选项如下所示：

--all (-1)	反汇编所有段，处理 .cinit 段
--noaddr (-a)	禁止列印分支目标地址和标签。
--bytes (-b)	将数据显示为字节而非字。
-c	转储目标文件信息。
--nodata (-d)	禁止显示数据段。
--hex (-e)	以十六进制显示整数值。
--help (-h)	显示当前帮助屏幕。
--data_as_text (-i)	将 .data 段反汇编为指令。
--text_as_data (-l)	将数据段反汇编为文本。
--loadtime_addr (-L)	如果加载地址和运行地址不同，则两者都显示。
--single_opcode (-o) ##	反汇编单个字 **##** 或 0x**##**，然后退出。
--quiet (-q)	(静默模式) 不显示横幅和所有进度信息。
--realquiet (-qq)	(超级静默模式) 不显示所有头文件。
--suppress (-s)	不列印地址和数据字。
--notext (-t)	不在列表文件中显示文本段。
--silicon_version (-v)	显示目标的系列。
--copy_tables (-y)	显示复制表和已复制的段。首先转储表信息，然后转储每个记录及其加载和运行数据。

input filename[*.ext*] 这是输入文件的名称。如果未指定可选扩展名，则按以下顺序搜索文件：

1. *infile*
2. *infile*.out，一个可执行文件
3. *infile*.obj，一个目标文件

output filename 是将反汇编代码写入其中的可选输出文件的名称。如果未指定输出文件名，则将反汇编代码写入标准输出。

13.3 调用名称实用程序

名称实用程序 *nm7x* 列印目标文件、可执行文件或归档库中定义和引用的名称列表。它还会列印符号值和符号类型的指示。隐藏符号列为 ""。若要调用名称实用程序，请输入以下命令：

nm7x [*-options*] [*input filenames*]

nm7x	是调用名称实用程序的命令。
input filename	是目标文件 (.obj)、可执行文件 (.out) 或归档库 (.lib)。
options	标识要使用的名称实用程序选项。选项不区分大小写，可出现在命令行中的调用之后的任意位置。在每个选项前面加连字符 (-)。名称实用程序选项如下：

--all (-a)	列印所有符号。
--prep_fname (-f)	在每个符号前面加上文件名。
--global (-g)	仅列印全局符号。
--help (-h)	展示了当前帮助屏幕。
--format:long (-l)	生成符号信息的详细列表。
--sort:value (-n)	按数字而不是按字母顺序对符号进行排序。
--output (-o) *file*	输出到给定文件。
--sort:none (-p)	使名称实用程序不对任何符号进行排序。
--quiet (-q)	（静默模式）不显示横幅和所有进度信息。
--sort:reverse (-r)	按相反顺序对符号进行排序。
--dynamic (-s)	列出 ELF 目标模块的动态符号表中的符号。
--undefined (-u)	仅列印未定义的符号。

13.4 调用符号去除实用程序

符号去除实用程序 *strip7x* 可从目标文件和可执行文件中删除符号表和调试信息。若要调用符号去除实用程序，请输入以下命令：

strip7x [**-p**] *input filename* [*input filename*]

strip7x	是调用符号去除实用程序的命令。
input filename	是目标文件 (.obj) 或可执行文件 (.out)。
options	标识要使用的符号去除实用程序选项。选项不区分大小写，可出现在命令行中的调用之后的任意位置。在每个选项前面加连字符 (-)。符号去除实用程序选项如下：

--help (-h)	显示帮助信息。
--outfile (-o) *filename*	将去除符号的输出写入文件名。
--postlink (-p)	删除执行不需要的所有信息。与默认行为相比，此选项会删除更多的信息，但目标文件会处于无法链接的状态。此选项应只与静态可执行文件一同使用。
--rom	去除只读段和程序段的符号。

在不带 -o 选项的情况下调用符号去除实用程序时，输入目标文件替换为已去除符号的版本。

章节 14

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。

14.1 调用 C++ 名称还原器

调用 C++ 名称还原器的语法如下：

dem7x [*options*] [*filenames*]

dem7x	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 dem7x 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。

14.2 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();
}
```

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：

```
dem7x 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()||:
;** --------------------------------------------------------------------------*
           MVC     .S1     RP,A9             ; [A_S1]
||         STD     .D1     A8,*SP(8)         ; [A_D1]
||         STD     .D2X    A9,*SP++(-24)     ; [A_D2]
           CALL    .B1     ||banana::banana()|| ; [A_B] |9|
||         ADDD    .D1     SP,0x10,A4        ; [A_D1] |9|
$C$RL0:    ; CALL OCCURS (||banana::banana()||) arg:{A4} ret:{}  ; [] |9|
           CALL    .B1     ||banana::calories()|| ; [A_B] |10|
||         ADDD    .D1     SP,0x10,A4        ; [A_D1] |10|
$C$RL1:    ; CALL OCCURS (||banana::calories()||) arg:{A4} ret:{A4}  ; [] |10|
           CALL    .B1     ||banana::~banana()|| ; [A_B] |10|
||         ADDD    .D1     SP,0x10,A4        ; [A_D1] |10|
||         MV      .D2     A4,A8             ; [A_D2] |10|
$C$RL2:    ; CALL OCCURS (||banana::~banana()||) arg:{A4} ret:{}  ; [] |10|
           MV      .D1     A8,A4             ; [A_D1] |10|
           MVC     .S1     A9,RP             ; [A_S1] BARRIER
           LDD     .D1     *SP(24),A9        ; [A_D1]
||         LDD     .D2     *SP(32),A8        ; [A_D2]
           RET     .B1     ; [A_B]
||         ADDD    .D1     SP,0x18,SP        ; [A_D1]
           ; RETURN OCCURS {RP}              ; []
```

This page intentionally left blank.

附录 A

XML 链接信息文件说明

C7000 链接器支持通过 --xml_link_info *file* 选项生成 XML 链接信息文件。此选项会使链接器生成一个格式良好的 XML 文件，其中包含有关链接结果的详细信息。这个文件中的信息包括由链接器生成的映射文件中当前生成的所有信息。

随着链接器的发展，XML 链接信息文件可扩展到包含其他有用信息，以对链接器结果进行静态分析。

本附录列举了链接器在 XML 链接信息文件中生成的所有元素。

A.1 XML 信息文件元素类型

链接器将生成以下元素类型：

- **容器元素**表示某对象包含描述该对象的其他元素。容器元素的 id 属性使其他元素可以访问它们。
- **字符串元素**包含其值的字符串表示。
- **常量元素**包含其值的 64 位 无符号长整型表示（0x 为前缀）。
- **引用元素**是包含 idref 属性的空元素，指定与另一容器元素的链接。

在节 A.2 中，在每个元素的元素说明之后，在括号中指定元素类型。例如，<link_time> 元素列出了执行链接的时间（字符串）。

A.2 文档元素

根元素或文档元素为 **<link_info>**。XML 链接信息文件中包含的所有其他元素都是 <link_info> 元素的子元素。以下各节描述了 XML 信息文件可以包含的元素。

A.2.1 标头元素

XML 链接信息文件中的第一个元素提供有关链接器和链接会话的一般信息：

- **<banner>** 元素列出了可执行文件的名称和版本信息（字符串）。
- **<copyright>** 元素列出了 TI 版权信息（字符串）。
- **<link_time>** 是链接时间的时间戳表示（无符号 32 位整数）。
- **<output_file>** 元素列出了生成的链接输出文件的绝对路径和名称（字符串）。
- **<entry_point>** 元素指定程序入口点，由链接器（容器）通过两个条目确定：
  - **<name>** 是入口点符号名称（字符串）（如有）。
  - **<address>** 是入口点地址（常量）。

hi.out 输出文件的标头元素

```
<banner>TMS320Cxx Linker           Version x.xx (Jan 6 2008)</banner>
<copyright>Copyright (c) 1996-2008 Texas Instruments Incorporated</copyright>
<link_time>0x43dfd8a4</link_time>
<output_file>/usr/mycode/hi.out</output_file>
<entry_point>
   <name>_c_int00</name>
   <address>0xaf80</address>
</entry_point>
```

A.2.2 输入文件列表

XML 链接信息文件的下一段是输入文件列表，用 **<input_file_list>** 容器元素分隔。<input_file_list> 可以包含任意数量的 <input_file> 元素。

每个 **<input_file>** 实例指定链接中涉及的输入文件。每个 <input_file> 都有一个 ID 属性，可以直接由其他元素（例如 <object_component>）引用。<input_file> 是包含以下元素的容器元素：

- **<path>** 元素在适用时指定目录路径的绝对名称（字符串）。
- **<kind>** 元素指定文件类型，可以是存档或目标（字符串）。
- **<file>** 元素指定存档名称或文件名（字符串）。
- **<name>** 元素指定目标文件名或存档成员名（字符串）。

hi.out 输出文件的输入文件列表

```
<input_file_list>
   <input_file id="fl-1">
      <kind>object</kind>
      <file>hi.obj</file>
      <name>hi.obj</name>
   </input_file>
   <input_file id="fl-2">
      <path>/usr/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>boot.obj</name>
   </input_file>
   <input_file id="fl-3">
      <path>/usr/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>exit.obj</name>
   </input_file>
   <input_file id="fl-4">
      <path>/usr/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>printf.obj</name>
   </input_file>
...
</input_file_list>
```

A.2.3 对象组件列表

XML 链接信息文件的下一个段包含链接中涉及的所有对象组件的规格。对象组件的一个示例是输入段。通常，对象组件是链接器可以操作的最小对象元素。

<object_component_list> 是一个包含任意数量的 <object_component> 元素的容器元素。

每个 **<object_component>** 指定一个对象组件。每个 <object_component> 都有一个 ID 属性，可以由其他元素（例如 <logical_group>）直接引用。<object_component> 是包含以下元素的容器元素：

- **<name>** 元素指定对象组件（字符串）。
- **<load_address>** 元素指定对象组件的加载时地址（常量）。
- **<run_address>** 元素指定对象组件的运行时地址（常量）。
- **<alignment>** 元素指定对象组件 (unsigned int) 的对齐方式。
- **<size>** 元素指定对象组件的大小（常量）。
- **<executable>** 元素指定目标组件是否允许执行访问（字符串）。如果对象具有可执行访问权限，则不能同时具有读写访问权限。虽然“false”为有效值，但仅当该元素对此对象组件为“true”时，链接器才会发出该元素。
- **<readonly>** 元素指定对象组件是否允许只读访问（字符串）。如果对象具有只读访问权限，则不能同时具有读写访问权限。虽然“false”为有效值，但仅当该元素对此对象组件为“true”时，链接器才会发出该元素。
- **<readwrite>** 元素指定对象组件是否允许读写访问（字符串）。如果对象具有读写访问权限，则不能同时具有只读访问权限或可执行访问权限。虽然“false”为有效值，但仅当该元素对此对象组件为“true”时，链接器才会发出该元素。
- **<uninitialized>** 元素指定是否已初始化目标组件（字符串）。虽然“false”为有效值，但仅当该元素对此对象组件为“true”时，链接器才会发出该元素。
- **<input_file_ref>** 元素指定对象组件起源的源文件（引用）。

fl-4 输入文件的对象组件列表

```
<object_component id="oc-20">
   <name>.text</name>
   <load_address>0xac00</load_address>
   <run_address>0xac00</run_address>
   <alignment>0x1</alignment>
   <size>0xc0</size>
   <readonly>true</readonly>
   <executable>false</executable>
   <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-21">
   <name>.data</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <alignment>0x1</alignment>
   <size>0x0</size>
   <readwrite>true</readwrite>
   <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-22">
   <name>.bss</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <alignment>0x1</alignment>
   <size>0x0</size>
   <readwrite>true</readwrite>
   <uninitialized>true</uninitialized>
   <input_file_ref idref="fl-4"/>
</object_component>
```

A.2.4 逻辑组列表

XML 链接信息文件的 **<logical_group_list>** 段类似于由链接器生成的映射文件中的输出段。但是，XML 链接信息文件包含 GROUP 和 UNION 输出段的规范，而映射文件中没有规范。<logical_group_list> 中会出现三种类型的列表项：

- **<logical_group>** 是包含目标组件或逻辑组成员列表的段或组的规范。每个 <logical_group> 元素提供一个 id，以便可以从其他元素引用它。每个 <logical_group> 是围住以下元素的容器元素：
  - **<name>** 元素指定逻辑组的名称（字符串）。
  - **<load_address>** 元素指定逻辑组的加载时地址（常量）。
  - **<run_address>** 元素指定逻辑组的运行时地址（常量）。
  - **<size>** 元素指定逻辑组的大小（常量）。
  - **<output_section_group>** 指定逻辑组是否为输出段组（字符串）。虽然“false”为有效值，但仅当该元素对此逻辑组为“true”时，链接器才会发出该元素。
  - **<contents>** 元素列出此逻辑组中包含的元素（容器）。这些元素引用此逻辑组中包含的每个成员对象：
    - **<object_component_ref>** 是此逻辑组中包含的目标组件（引用）。
    - **<logical_group_ref>** 是此逻辑组中包含的逻辑组（引用）。
- **<overlay>** 是特殊类型的逻辑组，表示 UNION 或共享相同存储器空间的一组对象（容器）。为每个 <overlay> 元素提供一个 id，以便可以从其他元素引用它（例如，从放置映射中的 <allocated_space> 元素引用）。每个 <overlay> 包含以下元素：
  - **<name>** 元素指定覆盖的名称（字符串）。
  - **<run_address>** 元素指定覆盖的运行时地址（常量）。
  - **<size>** 元素指定逻辑组的大小（常量）。
  - **<contents>** 容器元素列出此覆盖中包含的元素。这些元素引用此逻辑组中包含的每个成员对象：
    - **<object_component_ref>** 是此逻辑组中包含的目标组件（引用）。
    - **<logical_group_ref>** 是此逻辑组中包含的逻辑组（引用）。
- **<split_section>** 是另一个特殊类型的逻辑组，表示分拆到多个存储器区域的逻辑组集合。每个 <split_section> 元素提供一个 id，以便可以从其他元素引用它。id 由以下元素组成。
  - **<name>** 元素指定分拆段的名称（字符串）。
  - **<contents>** 容器元素列出此分拆段中包含的元素。<logical_group_ref> 元素引用此分拆段中包含的每个成员对象，引用的每个元素是此分拆段中包含的逻辑组（引用）。

fl-4 输入文件的逻辑组列表

```
<logical_group_list>
   ...
   <logical_group id="lg-7">
      <name>.text</name>
      <output_section_group>true</output_section_group>
      <load_address>0x20</load_address>
      <run_address>0x20</run_address>
      <size>0xb240</size>
      <contents>
         <object_component_ref idref="oc-34"/>
         <object_component_ref idref="oc-108"/>
         <object_component_ref idref="oc-e2"/>
         ...
      </contents>
   </logical_group>
   ...
   <overlay id="lg-b">
      <name>UNION_1</name>
      <run_address>0xb600</run_address>
      <size>0xc0</size>
      <contents>
         <object_component_ref idref="oc-45"/>
         <logical_group_ref idref="lg-8"/>
      </contents>
   </overlay>
   ...
   <split_section id="lg-12">
      <name>.task_scn</name>
      <size>0x120</size>
      <contents>
         <logical_group_ref idref="lg-10"/>
         <logical_group_ref idref="lg-11"/>
      </contents>
   </split_section>
   ...
</logical_group_list>
```

A.2.5 放置映射

<placement_map> 元素用于描述应用中所有指定存储器区域的存储器放置详细信息，包括特定存储器区域中已有逻辑组之间的未用空间。

<memory_area> 用于描述指定存储器区域（容器）内的放置详细信息。该描述包括以下项目：

- **<name>** 指定存储器区域的名称（字符串）。
- **<page_id>** 提供了用于定义此存储器区域的存储页面对应的 ID（常量）。
- **<origin>** 指定存储器区域的起始地址（常量）。
- **<length>** 指定存储器区域的长度（常量）。
- **<used_space>** 指定此区域中已分配的空间量（常量）。
- **<unused_space>** 指定此区域中的可用空间量（常量）。
- **<attributes>** 列出与此区域关联的 RWXI 属性（若有）（字符串）。
- **<fill_value>** 指定要在未用空间中填写的填充值，在为存储器区域指定了 fill 指令时使用（常量）。
- **<usage_details>** 会列出此存储器区域中每个已分配或可用片段的详细信息。如果片段已分配给逻辑组，则会提供 <logical_group_ref> 元素来协助访问该逻辑组的详细信息。所有片段规范均包含 <start_address> 和 <size> 元素。
  - **<allocated_space>** 元素提供此存储器区域内某个已分配片段的详细信息（容器）：
    - **<start_address>** 指定片段的地址（常量）。
    - **<size>** 指定片段的大小（常量）。
    - **<logical_group_ref>** 提供对已分配至此片段的逻辑组的引用（引用）。
  - **<Available_space** 元素提供此存储器区域内某个可用片段的详细信息（容器）：
    - **<start_address>** 指定片段的地址（常量）。
    - **<size>** 指定片段的大小（常量）。

fl-4 输入文件的放置映射

```
<placement_map> <memory_area> <name>PMEM</name> <page_id>0x0</page_id> <origin>0x20</origin>
<length>0x100000</length> <used_space>0xb240</used_space> <unused_space>0xf4dc0</unused_space>
<attributes>RWXI</attributes> <usage_details> <allocated_space> <start_address>0x20</start_address>
<size>0xb240</size> <logical_group_ref idref="lg-7"/> </allocated_space> <available_space>
<start_address>0xb260</start_address> <size>0xf4dc0</size> </available_space> </usage_details> </
memory_area> ... </placement_map>
```

A.2.6 Far Call Trampoline 列表

<far_call_trampoline_list> 是 <far_call_trampoline> 元素的列表。链接器支持生成 far call trampoline，以帮助调用站点到达超出范围的目的地。far call trampoline 函数可以保证到达被调用函数（被调用方），因为它可以利用对被调用函数的间接调用。

<far_call_trampoline_list> 枚举由链接器为某一特定链路生成的所有 far call trampoline。
<far_call_trampoline_list> 可以包含任意数量的 <far_call_trampoline> 元素。每个 **<far_call_trampoline>** 都是一个包含以下元素的容器：

- **<callee_name>** 元素命名目标函数（字符串）。
- **<callee_address>** 是被调用函数的地址（常量）。
- **<trampoline_object_component_ref>** 是对包含 trampoline 函数定义的对象组件的引用（引用）。
- **<trampoline_address>** 是 trampoline 函数的地址（常量）。
- **<caller_list>** 枚举利用此 trampoline 到达被调用函数的所有调用站点（容器）。
- **<trampoline_call_site>** 提供 trampoline 调用站点的详细信息（容器），包括以下各项：
  - **<caller_address>** 指定调用站点地址（常量）。
  - **<caller_object_component_ref>** 是调用站点所在的对象组件（引用）。

fl-4 输入文件的 Fall Call Trampoline 列表

```
<far_call_trampoline_list> ... <far_call_trampoline> <callee_name>_foo</callee_name>
<callee_address>0x08000030</callee_address> <trampoline_object_component_ref idref="oc-123"/>
<trampoline_address>0x2020</trampoline_address> <caller_list> <call_site> <caller_address>0x1800</
caller_address> <caller_object_component_ref idref="oc-23"/> </call_site> <call_site>
<caller_address>0x1810</caller_address> <caller_object_component_ref idref="oc-23"/> </call_site> </
caller_list> </far_call_trampoline> ... </far_call_trampoline_list>
```

A.2.7 符号表

<symbol_table> 包含链接中所含所有全局符号的列表。该列表提供有关符号的名称和值的信息。将来，symbol_table 列表可提供类型信息、定义符号的对象组件、存储类等信息。

<symbol> 是一个容器元素，用以下元素指定符号的名称和值：

- **<name>** 元素指定符号名称（字符串）。
- **<value>** 元素指定符号值（常量）。
- **<local>** 元素指定符号是否具有局部绑定（字符串）。虽然“false”为有效值，但仅当此符号的值为“true”时，链接器才会发出该元素。

fl-4 输入文件的符号表

```
<symbol_table>
   <symbol>
      <name>_c_int00</name>
      <value>0xaf80</value>
   </symbol>
   <symbol>
      <name>_main</name>
      <value>0xb1e0</value>
   </symbol>
   <symbol>
      <name>_printf</name>
      <value>0xac00</value>
      <local>true</local>
   </symbol>
   ...
</symbol_table>
```

This page intentionally left blank.

B.1 不受支持的工具和功能列表

C7000 编译器工具集不支持以下工具和功能：

- 编译器咨询
- 业务定向编译
- CCS Optimizer Assistant
- 缓存布局工具
- Hex Utility（建议改用第三方工具，例如 GNU“objcopy”或 ARM Ltd. Tools。）
- 交叉参考工具
- Absolute Lister
- C6x 线性汇编
- C7x 线性汇编
- C6x（传统）汇编兼容性
- MISRA-C:2004 和 MISRA-C:2012 校验（建议使用第三方工具，例如 LDRA）

This page intentionally left blank.

附录 C
术语表

别名消歧	一种决定两个指针表达式何时不能指向同一位置的技术，从而允许编译器自由地优化此类表达式。
别名使用	以多种方式访问单个对象的能力，例如当两个指针指向单个对象时。其会破坏优化，这是因为任何间接引用都可能引用任何其它对象。
分配	链接器计算输出段最终存储器地址的过程。
ANSI	美国国家标准协会；一个建立行业自愿遵循的标准的组织。
应用程序二进制接口 (ABI)	一项指定两个目标模块之间接口的标准。ABI 规定了如何调用函数以及如何将信息从一个程序组件传递到另一个程序组件。
存档库	由归档器将单独文件组合成单个文件的集合。
归档器	将多个单独文件集合成一个单个文件（称为存档库）的软件程序。借助归档器，可以添加、删除、提取或替换存档库的成员。
汇编器	根据包含汇编语言指令、指示和宏定义的源文件创建机器语言程序的软件程序。汇编器将绝对操作码替换为符号操作码，并将绝对地址或可重定位地址替换为符号地址。
赋值语句	用值来初始化变量的语句。
自动初始化	在程序开始执行之前，初始化全局 C 变量（包含在 .cinit 段中）的过程。
运行时的自动初始化	链接器在链接 C 代码时使用的自动初始化方法。在使用 --rom_model 链接选项调用链接器时，链接器会使用此方法。链接器将数据表的 .cinit 段加载到内存中，并在运行时初始化变量。
大端字节序	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 *小端字节序*
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。
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	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	复制器件的 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。
迭代计数	循环结束前执行的次数。以前称为“循环计数”。
内核	流水线循环逻辑程序和流水线循环收尾程序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。
小端字节序	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。
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。
逻辑程序	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 ^。
未配置的内存	未定义为存储器映射的一部分，且无法加载代码或数据的存储器。
未初始化段	在存储器映射中保留空间但没有实际内容的目标文件段。
无符号值	无论实际符号如何都会被当作非负数的值。
变量	表示可以假设一组值中的任何一个数的符号。
字	目标内存中的 32 位可寻址位置。

Changes from DECEMBER 15, 2023 to MARCH 15, 2024 (from Revision I (December 2023) to Revision J (March 2024)) **Page**

Changes from FEBRUARY 28, 2023 to DECEMBER 15, 2023 (from Revision H (February 2023) to Revision I (December 2023)) **Page**

- 添加了如下表述：--auto_stream、--assume_addresses_ok_for_stream 和 --diag_suppress 选项可与 FUNCTION_OPTIONS pragma 一起使用 113
- 添加了有关向量的部分元素谓词的注释 140

Changes from AUGUST 5, 2022 to FEBRUARY 28, 2023 (from Revision G (August 2022) to Revision H (February 2023)) **Page**

- 添加了 --silicon_errata 命令行选项 27
- 添加了 --silicon_errata 命令行选项 36
- 记录了 ptrdiff_t、size_t、wchar_t 和 C7000 矢量大小的预定义宏。 41
- 添加了流引擎的说明 75
- 添加了流地址生成器的说明。 76
- 介绍了使用流引擎和流地址生成器的优点。 76
- 记录了用于 MMA 和可扩展矢量编程的宏 143
- 阐明了链接器定义的符号的使用，包括何时以及如何使用 _symval() 运算符 272

Changes from OCTOBER 22, 2021 to AUGUST 5, 2022 (from Revision F (October 2021) to Revision G (August 2022)) **Page**

- --strict_compatibility 链接器选项不再起任何作用且已从文档中删除 33
- 删除了不受支持的 --rom 链接器选项的文档 33
- 添加了 --fp_single_precision_constant 编译器选项 36
- 为 C7504 添加了 --mma_version=2_256 设置 36
- 新增了 --silicon_version=7504 命令行选项。 38
- 为 C7504 器件添加了预定义的宏名称。 41
- 向 --opt_level 优化列表添加了指向其他信息的链接。 56
- 向某些优化说明添加了有关相应 --opt_level 的信息 71
- 本机向量类型现在称为“TI 向量类型”，并且添加了布尔向量类型。 94
- 修订了与 OpenCL 一致性相关的信息。 94
- 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。 135
- 为转换运算符提供了访问器。更新了相关示例以反映此更改。 137
- 添加了布尔向量的重新解释示例。 139
- 添加了布尔向量的信息和示例 140
- 添加了有关可扩展矢量编程的部分 143
- --strict_compatibility 链接器选项不再起任何作用，已从文档中删除。 222
- 删除了不受支持的 --rom 链接器选项的文档。 222
- 记录了 --absolute_exe 和 --relocatable 选项可能无法一同使用。 224
- 为 C7504 器件添加了预定义的宏名称。 226
- 更正链接器的头文件搜索路径的说明。 226

Changes from MARCH 19, 2021 to OCTOBER 22, 2021 (from Revision E (March 2021) to Revision F (October 2021)) **Page**

- 添加了 --mma_version 命令行选项 27
- 添加了 --mma_version 命令行选项 36
- 新增了 --silicon_version=7120 命令行选项。 38
- 为 C7120 器件添加了预定义的宏名称。 41

- 使用不同的 --silicon_version 或 --mma_version 编译的文件不支持链接时优化。 61
- 指定可以将 C7120 和更高版本的 ISA 配置为允许断言的 SA 加载 82
- 新增了几个加载操作的示例 83
- SET_DATA_SECTION pragma 优先于 --gen_data_subsections=on 选项。 121
- 记录了所有 C7000 代码和数据都必须位于 2GB 的虚拟地址空间内。 148
- 删除了 RTS 库中不再包含的运行时支持算术函数。 162
- 阐明了有关字符串处理函数的信息。 172
- 添加了关于时间和时钟 RTS 函数的信息 172
- SET_DATA_SECTION pragma 优先于 --gen_data_subsections=on 选项。 214
- 为 C7120 器件添加了预定义的宏名称。 226
- 记录了所有 C7000 代码和数据都必须位于 2GB 的虚拟地址空间内。 245

Changes from DECEMBER 15, 2020 to MARCH 19, 2021 (from Revision D (December 2020) to Revision E (March 2021)) **Page**

- 由编译器生成的矢量谓词存储在某些情况下可能会触发页面错误异常。可在链接器命令文件中更正此问题。 289

Changes from FEBRUARY 28, 2020 to DECEMBER 15, 2020 (from Revision C (February 2020) to Revision D (December 2020)) **Page**

- 更新了整个文档中的表格、图和交叉参考的编号格式。 13
- 删除了整个文档中对处理器 wiki 的引用 13
- 阐明了 --opt_level=4 必须位于 --run_linker 选项之前 61
- 为流地址生成器添加了断言的加载，并添加了一些示例，这些示例根据 SA 配置导致存储和加载操作出现定义明确的行为与不明确的行为 83
- 有文件证明不支持 C11 原子操作。 88
- 更新了有关枚举类型大小的信息。 94
- 阐明 --opt_level 和 FUNCTION_OPTIONS pragma 之间的交互。 113
- 为与 MUST_ITERATE pragma 对应的属性新增了 C++ 属性语法 115
- 添加了与 UNROLL pragma 对应的各个属性的 C++ 属性语法。 122
- 有文件证明不支持 C11 原子操作。 127
- 增加了使用位置属性的示例。 132
- 添加了有关 --gen_func_subsections 选项的默认值的信息 214
- 更正了有关 --gen_data_subsections 选项的默认值的信息。 214

下表列出了代码生成工具 v7.4 (SPRU187) 与 SPRUI03C (文档生成系统发生更改后) 之间文档的变化。左列标识了本文档出现该特定改动的版本。

早期修订版本

添加内容的版本	章节	位置	添加/修改/删除
SPRUIG8C	C/C++ 语言	节 5.3.2、节 5.14	添加了矢量不能通过 stdarg 传递也不能传递给 printf() 的注释。
SPRUIG8C	C/C++ 语言	节 5.8.23、节 5.8.30、节 5.8.34、节 5.13.3	C++ 属性语法对应于 MUST_ITERATE、PROB_ITERATE 和 UNROLL pragma。
SPRUIG8C	C/C++ 语言	节 5.8.28	`#pragma once` 现记录在头文件中使用。
SPRUIG8C	运行时环境，链接器	节 6.7.2.1、节 12.4.33	阐明了只有在使用 --rom_model 链接器选项时，才发生零初始化，使用 --ram_model 选项时则不发生。

早期修订版本（续）

添加内容的版本	章节	位置	添加/修改/删除
SPRUIG8C	程序加载	节 9.3.2.3	更正了有关 RAM 和 ROM 模型使用 CINIT 的信息。
SPRUIG8C	链接， 链接器	节 11.3.4、节 12.4.24	阐明了如果只有链接器在运行，则需要 --rom_model 或 --ram_model，但如果编译器在同一命令行中的 C/C++ 文件上运行，则 --rom_model 是默认选项。
SPRUIG8C	链接器	节 12.5.4.2、节 12.5.9.7 和节 12.5.9.8	添加了使用相关存储器区域中上一个已分配字节的运行时地址来定义符号的 LAST 操作符。
SPRUIG8B	优化	节 4.15	更新了流引擎和流地址生成器的类型。
SPRUIG8B	语言	节 5.2	目前不支持 MISRA C 2004 检查。
SPRUIG8B	语言	节 5.14.6	目前不支持矢量数据类型的转换修饰符。
SPRUIG8A	简介， 语言	节 1.3、 节 5.1 和 节 5.2	添加了对 C11 和 C++14 的支持。
SPRUIG8A	优化	节 4.15	添加了有关流引擎和流地址生成器的矢量长度和元素大小的信息。

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