ARM 优化 C/C++ 编译器 v20.2.0.LTS

User’s Guide

Literature Number: ZHCUAV8W
JANUARY 1998 – REVISED MARCH 2023

目录

请先阅读 9
关于本手册 9
标记规则 9
相关文档 10
德州仪器 (TI) 提供的相关文档 10
商标 11
1 软件开发工具简介 13
1.1 软件开发工具概述 14
1.2 编译器接口 15
1.3 ANSI/ISO 标准 15
1.4 输出文件 15
1.5 实用程序 16
2 使用 C/C++ 编译器 17
2.1 关于编译器 18
2.2 调用 C/C++ 编译器 18
2.3 使用选项更改编译器的行为 19
2.3.1 链接器选项 25
2.3.2 常用选项 27
2.3.3 其他有用的选项 27
2.3.4 运行时模型选项 28
2.3.5 符号调试和分析选项 31
2.3.6 指定文件名 31
2.3.7 更改编译器解释文件名的方式 32
2.3.8 更改编译器处理 C 文件的方式 32
2.3.9 更改编译器解释和命名扩展名的方式 32
2.3.10 指定目录 33
2.3.11 汇编器选项 33
2.3.12 已弃用的选项 33
2.4 通过环境变量控制编译器 34
2.4.1 设置默认编译器选项 (TI_ARM_C_OPTION) 34
2.4.2 命名一个或多个备用目录 (TI_ARM_C_DIR) 34
2.5 控制预处理器 35
2.5.1 预先定义的宏名称 35
2.5.2 #include 文件的搜索路径 38
2.5.3 支持#warning 和 #warn 指令 40
2.5.4 生成预处理列表文件 (--preproc_only 选项) 40
2.5.5 预处理后继续编译 (--preproc_with_compile 选项) 41
2.5.6 生成带有注释的预处理列表文件 (--preproc_with_comment 选项) 41
2.5.7 生成带有行控制详细信息的预处理列表 (--preproc_with_line 选项) 41
2.5.8 为 Make 实用程序生成预处理输出 (--preproc_dependency 选项) 41
2.5.9 生成包含#include 在内的文件列表 (--preproc_includes 选项) 41
2.5.10 在文件中生成宏列表 (--preproc_macros 选项) 41
2.6 将参数传递给 main() 41
2.7 了解诊断消息 42
2.7.1 控制诊断消息 43
2.7.2 如何使用诊断抑制选项 44

2.8 其他消息 44
2.9 生成交叉参考列表信息（--gen_cross_reference_listing 选项） 44
2.10 生成原始列表文件（--gen_preprocessor_listing 选项） 46
2.11 使用内联函数扩展 46
2.11.1 内联内在函数运算符 47
2.11.2 内联限制 47
2.12 使用交叉列出功能 49
2.13 控制应用程序二进制接口 49
2.14 VFP 支持 50
2.15 启用入口挂钩和出口挂钩函数 51
3 优化您的代码 53
3.1 调用优化 54
3.2 控制代码大小与速度 55
3.3 执行文件级优化（--opt_level=3 选项） 55
3.3.1 创建优化信息文件（--gen_opt_info 选项） 55
3.4 程序级优化（--program_level_compile 和 --opt_level=3 选项） 56
3.4.1 控制程序级优化（--call_assumptions 选项） 56
3.4.2 混合 C/C++ 和汇编代码时的优化注意事项 57
3.5 自动内联扩展（--auto_inline 选项） 58
3.6 链接时优化（--opt_level=4 选项） 58
3.6.1 选项处理 59
3.6.2 不兼容的类型 59
3.7 使用反馈制导优化 59
3.7.1 反馈向导优化 59
3.7.2 分析数据解码器 61
3.7.3 反馈制导优化 API 61
3.7.4 反馈制导优化总结 61
3.8 使用配置文件信息分析代码覆盖率 62
3.8.1 代码覆盖 62
3.8.2 相关的特征和功能 63
3.9 访问优化代码中的别名变量 64
3.10 在优化代码中谨慎使用 asm 语句 64
3.11 通过优化使用交叉列出特性 64
3.12 调试和分析优化代码 65
3.12.1 分析优化的代码 65
3.13 正在执行什么类型的优化？ 66
3.13.1 基于成本的寄存器分配 66
3.13.2 别名消歧 66
3.13.3 分支优化和控制流简化 66
3.13.4 数据流优化 67
3.13.5 表达式简化 67
3.13.6 函数的内联扩展 67
3.13.7 函数符号别名 67
3.13.8 归纳变量和强度降低 67
3.13.9 循环不变量代码运动 68
3.13.10 循环旋转 68
3.13.11 指令排程 68
3.13.12 尾部合并 68
3.13.13 自动增量寻址 68
3.13.14 块条件化 69
3.13.15 结语内联 69
3.13.16 删除与零的比较 69
3.13.17 用常数除数进行整数除法 69
3.13.18 分支链接 70
4 链接 C/C++ 代码 71
4.1 通过编译器调用链接器（-z 选项） 72

4.1.1 单独调用链接器 72
4.1.2 调用链接器作为编译步骤的一部分 73
4.1.3 禁用链接器（--compile_only 编译器选项） 73
4.2 链接器代码优化 74
4.2.1 生成死函数列表（--generate_dead_funcs_list 选项） 74
4.2.2 生成聚合数据子段（--gen_data_subsections 编译器选项） 74
4.3 控制链接过程 74
4.3.1 包含运行时支持库 75
4.3.2 运行时初始化 75
4.3.3 Cinit 的初始化和看门狗计时器保持 76
4.3.4 全局对象构造函数 76
4.3.5 指定全局变量初始化类型 76
4.3.6 指定在内存中分配段的位置 76
4.3.7 链接器命令文件示例 78
5 C/C++ 语言实现 79
5.1 ARM C 的特征 80
5.1.1 实现定义的行为 81
5.2 ARM C++ 的特征 84
5.3 使用 MISRA C 2004 85
5.4 使用 ULP Advisor 86
5.5 数据类型 87
5.5.1 枚举类型大小 88
5.6 文件编码和字符集 89
5.7 关键字 89
5.7.1 const 关键字 89
5.7.2 __interrupt 关键字 90
5.7.3 volatile 关键字 91
5.8 C++ 异常处理 92
5.9 寄存器变量和参数 92
5.9.1 本地寄存器变量和参数 92
5.9.2 全局寄存器变量 93
5.10 __asm 语句 94
5.11 pragma 指令 95
5.11.1 CALLS Pragma 96
5.11.2 CHECK_MISRA Pragma 96
5.11.3 CHECK_ULP Pragma 96
5.11.4 CODE_SECTION Pragma 97
5.11.5 CODE_STATE Pragma 97
5.11.6 DATA_ALIGN Pragma 98
5.11.7 DATA_SECTION Pragma 98
5.11.8 诊断消息 Pragma 98
5.11.9 DUAL_STATE Pragma 99
5.11.10 FORCEINLINE Pragma 100
5.11.11 FORCEINLINE_RECURSIVE Pragma 100
5.11.12 FUNC_ALWAYS_INLINE Pragma 101
5.11.13 FUNC_CANNOT_INLINE Pragma 101
5.11.14 FUNC_EXT_CALLED Pragma 101
5.11.15 FUNCTION_OPTIONS Pragma 102
5.11.16 INTERRUPT Pragma 103
5.11.17 LOCATION Pragma 104
5.11.18 MUST_ITERATE Pragma 104
5.11.19 NOINIT 和 PERSISTENT Pragma 105
5.11.20 NOINLINE Pragma 107
5.11.21 NO_HOOKS Pragma 107
5.11.22 once Pragma 108
5.11.23 pack Pragma 108
5.11.24 PROB_ITERATE Pragma 109
5.11.25 RESET_MISRA Pragma 109

5.11.26 RESET_ULP Pragma....110
5.11.27 RETAIN Pragma....110
5.11.28 SET_CODE_SECTION 和 SET_DATA_SECTION Pragma....111
5.11.29 SWI_ALIAS Pragma....112
5.11.30 TASK Pragma....113
5.11.31 UNROLL Pragma....113
5.11.32 WEAK Pragma....114
5.12 _Pragma 运算符....114
5.13 应用程序二进制接口....115
5.14 ARM 指令内在函数....115
5.15 目标文件符号命名规则 (链接名)....123
5.16 更改 ANSI/ISO C/C++ 语言模式....123
5.16.1 C99 支持 (--c99)....124
5.16.2 C11 支持 (--c11)....124
5.16.3 严格 ANSI 模式和宽松 ANSI 模式 (--strict_ansi 和 --relaxed_ansi)....125
5.17 GNU 、Clang 和 ACLE 语言扩展....126
5.17.1 扩展....126
5.17.2 函数属性....127
5.17.3 For 循环属性....129
5.17.4 变量属性....129
5.17.5 类型属性....130
5.17.6 内置函数....131
5.18 AUTOSAR....132
5.19 编译器限制....132
6 运行时环境....133
6.1 存储器模型....134
6.1.1 段....134
6.1.2 C/C++ 系统堆栈....135
6.1.3 动态存储器分配....135
6.2 对象表示....136
6.2.1 数据类型存储....136
6.2.2 位字段....140
6.2.3 字符串常量....142
6.3 寄存器惯例....143
6.4 函数结构和调用惯例....145
6.4.1 函数如何进行调用....146
6.4.2 被调用函数如何响应....147
6.4.3 C 异常处理程序调用惯例....147
6.4.4 访问参数和局部变量....148
6.5 访问 C 和 C++ 中的链接器符号....148
6.6 将 C 和 C++ 与汇编语言相连....148
6.6.1 使用汇编语言模块与 C/C++ 代码....148
6.6.2 从 C/C++ 访问汇编语言函数....149
6.6.3 从 C/C++ 访问汇编语言变量....149
6.6.4 与汇编源代码共享 C/C++ 头文件....151
6.6.5 使用内联汇编语言....151
6.6.6 修改编译器输出....151
6.7 中断处理....151
6.7.1 在中断期间保存寄存器....151
6.7.2 使用 C/C++ 中断例程....151
6.7.3 使用汇编语言中断例程....152
6.7.4 如何将中断例程映射到中断向量....153
6.7.5 使用软件中断功能....154
6.7.6 其他中断信息....154
6.8 固有运行时支持算术和转换例程....155
6.8.1 CPSR 寄存器和中断内在函数....155
6.9 内置函数....156

6.10 系统初始化 156
6.10.1 用于系统预初始化的引导挂钩函数 156
6.10.2 运行时栈 157
6.10.3 变量的自动初始化 157
6.10.4 初始化表 163
6.11 TIABI 下的双状态交互工作（已弃用） 165
6.11.1 双状态支持级别 165
6.11.2 实现 166
7 使用运行时支持函数并构建库 169
7.1 C 和 C++ 运行时支持库 170
7.1.1 将代码与对象库链接 170
7.1.2 头文件 170
7.1.3 修改库函数 171
7.1.4 支持字符串处理 171
7.1.5 极少支持国际化 171
7.1.6 时间和时钟函数支持 171
7.1.7 允许打开的文件数量 173
7.1.8 源码树中的非标准头文件 173
7.1.9 库命名规则 173
7.2 C I/O 函数 174
7.2.1 高级别 I/O 函数 175
7.2.2 低级 I/O 实现概述 175
7.2.3 器件驱动程序级别 I/O 函数 179
7.2.4 为 C I/O 添加用户定义的器件驱动程序 183
7.2.5 器件前缀 184
7.3 处理可重入性（_register_lock() 和 _register_unlock() 函数） 186
7.4 库构建流程 187
7.4.1 所需的非德州仪器 (TI) 软件 187
7.4.2 使用库构建流程 187
7.4.3 扩展 mklib 189
8 C++ 名称还原器 191
8.1 调用 C++ 名称还原器 192
8.2 C++ 名称还原器的示例用法 193
A 术语表 195
A.1 术语 195
B 修订历史记录 201

插图清单

图 1-1. ARM 软件开发流程 14
图 6-1. Char 和 Short 数据存储格式 136
图 6-2. 32 位数据存储格式 138
图 6-3. 双精度浮点数据存储格式 139
图 6-4. 以大端格式和小端格式打包位字段 141
图 6-5. 在函数调用期间使用堆栈 146
图 6-6. 运行时自动初始化 158
图 6-7. 加载时初始化 162
图 6-8. 构造函数表 162
图 6-9. .cinit 段中初始化记录的格式 163
图 6-10. .pinit 段中初始化记录的格式 164

表格清单

表 2-1. 处理器选项 19
表 2-2. 优化选项(1) 19
表 2-3. 高级优化选项(1) 20
表 2-4. 调试选项 20
表 2-5. Include 选项 20

表 2-6. ULP 顾问选项....20
表 2-7. 控制选项....20
表 2-8. 语言选项....21
表 2-9. 解析器预处理选项....21
表 2-10. 预定义宏选项....21
表 2-11. 诊断消息选项....22
表 2-12. 补充信息选项....22
表 2-13. 运行时模型选项....22
表 2-14. 入口/出口挂钩选项....23
表 2-15. 反馈选项....23
表 2-16. 汇编器选项....23
表 2-17. 文件类型说明符选项....23
表 2-18. 目录说明符选项....24
表 2-19. 默认文件扩展名选项....24
表 2-20. 命令文件选项....24
表 2-21. MISRA-C 2004 选项....24
表 2-22. 链接器基本选项....25
表 2-23. 文件搜索路径选项....25
表 2-24. 命令文件预处理选项....25
表 2-25. 诊断消息选项....25
表 2-26. 链接器输出选项....25
表 2-27. 符号管理选项....26
表 2-28. 运行时环境选项....26
表 2-29. 其他选项....26
表 2-30. 预定义 ARM 宏名称....35
表 2-31. ACLE 预定义宏....37
表 2-32. 原始列表文件标识符....46
表 2-33. 原始列表文件诊断标识符....46
表 3-1. 可与 --opt_level=3 结合使用的选项....55
表 3-2. 为 --gen_opt_info 选项选择一个级别....55
表 3-3. 为 --call_assumptions 选项选择一个级别....56
表 3-4. 使用 --call_assumptions 选项时的特殊注意事项....56
表 4-1. 由编译器创建的初始化段....77
表 4-2. 由编译器创建的未初始化段....77
表 5-1. ARM C/C++ 数据类型....87
表 5-2. 枚举器类型....87
表 5-3. 目标对 ARM 内在函数的支持....115
表 5-4. ARM 编译器内在函数....118
表 5-5. GCC 语言扩展....126
表 6-1. 段和存储器位置摘要....134
表 6-2. 寄存器和内存中的数据表示....136
表 6-3. 惯例对寄存器类型的影响....143
表 6-4. 寄存器的使用....143
表 6-5. VFP 寄存器使用....144
表 6-6. Neon 寄存器使用....145
表 6-7. CPSR 和中断 C/C++ 编译器内在函数....155
表 6-8. 选择双状态支持级别....165
表 7-1. __time32_t 和 __time64_t 之间的区别....172
表 7-2. mklib 程序选项....189

TEXAS INSTRUMENTS

关于本手册

ARM 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。

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：

armcl [*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：

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

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

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

- 有些指令的参数数量可变。例如，.byte 指令。此语法显示为 [, ..., *parameter*]。
- ARM® 16 位指令集被称为 16-BIS。

- ARM 32 位指令集被称为 32-BIS。

相关文档

以下书籍可以作为本用户指南的补充：

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

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

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

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

ISO/IEC 14882-2014, International Standard - Programming Languages - C++ (The 2014 C++ Standard), International Organization for Standardization

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

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

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

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

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

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

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

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

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

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

ARM C Language Extensions (ACLE) specification (ACLE Version ACLE Q2 2017)

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：

SPNU118 ***ARM 汇编语言工具用户指南。*** 介绍了用于 ARM 器件的汇编语言工具（汇编器、链接器以及其他用于开发汇编语言代码的工具）、汇编器指令、宏命令、通用目标文件格式和符号调试指令。

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

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

商标

This page intentionally left blank.

章节 1

软件开发工具简介

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

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

1.1 软件开发工具概述 14
1.2 编译器接口 15
1.3 ANSI/ISO 标准 15
1.4 输出文件 15
1.5 实用程序 16

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。

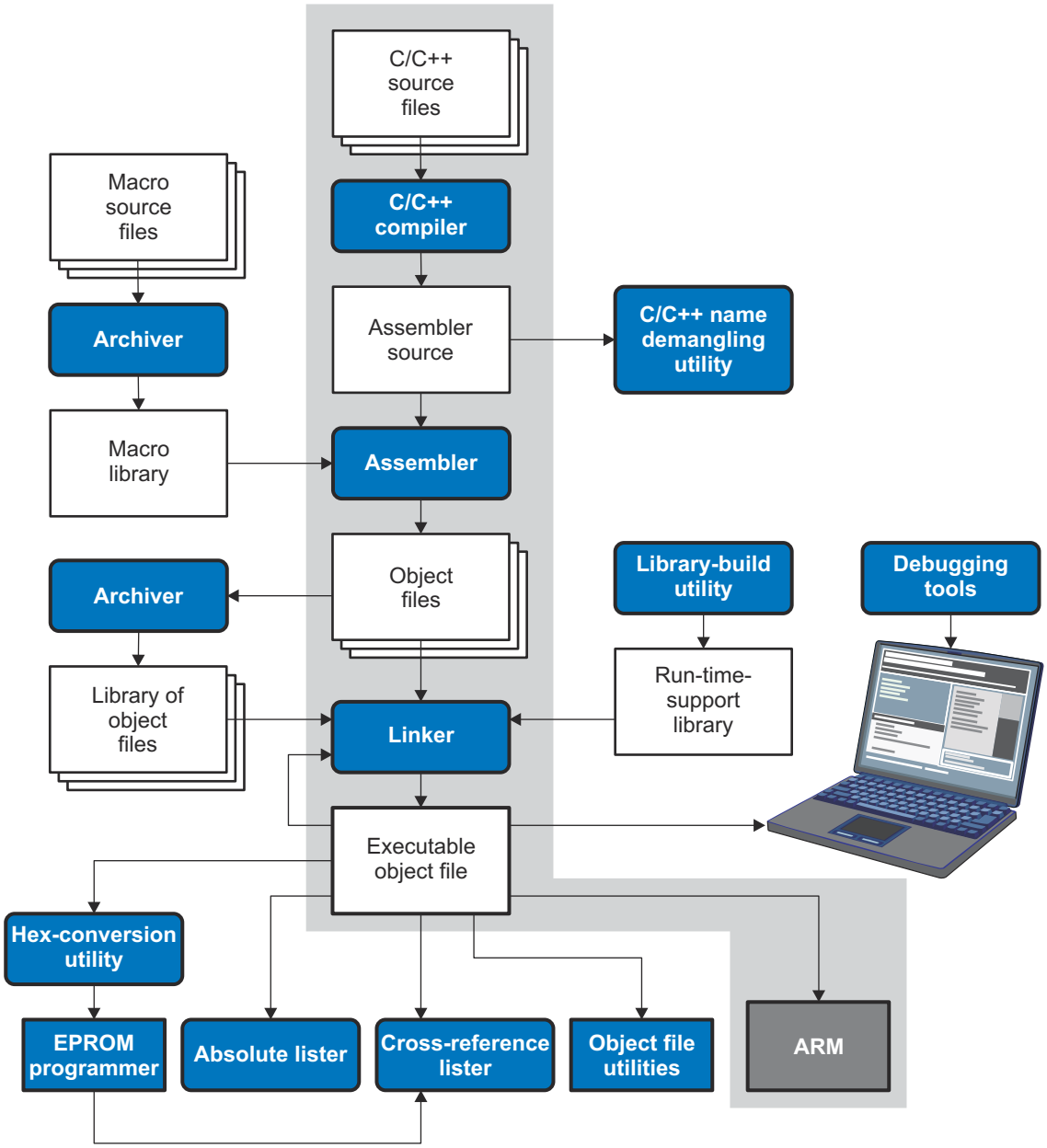


图 1-1. ARM 软件开发流程

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 ARM 汇编语言源代码。请参阅章节 2。
- **汇编器**将汇编语言源文件转换成机器语言可重定位的目标文件。请参阅 *ARM 汇编语言工具用户指南*。
- **链接器**将可重定位的目标文件组合成单个绝对可执行的目标文件。在创建可执行文件时，会执行重定位并解析外部引用。链接器接受可重定位的目标文件和对象库作为输入。有关链接器的概览信息，请参阅章节 4。请参阅 *ARM 汇编语言工具用户指南*，了解详细信息。
- **归档器**允许将一组文件收集到一个称为*库*的单个存档文件中。归档器允许通过删除、替换、提取或添加成员来修改这种库。归档器最有用的应用之一是构建目标文件库。请参阅 *ARM 汇编语言工具用户指南*。

- **运行时支持库**包含编译器支持的标准 ISO C 和 C++ 库函数、编译器实用程序函数、浮点算术函数和 C I/O 函数。请参阅章节 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。
- **十六进制转换实用程序**将目标文件转换为其他目标格式。可将转换后的文件下载到 EPROM 编程器。请参阅 *ARM 汇编语言工具用户指南*。
- **绝对列表器**接受链接的目标文件作为输入并创建 .abs 文件作为输出。可以组合这些 .abs 文件以生成包含绝对地址而不是相对地址的列表。如果没有绝对列表器，生成这样的列表将是冗长而乏味的，并且需要许多手动操作。请参阅 *ARM 汇编语言工具用户指南*。
- **交叉引用列表器**使用目标文件生成交叉引用列表，其中显示符号、其定义以及其在链接的源文件中的引用。请参阅 *ARM 汇编语言工具用户指南*。
- **C++ 名称还原器**是一种调试辅助工具，可将编译器改编的名称转换回其在 C++ 源代码中声明的原始名称。如图 1-1 所示，可对编译器输出的汇编文件使用 C++ 名称还原器；还可对汇编器列表文件和链接器映射文件使用此实用程序。请参阅章节 8。
- **反汇编器**解码目标文件以显示它们所表示的汇编指令。请参阅 *ARM 汇编语言工具用户指南*。
- 此开发流程的主要产品是可执行的目标文件，其可以在 **ARM** 器件上执行。

1.2 编译器接口

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

更多有关程序编译的信息，请参阅节 2.1。

编译器具有直接调用约定，因此可以编写相互调用的汇编和 C 函数。更多有关调用约定的信息，请参阅章节 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。以前使用的是 C++03。请参阅 C++ 标准 ISO/IEC 14882:2014。有关*不受支持的* C++ 特性的说明，请参阅节 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 5.16 。

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。用于 ARM 的 ELF 格式是应用程序二进制接口 (ABI) 规范的一部分，相关信息请参阅 ARM Infocenter 文档。

v15.6.0.STS 和更高版本的 TI 代码生成工具不支持 COFF 目标文件以及旧的 TIABI 和 TI ARM9 ABI 模式。如果希望生成 COFF 输出文件，请使用 v5.2 的 ARM 代码生成工具，并参考 SPNU151J 文档。

1.5 实用程序

这些功能是编译器实用程序：

- **库构建实用程序**

  库构建实用程序允许您从源代码中为运行时模型的任何组合自定义构建对象库。有关更多信息，请参阅 节 7.4。
- **C++ 名称还原器**

  C++ 名称还原器 (armdem) 是一种调试辅助工具，可将在编译器生成的汇编代码、反汇编输出或编译器诊断消息中检测到的每个已改编的名称转换为在 C++ 源代码中找到的原始名称。有关更多信息，请参阅 章节 8。
- **十六进制转换实用程序**

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

章节 2

使用 C/C++ 编译器

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

2.1 关于编译器 18
2.2 调用 C/C++ 编译器 18
2.3 使用选项更改编译器的行为 19
2.4 通过环境变量控制编译器 34
2.5 控制预处理器 35
2.6 将参数传递给 main() 41
2.7 了解诊断消息 42
2.8 其他消息 44
2.9 生成交叉参考列表信息（--gen_cross_reference_listing 选项） 44
2.10 生成原始列表文件（--gen_preprocessor_listing 选项） 46
2.11 使用内联函数扩展 46
2.12 使用交叉列出功能 49
2.13 控制应用程序二进制接口 49
2.14 VFP 支持 50
2.15 启用入口挂钩和出口挂钩函数 51

2.1 关于编译器

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

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

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

备注

调用链接器

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

有关汇编器和链接器的完整说明，请参阅《*ARM 汇编语言工具用户指南*》。

2.2 调用 C/C++ 编译器

要调用编译器，请输入：

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

armcl	用于运行编译器和汇编器的命令。
options	影响编译器对输入文件的处理方式的选项。表 2-7 到表 2-29 列出了这些选项。
filenames	一个或多个 C/C++ 源文件和 汇编语言源文件。
--run_linker (-z)	调用链接器的选项。--run_linker 选项的缩写形式为 -z。有关更多信息，请参阅章节 4。
link_options	控制链接过程的选项。
object files	链接过程的目标文件的名称。

编译器的参数分为三种类型：

- 编译器选项
- 链接选项
- 文件名

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

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

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

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

2.3 使用选项更改编译器的行为

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

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

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

表 2-1 到表 2-29 汇总了所有选项（包括链接选项）。使用表中的参考资料以获取更完整的选项说明。

表 2-1. 处理器选项

选项	别名	效果	段
--silicon_version={ 4 | 5e | 6 | 6M0 | 7A8 | 7M3 | 7M4 | 7R4 | 7R5 }	-mv	选择处理器版本：ARM V4 (ARM7)、ARM V5e (ARM9E)、ARM V6 (ARM11)、ARM V6M0 (Cortex-M0)、ARM V7A8 (Cortex-A8)、ARM V7M3 (Cortex-M3)、ARM V7M4 (Cortex-M4)、ARM V7R4 (Cortex-R4) 或 ARM V7R5 (Cortex-R5)。默认为 ARM V4。	节 2.3.4
--code_state={ 16 | 32 }		指定 ARM 编译模式。	节 2.3.4
--float_support={ vfpv2 | vfpv3 | vfpv3d16 | fpv4spd16 | none)		生成矢量浮点 (VFP) 协处理器指令。仅当目标硬件提供这一功能时才使用此选项。	节 2.14
--little_endian 或 --endian={ big | little }	-me	指定小端代码。默认为 big-endian。	节 2.3.4

表 2-2. 优化选项(1)

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

(1) **注意：**机器专用选项（参阅表 2-13）也会影响优化。

表 2-3. 高级优化选项(1)

选项	别名	效果	段
--auto_inline=[*size*]	-oi	设置自动内联大小（仅限 --opt_level=3）。如果未指定 *size*，则默认值为 1。	节 3.5
--call_assumptions=*n*	-op*n*	级别 0 (-op0) 指定了模块包含从提供给编译器的源代码外部调用或修改的函数和变量。 级别 1 (-op1) 指定了模块包含从提供给编译器的源代码外部修改的变量，但不使用从源代码外部调用的函数。 级别 2 (-op2) 指定了模块不包含从提供给编译器的源代码外部调用或修改的函数或变量（默认值）。 级别 3 (-op3) 指定了模块包含从提供给编译器的源代码外部调用的函数，但不使用从源代码外部修改的变量。	节 3.4.1
--disable_inlining		防止发生任何内联。	节 2.11
--fp_mode={relaxed|strict}		启用或禁用宽松浮点模式。	节 2.3.3
--fp_reassoc={on|off}		启用或禁用浮点算术的重新关联。	节 2.3.3
--gen_opt_info=*n*	-on*n*	级别 0 (-on0) 禁用优化信息文件。 级别 1 (-on2) 生成优化信息文件。 级别 2 (-on2) 生成详细的优化信息文件。	节 3.3.1
--optimizer_interlist	-os	交叉列出优化器注释与汇编语句。	节 3.11
--program_level_compile	-pm	组合源文件以执行程序级优化。	节 3.4
--sat_reassoc={on|off}		启用或禁用饱和算术的重新关联。默认为 --sat_reassoc=off。	节 2.3.3
--aliased_variables	-ma	表示使用了特定的别名技术。	节 3.9

(1) **注意：**机器专用选项（参阅表 2-13）也会影响优化。

表 2-4. 调试选项

选项	别名	效果	段
--symdebug:dwarf	-g	默认行为。启用符号调试。调试信息的生成不会影响优化。因此，默认情况下会生成调试信息。	节 2.3.5 节 3.12
--symdebug:dwarf_version=2|3|4		指定 DWARF 格式版本。	节 2.3.5
--symdebug:none		禁用所有符号调试。	节 2.3.5 节 3.12
--symdebug:skeletal		（已弃用；无效。）	

表 2-5. Include 选项

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

表 2-6. ULP 顾问选项

选项	别名	效果	段
--advice:power[={all|none|*rulespec*}]		启用检查指定的 ULP Advisor 规则。（默认为 all。）	节 2.3.3
--advice:power_severity={error|warning|remark|suppress}		设置 ULP Advisor 规则的诊断严重程度。	节 2.3.3

表 2-7. 控制选项

选项	别名	效果	段
--compile_only	-c	禁用链接（否定 --run_linker）。	节 4.1.3
--help	-h	打印（在标准输出设备上）编译器理解的选项的说明。	节 2.3.2
--run_linker	-z	导致从编译器命令行调用链接器。	节 2.3.2

表 2-7. 控制选项 (continued)

选项	别名	效果	段
--skip_assembler	-n	编译 C/C++ 源文件，从而生成汇编语言输出文件。汇编器不会运行，也不会生成目标文件。	节 2.3.2

表 2-8. 语言选项

选项	别名	效果	段
--c89		根据 ISO C89 标准处理 C 文件。	节 5.16
--c99		根据 ISO C99 标准处理 C 文件。	节 5.16
--c11		根据 ISO C11 标准处理 C 文件。	节 5.16
--c++14		根据 ISO C++14 标准处理 C++ 文件。 已弃用 --c++03 选项。	节 5.16
--cpp_default	-fg	将所有带有 C 扩展名的源文件作为 C++ 源文件处理。	节 2.3.7
--enum_type={int|packed}		选择是否使用紧凑整数类型来存储小型枚举类型。	节 2.3.4
--exceptions		启用 C++ 异常处理。	节 5.8
--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 2.3.3
--gen_cross_reference_listing	-px	生成交叉引用列表文件 (.crl)。	节 2.9
--pending_instantiations=#		指定在任何给定时间可能正在进行的模板实例化的数量。使用 0 来指定无限数量。	节 2.3.4
--plain_char={signed|unsigned}	-mc	指定如何处理普通字符，默认为无符号。	节 2.3.4
--printf_support={nofloat|full| minimal}		支持小型且版本受限的 printf 函数系列（sprintf、fprintf 等）和 scanf 函数系列（sscanf、fscanf 等）运行时支持函数的支持。	节 2.3.3
--relaxed_ansi	-pr	启用宽松模式；忽略严格的 ISO 违规。默认设置为 on。要禁用此模式，请使用 --strict_ansi 选项。	节 5.16.3
--rtti	-rtti	启用 C++ 运行时类型信息 (RTTI)。	--
--strict_ansi	-ps	启用严格的 ANSI/ISO 模式（适用于 C/C++，不适用于 K&R C）。在此模式下，禁用与 ANSI/ISO C/C++ 冲突的语言扩展。在严格的 ANSI/ISO 模式下，大多数 ANSI/ISO 违规都会报告为错误。被视为酌情处理的违规行为可能会报告为警告。	节 5.16.3
--wchar_t={32|16}		设置 C/C++ 类型 wchar_t 的大小。默认为 16 位。	节 2.3.4

表 2-9. 解析器预处理选项

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

表 2-10. 预定义宏选项

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

表 2-10. 预定义宏选项 (continued)

选项	别名	效果	段
--undefine=*name*	-U	未定义 *name*。	节 2.3.2

表 2-11. 诊断消息选项

选项	别名	效果	段
--compiler_revision		打印出编译器发布版本并退出。	--
--diag_error=*num*	-pdse	将 *num* 标识的诊断分类为错误。	节 2.7.1
--diag_remark=*num*	-pdsr	将 *num* 标识的诊断分类为备注。	节 2.7.1
--diag_suppress=*num*	-pds	抑制 *num* 标识的诊断。	节 2.7.1
--diag_warning=*num*	-pdsw	将 *num* 标识的诊断分类为警告。	节 2.7.1
--diag_wrap={on|off}		使诊断消息换行（默认为 on）。请注意，此命令行选项不能在 Code Composer Studio IDE 中使用。	
--display_error_number	-pden	显示诊断的标识符及其文本。请注意，此命令行选项不能在 Code Composer Studio IDE 中使用。	节 2.7.1
--emit_warnings_as_errors	-pdew	将警告视为错误。	节 2.7.1
--gen_func_info_listing		生成用户信息文件 (.aux)。	节 2.3.2
--issue_remarks	-pdr	发出备注（非严重警告）。	节 2.7.1
--no_warnings	-pdw	抑制诊断警告（仍会发出错误）。	节 2.7.1
--quiet	-q	抑制进度消息（静默）。	--
--set_error_limit=*num*	-pdel	将错误限制设置为 *num*。在在错误达到这个数量后，编译器放弃编译。（默认为 100。）	节 2.7.1
--super_quiet	-qq	超级静默模式。	--
--tool_version	-version	显示每个工具的版本号。	--
--verbose		显示横幅和函数进度信息。	--
--verbose_diagnostics	-pdv	提供详细的诊断消息，以自动换行的方式显示原始源代码。请注意，此命令行选项不能在 Code Composer Studio IDE 中使用。	节 2.7.1
--write_diagnostics_file	-pdf	生成诊断消息信息文件。编译器唯一选项。请注意，此命令行选项不能在 Code Composer Studio IDE 中使用。	节 2.7.1

表 2-12. 补充信息选项

选项	别名	效果	段
--gen_preprocessor_listing	-pl	生成原始列表文件 (.rl)。	节 2.10
--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 2.7.1

表 2-13. 运行时模型选项

选项	别名	效果	段
--common={on|off}		默认为 on。设置为 on 时，未初始化的文件范围变量作为通用符号发出。设置为 off 时，不会创建通用符号。	节 2.3.4
--embedded_constants={on|off}		控制编译器是否在函数中嵌入常量。	节 2.3.4
--gen_data_subsections={on|off}		将所有聚合数据（数组、结构和联合体）放入子段中。这使得链接器可以更好地控制在最终链接步骤期间删除未使用的数据。有关默认设置的详细信息，请参阅右侧的链接。	节 4.2.2
--global_register={r5|r6|r9}	-rr	禁止编译器使用 rx=[5,6,9]。	节 2.3.4
-neon		支持 Cortex-A8 Neon SIMD 指令集。	节 2.3.4
--ramfunc={on|off}		如果设置为 on，则指定所有函数都应放置在 RAM 中的 .TI.ramfunc 段中。	节 2.3.4
--unaligned_access={on|off}		控制未对齐访问的生成。	节 2.3.4

表 2-13. 运行时模型选项 (continued)

选项	别名	效果	段
--use_dead_funcs_list [=*fname*]		将文件中列出的每个函数放在单独段中。	节 2.3.4

表 2-14. 入口/出口挂钩选项

选项	别名	效果	段
--entry_hook[=*name*]		启用入口挂钩。	节 2.15
--entry_parm={none|name|address}		将函数的参数指定给 --entry_hook 选项。	节 2.15
--exit_hook[=*name*]		启用出口挂钩。	节 2.15
--exit_parm={none|name|address}		将函数的参数指定给 --exit_hook 选项。	节 2.15
--remove_hooks_when_inlining		删除自动内联函数的入口/出口挂钩。	节 2.15

表 2-15. 反馈选项

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

表 2-16. 汇编器选项

选项	别名	效果	段
--keep_asm	-k	保留汇编语言 (.asm) 文件。	节 2.3.11
--asm_listing	-al	生成汇编列表文件。	节 2.3.11
--c_src_interlist	-ss	交叉列出 C 源代码和汇编语句。	节 2.12 节 3.11
--src_interlist	-s	交叉列出优化器注释（如果可用）和汇编源语句；否则交叉列出 C 语言和汇编源语句。	节 2.3.2
--absolute_listing	-aa	启用绝对列表。	节 2.3.11
--asm_cross_reference_listing	-ax	生成交叉引用文件。	节 2.3.11
--asm_define=*name*[=*def*]	-ad	设置 *name* 符号。	节 2.3.11
--asm_dependency	-apd	执行预处理；仅列出程序集依赖项。	节 2.3.11
--asm_includes	-api	执行预处理；仅列出包含的 #include 文件。	节 2.3.11
--asm_undefine=*name*	-au	不对预定义的常量 *name* 进行定义。	节 2.3.11
--code_state={16|32}		开始将指令汇编为 16 位或 32 位指令。	节 2.3.11
--include_file=*filename*	-ahi	包含汇编模块的指定文件。	节 2.3.11

表 2-17. 文件类型说明符选项

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

表 2-18. 目录说明符选项

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

表 2-19. 默认文件扩展名选项

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

表 2-20. 命令文件选项

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

表 2-21. MISRA-C 2004 选项

选项	别名	效果	段
--check_misra[={all|required|advisory|none|*rulespec*}]		检查指定 MISRA-C:2004 规则。默认为 all。	节 2.3.3
--misra_advisory={error|warning|remark|suppress}		为建议性 MISRA-C:2004 规则设置诊断严重程度。	节 2.3.3
--misra_required={error|warning|remark|suppress}		为所需的 MISRA-C:2004 规则设置诊断严重程度。	节 2.3.3

2.3.1 链接器选项

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

表 2-22. 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 = 2K 字节。
--heap_size=*size*	[-]-heap	将堆大小（对于 C 中的动态存储器分配）设为 *size* 字节，并定义全局符号来指定栈大小。默认值 = 2K 字节。

表 2-23. 文件搜索路径选项

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

表 2-24. 命令文件预处理选项

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

表 2-25. 诊断消息选项

选项	别名	说明
--diag_error=*num*		将由 *num* 标识的诊断分类为错误。
--diag_remark=*num*		将由 *num* 标识的诊断分类为备注。
--diag_suppress=*num*		抑制由 *num* 标识的诊断。
--diag_warning=*num*		将由 *num* 标识的诊断分类为警告。
--display_error_number		显示诊断的标识符及其文本。
--emit_references:file[=*file*]		发出包含段信息的文件。这些信息包括段大小、定义的符号和对符号的引用。
--emit_warnings_as_errors	-pdew	将警告视为错误。
--issue_remarks		发出备注（非严重警告）。
--no_demangle		禁止解码诊断消息中的符号名称。
--no_warnings		抑制诊断警告（仍会发出错误）。
--set_error_limit=*count*		将错误限制设置为 *count*。在达到此错误数量后，链接器将放弃链接。（默认为 100。）
--verbose_diagnostics		提供详细的诊断消息，以换行方式显示原始源代码。
--warn_sections	-w	创建未定义的输出段时显示一条消息。

表 2-26. 链接器输出选项

选项	别名	说明
--absolute_exe	-a	生成绝对可执行目标文件。这是默认设置；如果 --absolute_exe 和 --relocatable 均未指定，链接器的行为就像指定了 --absolute_exe 一样。
--ecc={ on | off }		启用由链接器生成的错误校正码 (ECC)。默认关闭。

表 2-26. 链接器输出选项 (continued)

选项	别名	说明
--ecc:data_error		将指定的错误注入到输出文件中进行测试。
--ecc:ecc_error		将指定的错误注入到错误校正码 (ECC) 中进行测试。
--generate_dead_funcs_list		将链接器删除的死函数列表写入文件 fname。
--mapfile_contents=*attribute*		控制映射文件中包含的信息。
--relocatable	-r	生成不可执行的、可重定位输出目标文件。
--run_abs	-abs	生成绝对列表文件。
--xml_link_info=*file*		生成结构良好的 XML *file*，其中包含有关链接结果的详细信息。

表 2-27. 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		保存原本会被丢弃的段列表。
--scan_libraries	-scanlibs	扫描所有库中的重复符号定义。
--symbol_map=*refname*=*defname*		指定符号映射；对 *refname* 符号的引用被替换为对 *defname* 符号的引用。与 --opt_level=4 一同使用时，支持 --symbol_map 选项。
--undef_sym=*symbol*	-u	将 *symbol* 作为未解析符号添加到符号表中。
--unhide=*pattern*		排除与指定 *pattern* 匹配的符号，使其不被隐藏。

表 2-28. 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_hold_wdt={on|off}		RTS 自动初始化例程中的链接，该例程在 cinit 自动初始化期间保持（开启）或不保持（关闭）看门狗计时器。请参阅节 4.3.3。
-be32		强制链接器生成 BE-32 目标代码。
-be8		强制链接器生成 BE-8 目标代码。
--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。有关详细信息，请参阅节 4.3.5。
--rom_model	-c	在运行时自动初始化变量。有关详细信息，请参阅节 4.3.5。
--trampolines[=off|on]		生成 far call trampolines（参数是可选的，默认为“on”）。

表 2-29. 其他选项

选项	别名	说明
--compress_dwarf[=off|on]		积极减少输入目标文件中 DWARF 信息的大小。默认为 on。
--linker_help	[-]-help	显示有关语法和可用选项的信息。
--minimize_trampolines		放置段以最大限度地减少所需的 far trampolines 数量。
--preferred_order=*function*		为函数放置设定优先级。
--trampoline_min_spacing		当 trampoline 预留的间隔比指定的限值更近时，尝试使它们相邻。

表 2-29. 其他选项 (continued)

选项	别名	说明
--unused_section_elimination[=off|on]		消除可执行模块中不需要的段。默认为 on。
--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。

2.3.2 常用选项

以下是对可能会经常使用的选项的详细说明：

--c_src_interlist 调用交叉列出功能，该功能使原始 C/C++ 源代码与编译器生成的汇编语言交织在一起。交叉列出的 C 语句可能看起来是乱序的。可通过组合 --optimizer_interlist 和 --c_src_interlist 选项，将 交叉列出功能与优化器结合使用。请参阅节 3.11。--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：

```
armcl --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 TI_ARM_C_OPTION 环境变量中指定了 --run_linker 但又不希望链接时，请使用此选项。请参阅节 4.1.3。

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

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

--gen_func_info_listing 生成文件扩展名为 .aux 的用户信息文件。该文件包含每个文件级别上的链接器调用图信息。

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

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

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

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

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

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

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

--tool_version 打印编译器中每个工具的版本号。未发生编译。

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

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

2.3.3 其他有用的选项

以下是其他选项的详细说明：

--advice:power={all|none|*rulespec*} 允许根据 ULP (超低功耗) Advisor 规则检查代码，以避免可能出现的功耗不足。更多详细信息，请参阅 www.ti.com/ulpadvisor。rulespec 参数是用逗号分隔的说明符列表。有关详细信息，请参阅节 5.4。

--advice:power_severity={error|warning|remark|suppress}	设置 ULP Advisor 规则的诊断严重程度。
--check_misra={all|required|advisory|none|*rulespec*}	显示指定数量或类型的 MISRA-C 文档。如果要在源代码中启用 CHECK_MISRA 和 RESET_MISRA pragma，则必须使用此选项。*rulespec* 参数是用逗号分隔的说明符列表。有关详细信息，请参阅节 5.3。
--float_operations_allowed={none|all|32|64}	限制允许的浮点运算类型。默认为 all。如果设置为 none、32 或 64，则检查应用程序是否将在运行时执行运算。例如，如果在命令行上指定了 --float_operations_allowed=32，则编译器将在生成双精度运算时发出错误消息。这可以用来确保双精度运算不会意外地被引入到应用程序中。检查是在进行宽松模式优化后执行的，因此完全删除非法运算不会产生任何诊断消息。
--fp_mode={relaxed|strict}	默认的浮点模式为 strict。要启用宽松浮点模式，请使用 --fp_mode=relaxed 选项。宽松浮点模式会使双精度浮点计算和存储在可能的情况下转换为单精度浮点。这种行为不符合 ISO 要求，但会加快代码速度，准确性会有降低。宽松模式下会发生以下具体的变化： • 如果双精度浮点表达式的结果被分配给单精度浮点或整数，或者立即在单精度上下文中使用，则表达式的计算将转换为单精度计算。如果表达式中的双精度常量可以正确地表示为单精度常量，那么它们会转换为单精度常量。 • 如果所有参数都是单精度的并且结果将在单精度上下文中使用，则对 math.h 中的双精度函数的调用将转换为对应的单精度函数。必须包含 math.h 头文件才能使此优化正常运行。 • 除以一个常数被转换为逆乘法。 • 对 sqrt、sqrtf 和 sqrtl 的调用被直接转换为 VSQRT 指令。在这种情况下，不会为负输入设置 errno。 • 某些 C 标准浮点函数（例如 sqrt、sin、cos、atan、atan2 和 fmodf）会被重定向到优化的内联函数（如有可能）。
--fp_reassoc={on|off}	启用或禁用浮点算术的重新组合。如果指定了 --fp_mode=relaxed，则自动设置 --fp_reassoc=on。 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。
--misra_advisory={error|warning|remark|suppress}	为建议性 MISRA-C:2004 规则设置诊断严重程度。
--misra_required={error|warning|remark|suppress}	为所需的 MISRA-C:2004 规则设置诊断严重程度。
--preinclude=*filename*	在编译开始时包含 *filename* 的源代码。这可用于建立标准的宏定义。在包含搜索列表上的目录中搜索文件名。文件按照指定的顺序进行处理。
--printf_support={full|nofloat|minimal}	支持更小、有限版本的 printf 函数系列（sprintf、fprintf 等）和 scanf 函数系列（sscanf、fscanf 等）运行时支持函数。有效值为： • full：支持所有格式说明符。这是默认设置。 • nofloat：不支持打印和扫描浮点值。支持除 %a、%A、%f、%F、%g、%G、%e 和 %E 之外的所有格式说明符。 • minimal：支持打印和扫描没有宽度或精度标志的整数、字符或字符串值。具体来说，仅支持 %%、%d、%o、%c、%s 和 %x 格式说明符。 没有运行时错误检查来检测是否使用了未包含支持的格式说明符。--printf_support 选项位于 --run_linker 选项之前，并且必须在执行最终链接时使用。
--sat_reassoc={on|off}	启用或禁用饱和算术的重新组合。

2.3.4 运行时模型选项

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

ARM 编译器现在仅支持使用 ELF 目标文件格式和 DWARF 调试格式的嵌入式应用程序二进制接口 (EABI) ABI。如果希望支持传统的 COFF ABI，请使用 ARM v5.2 代码生成工具，并参阅 SPNU151J 和 SPNU118J 以查看相关文档。

--code_state={16|32}	生成 16 位 Thumb 代码。默认情况下生成 32 位代码。当选择支持 Cortex-R4、Cortex-M0、Cortex-M3 或 Cortex-A8 架构时，--code_state 选项会生成 Thumb-2 代码。有关 16 位与 32 位代码中的间接调用的详细信息，请参阅节 6.11.2.2。
--common={on|off}	当为 on（默认设置）时，未初始化的文件范围变量作为通用符号发出。当为 off 时，不会创建通用符号。允许创建通用符号的好处是生成的代码可以删除未使用的变量，否则会增加 .bss 段的大小。（大于 32 字节的未初始化变量通过放置在可以在链接时省略的单独子段中被单独地优化。）如果变量已分配到 .bss 以外的段或相对于另一个通用符号被定义，则变量不能作为通用符号。
--embedded_constants={on|off}	默认情况下，编译器会在函数中嵌入常量。这些常量可以包括文字、地址、字符串等。如果希望阻止从仅包含可执行代码的内存区域中读取，这将是一个问题。为了能够生成“仅执行的代码”，编译器提供了 --embedded_constants=[on|off] 选项。如果未指定此选项，则假设为 on。该选项在以下器件上可用：Cortex-A8、Cortex-M3、Cortex-M4 和 Cortex-R4。
--endian={ big | little }	为编译的代码指定大端或小端格式。默认情况下使用大端格式。
--enum_type={int|packed}	指定枚举类型的基础类型。默认为 packed，这会使基础枚举类型成为适应枚举常量的最小整数类型。使用 --enum_type=int 会导致基础类型始终为 int。值超出 int 范围的枚举常量会生成错误。
--float_support={ vfpv2 | vfpv3 | vfpv3d16 | fpv4spd16 | none }	为各种版本和库生成矢量浮点 (VFP) 协处理器指令。请参阅节 2.14。
--global_register={r5|r6|r9}	禁止编译器使用 rx=[5|6|9]。在命令行上只能使用一个 --global_register 选项；如果指定了多个这样的选项，则只有最后一个选项生效。
-md	禁用双状态互通支持。请参阅节 6.11.1。
-mv={4|5e|6|6M0|7A8|7M3 |7M4|7R4|7R5}	选择处理器版本：ARM V4 (ARM7)、ARM V5e (ARM9E)、ARM V6 (ARM11)、ARM V6M0 (Cortex-M0)、ARM V7A8 (Cortex-A8)、ARM V7M3 (Cortex-M3)、ARM V7M4 (Cortex-M4)、ARM V7R4 (Cortex-R4) 或 ARM V7R5 (Cortex-R5)。默认为 ARM V4。
--neon	编译器可以使用第 7 版 ARM 架构的 Neon 扩展中提供的 SIMD 指令来生成代码。优化器尝试对源代码进行矢量化，以利用这些 SIMD 指令。为了生成矢量化 SIMD Neon 代码，请使用 -mv=7A8 选项选择第 7 版架构，并使用 --neon 选项启用 Neon 指令支持。 优化器用于对源代码进行矢量化。尽管建议使用 3 级 (--opt_level=3) 和 --opt_for_speed 选项，但还是需要 至少 2 级优化（--opt_level=2 或 O2）。
--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。
--plain_char={signed|unsigned}	指定如何处理 C/C++ 普通字符变量。默认为无符号。
--ramfunc={on|off}	如果设置为 on，则指定所有函数都应放置在位于 RAM 中的 .TI.ramfunc 段中。如果设置为 off，则只有具有 ramfunc 函数属性的函数才会以此种方式被处理。请参阅节 5.17.2。 较新的 TI 链接器命令文件通过在 .TI.ramfunc 段中放置函数来自动支持 --ramfunc 选项。如果链接器命令文件不包含 .TI.ramfunc 段的段规格，则可以修改链接器命令文件以将此段放在 RAM 中。有关段放置位置的详细信息，请参阅《*ARM 汇编语言工具用户指南*》。
--silicon_version	选择指令集版本。选项是： • 4 = ARM V4 (ARM7) 这是默认设置。 • 5e = ARM V5e (ARM9E) • 6 = ARM V6 (ARM11) • 6M0 = ARM V6M0 (Cortex-M0) • 7A8 = ARM V7A8 (Cortex-A8) • 7M3 = ARM V7M3 (Cortex-M3) • 7M4 = ARM V7M4 (Cortex-M4) • 7R4 = ARM V7R4 (Cortex-R4) • 7R5 = ARM V7R5 (Cortex-R5) 使用 --silicon_version=7M4 选项自动设置 --float_support=fpv4spd16 选项。如需禁用硬件浮点支持，请使用 --float_support=none 选项。
--unaligned_access={on|off}	通知编译器目标器件支持未对齐的内存访问。通常，数据与其大小边界对齐。例如，32 位数据在 32 位边界上对齐，16 位数据在 16 位边界上对齐，8 位数据在 8 位边界上对齐。如果此选项设置为 on，则告知编译器为落在未对齐边界上的数据（16 位边界上的 32 位数据）生成加载和存储指令是合法的。可能发生未对齐数据访问的情况包括调用 memcpy() 和访问打包的结构体。默认情况下，所有 Cortex 器件都启用此选项。

--use_dead_funcs_list[=*fname*]	将文件中列出的每个函数放在单独的段中。如果指定了函数，将其放在 *fname* 段中。不建议在 Code Composer Studio IDE 中使用此选项和 --generate_dead_funcs_list，相反，请考虑使用 --opt_level=4、--program_level_compile 和/或 --gen_func_subsections。
--wchar_t={32|16}	设置 C/C++ 类型 wchar_t 的大小（以位为单位）。默认情况下，编译器生成 16 位 wchar_t。16 位 wchar_t 对象与 32 位 wchar_t 对象不兼容；如果将这两个对象组合在一起，则会产生错误。

2.3.5 符号调试和分析选项

下述选项用于选择符号调试或分析：

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

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

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

--symdebug:skeletal 已弃用。没有作用

2.3.6 指定文件名

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

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

备注

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

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

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

```
armcl *.cpp
```

备注

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

2.3.7 更改编译器解释文件名的方式

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

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

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

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

无法对文件名选项使用通配符规范。

备注

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

2.3.8 更改编译器处理 C 文件的方式

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

2.3.9 更改编译器解释和命名扩展名的方式

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

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

以下示例将汇编文件 fit.rrr ，并创建名为 fit.o 的目标文件：

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

扩展名中的句点 (.) 是可选的。 以上实例也可以写成：

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

2.3.10 指定目录

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

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

--asm_directory=*directory*
指定汇编文件的目录。例如：
```
armcl --asm_directory=d:\assembly
```

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

--obj_directory=*directory*
指定目标文件的目录。例如：
```
armcl --obj_directory=d:\object
```

--output_file=*filename*
指定编译输出文件名；可以覆盖 --obj_directory。例如：
```
armcl --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：
```
armcl --pp_directory=d:\preproc
```

--temp_directory=*directory*
指定临时中间文件的目录。例如：
```
armcl --temp_directory=d:\temp
```

2.3.11 汇编器选项

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

--absolute_listing
生成具有绝对地址而不是段偏移的列表。

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

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

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

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

--asm_listing
生成汇编列表文件。

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

--code_state={16|32}
生成 16 位 Thumb 代码。默认情况下生成 32 位代码。当选择支持 Cortex-R4、Cortex-M0、Cortex-M3 或 Cortex-A8 架构时，--code_state 选项会生成 Thumb-2 代码。有关 16 位与 32 位代码中的间接调用的详细信息，请参阅节 6.11.2.2。

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

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

2.3.12 已弃用的选项

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

2.4 通过环境变量控制编译器

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

备注

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

2.4.1 设置默认编译器选项 (TI_ARM_C_OPTION)

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

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

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

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

```
set TI_ARM_C_OPTION=--quiet --src_interlist --run_linker
```

备注

如果 TI_ARM_C_OPTION 环境变量与较旧的 TMS470_C_OPTION 环境变量均已定义，那么前者优先。如果只设置了 TMS470_C_OPTION，则将继续使用。

命令行或 TI_ARM_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 TI_ARM_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 TI_ARM_C_OPTION 设置如上所示：

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

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

2.4.2 命名一个或多个备用目录 (TI_ARM_C_DIR)

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

备注

如果 TI_ARM_C_DIR 环境变量与较旧的 TMS470_C_DIR 环境变量都已定义，则前者优先于后者。如果只设置了 TMS470_C_DIR，则将继续使用它。

2.5 控制预处理器

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

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

预处理器生成自解释的错误消息。出现错误的行号和文件名与诊断消息一同打印。

2.5.1 预先定义的宏名称

编译器维护并识别表 2-30 中列出的预定义宏名称。

表 2-30. 预定义 ARM 宏名称

宏名称	说明
__16bis__	如果选择了 16-BIS 状态（使用了 -code_state=16 选项），则已定义；否则未定义。
__32bis__	如果选择了 32-BIS 状态（未使用 -code_state=16 选项），则已定义；否则未定义。
_AEABI_PORTABILITY_LEVEL	定义为 1，以便在包含头文件时，全面移植目标文件。定义为 0 以便完全兼容 C 标准。有关详细信息，请参阅 ARM 标准。
__big_endian__	如果选择了大端模式（使用了 --endian=big 选项或未使用 --endian=little 选项），则已定义；否则未定义。
__DATE__ (1)	以 *mmm dd yyyy* 形式扩展到编译日期
__FILE__ (1)	扩展到当前源文件名
_INLINE	如果使用了优化（--opt_level 或 -O 选项），则扩展为 1；否则未定义。
__LINE__ (1)	扩展到当前行号
__little_endian__	如果选择了小端模式（使用了 --endian=little 选项），则已定义；否则未定义。
__PTRDIFF_T_TYPE__	定义为 ptrdiff_t 类型
__signed_chars__	如果 char 类型默认为有符号类型，则已定义
__SIZE_T_TYPE__	定义为 size_t 类型
__STDC__ (1)	定义为 1 以表示编译器符合 ISO C 标准。有关 ISO C 标准的例外情况，请参阅节 5.1。
__STDC_VERSION__	C 标准宏。
__STDC_HOSTED__	C 标准宏。始终定义为 1。

表 2-30. 预定义 ARM 宏名称 (continued)

宏名称	说明
__STDC_NO_THREADS__	C 标准宏。始终定义为 1。
__TI_COMPILER_VERSION__	已定义为 7-9 位整数，具体取决于 X 是 1、2 还是 3 位。该数字不包含小数。例如，版本 3.2.1 表示为 3002001。去掉前导零以防止数字被解释为八进制。
__TI_EABI_SUPPORT__	如果启用了 EABI ABI（这是默认设置），则定义为 1；否则未定义。
__TI_FPALIB_SUPPORT__	如果 FPA 字节序用于存储双精度浮点值，则定义为 1；否则未定义。
__TI_GNU_ATTRIBUTE_SUPPORT__	如果启用了 GCC 扩展（这是默认设置），则定义为 1
__TI_NEON_SUPPORT__	如果目标是 NEON SIMD 扩展（使用了 --neon 选项），则定义为 1；否则未定义。
__TI_STRICT_ANSI_MODE__	如果启用了严格的 ANSI/ISO 模式（使用了 --strict_ansi 选项），则定义为 1；否则定义为 0。
__TI_STRICT_FP_MODE__	如果使用了 --fp_mode=strict（默认设置），则定义为 1；否则定义为 0。
__TI_ARM__	始终已定义
__TI_ARM_V4__	如果目标是 v4 架构 (ARM7)（使用 -mv4 选项），则已定义为 1；否则未定义。
__TI_ARM_V5__	如果目标是 v5E 架构 (ARM9E)（使用 -mv5e 选项），则已定义为 1；否则未定义。
__TI_ARM_V6__	如果目标是 v6 架构 (ARM11)（使用了 -mv6 选项），则已定义为 1；否则未定义。
__TI_ARM_V6M0__	如果目标是 v6M0 架构 (Cortex-M0)（使用了 -mv6M0 选项），则已定义为 1；否则未定义。
__TI_ARM_V7__	如果目标是任意 v7 架构 (Cortex)，则定义为 1；否则未定义。
__TI_ARM_V7A8__	如果目标是 v7A8 架构 (Cortex-A8)（使用了 -mv7A8 选项），则定义为 1；否则未定义。
__TI_ARM_V7M3__	如果目标是 v7M3 架构 (Cortex-M3)（使用了 -mv7M3 选项），则已定义为 1；否则未定义。
__TI_ARM_V7M4__	如果目标是 v7M4 架构 (Cortex-M4)（使用了 -mv7M4 选项），则已定义为 1；否则未定义。
__TI_ARM_V7R4__	如果目标是 v7R4 架构 (Cortex-R4)（使用了 -mv7R4 选项），则定义为 1；否则未定义。
__TI_ARM_V7R5__	如果目标是 v7R5 架构 (Cortex-R5)（使用了 -mv7R5 选项），则定义为 1；否则未定义。
__TI_FPV4SPD16_SUPPORT__	如果启用了 VFP 协处理器（使用了 --float_support=fpv4spd16 选项），则定义为 1；否则未定义。
__TI_VFP_SUPPORT__	如果启用了 VFP 协处理器（使用了任意 --float_support 选项），则定义为 1；否则未定义。
__TI_VFPLIB_SUPPORT__	如果 VFP 字节序用于存储双精度浮点值，则定义为 1；否则未定义。
__TI_VFPV3_SUPPORT__	如果启用了 VFP 协处理器（使用了 --float_support=vfpv3 选项），则定义为 1；否则未定义。
__TI_VFPV3D16_SUPPORT__	如果启用了 VFP 协处理器（使用了 --float_support=vfpv3d16 选项），则定义为 1；否则未定义。
__TI_WCHAR_T_BITS__	定义为 wchar_t 类型。
__TIME__ (1)	以“*hh:mm:ss*”形式扩展到编译时间
__unsigned_chars__	如果 char 类型默认为无符号类型（默认设置），则已定义
__WCHAR_T_TYPE__	定义为 wchar_t 类型。

(1) 由 ISO 标准指定

备注

名称中包含 __TI_ARM 的宏是旧的 __TI_TMS470 宏的副本。例如，__TI_ARM_V7__ 是 __TI_TMS470_V7__ 宏的新名称。旧的宏名称仍然存在并且可以继续使用。

可以按照与任何其他已定义名称相同的方式使用表 2-30 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");
```

此外，ARM C 语言扩展 (ACLE) v2.0 规范描述了标识 ARM 架构特征的宏以及 C/C++ 实现如何使用该架构。所有 ACLE 预定义宏都以前缀 __ARM 开头。表 2-31 列出了 ACLE 规范中提到的宏以及规范中可提供更多信息的章节。有些宏未定义，是因为这些宏不适用于任何 Cortex-M 或 Cortex-R 处理器变体。

表 2-31. ACLE 预定义宏

宏名称	说明	ACLE 规范章节
__ARM_32BIT_STATE	如果编译器为 ARM 32 位处理器变体生成代码（-mv6m0、-mv7m3、-mv7m4、-mv7a8、-mv7r4 和 -mv7r5），则定义为 1；否则未定义。	（第 5.4.1 节）
__ARM_64BIT_STATE	未定义	（第 5.4.1 节）
__ARM_ACLE	对于所有 Cortex-M 和 Cortex-R 处理器变体（-mv6m0、-mv7m3、-mv7m4、-mv7r4 和 -mv7r5），定义为 200。	（第 3.4、5.2 节）
__ARM_ALIGN_MAX_PWR	不支持	（第 6.5.2 节）
__ARM_ALIGN_MAX_STACK_PWR	不支持	（第 6.5.3 节）
__ARM_ARCH	确定在编译器命令行上选择的 ARM 架构版本。 • 4 表示 -mv4 • 5 表示 -mv5e • 6 表示 -mv6 或 -mv6m0 • 7 表示 -mv7a8、-mv7m3、-mv7m4、-mv7r4 或 -mv7r5	（第 5.1 节）
__ARM_ARCH_ISA_A64	未定义	（第 5.4.1 节）
__ARM_ARCH_ISA_ARM	如果编译器为支持 ARM 指令集的处理器变体生成代码（-mv7a8、-mv7r4 和 -mv7r5），则定义为 1；否则未定义。	（第 5.4.1 节）
__ARM_ARCH_ISA_THUMB	如果编译器为支持 THUMB-1 指令集的处理器变体生成代码，则定义为 1。如果编译器为支持 THUMB-2 指令集的处理器变体生成代码，则定义为 2；否则未定义。	（第 5.4.1 节）
__ARM_ARCH_PROFILE	不支持	（第 5.4.2 节）
__ARM_BIG_ENDIAN	默认定义为 1；如果使用了 --little-endian (-me) 选项，则未定义。	（第 5.3 节）
__ARM_FEATURE_CLZ	如果编译器为支持 CLZ 指令的处理器变体生成代码（-mv7m3、-mv7m4、-mv7a8、-mv7r4 和 -mv7r5），则定义为 1；否则未定义。	（第 5.4.5 节）
__ARM_FEATURE_COPROC	不支持	（第 5.9 节）
__ARM_FEATURE_CRC32	未定义	（第 5.5.8 节）
__ARM_FEATURE_CRYPTO	未定义	（第 5.5.7 节）
__ARM_FEATURE_DIRECTED_ROUNDING	未定义	（第 5.5.9 节）
__ARM_FEATURE_DSP	如果编译器为支持 DSP 指令/内在函数的 Cortex-M 或 Cortex-R 处理器生成代码（-mv7m4、-mv7r4 和 -mv7r5），则定义为 1；否则未定义。	（第 5.4.7 节）
__ARM_FEATURE_FMA	不支持	（第 5.5.3 节）
__ARM_FEATURE_FP16_SCALAR_ARITHMETIC	未定义	（第 3.4、5.5.13 节）
__ARM_FEATURE_FP16_VECTOR_ARITHMETIC	未定义	（第 5.5.13 节）
__ARM_FEATURE_IDIV	不支持	（第 5.4.10 节）
__ARM_FEATURE_JCVT	未定义	（第 5.5.14 节）
__ARM_FEATURE_LDREX	未定义	（第 5.4.4 节）
__ARM_FEATURE_NUMERIC_MAXMIN	未定义	（第 5.5.10 节）
__ARM_FEATURE_QBIT	不支持	（第 5.4.6 节）
__ARM_FEATURE_QRDMX	未定义	（第 5.5.12 节）

表 2-31. ACLE 预定义宏 (continued)

宏名称	说明	ACLE 规范章节
__ARM_FEATURE_SAT	如果编译器为支持 SSAT/USAT 指令/内在函数的处理器变体生成代码（-mv7m3、-mv7m4、-mv7a8、-mv7r4 和 -mv7r5），则定义为 1；否则未定义。	（第 5.4.8 节）
__ARM_FEATURE_SIMD32	如果编译器为支持所有 SIMD 指令/内在函数的处理器变体生成代码（-mv7m4、-mv7r4 和 -mv7r5），则定义为 1；否则未定义。	（第 5.4.9 节）
__ARM_FEATURE_UNALIGNED	如果编译器为支持对存储器进行无符号访问的处理器变体生成代码（-mv7m3、-mv7m4、-mv7a8、-mv7r4 和 -mv7r5），则定义为 1；否则未定义。	（第 5.4.3 节）
__ARM_FP	对于 --float_support={fpv4spd16 | fpv5spd16}，定义为 6。对于 --float_support={vfpv2 | vfpv3 | vfpv3d16}，定义为 12；否则未定义。	（第 5.5.1 节）
__ARM_FP16_ARGS	如果将 16 位浮点类型可以用于参数和/或结果，则定义为 1；否则未定义。	（第 5.5.11 节）
__ARM_FP16_FORMAT_ALTERNATIVE	未定义	（第 5.5.2 节）
__ARM_FP16_FORMAT_IEEE	如果使用了 IEEE 格式的 16 位浮点（根据 IEEE 754-2008 标准），则定义为 1；否则未定义。	（第 5.5.2 节）
__ARM_FP_FAST	不支持	（第 5.6 节）
__ARM_FP_FENV_ROUNDING	不支持	（第 5.6 节）
__ARM_NEON	未定义	（第 3.4、5.5.4 节）
__ARM_NEON_FP	未定义	（第 5.5.5 节）
__ARM_PCS	如果编译器可以假设转换单元的默认程序调用标准符合 ARM 架构程序调用标准 (AAPCS) 规范中规定的“基础程序调用标准”（-mv7m3、-mv7m4、-mv7r4 和 -mv7r5)，则定义为 1；否则未定义。	（第 5.7 节）
__ARM_PCS_AAPCS64	未定义	（第 5.7 节）
__ARM_PCS_VFP	如果默认程序调用约定是在硬件浮点寄存器中传递浮点参数/返回值，则定义为 1；否则未定义。	（第 5.7 节）
__ARM_ROPI	未定义	（第 5.8 节）
__ARM_RWPI	未定义	（第 5.8 节）
__ARM_SIZEOF_MINIMAL_ENUM	定义为最小的枚举类型大小（packed 为 1 个字节，int 为 4 个字节）。这反映了 --enum_type=[packed | int] 选项，其中 packed 是默认值。	（第 3.1.1 节）
__ARM_SIZEOF_WCHAR_T	如果 --wchar_t=16（默认设置），则定义为 2。如果 --wchar_t=32，则定义为 4。	（第 3.1.1 节）
__ARM_WMMX	未定义	（第 5.5.6 节）
__STDC_IEC_559__	未定义	（第 5.6 节）
__SUPPORT_SNAN__	不支持	（第 5.6 节）

2.5.2 #include 文件的搜索路径

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

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

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

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

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

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

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

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

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

假设 alt.h 的完整路径名是：

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

下表显示了如何调用编译器。选择适用操作系统的命令：

操作系统	输入
UNIX	`armcl --include_path=/tools/files source.c`
Windows	`armcl --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 TI_ARM_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 TI_ARM_C_DIR：

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

或使用以下命令调用编译器：

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

且 file.c 包含以下行：

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

结果是包含的文件位于以下路径中：

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

2.5.3 支持#warning 和 #warn 指令

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

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

2.5.4 生成预处理列表文件（--preproc_only 选项）

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

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

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

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

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

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

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

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

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

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

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

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

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

2.5.10 在文件中生成宏列表（--preproc_macros 选项）

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

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

2.6 将参数传递给 main()

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

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

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

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

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

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

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

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

对于基于 SYS/BIOS 的可执行文件，.args 段中的元素如下：

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

有关更多详细信息，请参阅“Scripting Console”页面。

2.7 了解诊断消息

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

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

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

诊断消息的严重性如下：

- **致命错误**表示问题严重到无法继续编译。此类问题的示例包括命令行错误、内部错误和缺少包含文件。如果正在编译多个源文件，则不会编译当前源文件之后的任何其他源文件。
- **错误**表示违反了 C/C++ 语言的语法或语义规则。编译可以继续，但不会生成目标代码。
- **警告**表示可能有问题但不能证明是错误。例如，编译器会针对未使用的变量发出警告。未使用的变量不会影响程序执行，但它的存在表明您可能有意使用它。编译会继续并生成目标代码（如果没有检测到错误）。
- **备注**不如警告那么严重。它可以表示在极少数情况下存在潜在问题，或者该备注可能只是为了提供参考信息。编译会继续并生成目标代码（如果没有检测到错误）。默认情况下不会发出备注。使用 --issue_remarks 编译器选项可启用备注。
- **建议**提供有关推荐用法的信息。这种类别的提供方式与此处所述的其他诊断类别不同。相反，此类别只在 Code Composer Studio 的“建议”区域可用（此类型是显示在“问题”选项卡旁边的选项卡）。不能通过命令行来控制或访问此建议。提供的建议包括 --opt_level 和 --opt_for_speed 选项的建议设置。另外，此选项卡中还提供了来自 ULP（超低功耗）Advisor 的代码更改建议的相关消息。

诊断消息以类似于以下示例的形式写入标准错误：

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

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

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

必要时，长消息会换行到其他行。

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

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

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

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

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

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

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

没有上下文信息，就很难确定错误指的是什么。

2.7.1 控制诊断消息

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

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

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

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

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

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

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

--issue_remarks
发出默认情况下被抑制的备注（非严重警告）。

--no_warnings
抑制诊断警告（仍会发出错误）。

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

--set_error_limit=*num*	将错误限制设置为 *num*，可以是任何十进制值。在出现此数量的错误后，编译器将放弃编译。（默认为 100。）
--verbose_diagnostics	提供详细的诊断消息，以换行方式显示原始源，并指示错误在源行中的位置。请注意，此命令行选项不能在 Code Composer Studio IDE 中使用。
--write_diagnostics_file	生成具有相同源文件名且扩展名为 *.err* 的诊断消息信息文件。（链接器不支持 --write_diagnostics_file 选项。）请注意，此命令行选项不能在 Code Composer Studio IDE 中使用。

2.7.2 如何使用诊断抑制选项

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

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

如果使用 --quiet 选项调用编译器，结果如下：

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

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

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

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

备注

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

2.8 其他消息

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

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

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

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

交叉引用列表文件中的信息使用以下格式显示：

sym-id name X filename line number column number

sym-id	唯一分配给每个标识符的整数
name	标识符名称
X	以下值之一：
	D 定义
	d 声明（不是定义）
	M 修改
	A 已取地址
	U 已用
	C 已更改（已在单个操作中使用和修改）
	R 任何其他类型的引用
	E 错误；引用是不确定的
filename	源文件
line number	源文件中的行号
column number	源文件中的列号

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

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

原始列表文件包含以下信息：

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

原始列表文件中的每个源代码行都以表 2-32 中列出的标识符之一开头。

表 2-32. 原始列表文件标识符

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

--gen_preprocessor_listing 选项还包括表 2-33 中定义的诊断标识符。

表 2-33. 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 numbercolumn numberdiagnostic

S	表 2-33 中的标识符之一指示诊断的严重程度
filename	源文件
line number	源文件中的行号
column number	源文件中的列号
diagnostic	用于诊断的消息文本

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

2.11 使用内联函数扩展

当调用内联函数时，在调用点插入该函数的 C/C++ 源代码副本。这就是所谓的内联函数扩展，通常称为*函数内联*或简称*内联*。内联函数扩展可以通过消除函数调用开销来加快执行速度。这对于经常被调用的非常小的函数特别

有用。函数内联涉及到在执行速度和代码大小之间进行权衡，因为代码在每个函数调用点都是重复的。在许多位置被调用的大型函数不适合内联。

备注

过多内联会降低性能：过多内联会使编译器显著变慢并降低所生成代码的性能。

以下情况会触发函数内联：

- 使用内置的内在函数运算。内在函数运算看起来像函数调用，即使不存在函数体，也会自动内联。
- 使用 内联 关键字或等效的 __内联 关键字。如果设置 --opt_level=0 或更大值，则使用内联关键字声明的函数可能会被编译器内联。内联关键字是程序员对编译器提出的建议。即使优化级别很高，内联对于编译器来说仍然是可选的。编译器根据函数的长度、函数被调用的次数、--opt_for_speed 设置以及函数中任何不允许函数内联的内容来决定是否内联函数（请参阅节 2.11.2）。如果函数体在同一模块中可见，或者使用了 -pm 且函数在正在编译的模块之一中可见，则可以在 --opt_level=0 或更高级别内联函数。如果包含定义信息的文件和调用点都使用了 --opt_level=4 进行编译，则可以在链接时内联函数。同时定义为静态和内联的函数更有可能被内联。
- 当使用 --opt_level=3 或更高级别时，编译器可能会自动内联符合条件的函数，即使这些函数没有被声明为内联函数也是如此。此过程会用到使用内联关键字显式定义的函数对应列出的相同决策因素列表。有关自动函数内联的更多信息，请参阅节 3.5。
- 除非 --opt_level=off，否则 pragma FUNC_ALWAYS_INLINE（节 5.11.12）和等效的 `always_inline` 属性（节 5.17.2）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 5.11.10）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 和 C11 的严格 ANSI 模式中都受支持。该关键字在 C89 的严格的 ANSI 模式中被禁用，因为它是一种可能与严格遵守标准的程序相冲突的语言扩展。如果要在严格 ANSI C89 模式下定义内联函数，请使用备用关键字 `__inline`。

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

2.11.1 内联内在函数运算符

编译器具有大量的内置函数式运算，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 5.14。除了所列出的这些之外，abs 和 memcpy 也作为内在函数实现。

2.11.2 内联限制

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

如果函数符合以下条件，编译器将保留调用：

- 具有与调用站点不同数量的参数
- 一个参数的类型与相应的调用站点参数不兼容
- 未声明为内联并返回 void 但需要其返回值
- 是代码状态与其调用方不同的 ARM 函数

如果函数具有会给编译器带来困难情形的特性，编译器也不会内联调用：

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

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

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

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

2.12 使用交叉列出功能

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

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

```
armcl --c_src_interlist function
```

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

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

有关将交叉列出功能与优化器一起使用的信息，请参阅节 3.11。使用 --c_src_interlist 选项会导致性能和/或代码大小下降。

以下示例显示了一个典型的交叉列出的汇编文件。

```
_main:
        STMFD     SP!, {LR}
;----------------------------------------------------------------------
;   5 | printf("Hello, world\n");
;----------------------------------------------------------------------
        ADR       A1, SL1
        BL        _printf
;----------------------------------------------------------------------
;   6 | return 0;
;----------------------------------------------------------------------
        MOV       A1, #0
        LDMFD     SP!, {PC}
```

2.13 控制应用程序二进制接口

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

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

ARM 编译器现在仅支持嵌入式应用程序二进制接口 (EABI) ABI，这种接口使用 ELF 目标文件格式和 DWARF 调试格式。如果希望支持传统的 TI_ARM9_ABI 和 TIARM ABI，请使用 ARM v5.2 代码生成工具，并参阅 SPNU151J 和 SPNU118J 的文档。

由 ARM Ltd 创立的行业联盟为用于 ARM 架构的二进制代码定义了一种标准 ABI。此 ABI 称为应用程序二进制接口 (ABI) ，用于第 2 版本的 ARM 架构(ARM ABIv2)。此 ABI 也称为嵌入式应用程序二进制接口 (EABI)。术语 ABIv2 和 EABI 可以互换使用。

更多有关 ABI 的详细信息，请参阅节 5.13。

2.14 VFP 支持

编译器通过 --float_support=*vfp* 选项支持生成向量浮点 (VFP) 协处理器指令。VFP 协处理器在许多 ARM11 及更高版本号可用 。有效的 *vfp* 条目为：

vfpv2	允许为 ARM9E 生成浮点指令。
vfpv3	允许为 Cortex-A8 生成浮点指令。
vfpv3d16	允许为 Cortex-R4 生成浮点指令。
fpv4spd16	允许为 Cortex-M4 生成浮点指令。
无	禁用硬件浮点支持。指定编译器在软件中实现浮点运算。

使用 --silicon_version=7M4 命令行选项会自动设置 --float_support=fpv4spd16 选项。要禁用硬件浮点支持，请使用 --float_support=none 选项。

这是当前对 VFP 的支持：

- 必须将任何 VFP 编译的代码与独立版本的运行时支持库一起链接。有关库命名约定的信息，请参阅节 7.1.9。
- 对于合格的 VFP 参数，编译器遵循 VFP 参数传递和返回调用约定。
- *不包含* 任何带有浮点参数或返回值的函数的目标文件可与 VFP 和非 VFP 文件一起链接。
- *包含* 带有浮点参数或返回值的函数的目标文件只能与使用匹配的 VFP 支持进行编译的对象一起链接。
- 所有手工编码的 VFP 汇编代码都必须遵循 VFP 调用约定和 EABI 约定才能正确编译和链接。除此之外，还必须为 EABI 正确设置 合适的 VFP 构建属性。
- 可使用编译时预定义宏 __TI_VFP_SUPPORT__ 对用户代码进行条件编译/汇编。VFP 专用的用户代码可以使用此宏来确保仅在启用 VFP 时才编译按条件包含的代码。

更多有关 VFPv3 和 VFPv3D16 架构和 ISA 的详细信息，请参阅 ARM 架构手册。更多有关 VFP 调用约定和构建属性的详细信息，请参阅 ARM AAPCS 和 EABI 文档。

2.15 启用入口挂钩和出口挂钩函数

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

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

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

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

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

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

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

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

有关 NO_HOOKS pragma 的信息，请参阅节 5.11.21。

This page intentionally left blank.

章节 3

优化您的代码

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

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

3.1 调用优化 54
3.2 控制代码大小与速度 55
3.3 执行文件级优化（--opt_level=3 选项） 55
3.4 程序级优化（--program_level_compile 和 --opt_level=3 选项） 56
3.5 自动内联扩展（--auto_inline 选项） 58
3.6 链接时优化（--opt_level=4 选项） 58
3.7 使用反馈制导优化 59
3.8 使用配置文件信息分析代码覆盖率 62
3.9 访问优化代码中的别名变量 64
3.10 在优化代码中谨慎使用 asm 语句 64
3.11 通过优化使用交叉列出特性 64
3.12 调试和分析优化代码 65
3.13 正在执行什么类型的优化？ 66

3.1 调用优化

C/C++ 编译器能够执行各种优化，这些优化由优化器和代码生成器执行：

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

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

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

- **--opt_level=off** 或 **-Ooff**
  - 不执行优化
- **--opt_level=0** 或 **-O0**
  - 执行控制流图简化（节 3.13.3）
  - 执行循环旋转（节 3.13.10）
  - 消除未使用的代码
  - 简化表达式和语句（节 3.13.5）
  - 扩展对声明的内联函数的调用（节 3.13.6）
- **--opt_level=1** 或 **-O1** 执行所有 --opt_level=0 (-O0) 优化，加上：
  - 执行本地复制/常量传播（节 3.13.4）
  - 删除未使用的赋值（节 3.13.4）
  - 消除局部公用表达式
- **--opt_level=2** 或 **-O2** 执行所有 --opt_level=1 (-O1) 优化，加上：
  - 执行循环优化
  - 消除全局公用子表达式（节 3.13.4）
  - 消除全局未使用的赋值（节 3.13.4）
  - 执行循环展开（节 5.11.31）
- **--opt_level=3** 或 **-O3** 执行所有 --opt_level=2 (-O2) 优化，加上：
  - 删除所有从未调用过的函数（节 3.4）
  - 简化返回值从未使用过的函数（节 3.4）
  - 内联函数对小函数的调用（节 2.11 和节 3.5）
  - 重新排序函数声明；当调用方被优化后，被调用函数的属性是已知的
  - 当所有调用在相同的参数位置传递相同的值时，将参数传播到函数体中
  - 识别文件级变量特征（节 3.4）
  - 执行其他优化（节 3.3 和节 3.4）
- **--opt_level=4** 或 **-O4**
  - 执行链接时优化。(节 3.6)

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

调试默认启用，并且优化级别不受调试信息生成的影响。

3.2 控制代码大小与速度

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

- --opt_for_speed=0

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

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

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

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

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

  优化能够恶化或影响代码大小的风险*较为 高* 的代码性能/速度。

如果未使用参数指定 --opt_for_speed 选项，则默认设置为 --opt_for_speed=4。如果未指定 --opt_for_speed 选项，则默认设置为 1

当将 --opt_for_speed 设置为级别 1 或 2 时，可以观察到缓存器件具有最佳性能。

3.3 执行文件级优化（--opt_level=3 选项）

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

表 3-1. 可与 --opt_level=3 结合使用的选项

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

3.3.1 创建优化信息文件（--gen_opt_info 选项）

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

表 3-2. 为 --gen_opt_info 选项选择一个级别

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

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

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

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

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

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

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

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

备注

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

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

3.4.1 控制程序级优化（--call_assumptions 选项）

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

表 3-3. 为 --call_assumptions 选项选择一个级别

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

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

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

如果 --call_assumptions 为...	在以下条件下...	则 --call_assumptions 级别...
未指定	指定了 --opt_level=3 优化级别	默认为 --call_assumptions=2
未指定	编译器在 --opt_level=3 优化级别下发现对外部函数的调用	恢复为 --call_assumptions=0
未指定	未定义 main	恢复为 --call_assumptions=0

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

如果 --call_assumptions 为...	在以下条件下...	则 --call_assumptions 级别...
--call_assumptions=1 或 --call_assumptions=2	没有将 main 定义为入口点的函数，也没有定义中断函数，也 没有由 FUNC_EXT_CALLED pragma 标识的函数	恢复为 --call_assumptions=0
--call_assumptions=1 或 --call_assumptions=2	定义了 main 函数，*或* 定义了中断函数，*或* 者用 FUNC_EXT_CALLED pragma 标识了函数	保留 --call_assumptions=1 或 --call_assumptions=2
--call_assumptions=3	任何条件下	保留 --call_assumptions=3

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

3.4.2 混合 C/C++ 和汇编代码时的优化注意事项

如果程序中有任何汇编函数，则在使用 --program_level_compile 选项时，请谨慎操作。编译器只识别 C/C++ 源代码，而不识别任何可能存在的汇编代码。由于编译器无法识别对 C/C++ 函数的汇编代码调用和变量修改，因此 --program_level_compile 选项会优化这些 C/C++ 函数。要保留这些函数，请将 FUNC_EXT_CALLED pragma（请参阅节 5.11.14）放在对要保留的函数的任何声明或引用之前。

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

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

如果您的应用程序出现以下任一情况，请使用建议的解决方案：

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

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

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

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

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

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

3.5 自动内联扩展（--auto_inline 选项）

当使用 --opt_level=3 选项（别名为 -O3）进行优化时，编译器自动内联小函数。命令行选项 --auto_inline=*size* 指定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。
- 在 --opt_level=4 时，如果编译的是性能时，则编译器会主动地自动内联。

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

备注

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 2.11.2 中的内联限制。

备注

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

备注

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

3.6 链接时优化（--opt_level=4 选项）

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

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

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

- 每个源文件都可以单独编译。程序级编译的一个问题是其要求所有源文件都要一次性传递给编译器。这通常需要对客户的构建过程进行重大修改。使用链接时优化，所有文件都可以单独编译。
- 自动处理对程序集的 C/C++ 符号的引用。在进行程序级编译时，编译器不知道符号是否被外部引用。当在最后一个链接中执行链接时优化时，链接器可以确定哪些符号被外部引用，并在优化过程中防止消除这些符号。

- 第三方目标文件可以参与优化。如果第三方供应商提供了使用 --opt_level=4 选项编译的目标文件，这些文件将与用户生成的文件一起参与优化。这包括作为 TI 运行时支持的一部分提供的目标文件。未使用 –opt_level=4 编译的目标文件仍可在执行链接时优化的链接中使用。未使用 –opt_level=4 进行编译的那些文件则不参与优化。
- 可以使用不同的选项集编译源文件。对于程序级编译，必须使用相同的选项集编译所有源文件。借助链接时优化，可以使用不同的选项来编译文件。如果编译器确定两个选项不兼容，就会发出错误。

3.6.1 选项处理

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

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

3.6.2 不兼容的类型

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

3.7 使用反馈制导优化

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

3.7.1 反馈向导优化

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

3.7.1.1 第 1 阶段 - 收集程序分析信息

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

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

3.7.1.2 第 2 阶段 - 使用应用程序分析信息进行优化

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

编译器使用分析反馈文件中的数据来指导对频繁执行的程序路径进行某些优化。

3.7.1.3 生成和使用配置文件信息

有两个选项可以控制反馈定向优化：

--gen_profile_info	告知编译器添加检测代码以收集配置文件信息。当程序执行 run-time-support exit() 函数时，配置文件数据会被写入 PDAT 文件。此选项适用于在命令行上编译的所有 C/C++ 源文件。 如果设置了主机上的环境变量 TI_PROFDATA，则将数据写入指定的文件中。否则，它使用默认文件名：pprofout.pdat。可以使用 TI_PROFDATA 主机环境变量指定 PDAT 文件的完整路径名（包括目录名）。 默认情况下，RTS 配置文件数据输出例程使用 C I/O 机制将数据写入 PDAT 文件。您可以为 PPHNDL 器件安装器件处理程序，以将配置文件数据重定向到自定义器件驱动程序例程。例如，这可用于将配置文件数据发送到不使用文件系统的器件。 反馈定向优化要求您在使用 --gen_profile_info 选项时至少打开一些调试信息。这使编译器能够输出调试信息，以允许 armpdd 关联已编译的函数及其相关配置文件数据。
--use_profile_info	指定用于执行反馈定向优化的第 2 阶段的配置文件信息文件。可以在命令行上指定多个配置文件信息文件；编译器使用来自多个信息文件的所有输入数据。此选项的语法为： **--use_profile_info=***file1*[, *file2*, ..., *filen*] 如果未指定文件名，编译器将在调用编译器的目录中查找名为 pprofout.prf 的文件。

3.7.1.4 反馈制导优化的应用示例

这些步骤说明了反馈制导优化的创建和应用。

1. 生成分析信息。

```
armcl --opt_level=2 --gen_profile_info foo.c --run_linker --output_file=foo.out
     --library=lnk.cmd --library=rtsv4_A_be_eabi.lib
```

2. 执行应用程序。

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

3. 处理分析数据。

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

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

4. 使用分析反馈文件重新编译。

```
armcl --opt_level=2 --use_profile_info=pprofout.prf foo.c --run_linker
     --output_file=foo.out --library=lnk.cmd --library=rtsv4_A_be_eabi.lib
```

3.7.1.5 .ppdata 段

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

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

3.7.1.6 反馈制导优化和代码大小调整

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

3.7.1.7 检测程序执行开销

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

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

3.7.1.8 无效的分析数据

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

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

3.7.2 分析数据解码器

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

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

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

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

备注

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

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

3.7.3 反馈制导优化 API

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

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

3.7.4 反馈制导优化总结

选项

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

--analyze_only	仅生成代码覆盖率信息文件。必须与 --use_profile_info 一起使用。同时指定 --analyze=codecov 和 --analyze_only 才能对检测的应用程序进行代码覆盖率分析。

主机环境变量

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

API

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

创建的文件

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

3.8 使用配置文件信息分析代码覆盖率

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

3.8.1 代码覆盖

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

3.8.1.1 第 1 阶段 - 收集程序分析信息

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

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

3.8.1.2 第 2 阶段 -- 生成代码覆盖信息报告

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

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

可以指定两个环境变量来控制代码覆盖信息文件的目标。

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

如果代码覆盖数据文件已存在，编译器会在文件末尾附加新数据集。

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

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

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

完整的文件名、函数名和注释用引号 (") 引起来。例如：

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

可使用其他工具（例如电子表格程序）来格式化和查看代码覆盖数据。

3.8.2 相关的特征和功能

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

3.8.2.1 路径分析器

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

armcl --gen_profile ... *file.c*
armcl --use_profile ... *file.c*

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

3.8.2.2 分析选项

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

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

--analyze=codecov	指示编译器生成代码覆盖率分析信息。此选项取代了先前的 --codecov 选项。
--analyze_only	在分析信息生成完成后停止编译。

3.8.2.3 环境变量

为了协助管理输出 CSV 分析文件，armpprof 支持以下环境变量：

TI_ANALYSIS_DIR	指定输出分析文件将在其内生成的目录。

3.9 访问优化代码中的别名变量

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。优化器会分析代码以确定哪里可以出现别名，哪里或不可以出现别名，然后在保持程序正确性的同时尽可能进行优化。优化器谨慎执行其行为。如果两个指针有可能指向同一个对象，那么优化器就会假设这两个指针确实指向同一个对象。

编译器认为，如果将局部变量的地址传递给某个函数，则该函数会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，被调用的函数不能将局部变量的地址分配给全局变量或返回局部变量的地址。如果此假设无效，，请使用 --aliased_variables 编译器选项强制编译器采用最坏情况下的别名。在最坏情况下的别名中，任何间接引用都可以引用这样的变量。

3.10 在优化代码中谨慎使用 asm 语句

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

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

3.11 通过优化使用交叉列出特性

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

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

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

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

备注

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

当 --c_src_interlist 和 --optimizer_interlist 选项与优化一起使用时，编译器会插入其注释，并且交叉列出功能在汇编器之前运行，从而将原始 C/C++ 源代码合并到汇编文件中。

例如，假设下述 C 代码是使用优化 (--opt_level=2) 和 --optimizer_interlist 选项编译的：

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

汇编文件包含与汇编代码交叉列出的编译器注释。

```
_main:
        STMFD     SP!, {LR}
;** 5-----------------------    printf("Hello, world\n");
        ADR       A1, SL1
        BL        _printf
;** 6-----------------------    return 0;
        MOV       A1, #0
        LDMFD     SP!, {PC}
```

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

```
_main:
        STMFD     SP!, {LR}
;** 5-----------------------    printf("Hello, world\n");
;------------------------------------------------------------------------------
;   5 | printf("Hello, world\n");
;------------------------------------------------------------------------------
        ADR       A1, SL1
        BL        _printf
;** 6-----------------------    return 0;
;------------------------------------------------------------------------------
;   6 | return 0;
;------------------------------------------------------------------------------
        MOV       A1, #0
        LDMFD     SP!, {PC}
```

3.12 调试和分析优化代码

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

默认的优化级别--symdebug:dwarf (-g) 选项的使用。如果指定了 --symdebug:dwarf，则默认的优化级别为 off。否则，默认的优化级别为 3。

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

3.12.1 分析优化的代码

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

3.13 正在执行什么类型的优化？

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

优化	请参阅
基于成本的寄存器分配	节 3.13.1
别名消歧	节 3.13.2
分支优化和控制流简化	节 3.13.3
数据流优化 • 复制传播 • 通用子表达式消除 • 冗余分配消除	节 3.13.4
表达式简化	节 3.13.5
函数的内联扩展	节 3.13.6
函数符号别名	节 3.13.7
归纳变量和强度降低	节 3.13.8
循环不变量代码运动	节 3.13.9
循环旋转	节 3.13.10
指令调度	节 3.13.11

ARM 专用优化	请参阅
尾部合并	节 3.13.12
自动增量寻址	节 3.13.13
块条件化	节 3.13.14
结语内联	节 3.13.15
将比较值删除为 0	节 3.13.16
带常数除数的整数除法	节 3.13.17
分支链接	节 3.13.18

3.13.1 基于成本的寄存器分配

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

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

3.13.2 别名消歧

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

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

3.13.3 分支优化和控制流简化

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

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

这种类型的优化通过 --opt_level=0 和更高的优化设置来启用。

3.13.4 数据流优化

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

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

3.13.5 表达式简化

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

这种类型的优化通过 --opt_level=0 和更高的优化设置来启用。

3.13.6 函数的内联扩展

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

这种类型的优化通过 --opt_level=0 和更高的优化设置来启用。

3.13.7 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 5.17.2。

3.13.8 归纳变量和强度降低

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

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

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

这种类型的优化通过 --opt_level=2 和更高的优化设置来启用。

3.13.9 循环不变量代码运动

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

3.13.10 循环旋转

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

这种类型的优化通过 --opt_level=0 和更高的优化设置来启用。

3.13.11 指令排程

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

3.13.12 尾部合并

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

3.13.13 自动增量寻址

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

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

3.13.14 块条件化

因为所有 32 位指令都可以是有条件的，所以可以对指令进行条件化去除分支。

在示例 3-1 中，通过简单地对加法和减法进行条件化，可以去除加法和减法周围的分支。

示例 3-1. 块条件化 C 源代码

```
int main(int a)
{
   if (a < 0)
      a = a-3;
   else
      a = a*3;
   return ++a;
}
```

示例 3-2. 示例 3-1 的 C/C++ 编译器输出

```
;*****************************************************
;* FUNCTION DEF: _main                               *
;*****************************************************
_main:
          CMP       A1, #0
          ADDPL     A1, A1, A1, LSL #1
          SUBMI     A1, A1, #3
          ADD       A1, A1, #1
          BX        LR
```

3.13.15 结语内联

如果函数的结语是单个指令，则该指令将替换结语的所有分支。这是通过删除分支来提高执行速度。

3.13.16 删除与零的比较

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

3.13.17 用常数除数进行整数除法

优化器尝试用常数除数重写整数除法运算。整数除法将重写为乘以除数的倒数。这种情况发生在优化级别 2 (--opt_level=2 或 -O2) 及更高级别上。还必须使用 --opt_for_speed 选项进行编译。该选项将选择编译表示速度。

3.13.18 分支链接

跳转到所需目标的分支称为“分支链接”。仅在 16-BIS 模式下支持分支链接。考虑以下代码序列：

```
LAB1:   BR  L10
        ....
LAB2:   BR  L10
        ....
L10:
```

如果 L10 离 LAB1 很远（大偏移量），汇编器会将 BR 转换为无条件分支和其周围分支的序列，从而产生四个或六个字节长的两条指令。相反，如果 LAB1 处的分支可以跳转到 LAB2，并且距离 LAB2 足够近，以至于 BR 可以被单个短分支指令替换，则生成的代码会更小，因为 LAB1 中的 BR 将被转换为一条两个字节长的指令。如果 L10 离 LAB2 太远，LAB2 会跳转到另一个分支。因此，分支链接可以扩展到任意深度。

当在 thumb 模式下 (--code_state=16) 对代码大小（未使用 --opt_for_speed）进行编译时，编译器生成两条伪指令：

- BTcc 取代了 BRcc。格式为 **BRcc** *target*, #[*depth*]。

  #depth 是可选参数。如果未指定 深度，则将其设置为默认的分支链接深度。如果已指定，则此分支指令的分支链接深度设置为 #depth。如果 #depth 小于零，则汇编器发出警告，并将此指令的分支链深度设置为零。
- BQcc 取代了 Bcc。格式为 **BQcc** *target*, #[*depth*]。

  #depth 已 BTcc 伪指令的相同。

BT 伪指令取代了 BR（伪分支）指令。同样，BQ 取代了 B。如果启用了分支链接，汇编器会为这些指令执行分支链优化。汇编器将 BT 和 BQ 跳转目标替换为这些指令跳转到的分支的偏移量。

默认分支链接深度为 10。此限制旨在防止较长的分支链对性能造成影响。

可以在汇编语言程序中使用 BT 和 BQ 指令使汇编器能够执行分支链接。可以通过指定（可选）*#depth* 参数来控制每条指令的分支链接深度。必须使用 BR 和 B 指令来防止任何 BT 或 BQ 分支的分支链接。

章节 4

链接 C/C++ 代码

C/C++ 代码生成工具提供了两种链接程序的方法：

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

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

4.1 通过编译器调用链接器（-z 选项） 72
4.2 链接器代码优化 74
4.3 控制链接过程 74

4.1 通过编译器调用链接器（-z 选项）

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

4.1.1 单独调用链接器

将 C/C++ 程序作为单独步骤进行链接的一般语法如下：

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

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

备注

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

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

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

```
armcl --run_linker --rom_model prog1 prog2 prog3 --output_file=prog.out
     --library=rtsv4_A_be_eabi.lib
```

4.1.2 调用链接器作为编译步骤的一部分

在编译步骤中链接 C/C++ 程序的一般语法如下：

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

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

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

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

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

```
armcl prog1.c prog2.c prog3.c --run_linker --rom_model --output_file=prog.out
     --library=rtsv4_A_be_eabi.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 armcl --run_linker 时，默认情况下会在运行时使用 **--rom_model** 进行变量的自动初始化。有关使用 --rom_model 和 --ram_model 选项的详细信息，请参阅节 4.3.5。

备注

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

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

4.1.3 禁用链接器（--compile_only 编译器选项）

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

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

4.2 链接器代码优化

这些技术用于进一步优化您的代码。

4.2.1 生成死函数列表（--generate_dead_funcs_list 选项）

为了便于删除未使用的代码，链接器生成一个反馈文件，其包含从未被引用的函数列表。下次编译源文件时必须使用该反馈文件。--generate_dead_funcs_list 选项的语法如下：

--generate_dead_funcs_list= *filename*

如果未指定 *filename*，则会使用 dead_funcs.txt 的默认文件名。

正确创建和使用反馈文件需要以下步骤：

1. 使用 --gen_func_subsections 编译器选项编译所有的源文件。例如：

```
armcl file1.c file2.c --gen_func_subsections
```

2. 在链接器中，使用 --generate_dead_funcs_list 选项根据生成的目标文件生成反馈文件。例如：

```
armcl --run_linker file1.c.obj file2.c.obj --generate_dead_funcs_list=feedback.txt
```

   或者，可以将步骤 1 和 2 合并为一个步骤。如果这样做，编译源文件时不需要指定 --gen_func_subsections，因为这会自动完成。例如：

```
armcl file1.c file2.c --run_linker --generate_dead_funcs_list=feedback.txt
```

3. 一旦获得反馈文件后，重新编译源代码。使用 --use_dead_funcs_list 选项将反馈文件提供给编译器。此选项强制文件中列出的每个死函数放入其自己的子段。例如：

```
armcl file1.c file2.c --use_dead_funcs_list=feedback.txt
```

4. 使用新创建的目标文件调用链接器。链接器删除子段。例如：

```
armcl --run_linker file1.c.obj file2.c.obj
```

   或者，可以将步骤 3 和 4 合并为一个步骤。例如：

```
armcl file1.c file2.c --use_dead_funcs_list=feedback.txt --run_linker
```

备注

死函数反馈：严格控制使用 --generate_dead_funcs_list 生成的反馈文件的格式。该文件必须由链接器生成才能被编译器正确处理。该文件的格式可能会随着时间的推移而改变，因此该文件包含一个版本格式号以支持向后兼容。

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

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

如果未使用此选项，则默认值为“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。

4.3 控制链接过程

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

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

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

4.3.1 包含运行时支持库

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

4.3.1.1 自动选择运行时支持库

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

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

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

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

示例 4-1. 使用 --issue_remarks 选项

```
armcl --code_state=16 --issue_remarks main.c --run_linker --rom_model
<Linking>
remark: linking in "libc.a"
remark: linking in "rtsv4_A_be_eabi.lib" in place of "libc.a"
```

4.3.1.2 手动选择运行时支持库

通过显式指定要使用的所需运行时支持库，可以避开自动选择库。使用 --library 链接器选项指定库的名称。链接器将搜索由 --search_path 选项指定的路径，然后搜索命名库的 TI_ARM_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。

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

4.3.1.3 用于搜索符号的库顺序

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

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

4.3.2 运行时初始化

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

1. 切换到用户模式并建立用户模式栈
2. 设置状态寄存器和配置寄存器

3. 设置栈
4. 处理特殊二进制符号复制表（若存在）。
5. 处理运行时初始化表以自动初始化全局变量（使用 --rom_model 选项时）
6. 调用所有全局构造函数
7. 调用 main() 函数
8. 当 main() 返回时调用 exit()

备注

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

4.3.3 Cinit 的初始化和看门狗计时器保持

可以使用 --cinit_hold_wdt 选项指定在 cinit 自动初始化期间看门狗计时器是否应保持 (on) 或不保持 (off)。设置此选项会导致 RTS 自动初始化例程被连接到程序中，以处理所需的看门狗计时器行为。

4.3.4 全局对象构造函数

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

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

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

节 6.10.3.6 讨论了 EABI 模式的全局构造函数表的格式。

4.3.5 指定全局变量初始化类型

C/C++ 编译器生成用于初始化全局变量的数据表。节 6.10.3.4 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 6.10.3.3）。
- 在*加载时*初始化全局变量。使用 --ram_model 链接器选项（请参阅节 6.10.3.5）。

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 4.1）。

有关 EABI 使用 --rom_model 和 --ram_model 的链接约定的信息，请分别参阅节 6.10.3.3 和节 6.10.3.5。

备注

引导加载程序：加载器不包含在 C/C++ 编译器工具中。可将 ARM 模拟器或仿真器与源代码调试器一起用作加载器。有关启动加载的更多信息，请参阅 *ARM 汇编语言工具用户指南* 中的“程序加载和运行”一章。

4.3.6 指定在内存中分配段的位置

编译器生成可重定位的代码块和数据块。这些块，称为*段*，以各种方式分配在内存中，以符合各种系统配置。有关编译器如何使用这些段的完整说明，请参阅节 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 4-1 总结了初始化段。表 4-2 总结了未初始化段。

表 4-1. 由编译器创建的初始化段

名称	内容
.binit	引导时间复制表（有关链接器命令文件中 BINIT 的信息，请参阅*汇编语言工具用户指南*。）
.cinit	用于显式初始化全局和静态变量的表。
.const	显式初始化的全局和静态常量变量。
.data	显式初始化的全局和静态非常量变量。
.init_array	启动时要调用的构造函数表。
.ovly	复制除引导时间 (.binit) 复制表以外的表。只读数据。
.text	可执行代码和常量。还包含字符串文字和切换表。有关例外情况，请参阅节 6.1.1。
.TI.crctab	生成的 CRC 校验表。只读数据。

表 4-2. 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	栈
.sysmem	用于动态内存分配（malloc 等）的内存池（堆）

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

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 *ARM 汇编语言工具用户指南*。

4.3.7 链接器命令文件示例

链接器命令文件 显示了一个链接 32 位 C 程序的典型链接器命令文件。本示例中的命令文件名为 lnk32.cmd，且该命令文件列出了几个链接选项：

--rom_model	通知链接器在运行时使用自动初始化功能
--stack_size	通知链接器将 C 栈大小设置为 0x8000 字节
--heap_size	通知链接器将堆大小设置为 0x2000 字节

要链接该程序，请使用以下语法：

armcl --run_linker *object_file*(s) **--output_file** *outfile* **--map_file** *mapfile* **lnk32.cmd**

链接器命令文件

```
--rom_model                                 /* LINK USING C CONVENTIONS      */
--stack_size=0x8000                         /* SOFTWARE STACK SIZE           */
--heap_size=0x2000                          /* HEAP AREA SIZE                */
/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
    P_MEM    : org = 0x00000000   len = 0x00030000   /* PROGRAM MEMORY (ROM) */
    D_MEM    : org = 0x00030000   len = 0x00050000   /* DATA MEMORY    (RAM) */
}
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
SECTIONS
{
    .intvecs : {} > 0x0                 /* INTERRUPT VECTORS                 */
    .bss     : {} > D_MEM               /* GLOBAL & STATIC VARS              */
    .sysmem  : {} > D_MEM               /* DYNAMIC MEMORY ALLOCATION AREA    */
    .stack   : {} > D_MEM               /* SOFTWARE SYSTEM STACK             */
    .text    : {} > P_MEM               /* CODE                              */
    .cinit   : {} > P_MEM               /* INITIALIZATION TABLES             */
    .const   : {} > P_MEM               /* CONSTANT DATA                     */
    .pinit   : {} > P_MEM               /* TEMPLATE INITIALIZATION TABLES    */
```

章节 5

C/C++ 语言实现

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

5.1 ARM C 的特征 80
5.2 ARM C++ 的特征 84
5.3 使用 MISRA C 2004 85
5.4 使用 ULP Advisor 86
5.5 数据类型 87
5.6 文件编码和字符集 89
5.7 关键字 89
5.8 C++ 异常处理 92
5.9 寄存器变量和参数 92
5.10 __asm 语句 94
5.11 pragma 指令 95
5.12 _Pragma 运算符 114
5.13 应用程序二进制接口 115
5.14 ARM 指令内在函数 115
5.15 目标文件符号命名规则（链接名） 123
5.16 更改 ANSI/ISO C/C++ 语言模式 123
5.17 GNU 、Clang 和 ACLE 语言扩展 126
5.18 AUTOSAR 132
5.19 编译器限制 132

5.1 ARM 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 5.17）。

在支持 C89 的默认宽松 ANSI 模式下，编译器支持 C99 和 C11 的某些功能。它支持 C99 模式下 C99 的所有语言功能以及 C11 模式下 C11 的所有语言功能。请参阅节 5.16。

在宽松 ANSI 模式下（默认情况下为打开）和 C11 模式下支持 C11 中的原子操作，如下所示：

- 在 ARM V7A8 (Cortex-A8)、ARM V7M3 (Cortex-M3)、ARM V7M4 (Cortex-M4)、ARM V7R4 (Cortex-R4) 和 ARM V7R5 (Cortex-R5) 上，原子操作使用处理器支持的独占访问指令来实现。
- 在 ARM V6M0 (Cortex-M0) 上，原子操作是通过禁用运算中的中断来实现的。
- 在 ARM V4 (ARM7)、ARM V5e (ARM9E) 和 ARM V6 (ARM11) 上，不支持原子操作。

此外，编译器还支持 ARM C 语言扩展 (ACLE) 规范中描述的许多功能。这些功能适用于 Cortex-M 和 Cortex-R 处理器变体。ACLE 支持会影响您可能在 C/C++ 代码中使用的预定义宏命令（表 2-31）、函数属性（节 5.17.2）和内在函数（节 5.14）。实现这些功能以支持源代码开发，这些源代码可以使用多个供应商提供的 ACLE 兼容编译器为各种 ARM 处理器进行编译。

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

不受支持的 C 库功能包括：

- 运行时库对宽字符的支持很少。类型 wchar_t 实现为 unsigned short（16 位），但如果设置 --wchar_t=32 选项，也可以是 int。宽字符集相当于 char 类型的值集。该库包含头文件 <wchar.h> 和 <wctype.h>，但不包含标准中指定的所有函数。请参阅节 5.6，了解有关扩展字符集和多字节字符集的信息。
- 运行时库包含头文件 <locale.h>，但极少实现。唯一受支持的区域设置是 C 区域设置。也就是说，指定为随区域设置而变化的库行为被硬编码为 C 区域设置的行为，并且通过调用 setlocale() 来安装不同的区域设置的尝试将返回 NULL。
- 不支持 C99/C11 规范中的某些运行时函数和功能。请参阅节 5.16。

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 5.6 (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 5.6。(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 5.5。(3.6)
- 执行字符集与基本执行字符集相同，为纯 ASCII。还支持 ISO 8859 扩展字符集中的字符。(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

转义序列	ASCII 含义	整数值
\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 unsigned char，但可以通过 --plain_char=signed 选项更改为 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 5.6。（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“舍入为偶数”。(6.3.1.4)
- 浮点数转换为更窄浮点数时的舍入方向是 IEEE-754“舍入到偶数”。(6.3.1.5)
- 对于不能准确表示的浮点常量，实现方案使用最接近的可表示值。(6.4.4.2)
- 编译器不会收缩浮点表达式。(6.5)
- FENV_ACCESS pragma 的默认状态为“关闭”。(7.6.1)
- TI 编译器不会定义任何额外的浮点异常。（7.6，7.12）
- FP_CONTRACT pragma 的默认状态为“关闭”。(7.12.2)
- 如果舍入结果等于数学结果，则不会产生“不精确”的浮点异常。(F.9)
- 如果结果很小但不是不精确，则不会产生“下溢”和“不精确”的浮点异常。(F.9)

J.3.7 数组和指针

- 在将指针转换为整数或将整数转换为指针时，该指针被视为是具有相同大小的无符号整数，并且适用普通整数转换规则。
- 在将指针转换为整数或将长整型转换为指针时，如果目标的按位表示可以保存源命令的按位表示中的所有位，则这些位被精确复制。(6.3.2.3)
- 两个指向同一数组元素的指针相减的结果大小就是 ptrdiff_t 的大小，如节 5.5 中所定义。(6.5.6)

J.3.8 提示

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

J.3.9 结构体、联合体、枚举和位字段

- “plain”整数位字段会被视为有符号整数位字段。（6.7.2，6.7.2.1）
- 除了 _Bool、signed int 和 unsigned int，编译器还允许将 char、signed char、unsigned char、signed short、unsigned shot、signed long、unsigned long、signed long long、unsigned long long 和枚举类型作为位字段类型。(6.7.2.1)

- 位字段不能跨越存储单元边界。(6.7.2.1)
- 位字段在一个单元内按字节顺序分配。请参阅节 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 5.5.1 中所述。(6.7.2.2)

J.3.10 限定符

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

J.3.11 预处理指令

- Include 指令可能为以下两种形式之一，" " 或 < >。对于这两种形式，编译器将使用头文件搜索路径通过该名称查找磁盘上的真实文件。请参阅节 2.5.2。(6.4.7)
- 控制条件包含的常量表达式中的字符常量的值与执行字符集中的同一字符常量的值相匹配（两者都是 ASCII）。(6.10.1)
- 编译器使用文件搜索路径来搜索包含的 < > 分隔的头文件。请参阅节 2.5.2。(6.10.2)
- 编译器使用文件搜索路径来搜索包含的 " " 分隔的头文件。请参阅节 2.5.2。(6.10.2)
- #include 处理没有任意的嵌套限制。(6.10.2)
- 有关公认的非标准 pragma 的说明，请参阅节 5.11。(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 5.16.1 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 5.5 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 ARM C++ 的特征

根据 ANSI/ISO/IEC 14882:2014 标准 (C++14) 中的定义，ARM 编译器支持 C++，包括以下特性：

- 支持完整的 C++ 标准库，但具有以下例外情况。
- 模板
- 异常，通过 --exceptions 选项启用；请参阅节 5.8。
- 运行时类型信息 (RTTI)，可通过 --rtti 编译器选项启用。

编译器支持 ISO 标准化的 2014 年标准 C++。但是，以下特性*未*实现或完全受支持：

- 编译器不支持嵌入式 C++ 运行时支持库。
- 此库支持宽字符 (wchar_t)，因为为字符定义的模板函数和类也适用于 wchar_t。例如，实现了宽字符流类 wios、wiostream、wstreambuf 等（对应于字符类 ios、iostream、streambuf）。但是，没有用于宽字符的低级别文件 I/O。此外，C 库接口对宽字符的支持（通过 C++ 标头 <cwchar> 和 <cwctype>）是有限的，如上面 C 库中所述。
- 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）。
- 不支持强比较和交换（在 C++11 标准中引入）。
- 不支持双向围栏（在 C++11 标准中引入）。
- 不支持内存模型（在 C++11 标准中引入）。
- 不支持传播异常（在 C++11 标准中引入）。
- 不支持线程本地存储（在 C++11 标准中引入）。
- 不支持动态初始化和并发销毁（在 C++11 标准中引入）。

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

5.3 使用 MISRA C 2004

MISRA C 是面向 C 编程语言的一套软件开发指南，它用于促进与道路车辆中安全相关电子系统和其他嵌入式系统的最佳开发实践。MISRA C 最初于 1998 年由汽车工业软件可靠性协会发布，此后在各行各业得到广泛采用。MISRA C:2004 作为对该指南的后续更新发布

您可以更改代码，以符合 MISRA C:2004 规则。以下选项和 pragma 可用于启用/禁用规则：

- --check_misra 选项启用对指定 MISRA C:2004 规则的检查。如果您希望使用 CHECK_MISRA 和 RESET_MISRA pragma 来进一步控制检查，则必须使用此编译器选项。
- CHECK_MISRA pragma 在源代码级别启用/禁用 MISRA C:2004 规则。请参阅节 5.11.2。
- RESET_MISRA pragma 会将指定的 MISRA C:2004 规则重置为处理 CHECK_MISRA pragma 之前的状态。请参阅节 5.11.25。

选项和 pragma 的语法为：

--check_misra={all|required|advisory|none|*rulespec*}
#pragma CHECK_MISRA ("{all|required|advisory|none|*rulespec*}**")**
#pragma RESET_MISRA ("{all|required|advisory|*rulespec*}**")**

rulespec 参数是要启用或禁用的以逗号分隔的规则编号列表。

示例：--check_misra=1.1,1.4,1.5,2.1,2.7,7.1,7.2,8.4
- 启用 1.1、1.4、1.5、2.1、2.7、7.1、7.2 和 8.4 规则的检查。

示例：#pragma CHECK_MISRA("-7.1,-7.2,-8.4")
- 禁用规则 7.1、7.2 和 8.4 的检查。

一种典型应用场景是在命令行中使用 --check_misra 选项，指定在多数代码中应检查的规则。然后使用带 *rulespec* 的 CHECK_MISRA pragma，针对特定代码区域启用或停用特定规则。

以下两个选项可控制特定 MISRA C:2004 规则的严重性：

- --misra_required 选项设置必查 MISRA C:2004 规则的诊断严重性。
- --misra_advisory 选项设置推荐 MISRA C:2004 规则的诊断严重性。

这些选项的语法为：

--misra_advisory={error|warning|remark|suppress}
--misra_required={error|warning|remark|suppress}

5.4 使用 ULP Advisor

您可以从 ULP（超低功耗）Advisor 获得有关代码的反馈。有关 ULP 规则的列表和描述，请参阅 www.ti.com/ulpadvisor。您可以使用以下任一方式启用/禁用规则。允许在命令行中使用多个 --advice 选项。

- --advice:power 选项可指定要检查的规则。
- --advice:power_severity 选项可指定违反 ULP Advisor 规则的情况是错误、警告、备注还是未报告。
- CHECK_ULP pragma 在源代码级别启用/禁用 ULP Advisor 规则。此 pragma 具有与使用 --advice:power 选项相同的效果。请参阅节 5.11.3。
- RESET_ULP pragma 将指定的 ULP Advisor 规则重置为处理任何 CHECK_ULP pragmas 之前的状态。请参阅节 5.11.26。

--advice:power 选项支持检查指定的 ULP Advisor 规则。语法为：

--advice:power={all|none|*rulespec*}

rulespec 参数是要启用的以逗号分隔的规则编号列表。例如 `--advice:power=1.1,7.2,7.3,7.4` 启用 1.1、7.2、7.3 和 7.4 规则。

--advice:power_severity 选项设置 ULP Advisor 规则的诊断严重性。语法为：

--advice:power_severity={error|warning|remark|suppress}

pragmas 的语法为：

#pragma CHECK_ULP ("{all|none|*rulespec*}**")**
#pragma RESET_ULP ("{all|*rulespec*}**")**

5.5 数据类型

表 5-1 列出了 ARM 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. ARM C/C++ 数据类型

			范围	
类型	大小	表示	最小值	最大值
signed char	8 位	ASCII	-128	127
char (1)	8 位	ASCII	0 (1)	255 (1)
unsigned char	8 位	ASCII	0	255
bool、_Bool	8 位	ASCII	0 (false)	1 (true)
short、signed short	16 位	二进制	-32 768	32 767
unsigned short、wchar_t (2)	16 位	二进制	0	65 535
int、signed int	32 位	二进制	-2 147 483 648	2 147 483 647
unsigned int	32 位	二进制	0	4 294 967 295
long、signed long	32 位	二进制	-2 147 483 648	2 147 483 647
unsigned long	32 位	二进制	0	4 294 967 295
long long、signed long long	64 位(3)	二进制	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 位(3)	二进制	0	18 446 744 073 709 551 615
enum (仅 TI_ARM9_ABI 和 TIABI) (4)	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(5)	3.40 282 346e+38
double	64 位(3)	IEEE 64 位	2.22 507 385e-308(5)	1.79 769 313e+308
long double	64 位(3)	IEEE 64 位	2.22 507 385e-308(5)	1.79 769 313e+308
指针、引用、数据成员指针	32 位	二进制	0	0xFFFFFFFF

(1) "Plain" char 具有与 signed char 或 unsigned char 相同的表示形式。--plain_char 选项指定 "plain" char 是有符号型还是无符号型。默认为无符号型。
(2) 这是 wchar_t 的默认类型。您可以使用 --wchar_t 选项将 wchar_t 类型更改为 32 位 unsigned int 类型。
(3) 64 位数据在 64 位边界上对齐。
(4) 有关枚举类型大小的详细信息，请参阅节 5.5.1。有关大小，另请参阅表 5-2。
(5) 数字是最低精度。

有符号类型的负值用 2 的补码表示。

枚举类型的存储容器类型是包含所有枚举值的最小整数类型。枚举器的容器类型如表 5-2 中所示

表 5-2. 枚举器类型

下限范围	上限范围	枚举器类型
0 至 255	0 至 255	unsigned char
-128 至 1	-128 至 127	signed char
0 至 65 535	256 至 65 535	unsigned short
-128 至 1 -32 768 至 -129	128 至 32 767 -32 768 至 32 767	short、signed short
0 至 4 294 967 295	2 147 483 648 至 4 294 967 295	unsigned int
-32 768 至 -1 -2 147 483 648 至 -32 769 0 至 2 147 483 647	32 767 至 2 147 483 647 -2 147 483 648 至 2 147 483 647 65 536 至 2 147 483 647	int、signed int

编译器根据枚举器的最低和最高元素的范围确定类型。例如，以下代码导致枚举器类型为 int：

```
   enum COLORS
{  green  = -200,
   blue   = 1,
   yellow = 2,
   red    = 60000
};
```

以下 代码导致枚举器类型为 short：

```
   enum COLORS
{  green  = -200,
   blue   = 1,
   yellow = 2,
   red    = 3
};
```

5.5.1 枚举类型大小

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

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

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

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

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

您可以通过使用 --enum_type 命令行选项和/或通过使用属性，来控制挑选枚举类型的策略。如果您使用 --enum_type=packed 选项（默认），则编译器会为枚举类型使用最小类型。如果使用 --enum_type=int 选项，则基础类型将是 int。值超出 int 范围的枚举常量会生成错误。

对于枚举类型，如果 --enum_type=packed，编译器会选择此列表中第一个足够大且符号正确的类型来表示所有枚举常量值：

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

会跳过“long”类型，因为其与“int”类型大小相同。

例如，下述枚举类型将会以“unsigned char”作为其基础类型：

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

但下述类型将会以“signed char”作为其基础类型：

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

而下述类型将会以“signed short”作为其基础类型：

```
enum ss { a, b, c, d = -1, e = UCHAR_MAX };
```

对于 C++，枚举常量全都具有与枚举类型相同的类型。

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

5.6 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。

5.7 关键字

ARM C/C++ 编译器支持所有标准 C89 关键字，包括 const、volatile 和 register。它支持所有标准的 C99 关键字，包括 inline 和 restrict。它支持所有标准的 C11 关键字。它还支持 TI 扩展关键字 __interrupt、和 __asm。某些关键字在严格 ANSI 模式下不可用。

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

- 陷阱[trap]
- reentrant
- cregister

5.7.1 const 关键字

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

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

- 如果对象定义中也指定了 *volatile*。例如，`volatile const int x`。假设将 Volatile 关键字分配给 RAM。（不允许程序修改 const volatile 对象，但可能会修改程序外部的内容。）
- 如果对象具有自动存储功能（函数作用域）。
- 如果对象是具有“可变”成员的 C++ 对象。
- 如果使用编译时未知的值（例如另一个变量的值）来初始化对象。

在这些情况下，对象的存储与未使用 const 关键字时相同。

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

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

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

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

5.7.2 __interrupt 关键字

编译器通过添加 __interrupt 关键字来扩展 C/C++ 语言，该关键字指定函数被视为中断函数。此关键字是一个 IRQ 中断。除了在严格 ANSI C 或 C++ 模式中，还可以使用备用关键字“interrupt”。

请注意，节 5.11.16 中描述的中断函数属性是声明中断函数的推荐语法。

处理中断的函数遵循特殊的寄存器保存规则和特殊的返回序列。该实现方案强调安全性。中断例程不假定各种 CPU 寄存器和状态位的 C 运行时惯例有效；相反，它会重新建立运行时环境假定的任何值。当 C/C++ 代码被中断时，中断例程必须保留例程或例程所调用任何函数使用的所有机器寄存器的内容。在函数定义中使用 __interrupt 关键字时，编译器会根据中断函数的规则和中断的特殊返回序列生成寄存器保存。

您只能将 __interrupt 关键字与定义为返回 void 且没有参数的函数一同使用。中断函数的主体可以有局部变量，并且可以自由地使用栈或全局变量。例如：

```
__interrupt void int_handler()
{
    unsigned int flags;
    ...
}
```

名称 c_int00 是 c/c++ 的入口点。此名称是为系统复位中断而保留的。这个特殊的中断例程可初始化系统并调用 main() 函数。因为它没有调用方，所以 c_int00 不保存任何寄存器。

备注

Hwi 对象和 __interrupt 关键字：将 SYS/BIOS Hwi 对象与 C 函数一同使用时，不得使用 __interrupt 关键字。Hwi_enter/Hwi_exit 宏命令和 Hwi 调度程序已经包含此功能；使用 C 修饰符可能导致出现不希望的冲突。

5.7.3 volatile 关键字

C/C++ 编译器在所有模式下都支持 *volatile* 关键字，但 。此外，在 C89、C99、C11 和 C++ 的宽松 ANSI 模式下支持 __volatile 关键字。

volatile 关键字指示编译器如何访问变量，这要求编译器不得投机取巧地优化涉及该变量的表达式。例如，外部程序、中断、另一个线程或外围设备也以访问该变量。

编译器会使用数据流分析来确定访问是否合法，从而尽可能消除冗余的存储器访问。不过，一些存储器访问可能在编译器未看到的方面比较特殊，在这类情况下，您应当使用 volatile 关键字来防止编译器优化掉某些重要内容。对于已声明为 volatile 的变量，编译器不会优化掉对该变量的任何访问。volatile 读取和写入的次数将与 C/C++ 代码中的完全相同，不多不少而且顺序也完全相同。

任何可能由明显程序控制流程外部的事物 (例如中断服务例程) 进行修改的变量必须声明为 volatile。这会告诉编译器，中断函数可能会随时修改该值，因此编译器不应执行会更改该变量的编号或访问顺序的优化。这就是 volatile 关键字的主要用途。在下述示例中，循环旨在等待位置被读取为 0xFF ：

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

不过，在此示例中，*ctrl 是循环不变量表达式，因此循环会优化为单个存储器读取。若要获取所需结果，应将 ctrl 定义为：

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

其中，*ctrl 指针旨在引用一个硬件位置，例如中断标志。

访问表示存储器映射外围设备的存储器位置时，也必须使用 volatile 关键字。此类存储器位置可能会以编译器无法预测的方式更改值。这些位置可能会在被访问时、或者当其他存储器位置被访问时或者出现某些信号时发生改变。

在调用 setjmp 的函数中，如果局部变量的值需要在发生 longjmp 时保持有效，则 volatile 也必须用于局部变量。

```
#include <stdlib.h>
jmp_buf context;
void function()
{
    volatile int x = 3;
    switch(setjmp(context))
    {
        case 0: setup(); break;
        default:
        {
           /* We only reach here if longjmp occurs.因为 x 的生命周期在 setjmp 之前开始并持续至 longjmp，C
标准要求将 x 声明为 "volatile"。*/
           printf("x == %d\n", x);
           break;
        }
    }
}
```

5.8 C++ 异常处理

编译器支持根据 ANSI/ISO 14882 C++ 标准定义的 C++ 异常处理功能。请参阅由 Bjarne Stroustrup 编写的《*C++ 编程语言》第三版*。编译器的 --exceptions 选项启用异常处理功能。编译器的默认设置是不支持异常处理。

若要在异常下正常工作，应用程序中的所有 C++ 文件都必须使用 --exceptions 选项进行编译，而不管该文件中是否存在异常。混合使用启用了异常和禁用了异常的目标文件和库可能导致未定义的行为。

异常处理需要在运行时支持库中得到支持，该库以启用异常和禁用异常的形式提供；您必须使用正确的表单链接。使用自动选择库（默认）选项时，链接器会自动选择正确的库，请参阅节 4.3.1.1。如果手动选择库，并且启用异常，则必须使用名称中包含 _eh 的运行时支持库。

使用 --exceptions 选项会导致编译器插入异常处理代码。这段代码会增加程序的大小。，但 EABI 不会大幅增加代码大小，并且如果从未引发异常，则执行时间开销极小。它略微增加了异常处理表的数据大小。

有关运行时库的详细信息，请参阅节 7.1。

5.9 寄存器变量和参数

C/C++ 编译器允许在全局和局部寄存器变量和参数上使用关键字 register。本节介绍了针对此限定符的编译器实现方案。

5.9.1 本地寄存器变量和参数

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栈，从而加快对函数内参数的访问。

更多有关寄存器惯例的信息，请参阅节 6.3。

5.9.2 全局寄存器变量

C/C++ 编译器通过向寄存器存储类指示符添加特殊约定来扩展 C 语言，以允许分配全局寄存器。此特殊全局声明具有以下格式：

register *type regid*

regid 参数可以是 __R5、__R6 或 __R9。标识符 _ _R5、_ _R6 和 _ _R9 分别绑定至与其对应的寄存器 R5、R6 和 R9。

当您在文件级别使用此声明时，该寄存器将永久保留，不允许优化器和代码生成器将该文件用于任何其他用途。您不能为寄存器分配初始值。您可以使用 #define 指令为寄存器分配一个有意义的名称；例如：

```
register struct data_struct *__R5
#define data_pointer __R5
data_pointer->element;
data_pointer++;
```

您可能会使用全局寄存器变量的原因有两个：

- 您正在整个程序中使用一个全局变量，而将这个变量分配给固定的寄存器会显著减小代码大小并降低执行速度。
- 您正在使用一个被频繁调用的中断服务例程，如果该例程不必在每次调用时都保存和恢复它使用的寄存器，则会显著降低执行速度。

您需要非常仔细地考虑保留全局寄存器变量的含义。寄存器是编译器的宝贵资源，不加区别地使用此功能可能会导致代码质量较差。

您还需要仔细考虑具有全局声明的寄存器变量的代码如何与其他代码（包括库函数）交互，这些代码无法识别对寄存器施加的限制。

因为可以作为全局寄存器变量的寄存器是入口保存寄存器，所以正常的函数调用和返回不会影响寄存器中的值，也不会影响正常的中断。但是，当您将具有全局声明的寄存器变量的代码与没有保留寄存器的代码混合使用时，寄存器中的值仍有可能损坏。为避免可能发生损坏，您必须遵守以下规则：

- 不了解全局寄存器的函数不能调用会更改全局寄存器变量的函数。使用 -r shell 选项在不了解全局寄存器声明的代码中保留寄存器。如果将指向函数的指针作为参数传递，则必须小心。如果传递的函数改变了全局寄存器变量并且被调用的函数保存了寄存器，则寄存器中的值将被损坏。
- 除非重新编译所有代码（包括所有库）以保留寄存器，否则无法访问中断服务例程中的全局寄存器变量。这是因为中断例程可以从程序中的任何一点调用。
- longjmp ( ) 函数将全局寄存器变量恢复为它们在 setjmp ( ) 位置的值。如果这会给您的代码带来问题，您必须更改函数的代码并重新编译 rts.src。

  使用 -r *register* 编译器命令行选项，您可以阻止编译器使用命名寄存器。如果您需要编译模块以防止发生上述某些情况，这使您能够在没有全局寄存器变量声明的模块（例如运行时支持库）中保留命名寄存器。

5.10 __asm 语句

C/C++ 编译器可以将汇编语言指令或指示直接嵌入到编译器的汇编语言输出中。此功能是对 C/C++ 语言的扩展，通过 ***__asm*** 关键字来实现。__asm 关键字提供了对 C/C++ 无法提供的硬件功能的访问。

除了在严格 ANSI C 模式中，也可以使用备用关键字“asm”。该关键字可以在宽松 C 和 C++ 模式下使用。

使用 __asm 在语法上是调用名为 __asm 的函数，其带有一个字符串常量参数：

__asm(" *assembler text* **");**

编译器将参数字符串直接复制到输出文件中。汇编器文本必须用双引号括起来。所有常用的字符串转义码都保留其定义。例如，可插入包含引号的 .byte 指令，如下所示：

```
__asm("STR: .byte \"abc\"");
```

naked 函数属性可用于识别使用 __asm 语句写为嵌入式汇编函数的函数。请参阅节 5.17.2。

插入的代码必须是合法的汇编语言语句。与所有汇编语言语句一样，引号内的代码行必须以标签、空格、制表符或注释（星号或分号）开头。编译器不检查字符串；如果有错误，汇编器会检测到错误。有关汇编语言语句的更多信息，请参阅 *ARM 汇编语言工具用户指南*。

__asm 语句不遵循普通 C/C++ 语句的语法限制。每个语句都可以显示为语句或声明，甚至在块之外。这对于在编译模块的最开始插入指令非常有用。

__asm 语句不提供任何引用局部变量的方法。如果汇编代码需要引用局部变量，则需要在汇编代码中编写整个函数。

如需更多信息，请参阅节 6.6.5。

备注

避免使用 asm 语句破坏 C/C++ 环境

注意请勿使用 __asm 语句破坏 C/C++ 环境。编译器不会检查插入的指令。在 C/C++ 代码中插入跳转指令和标签可能会导致在插入的代码中或其周围操作的变量产生不可预测的结果。更改段或以其他方式影响汇编环境的指令也可能会造成很多麻烦。

在使用 __asm 语句进行优化时需谨慎。尽管编译器无法删除 __asm 语句，但会大规模重新排列它们附近的代码，并导致不期望的结果。

5.11 pragma 指令

以下 pragma 指令告知编译器如何处理某个函数、对象或代码段。

- CALLS (请参阅节 5.11.1)
- CHECK_MISRA (请参阅节 5.11.2)
- CHECK_ULP (请参阅节 5.11.3)
- CODE_SECTION (请参阅节 5.11.4)
- CODE_STATE (请参阅节 5.11.5)
- DATA_ALIGN (请参阅节 5.11.6)
- DATA_SECTION (请参阅节 5.11.7)
- diag_suppress、diag_remark、diag_warning、diag_error、diag_default、diag_push、diag_pop (请参阅节 5.11.8)
- DUAL_STATE (请参阅节 5.11.9)
- FORCEINLINE (请参阅节 5.11.10)
- FORCEINLINE_RECURSIVE (请参阅节 5.11.11)
- FUNC_ALWAYS_INLINE (请参阅节 5.11.12)
- FUNC_CANNOT_INLINE (请参阅节 5.11.13)
- FUNC_EXT_CALLED (请参阅节 5.11.14)
- FUNCTION_OPTIONS (请参阅节 5.11.15)
- INTERRUPT (请参阅节 5.11.16)
- LOCATION (请参阅节 5.11.17)
- MUST_ITERATE (请参阅节 5.11.18)
- NOINIT (请参阅节 5.11.19)
- NOINLINE (请参阅节 5.11.20)
- NO_HOOKS (请参阅节 5.11.21)
- once (请参阅节 5.11.22)
- pack (请参阅节 5.11.23)
- PERSISTENT (请参阅节 5.11.19)
- PROB_ITERATE (请参阅节 5.11.24)
- RESET_MISRA (请参阅节 5.11.25)
- RESET_ULP (请参阅节 5.11.26)
- RETAIN (请参阅节 5.11.27)
- SET_CODE_SECTION (请参阅节 5.11.28)
- SET_DATA_SECTION (请参阅节 5.11.28)
- SWI_ALIAS (请参阅节 5.11.29)
- TASK (请参阅节 5.11.30)
- UNROLL (请参阅节 5.11.31)
- WEAK (请参阅节 5.11.32)

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。

5.11.1 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《*ARM 汇编语言工具用户指南*》中的“调用目标文件显示实用程序”部分。

CALLS pragma 可以位于调用函数的定义或声明之前。在 C 语言中，pragma 必须至少有 2 个参数。第一个参数是调用函数，后面至少有一个将从调用函数间接调用的函数。在 C++ 语言中，pragma 应用于所声明或定义的下一个函数，并且 pragma 必须至少有一个参数。

C 语言中 CALLS pragma 的语法如下所示。这表明 calling_function 可以通过 function_n 间接调用 function_1。

#pragma CALLS (*calling_function*, *function_1*, *function_2*, ..., *function_n* **)**

C++ 中 CALLS pragma 的语法是：

#pragma CALLS (*function_1_mangled_name*, ..., *function_n_mangled_name* **)**

注意，在 C++ 语言中，CALLS pragma 的参数必须是可从调用函数间接调用的函数的全名。

GCC 样式的“调用”函数属性（请参阅节 5.17.2）具有与 CALLS pragma 相同的效果，语法如下：

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

5.11.2 CHECK_MISRA Pragma

CHECK_MISRA pragma 在源代码级启用/禁用 MISRA C:2004 规则。编译器选项 --check_misra 必须用于启用检查，以使该 pragma 在源代码级发挥作用。

C 中 pragma 的语法为：

#pragma CHECK_MISRA (" {all|required|advisory|none|*rulespec*} **")**

rulespec 参数是以逗号分隔的规则编号列表。有关详细信息，请参阅节 5.3。

RESET_MISRA 可用于复位任何 CHECK_MISRA pragma；请参阅节 5.11.25。

5.11.3 CHECK_ULP Pragma

CHECK_ULP pragma 在源代码级别启用/禁用 ULP Advisor 规则。此 pragma 具有与使用 --advice:power 选项相同的效果。

C 中 pragma 的语法为：

#pragma CHECK_ULP (" {all|none|*rulespec*} **")**

rulespec 参数是以逗号分隔的规则编号列表。请参阅节 5.4 以了解语法。请参阅 www.ti.com/ulpadvisor 以了解规则列表。

RESET_ULP pragma 可用于复位任何 CHECK_ULP pragma；请参阅节 5.11.26。

5.11.4 CODE_SECTION Pragma

CODE_SECTION pragma 在一个名为 *section name* 的段中，为 C 中的 *symbol* 或在 C++ 中声明的下一个 symbol 分配空间。如果要将代码对象链接到与 .text 段分开的区域，CODE_SECTION pragma 会非常有用。CODE_SECTION pragma 具有与使用 GCC 样式 `section` 函数属性相同的效果。请参阅节 5.17.2。

C 中 pragma 的语法为：

#pragma CODE_SECTION (*symbol* **, "** *section name* **")**

C++ 中 pragma 的语法为：

#pragma CODE_SECTION (" *section name* **")**

以下示例演示了 CODE_SECTION pragma 的用法。

在 C 中使用 CODE_SECTION Pragma

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

以下示例 C 代码将生成下列汇编代码：

```
            .sect     "my_sect"
            .align    4
            .state32
            .global   fn
;******************************************************************************
;* FUNCTION NAME: fn                                                          *
;*                                                                            *
;*   Regs Modified     : SP                                                   *
;*   Regs Used         : A1,SP                                                *
;*   Local Frame Size  : 0 Args + 4 Auto + 0 Save = 4 byte                    *
;******************************************************************************
fn:
;* ---------------------------------------------------------------------------*
        SUB       SP, SP, #8
        STR       A1, [SP, #0]          ; |4|
        ADD       SP, SP, #8
        BX        LR
```

5.11.5 CODE_STATE Pragma

CODE_STATE 在函数级别覆盖文件的编译状态。例如，如果文件是在 Thumb 模式下编译的，但希望该文件中的函数在 32 位模式下编译，则应在该文件中添加此 pragma。函数的编译状态更改为 16 位模式 (Thumb) 或 32 位模式。

C 中 pragma 的语法为：

#pragma CODE_STATE (*function* **,** {16|32} **)**

C++ 中 pragma 的语法为：

#pragma CODE_STATE (*code state* **)**

5.11.6 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 5.17.4。

C 中 pragma 的语法为：

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

C++ 中 pragma 的语法为：

#pragma DATA_ALIGN (*constant* **)**

5.11.7 DATA_SECTION Pragma

DATA_SECTION pragma 在一个名为 *section name* 的段中，为 C 中的 *symbol* 或在 C++ 中声明的下一个 symbol 分配空间。如要将数据对象链接至一个独立于 .bss 段的区域，此 pragma 很有用。

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 5.17.4。

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 5-1 到示例 5-3 演示了 DATA_SECTION pragma 的用法。

示例 5-1. 使用 DATA_SECTION Pragma C 源文件

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

示例 5-2. 使用 DATA_SECTION Pragma C++ 源文件

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

示例 5-3. 使用 DATA_SECTION Pragma 汇编源文件

```
          .global _bufferA
          .bss    _bufferA,512,4
          .global _bufferB
_bufferB: .usect  "my_sect",512,4
```

5.11.8 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。

5.11.9 DUAL_STATE Pragma

默认情况下（即，没有编译器 -md 选项），所有具有外部链接的函数都支持双状态交互工作。此支持假定大多数调用不涉及状态的更改，因此针对不需要状态更改的调用进行了优化（在代码大小和执行速度方面）。使用 DUAL_STATE pragma 不会更改双状态支持的功能性，但它断言对应用函数的调用通常需要更改状态。因此，这种支持针对状态更改进行了优化。

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* 是函数名。在 C++ 中，pragma 应用于所声明的下一个函数。

C 中 pragma 的语法为：

#pragma DUAL_STATE (*func* **)**

C++ 中 pragma 的语法为：

#pragma DUAL_STATE

有关双状态交互工作的详细信息，请参阅节 6.11。

5.11.10 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 2.11。

请注意，FORCEINLINE、FORCEINLINE_RECURSIVE 和 NOINLINE pragma 只影响 pragma 后面的 C/C++ 语句。FUNC_ALWAYS_INLINE 和 FUNC_CANNOT_INLINE pragma 影响整个函数。

5.11.11 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 2.11。

5.11.12 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 2.11。

此 pragma 必须出现在针对要内联的函数进行的任何声明或引用之前。在 C 语言中，参数 *func* 是将被内联的函数名称。在 C++ 中，pragma 适用于下一个声明的函数。

FUNC_ALWAYS_INLINE pragma 与使用 GCC 样式 `always_inline` 的函数的效果相同。请参阅节 5.17.2。

C 中 pragma 的语法为：

#pragma FUNC_ALWAYS_INLINE (*func*)

C++ 中 pragma 的语法为：

#pragma FUNC_ALWAYS_INLINE

下述示例使用此 pragma：

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

备注

谨慎使用 FUNC_ALWAYS_INLINE Pragma

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

5.11.13 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 2.11。

此 pragma 必须出现在要保留的函数的任何声明或引用之前。在 C 语言中，参数 *func* 是不能内联的函数名。在 C++ 中，pragma 应用于所声明的下一个函数。

FUNC_CANNOT_INLINE pragma 具有与使用 GCC 样式 `noinline` 函数属性相同的效果。请参阅节 5.17.2。

C 中 pragma 的语法为：

#pragma FUNC_CANNOT_INLINE (*func*)

C++ 中 pragma 的语法为：

#pragma FUNC_CANNOT_INLINE

5.11.14 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++ 函数可能而不是通过 main() 调用。

FUNC_EXT_CALLED pragma 指定优化器应保留这些 C 函数或这些 C/C++ 函数调用的任何函数。这些函数充当 C/C++ 的入口点。此 pragma 必须出现在针对要保留的函数进行的任何声明或引用之前。在 C 语言中，参数 *func* 是要保留的函数名。在 C++ 中，pragma 适用于下一个声明的函数。

C 中 pragma 的语法为：

```
#pragma FUNC_EXT_CALLED ( func )
```

C++ 中 pragma 的语法为：

```
#pragma FUNC_EXT_CALLED
```

除了为 C/C++ 程序系统复位中断预留的名称 _c_int00 之外，中断的名称（*func* 参数）无需遵循命名惯例。

使用程序级优化时，可能需要使用 FUNC_EXT_CALLED pragma 和某些选项。请参阅节 3.4.2。

5.11.15 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 和 --opt_for_speed。

为了将 --opt_level 和 --auto_inline 与 FUNCTION_OPTIONS pragma 一同使用，必须在某种优化级别（即至少 --opt_level=0）调用编译器。如果 --opt_level=off，则忽略 FUNCTION_OPTIONS pragma。
FUNCTION_OPTIONS pragma 不能用于完全禁用编译函数的优化器；可以指定的最低优化级别是 --opt_level=0。

5.11.16 INTERRUPT Pragma

借助 INTERRUPT pragma，您可以使用 C 代码来直接处理中断。该 pragma 指定该函数为中断。中断类型由该 pragma 指定；如果未指定，则假定 中断类型为 IRQ（中断请求）。

C 中 pragma 的语法为：

```
#pragma INTERRUPT ( func [,interrupt_type] )
```

C++ 中 pragma 的语法为：

```
#pragma INTERRUPT [(interrupt_type)]
void func ( void )
```

GCC 中断属性具有与 INTERRUPT pragma 相同的效果，语法如下所示。请注意，该中断属性可以放在函数的定义或其声明之前。

```
__attribute__((interrupt [("interrupt_type")] )) void func ( void )
```

在 C 语言中，参数 *func* 是函数的名称。在 C++ 中，pragma 应用于下一个声明的函数。可选参数 *interrupt_type* 指定中断类型。保存的寄存器以及返回序列取决于中断类型。如果 interrupt pragma 中省略了中断类型，则假定中断类型为 IRQ。有效的中断类型如下：

中断类型	说明
DABT	数据中止
FIQ	快速中断请求
IRQ	中断请求
PABT	预取中止
RESET	系统复位
SWI	软件中断
UDEF	未定义的指令

除了为 C 程序系统复位中断预留的名称 _c_int00，中断的名称（*func* 参数）无需遵循命名惯例。

对于 Cortex-M 架构，*interrupt_type* 可以不指定（默认）或为 SWI。硬件会为中断执行必要的上下文保存和恢复操作。因此，编译器并不区分不同的中断类型。唯一的例外是软件中断 (SWI)，该中断允许包含参数（对于 Cortex-M 架构，C SWI 处理程序无法返回值）。

```
#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。

5.11.17 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 5.17.2。

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 5.17.4）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 5.11.19。

5.11.18 MUST_ITERATE Pragma

MUST_ITERATE pragma 为编译器指定循环的某些属性。使用此 pragma，即表示向编译器保证循环会执行特定的次数或在指定范围内执行多次。

只要向循环应用 UNROLL pragma，则应向同一循环应用 MUST_ITERATE。对于循环，MUST_ITERATE pragma 的第三个参数 multiple 最为重要，并且始终应该指定。

另外，应当尽可能多地向任何其他循环应用 MUST_ITERATE pragma。这是因为通过该 pragma 提供的信息（尤其是最小迭代次数）能够帮助编译器选择最优循环和循环变换（即嵌套循环变换）。此外，该 pragma 还可帮助编译器缩减代码大小。

MUST_ITERATE pragma 与其适用的 for、while 或 do-while 循环之间不能包含任何语句。不过，MUST_ITERATE pragma 与相应循环之间可以存在 UNROLL 等其他 pragma。

5.11.18.1 MUST_ITERATE Pragma 语法

该 pragma 的 C 和 C++ 语法为：

```
#pragma MUST_ITERATE ( min, max, multiple )
```

对应属性的 C++ 语法如下所示。没有可用的 C 属性语法。

```
[[TI::must_iterate( min, max, multiple )]]
```

参数 *min* 和 *max* 是由程序员保证的最小和最大行程计数。行程计数是指循环迭代的次数。循环的行程计数必须能够被 *multiple* 整除。所有参数都是可选的。例如，如果行程计数可以为 5 或更大值，那么您可以按如下所示指定参数列表：

```
#pragma MUST_ITERATE(5)
```

不过，如果行程计数可以为 5 的任何非零倍数，该 pragma 会类似如下：

```
#pragma MUST_ITERATE(5, , 5) /* Note the blank field for max */
```

有时为了让编译器展开，需要提供 min 和 multiple。当编译器无法轻松地确定循环要执行的迭代次数（即循环具有复杂的退出条件）时，尤其如此。

通过 MUST_ITERATE pragma 指定 multiple 时，如果行程计数不能被 multiple 整除，程序的结果会为 undefined。另外，如果行程计数小于指定的最小值或大于指定的最大值，程序的结果也会是 undefined。

如果未指定 min，则会使用 0。如果未指定 max，则会使用可能的最大值。如果为同一循环指定了多个 MUST_ITERATE pragma，则会使用最小的 max 和最大的 min。

下述示例使用 must_iterate C++ 属性语法：

```
void myFunc (int *a, int *b, int * restrict c, int n)
{
    ...
    [[TI::must_iterate(32, 1024, 16)]]
    for (int i = 0; i < n; i++)
    {
        c[i] = a[i] + b[i];
    }
    ...
}
```

5.11.18.2 使用 MUST_ITERATE 扩展编译器对循环的了解

通过使用 MUST_ITERATE pragma，可以保证循环执行一定的次数。下述示例会告知编译器，循环保证可以正好运行 10 次：

```
#pragma MUST_ITERATE(10,10)
for(i = 0; i < trip_count; i++)  { ...
```

在此示例中，即使没有 pragma，编译器也尝试生成循环。但如果没有为这样的循环指定 MUST_ITERATE，编译器会生成代码绕过循环，以解决可能出现的 0 次迭代。利用 pragma 规范，编译器知道循环至少会迭代一次，可以消除循环绕过代码。

MUST_ITERATE 可用于指定循环计数的范围以及循环计数的系数。下述示例会告知编译器，循环执行 8 次到 48 次之间，trip_count 变量是 8 的倍数（8、16、24、32、40、48）。倍数参数支持编译器展开循环。

```
#pragma MUST_ITERATE(8, 48, 8)
for(i = 0; i < trip_count; i++)  { ...
```

对于具有复杂边界的循环，应考虑使用 MUST_ITERATE。在下述示例中，编译器不得不生成一个除法函数调用，以便在运行时确定所执行的迭代次数。

```
for(i2 = ipos[2]; i2 < 40; i2 += 5)  { ...
```

编译器不会执行上述操作。在这种情况下，使用 MUST_ITERATE 指定循环始终执行八次，编译器将尝试生成循环：

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

5.11.19 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 5.17.4。

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;
```

5.11.20 NOINLINE Pragma

NOINLINE pragma 可以放置在一条语句之前，用于防止该语句中所做的任何函数调用发生内联。该 pragma 对相同函数的其他调用则没有影响。

此 pragma 在 C/C++ 中的语法为：

```
#pragma NOINLINE
```

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

5.11.21 NO_HOOKS Pragma

NO_HOOKS pragma 用于防止为一个函数而生成入口和出口钩子程序调用。

C 中 pragma 的语法为：

```
#pragma NO_HOOKS ( func )
```

C++ 中 pragma 的语法为：

```
#pragma NO_HOOKS
```

有关入口和出口钩子程序的详细信息，请参阅节 2.15。

5.11.22 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。

5.11.23 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 5.17.5。

5.11.24 PROB_ITERATE Pragma

PROB_ITERATE pragma 为编译器指定循环的某些属性。您可以断言这些属性在常见情况下为 true。PROB_ITERATE pragma 能够帮助编译器选择最优循环和循环变换（也即软件流水线和嵌套循环变换）。仅当未使用 MUST_ITERATE pragma 时或 PROB_ITERATE 参数比 MUST_ITERATE 参数具有更多限制时，PROB_ITERATE 才有用。

PROB_ITERATE pragma 与其适用的 for、while 或 do-while 循环之间不能包含任何语句。不过，MUST_ITERATE pragma 与相应循环之间可以存在 UNROLL 和 PROB_ITERATE 和 PROB_ITERATE 等其他 pragma。该 pragma 的 C 和 C++ 语法为：

#pragma PROB_ITERATE(*min* **,** *max* **)**

对应属性的 C++ 语法如下所示。不存在 C 属性语法。有关使用类似语法的示例，请参阅节 5.11.18.1。

[[TI::prob_iterate(*min*, *max* **)]]**

其中，min 和 max 是循环在常见情况下的最小和最大行程计数。行程计数是指循环的迭代次数。这两个参数都是可选的。

例如，PROB_ITERATE 可应用于大多数情况下执行八次迭代（但有时可能执行超过或少于八次迭代）的循环：

```
#pragma PROB_ITERATE(8, 8)
```

如果仅知道预期的最小行程计数（例如 5 次），则该 pragma 与以下所示类似：

```
#pragma PROB_ITERATE(5)
```

如果仅知道预期的最大行程计数（例如 10 次），则该 pragma 与以下所示类似：

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

5.11.25 RESET_MISRA Pragma

RESET_MISRA pragma 会将指定的 MISRA C:2004 规则重置为处理任何 CHECK_MISRA pragma（请参阅节 5.11.2）之前的状态。例如，如果在命令行中启用了某个规则，但在源代码中禁用了这个规则，RESET_MISRA pragma 会将它重置为已启用。此 pragma 接受与 --check_misra 选项相同的格式，但“none”关键字除外。

必须使用 --check_misra 编译器命令行选项来启用 MISRA C:2004 规则检查，此 pragma 才能在源代码级别正常工作。

C 中 pragma 的语法为：

#pragma RESET_MISRA (" {all|required|advisory|*rulespec*} **")**

rulespec 参数是以逗号分隔的规则编号列表。有关详细信息，请参阅节 5.3。

5.11.26 RESET_ULP Pragma

RESET_ULP pragma 会将指定的 ULP Advisor 规则重置为处理任何 CHECK_ULP pragma（请参阅节 5.11.3）之前的状态。例如，如果在命令行中启用了某个规则，但在源代码中禁用了这个规则，RESET_ULP pragma 会将它重置为已启用。此 pragma 与 --advice:power 选项接受相同的格式，但“none”关键字除外。

C 中 pragma 的语法为：

```
#pragma RESET_ULP (" {all|rulespec} ")
```

rulespec 参数是以逗号分隔的规则编号列表。相关详细信息，请参阅节 5.4。有关规则列表，请访问 www.ti.com/ulpadvisor。

5.11.27 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 5.17.2 和节 5.17.4。

C 中 pragma 的语法为：

```
#pragma RETAIN ( symbol )
```

C++ 中 pragma 的语法为：

```
#pragma RETAIN
```

5.11.28 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。

5.11.29 SWI_ALIAS Pragma

SWI_ALIAS pragma 让您可以以函数名称的方式来引用特定的软件中断，以及以函数调用的方式来调用软件中断。函数名称只是软件中断的别名，因此函数名称不存在函数定义。

C 中 pragma 的语法为：

#pragma SWI_ALIAS(*func* **,** *swi_number* **)**

C++ 中 pragma 的语法为：

#pragma SWI_ALIAS(*swi_number* **)**

对所应用函数的调用会被编译为软件中断，其编号为 *swi_number*。*swi_number* 变量必须为整数常量。

该别名必须存在函数原型，并且这个函数原型必须位于该 pragma 之后和使用的别名之前。其编号在运行时之前未知的软件中断不受支持。

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 5.17.2。

有关使用软件中断的更多信息，包括参数传递和寄存器使用方面的限制，请参阅节 6.7.5。

使用 SWI_ALIAS Pragma C 源文件

```
#pragma SWI_ALIAS(put, 48)    /* #pragma SWI_ALIAS(48) for C++ */
int  put  (char *key, int value);
void error();
main()
{
   if (!put("one", 1))  /* calling "put" invokes SWI #48 with 2 arguments */
      error();          /* and returns a result.                          */
}
```

生成的汇编文件

```
;****************************************************************
;* FUNCTION DEF: _main                                          *
;****************************************************************
_main:
        STMFD     SP!, {LR}
        ADR       A1, SL1
        MOV       A2, #1
        SWI       #48             ; SWI #48 is generated for the function call
        CMP       A1, #0
        BLEQ      _error
        MOV       A1, #0
        LDMFD     SP!, {PC}
SL1:    .string "one",0
```

5.11.30 TASK Pragma

TASK pragma 将其适用的函数指定为任务。任务是被调用但从不返回值的函数。通常，它们由一个只负责分派其他活动的无限循环构成。它们从不返回值，因此无需保存（和恢复）其他情况下要保存和恢复的寄存器。这可以节省 RAM 空间以及一些代码空间。

C 中 pragma 的语法为：

```
#pragma TASK( func )
```

C++ 中 pragma 的语法为：

```
#pragma TASK
```

5.11.31 UNROLL Pragma

UNROLL pragma 向编译器指定了循环应该展开的次数。必须调用优化器（使用 --opt_level=[1|2|3] 或者 -O1、-O2 或 -O3），该 pragma 指定的循环才会展开。编译器具有忽略此 pragma 的选项。

UNROLL pragma 与其适用的 for、while 或 do-while 循环之间不能包含任何语句。不过，UNROLL pragma 与相应循环之间可以存在 MUST_ITERATE 等其他 pragma。

C 和 C++ 中该 pragma 语法为：

```
#pragma UNROLL( n )
```

对应属性的 C++ 语法如下所示。不存在 C 属性语法。有关使用类似语法的示例，请参阅节 5.11.18.1。

```
[[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。

5.11.32 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 5.17.2 和节 5.17.4。

5.12 _Pragma 运算符

ARM 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。

5.13 应用程序二进制接口

应用程序二进制接口 (ABI) 定义单独编写、单独编译或汇编的函数如何协同工作。这涉及到数据类型存储、寄存器惯例、函数结构和调用惯例的标准化。它应该根据 C 符号定义链接名称生成。它定义了目标文件格式和调试格式。它应该记录系统初始化的方式。如果是 C++，它则定义了对 C++ 名称的处理和异常处理支持。

v15.6.0.STS 和更高版本的 TI 代码生成工具不支持 COFF ABI。如果要生成 COFF 输出文件，请使用 v5.2 的 ARM 工具，并参阅 SPRU151J。

ARM ABIv2 已经成为 ARM 架构的行业标准。它具有以下优势：

- 支持使用不同工具链构建的对象之间互连。例如，使用 RVCT 构建的库因此能与使用 ARM 4.6 工具集构建的应用程序相连接。
- 这是有详尽文档记录的。完整的 ARM ABI 规范位于 ARM 信息中心。
- 它是现代化的。EABI 需要 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。

ARM ABIv2 允许供应商在裸机模式下定义系统初始化。节 6.10.3 中描述了有关 EABI 模式的 TI 特定信息。如果为 EABI 编译，则定义 __TI_EABI_ASSEMBLER 预定义符号并将其设置为 1。

5.14 ARM 指令内在函数

可使用下表中的内在函数生成汇编指令。表 5-3 显示了不同 ARM 目标上可用的内在函数。表 5-4 显示了每个内在函数的调用语法，以及相应的汇编指令和说明。节 6.8.1 中提供了用于获取和设置 CPSR 寄存器、启用/禁用中断的其他内在函数。

表 5-3. 目标对 ARM 内在函数的支持

C/C++ 编译器内在函数	ARM V5e (ARM9E)	ARM V6 (ARM11)	ARM V6M0 (Cortex-M0)	ARM V7M3 (Cortex-M3)	ARM V7M4 (Cortex-M4)	ARM V7R (Cortex-R4)	ARM V7A8 (Cortex-A8)
__clz	是	是		是	是	是	是
__delay_cycles			是	是	是	是	
__get_MSP			是	是	是		
__get_PRIMASK			是	是	是		
__ldrex		是		是	是	是	是
__ldrexb		是		是	是	是	是
__ldrexd		是				是	是
__ldrexh		是		是	是	是	是
__MCR	是	是		是	是	是	是
__MRC	是	是		是	是	是	是
__ nop	是	是	是	是	是	是	是
_norm	是	是		是	是	是	是
__rev		是	是		是	是	是
__rev16		是	是		是	是	是
__revsh		是	是		是	是	是
__rbit		是			是	是	是
__ror	是	是	是	是	是	是	是
_pkhbt		是			是	是	是
_pkhtb		是			是	是	是
_qadd16		是			是	是	是
_qadd8		是			是	是	是
_qaddsubx		是			是	是	是
_qsub16		是			是	是	是
_qsub8		是			是	是	是
_qsubaddx		是			是	是	是

表 5-3. 目标对 ARM 内在函数的支持 (continued)

C/C++ 编译器内在函数	ARM V5e (ARM9E)	ARM V6 (ARM11)	ARM V6M0 (Cortex-M0)	ARM V7M3 (Cortex-M3)	ARM V7M4 (Cortex-M4)	ARM V7R (Cortex-R4)	ARM V7A8 (Cortex-A8)
_sadd	是	是			是	是	是
_sadd16		是			是	是	是
_sadd8		是			是	是	是
_saddsubx		是			是	是	是
_sdadd	是	是			是	是	是
_sdsub	是	是			是	是	是
_sel		是			是	是	是
__set_MSP			是	是	是		
__set_PRIMASK			是	是	是		
_shadd16		是			是	是	是
_shadd8		是			是	是	是
_shsub16		是			是	是	是
_shsub8		是			是	是	是
_smac	是	是			是	是	是
_smlabb	是	是			是	是	是
_smlabt	是	是			是	是	是
_smlad		是			是	是	是
_smladx		是			是	是	是
_smlalbb	是	是			是	是	是
_smlalbt	是	是			是	是	是
_smlald		是			是	是	是
_smlaldx		是			是	是	是
_smlaltb	是	是			是	是	是
_smlaltt	是	是			是	是	是
_smlatb	是	是			是	是	是
_smlatt	是	是			是	是	是
_smlawb	是	是			是	是	是
_smlawt	是	是			是	是	是
_smlsd		是			是	是	是
_smlsdx		是			是	是	是
_smlsld		是			是	是	是
_smlsldx		是			是	是	是
_smmla		是			是	是	是
_smmlar		是			是	是	是
_smmls		是			是	是	是
_smmlsr		是			是	是	是
_smmul		是			是	是	是
_smmulr		是			是	是	是
_smuad		是			是	是	是
_smuadx		是			是	是	是
_smusd		是			是	是	是
_smusdx		是			是	是	是
_smpy	是	是			是	是	是
_smsub	是	是			是	是	是

表 5-3. 目标对 ARM 内在函数的支持 (continued)

C/C++ 编译器内在函数	ARM V5e (ARM9E)	ARM V6 (ARM11)	ARM V6M0 (Cortex-M0)	ARM V7M3 (Cortex-M3)	ARM V7M4 (Cortex-M4)	ARM V7R (Cortex-R4)	ARM V7A8 (Cortex-A8)
_smulbb	是	是			是	是	是
_smulbt	是	是			是	是	是
_smultb	是	是			是	是	是
_smultt	是	是			是	是	是
_smulwb	是	是			是	是	是
_smulwt	是	是			是	是	是
__sqrt	是	是				是	是
__sqrtf	是	是			是	是	是
_ssat16		是			是	是	是
_ssata	是	是		是	是	是	是
_ssatl	是	是		是	是	是	是
_ssub	是	是			是	是	是
_ssub16		是			是	是	是
_ssub8		是			是	是	是
_ssubaddx		是			是	是	是
__strex		是		是	是	是	是
__strexb		是		是	是	是	是
__strexd		是				是	是
__strexh		是		是	是	是	是
_subc	是	是			是	是	是
_sxtab		是			是	是	是
_sxtab16		是			是	是	是
_sxtah		是			是	是	是
_sxtb	是	是		是	是	是	是
_sxtb16		是			是	是	是
_sxth	是	是		是	是	是	是
_uadd16		是			是	是	是
_uadd8		是			是	是	是
_uaddsubx		是			是	是	是
_uhadd16		是			是	是	是
_uhadd8		是			是	是	是
_uhsub16		是			是	是	是
_uhsub8		是			是	是	是
_umaal		是			是	是	是
_uqadd16		是			是	是	是
_uqadd8		是			是	是	是
_uqaddsubx		是			是	是	是
_uqsub16		是			是	是	是
_uqsub8		是			是	是	是
_uqsubaddx		是			是	是	是
_usad8		是			是	是	是
_usat16		是			是	是	是
_usata	是	是		是	是	是	是
_usatl	是	是		是	是	是	是

表 5-3. 目标对 ARM 内在函数的支持 (continued)

C/C++ 编译器内在函数	ARM V5e (ARM9E)	ARM V6 (ARM11)	ARM V6M0 (Cortex-M0)	ARM V7M3 (Cortex-M3)	ARM V7M4 (Cortex-M4)	ARM V7R (Cortex-R4)	ARM V7A8 (Cortex-A8)
_usub16		是			是	是	是
_usub8		是			是	是	是
_usubaddx		是			是	是	是
_uxtab		是			是	是	是
_uxtab16		是			是	是	是
_uxtah		是			是	是	是
_uxtb	是	是		是	是	是	是
_uxtb16		是			是	是	是
_uxth	是	是		是	是	是	是
_ _wfe			是	是	是	是	是
_ _wfi			是	是	是	是	是

表 5-4 显示了每个内在函数的调用语法，以及相应的汇编指令和说明。请参阅表 5-3，以查看有关不同 ARM 目标的可用内在函数列表。节 6.8.1 中提供了用于获取和设置 CPSR 寄存器、启用/禁用中断的其他内在函数。

表 5-4. ARM 编译器内在函数

C/C++ 编译器内在函数	汇编指令	说明
int *count* = **_ _clz**(int *src*);	**CLZ** *count* , *src*	返回前导零的计数。
void **_ _delay_cycles(** unsigned int *cycles*);	不尽相同	将执行指定周期数的延迟。周期数必须是常数。__delay_cycles 内在函数会插入代码，以精确地使用指定的周期数，而不会产生任何副作用。延迟的周期数必须是编译时间常量。 **注意：**循环计时基于 0 个等待状态。结果因其他等待状态而异。该实现方案不考虑动态预测。鉴于流水线清除行为，较低的延迟周期计数可能不太准确。
unsigned int *dst* = **_ _get_MSP**(void);	**MRS** *dst* , **MSP**	返回主栈指针的当前值。
unsigned int *dst* = **_ _get_PRIMASK**(void);	**MRS** *dst* , **PRIMASK**	返回优先级屏蔽寄存器的当前值。如果该值为 1，则禁止激活具有可配置优先级的所有异常。
unsigned int *dest* = **_ _ldrex**(void* *src*);	**LDREX** *dst* , *src*	从包含文字（32 位）数据的内存地址加载数据
unsigned int *dest*= **_ _ldrexb**(void* *src*);	**LDREXB** *dst* , *src*	从包含字节数据的内存地址加载数据
unsigned long long *dest* = **_ _ldrexd**(void* *src*);	**LDREXD** *dst* , *src*	从支持 long-long 的内存地址加载数据
unsigned int *dest* = **_ _ldrexh**(void* *src*);	**LDREXH** *dst* , *src*	从包含半字（16 位）数据的内存地址加载数据
void **__MCR** (unsigned int *coproc*, unsigned int *opc1*, unsigned int *src*, unsigned int *coproc_reg1*, unsigned int *coproc_reg2*, unsigned int *opc2*);	**MCR** *coproc*, *opc1*, *src*, *CR<coproc_reg1>*, *CR<coproc_reg2>*, *opc2*	访问协处理器寄存器
unsigned int **__MRC**(unsigned int *coproc*, unsigned int *opc1*, unsigned int *coproc_reg1*, unsigned int *coproc_reg2*, unsigned int *opc2*);	**MRC** *coproc*, *opc1*, *src*, *CR<coproc_reg1>*, *CR<coproc_reg2>*, *opc2*	访问协处理器寄存器
void **_ _nop(** void);	**NOP**	执行不做任何事情的指令。
int *dst* = **_norm**(int *src*);	**CLZ** *dst* , *src*	计算前导零位数。在实现整数规范化时，可以使用此内在函数。
int *dst* = **_pkhbt**(int *src1* , int *src2* , int *shift*);	**PKHBT** *dst* , *src1* , *src2* , **#** *shift*	组合 src1 的底部半字和 src2 的移位顶部半字
nt *dst* = **_pkhtb**(int *src1* , int *src2* , int *shift*);	**PKHTB** *dst* , *src1* , *src2* , **#** *shift*	组合 src1 的顶部半字和 src2 的移位底部半字
int *dst* = **_qadd16**(int *src1* , int *src2*);	**QADD16** *dst* , *src1* , *src2*	执行两个有符号半字饱和加法
int *dst* = **_qadd8**(int *src1* , int *src2*);	**QADD8** *dst* , *src1* , *src2*	执行四个有符号饱和 8 位加法

表 5-4. ARM 编译器内在函数 (continued)

C/C++ 编译器内在函数	汇编 指令	说明
int *dst* = **_qaddsubx(**int *src1* , int *src2* **);**	**QASX** *dst* , *src1* , *src2*	交换 src2 的半字，对顶部半字执行有符号饱和加法，对底部半字执行有符号饱和减法。
int *dst* = **_qsub16(**int *src1* , int *src2* **);**	**QSUB16** *dst* , *src1* , *src2*	执行两个有符号饱和半字减法
int *dst* = **_qsub8(**int *src1* , int *src2* **);**	**QSUB8** *dst* , *src1* , *src2*	执行四个有符号饱和 8 位减法
int *dst* = **_qsubaddx(**int *src1* , int *src2* **);**	**QSAX** *dst* , *src1* , *src2*	交换 src2 的半字，对顶部半字执行有符号饱和减法，对底部半字执行有符号饱和加法。
int *dst* = **_ _rbit(**int *src* **);**	**RBIT** *dst* , *src*	反转文字中的位顺序。
int *dst* = **_ _rev(**int *src* **);**	**REV** *dst* , *src*	反转文字中的字节顺序。也就是说，在大端字节序和小端字节序之间转换 32 位数据，反之亦然。
int *dst* = **_ _rev16(**int *src* **);**	**REV16** *dst* , *src*	独立反转文字中每个字节的字节顺序。也就是说，在大端字节序和小端字节序之间转换 16 位数据，反之亦然。
int *dst* = **_ _revsh(**int *src* **);**	**REVSH** *dst* , *src*	反转文字的低位字节的字节顺序，并将符号扩展到 32 位。也就是说，将 16 位有符号数据转换为 32 位有符号数据，同时在大端字节序和小端字节序之间进行转换，反之亦然。
int *dst* = **_ _ror(**int *src* , int *shift* **);**	**ROR** *dst* , *src* , *shift*	将值向右旋转指定的位数。从右端旋转的位放入左端的空位。
int *dst* =**_sadd(**int *src1* , int *src2* **);**	**QADD** *dst* , *src1* , *src2*	饱和加法
int *dst* = **_sadd16(**int *src1* , int *src2* **);**	**SADD16** *dst* , *src1* , *src2*	执行两个有符号半字加法
int *dst* = **_sadd8(**int *src1* , int *src2* **);**	**SADD8** *dst* , *src1* , *src2*	执行四个有符号 8 位加法
int *dst* = **_saddsubx(**int *src1* , int *src2* **);**	**SASX** *dst* , *src1* , *src2*	交换 src2 的半字，添加顶部半字并减去底部半字
int *dst* =**_sdadd(**int *src1* , int *src2* **);**	**QDADD** *dst* , *src1* , *src2*	饱和双加
int *dst* =**_sdsub(**int *src1* , int *src2* **);**	**QDSUB** *dst* , *src1* , *src2*	饱和双减
int *dst* = **_sel(**int *src1* , int *src2* **);**	**SEL** *dst* , *src1* , *src2*	如果设置了 GE 位 n，则从 src1 选择字节 n；如果未设置 GE 位 n，则从 src2 选择字节 n，其中 n 的范围为 0 到 3。
void **_ _set_MSP(**unsigned int *src* **);**	**MSR MSP,** *src*	将主栈指针的值设置为 src。
unsigned int *dst* = **_ _set_PRIMASK(**unsigned int *src* **);**	**MRS** *dst* , **PRIMASK** (optional) **MSR PRIMASK,** *src*	将优先级屏蔽寄存器设置为 src 值，并返回设置为 dst 之前的值。将此寄存器设置为 1 可防止激活具有可配置优先级的所有异常。
int *dst* = **_shadd16(**int *src1* , int *src2* **);**	**SHADD16** *dst* , *src1* , *src2*	执行两个带符号的半字加法并将结果减半
int *dst* = **_shadd8(**int *src1* , int *src2* **);**	**SHADD8** *dst* , *src1* , *src2*	执行四个带符号的 8 位加法并将结果减半
int *dst* = **_shsub16(**int *src1* , int *src2* **);**	**SHSUB16** *dst* , *src1* , *src2*	执行两个带符号的半字减法并将结果减半
int *dst* = **_shsub8**int *src1* , int *src2* **);**	**SHSUB8** *dst* , *src1* , *src2*	执行四个带符号的 8 位减法并将结果减半
int *dst* =**_smac(**int *dst* , int *src1* , int *src2* **);**	**SMULBB** *tmp* , *src1* , *src2* **QDADD** *dst* , *dst* , *tmp*	饱和乘法累加
int *dst* =**_smlabb(**int *dst* , short *src1* , short *src2* **);**	**SMLABB** *dst* , *src1* , *src2*	有符号乘法累加底部半字
int *dst* =**_smlabt(**int *dst* , short *src1* , int *src2* **);**	**SMLABT** *dst* , *src1* , *src2*	有符号乘法累加底部半字和顶部半字
int *dst* **_smlad(**int *src1* , int *src2* , int *acc* **);**	**SMLAD** *dst* , *src1* , *src2* , *acc*	对 src1 和 src2 的顶部半字和底部半字执行两次有符号 16 位乘法，并将结果添加到 acc。
int *dst* **_smladx(**int *src1* , int *src2* , int *acc* **);**	**SMLADX** *dst* , *src1* , *src2* , *acc*	与 _smlad 相同，只是 src2 中的半字在乘法之前交换。
long long *dst* =**_smlalbb(**long long *dst* , short *src1* , short *src2* **);**	**SMLALBB** *dstlo* , *dsthi* , *src1* , *src2*	有符号长整数乘法并累加底部半字

表 5-4. ARM 编译器内在函数 (continued)

C/C++ 编译器内在函数	汇编 指令	说明
long long *dst* =**_smlalbt(**long long *dst* , short *src1* , int *src2* **);**	**SMLALBT** *dstlo* , *dsthi* , *src1* , *src2*	有符号长整数乘法并累加底部半字和顶部半字
long long *dst* **_smlald(**long long *acc* , int *src1* , int *src2* **);**	**SMLALD** *dst* , *src1* , *src2*	对 src1 和 src2 的顶部半字和底部半字执行两次 16 位乘法，并将结果添加到 64 位 acc 操作数
long long *dst* **_smlaldx(**long long *acc* , int *src1* , int *src2* **);**	**SMLALDX** *dst* , *src1* , *src2*	与 _smlald 相同，只是 src2 中的半字已交换。
long long *dst* =**_smlaltb(**long long *dst* , int *src1* , short *src2* **);**	**SMLALTB** *dstlo* , *dsthi* , *src1* , *src2*	有符号长整数乘法并累加顶部半字和底部半字
long long *dst* =**_smlaltt(**long long *dst* , int *src1* , int *src2* **);**	**SMLALTT** *dstlo* , *dsthi* , *src1* , *src2*	有符号长整数乘法并累加顶部半字
int *dst* =**_smlatb(**int *dst* , int *src1* , short *src2* **);**	**SMLATB** *dst* , *src1* , *src2*	有符号乘法累加顶部半字和底部半字
int *dst* =**_smlatt(**int *dst* , int *src1* , int *src2* **);**	**SMLATT** *dst* , *src1* , *src2*	有符号乘法累加顶部半字
int *dst* **_smlawb(**int *src1* , short *src2* , int *acc* **);**	**SMLAWB** *dst* , *src1* , *src2*	有符号乘法累加文字和底部半字
int *dst* **_smlawt(**int *src1* , short *src2* , int *acc* **);**	**SMLAWT** *dst* , *src1* , *src2*	有符号乘法累加文字和顶部半字
int *dst* **_smlsd(**int *src1* , int *src2* , int *acc* **);**	**SMLSD** *dst* , *src1* , *src2* , *acc*	对 src1 和 src2 的顶部半字和底部半字执行两次有符号 16 位乘法，并将结果差异添加到 acc。
int *dst* **_smlsdx(**int *src1* , int *src2* , int *acc* **);**	**SMLSDX** *dst* , *src1* , *src2* , *acc*	与 _smlsd 相同，只是 src2 中的半字在乘法之前交换。
long long *dst* **_smlsld(**long long *acc* , int *src1* , int *src2* **);**	**SMLSLD** *dst* , *src1* , *src2*	对 src1 和 src2 的顶部半字和底部半字执行两次 16 位乘法，并将结果差异添加到 64 位 acc 操作数。
long long *dst* **_smlsldx(**long long *acc* , int *src1* , int *src2* **);**	**SMLSLDX** *dst* , *src1* , *src2*	与 _smlsld 相同，只是 src2 中的半字已交换。
int *dst* **_smmla(**int *src1* , int *src2* , int *acc* **);**	**SMMLA** *dst* , *src1* , *src2* , *acc*	对 src1 和 src2 执行有符号乘法，提取结果的最高有效 32 位，并添加累积值。
int *dst* **_smmlar(**int *src1* , int *src2* , int *acc* **);**	**SMMLAR** *dst* , *src1* , *src2* , *acc*	与 _smmla 相同，只是结果是四舍五入的，而不是被截断的。
int *dst* **_smmls(**int *src1* , int *src2* , int *acc* **);**	**SMMLS** *dst* , *src1* , *src2* , *acc*	对 src1 和 src2 执行有符号乘法，从左移 32 位的累积值中减去结果，并提取减法结果的最高有效 32 位。
int *dst* **_smmlsr(**int *src1* , int *src2* , int *acc* **);**	**SMMLSR** *dst* , *src1* , *src2* , *acc*	与 _smmls 相同，只是结果是四舍五入的，而不是被截断的。
int *dst* **_smmul(**int *src1* , int *src2* , int *acc* **);**	**SMMUL** *dst* , *src1* , *src2* , *acc*	对 src1 和 src2 执行有符号 32 位乘法，并提取结果的最高有效 32 位。
int *dst* **_smmulr(**int *src1* , int *src2* , int *acc* **);**	**SMMULR** *dst* , *src1* , *src2* , *acc*	与 _smmul 相同，只是结果是四舍五入的，而不是被截断的。
int *dst* =**_smpy(**int *src1* , int *src2* **);**	**SMULBB** *dst* , *src1* , *src2* **QADD** *dst* , *dst* , *dst*	饱和乘法
int *dst* =**_smsub(**int *src1* , int *src2* **);**	**SMULBB** *tmp* , *src1* , *src2* **QDSUB** *dst* , *dst* , *tmp*	饱和乘减
int *dst* **_smuad(**int *src1* , int *src2* **);**	**SMUAD** *dst* , *src1* , *src2*	对顶部半字和底部半字执行两次有符号 16 位乘法，并将乘积相加。
int *dst* **_smuadx(**int *src1* , int *src2* **);**	**SMUADX** *dst* , *src1* , *src2*	与 _smuad 相同，只是 src2 中的半字在乘法之前交换。
int *dst* =**_smulbb(**int *src1* , int *src2* **);**	**SMULBB** *dst* , *src1* , *src2*	有符号乘法底部半字
int *dst* =**_smulbt(**int *src1* , int *src2* **);**	**SMULBT** *dst* , *src1* , *src2*	有符号乘法底部半字和顶部半字
int *dst* =**_smultb(**int *src1* , int *src2* **);**	**SMULTB** *dst* , *src1* , *src2*	有符号乘法顶部半字和底部半字
int *dst* =**_smultt(**int *src1* , int *src2* **);**	**SMULTT** *dst* , *src1* , *src2*	有符号乘法顶部半字
int *dst* **_smulwb(**int *src1* , short *src2* , int *acc* **);**	**SMULWB** *dst* , *src1* , *src2*	有符号乘法文字和底部半字
int *dst* **_smulwt(**int *src1* , short *src2* , int *acc* **);**	**SMULWT** *dst* , *src1* , *src2*	有符号乘法文字和顶部半字

表 5-4. ARM 编译器内在函数 (continued)

C/C++ 编译器内在函数	汇编 指令	说明
int *dst* **_smusd(**int *src1* , int *src2* **);**	**SMUSD** *dst* , *src1* , *src2*	对顶部半字和底部半字执行两次有符号 16 位乘法，并减去乘积。
int *dst* **_smusdx(**int *src1* , int *src2* **);**	**SMUSDX** *dst* , *src1* , *src2*	与 _smusd 相同，只是 src2 中的半字在乘法之前交换。
double **__sqrt(** double **);**	**VSQRT** *dst* , *src1*	返回指定双精度值的平方根。如果 --fp_mode=relaxed，则直接使用 VSQRT 指令替换此内在函数。如果使用严格浮点模式并且发生域错误，函数还必须设置 errno。
float **__sqrtf(** float **);**	**VSQRT** *dst* , *src1*	返回指定浮点值的平方根。如果 --fp_mode=relaxed，则直接使用 VSQRT 指令替换此内在函数。如果使用严格浮点模式并且发生域错误，函数还必须设置 errno。
int *dst* =**_ssat16(**int *src* , int *bitpos* **);**	**SSAT16** *dst* , **#** *bitpos*	对 bitpos 指定的可选有符号范围执行两个半字饱和
int *dst* =**_ssata(**int *src* , int *shift* , int *bitpos* **);**	**SSAT** *dst* , **#** *bitpos* , *src* , **ASR #** *shift*	右移 src 并饱和到 bitpos 指定的可选有符号范围
int *dst* =**_ssatl(**int *src* , int *shift* , int *bitpos* **);**	**SSAT** *dst* , **#** *bitpos* , *src* , **LSL #** *shift*	左移 src 并饱和到 bitpos 指定的可选有符号范围
int *dst* =**_ssub(**int *src1* , int *src2* **);**	**QSUB** *dst* , *src1* , *src2*	饱和减法
int *dst* = **_ssub16(**int *src1* , int *src2* **);**	**SSUB16** *dst* , *src1* , *src2*	执行两个有符号半字减法
int *dst* = **_ssub8(**int *src1* , int *src2* **);**	**SSUB8** *dst* , *src1* , *src2*	执行四个有符号 8 位减法
int *dst* = **_ssubaddx(**int *src1* , int *src2* **);**	**SSAX** *dst* , *src1* , *src2*	交换 src2 的半字，减去顶部半字并添加底部半字
int *status* = **_ _strex(**unsigned int *src* , void* *dst* **);**	**STREX** *status* , *src* , *dest*	在内存地址中存储文字（32 位）数据
int *status* = **_ _strexb(**unsigned char *src* , void* *dst* **);**	**STREXB** *status* , *src* , *dest*	在内存地址中存储字节数据
int *status* = **_ _strexd(**unsigned long long *src* , void* *dst* **);**	**STREXD** *status* , *src* , *dest*	在内存地址中存储超长整型数据
int *status* = **_ _strexh(**unsigned short *src* , void* *dst* **);**	**STREXH** *status* , *src* , *dest*	在内存地址中存储半字（16 位）数据
int *dst* = **_subc(**int *src1* , int *src2* **);**	**SUBC** *dst* , *src1* , *src2*	带进位的减法
int *dst* **_sxtab(**int *src1* , int *src2* , int *rotamt* **);**	**SXTAB** *dst* , *src1* , *src2* , **ROR #** *rotamt*	从 src2 中提取一个可选的旋转 8 位值，并将其符号扩展到 32 位，然后将该值添加到 src1。旋转量可以是 0、8、16 或 24。
int *dst* **_sxtab16(**int *src1* , int *src2* , int *rotamt* **);**	**SXTAB16** *dst* , *src1* , *src2* , **ROR #** *rotamt*	从 src2 中提取两个可选的旋转 8 位值，并将其符号扩展到每个 16 位，然后将这些值添加到 src1 中的两个 16 位值。旋转量应为 0、8、16 或 24。
int *dst* **_sxtah(**int *src1* , int *src2* , int *rotamt* **);**	**SXTAH** *dst* , *src1* , *src2* , **ROR #** *rotamt*	从 src2 中提取一个可选的旋转 16 位值，并将其符号扩展到 32 位，然后将结果添加到 src1。旋转量可以是 0、8、16 或 32。
int *dst* **_sxtb(**int *src1* , int *rotamt* **);**	**SXTB** *dst* , *src1* , **ROR #** *rotamt*	从 src1 中提取一个可选的旋转 8 位值，并将其符号扩展到 32 位。旋转量可以是 0、8、16 或 24。
int *dst* **_sxtb16(**int *src1* , int *rotamt* **);**	**SXTAB16** *dst* , *src1* , **ROR #** *rotamt*	从 src1 中提取两个可选的旋转 8 位值，并将其符号扩展到 16 位。旋转量可以是 0、8、16 或 24。
int *dst* **_sxth(**int *src1* , int *rotamt* **);**	**SXTH** *dst* , *src1* , **ROR #** *rotamt*	从 src2 中提取一个可选的旋转 16 位值，并将其符号扩展到 32 位。旋转量可以是 0、8、16 或 24。
int *dst* = **_uadd16(**int *src1* , int *src2* **);**	**UADD16** *dst* , *src1* , *src2*	执行两个无符号半字加法
int *dst* = **_uadd8(**int *src1* , int *src2* **);**	**UADD8** *dst* , *src1* , *src2*	执行四个无符号 8 位加法

表 5-4. ARM 编译器内在函数 (continued)

C/C++ 编译器内在函数	汇编 指令	说明
int *dst* = **_uaddsubx(**int *src1* , int *src2* **);**	**UASX** *dst* , *src1* , *src2*	交换 src2 的半字，添加顶部半字并减去底部半字
int *dst* = **_uhadd16(**int *src1* , int *src2* **);**	**UHADD16** *dst* , *src1* , *src2*	执行两个无符号的半字加法并将结果减半
int *dst* = **_uhadd8(**int *src1* , int *src2* **);**	**UHADD8** *dst* , *src1* , *src2*	执行四个无符号的 8 位加法并将结果减半
int *dst* = **_uhsub16(**int *src1* , int *src2* **);**	**UHSUB16** *dst* , *src1* , *src2*	执行两个无符号的半字减法并将结果减半
int *dst* = **_uhsub8(**int *src1* , int *src2* **);**	**UHSUB8** *dst* , *src1* , *src2*	执行四个无符号的 8 位减法并将结果减半
int *dst* = **_umaal(**long long *acc* , int *src1* , int *src2* **);**	**UMAAL** *dst1* , *dst2* , *src1* , *src2*	对 src1 和 src2 执行无符号 32 位乘法，然后在 acc 中添加两个无符号 32 位值。
int *dst* = **_uqadd16(**int *src1* , int *src2* **);**	**UQADD16** *dst* , *src1* , *src2*	执行两个无符号半字饱和加法
int *dst* = **_uqadd8(**int *src1* , int *src2* **);**	**UQADD8** *dst* , *src1* , *src2*	执行四个无符号饱和 8 位加法
int *dst* = **_uqaddsubx(**int *src1* , int *src2* **);**	**UQASX** *dst* , *src1* , *src2*	交换 src2 的半字，对顶部半字执行无符号饱和加法，对底部半字执行无符号饱和减法。
int *dst* = **_uqsub16(**int *src1* , int *src2* **);**	**UQSUB16** *dst* , *src1* , *src2*	执行两个无符号饱和半字减法
int *dst* = **_uqsub8(**int *src1* , int *src2* **);**	**UQSUB8** *dst* , *src1* , *src2*	执行四个无符号饱和 8 位减法
int *dst* = **_uqsubaddx(**int *src1* , int *src2* **);**	**UQSAX** *dst* , *src1* , *src2*	交换 src2 的半字，对顶部半字执行无符号饱和减法，对底部半字执行无符号饱和加法。
int *dst* = **_usad8(**int *src1* , int *src2* **);**	**USAD8** *dst* , *src1* , *src2*	执行四个无符号 8 位减法，并将差值的绝对值相加。
int *dst* =**_usat16(**int *src* , int *bitpos* **);**	**USAT16** *dst* , **#** *bitpos*	对 bitpos 指定的可选无符号范围执行两个半字饱和
int *dst* =**_usata(**int *src* , int *shift* , int *bitpos* **);**	**USAT** *dst* , **#** *bitpos* , *src* , **ASR #** *shift*	右移 src 并饱和到 bitpos 指定的可选无符号范围
int *dst* =**_usatl(**int *src* , int *shift* , int *bitpos* **);**	**USAT** *dst* , **#** *bitpos* , *src* , **LSL #** *shift*	左移 src 并饱和到 bitpos 指定的可选无符号范围
int *dst* = **_usub16(**int *src1* , int *src2* **);**	**USUB16** *dst* , *src1* , *src2*	执行两个无符号半字减法
int *dst* = **_usub8(**int *src1* , int *src2* **);**	**USUB8** *dst* , *src1* , *src2*	执行四个无符号 8 位减法
int *dst* = **_usubaddx(**int *src1* , int *src2* **);**	**USAX** *dst* , *src1* , *src2*	交换 src2 的半字，减去顶部半字并添加底部半字
int *dst* **_uxtab(**int *src1* , int *src2* , int *rotamt* **);**	**UXTAB** *dst* , *src1* , *src2* , **ROR #** *rotamt*	从 src2 中提取一个可选的旋转 8 位值，并将 0 扩展到 32 位，然后将该值添加到 src1。旋转量可以是 0、8、16 或 24。
int *dst* **_uxtab16(**int *src1* , int *src2* , int *rotamt* **);**	**UXTAB16** *dst* , *src1* , *src2* , **ROR #** *rotamt*	从 src2 中提取两个可选的旋转 8 位值，并将 0 扩展到每个 16 位，然后将这些值添加到 src1 中的两个 16 位值。旋转量应为 0、8、16 或 24。
int *dst* **_uxtah(**int *src1* , int *src2* , int *rotamt* **);**	**UXTAH** *dst* , *src1* , *src2* , **ROR #** *rotamt*	从 src2 中提取一个可选的旋转 16 位值，并将 0 扩展到 32 位，然后将结果添加到 src1。旋转量可以是 0、8、16 或 32。
int *dst* **_uxtb(**int *src1* , int *rotamt* **);**	**UXTB** *dst* , *src1* , **ROR #** *rotamt*	从 src2 中提取一个可选的旋转 8 位值，并将 0 扩展到 32 位。旋转量可以是 0、8、16 或 24。
int *dst* **_uxtb16(**int *src1* , int *rotamt* **);**	**UXTB16** *dst* , *src1* , **ROR #** *rotamt*	从 src1 中提取两个可选的旋转 8 位值，并将 0 扩展到 16 位。旋转量可以是 0、8、16 或 24。
int *dst* **_uxth(**int *src1* , int *rotamt* **);**	**UXTH** *dst* , *src1* , **ROR #** *rotamt*	从 src2 中提取一个可选的旋转 16 位值，并将 0 扩展到 32 位。旋转量可以是 0、8、16 或 24。
void **_ _wfe(** void **);**	**WFE**	等待事件。通过等待异常或事件来节省电源。
void **_ _wfi(** void **);**	**WFI**	等待中断。进入待机、休眠或关机模式，在此类模式中，需要中断才能将处理器唤醒。

此外，编译器还支持 ARM C 语言扩展 (ACLE) 规范中所述的许多内在函数。这些内在函数适用于 Cortex-M 和 Cortex-R 处理器变体。实现 ACLE 内在函数以支持源代码开发，这些源代码可以使用多个供应商提供的 ACLE 兼

容编译器为各种 ARM 处理器进行编译。许多内在函数与上表中列出的内在函数重复，但名称略有不同（例如一个与两个前导下划线）。

编译器不支持 ACLE 规格中列出的所有 ACLE 内在函数。例如，CLS 指令在 Cortex-M 或 Cortex-R 架构上不可用，因此不支持 __cls、__clsl 和 __clsll ACLE 内在函数。

为了使用 ACLE 内在函数，您的代码必须包含提供的 `arm_acle.h` 头文件。有关 ACLE 内在函数的详细信息，请参阅 ACLE 规格。有关支持哪些 ACLE 内在函数的信息，请参阅 `arm_acle.h` 文件。在适用的情况下，不受支持的 ACLE 内在函数声明将包含在该头文件中的注释中，并简要说明不支持该内在函数的原因，以及包含对 ACLE 规格中描述该内在函数的相应章节的引用。

5.15 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 32 位函数的 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 compiled in 16-bit mode
```

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 16 位函数。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 8，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

备注

EABI 模式 C++ 还原： EABI 模式具有不同的 C++ 还原方案。例如，没有前缀（_ 或 $）。有关详细信息，请参阅 ARM 信息中心。

5.16 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 5.16.1。）

- ANSI/ISO C11（--c11，请参阅节 5.16.2）
- 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 5.16.3。）

如果您想将使用 TI CodeGen 工具创建的目标文件与其他编译器工具链生成的目标文件链接起来，根据 ARM 标准的要求，您应先定义 _AEABI_PORTABILITY_LEVEL 预处理器符号（如下所示），然后再包含任何标准头文件，如 <stdlib.h>。

```
#define _AEABI_PORTABILITY_LEVEL 1
```

此定义可实现完全可移植性。将符号定义为 0 指定将使用“C 标准”可移植性级别。

5.16.1 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()
- 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

5.16.2 C11 支持 (--c11)

编译器支持 ISO 标准化的 2011 年标准 C。但是，除了节 5.16.1 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

5.16.3 严格 ANSI 模式和宽松 ANSI 模式 (--strict_ansi 和 --relaxed_ansi)

在宽松 ANSI/ISO 模式 (默认模式) 下，编译器接受可能与严格遵循 ANSI/ISO C/C++ 的程序相冲突的语言扩展。在严格 ANSI 模式下，这些语言扩展遭到抑制，因此编译器将接受所有严格遵循规范的程序。

当您知道您的程序是一个遵循规范的程序，并且不会在宽松模式下编译时，请使用 --strict_ansi 选项。在此模式下，与 ANSI/ISO C/C++ 相冲突的语言扩展将被禁用，编译器将在标准要求时发出错误消息。本标准视为酌情处理的违规行为可作为警告发出。

示例：

以下是严格遵循规范的 C 代码，但在默认宽松模式下将不被编译器接受。若要使编译器接受这种代码，请使用严格 ANSI 模式。编译器将抑制 interrupt 关键字语言异常，然后，interrupt 可用作代码中的标识符。

```
int main()
{
    int interrupt = 0;
    return 0;
}
```

以下是未严格遵循规范的代码。编译器将不接受这种严格 ANSI 模式下的代码。若要使编译器接受这种代码，请使用宽松 ANSI 模式。 编译器将提供 interrupt 关键字扩展并接受此代码。

```
interrupt void isr(void);
int main()
{
    return 0;
}
```

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

```
__interrupt void isr(void);
int main()
{
    return 0;
}
```

默认模式为宽松 ANSI。 可以通过 --relaxed_ansi (或 -pr) 选项来选择此模式。宽松 ANSI 模式接受种类最多的程序，以及所有 TI 语言扩展，即使是那些与 ANSI/ISO 相冲突的扩展，也会忽略一些编译器能够合理处理的 ANSI/ISO 冲突。节 5.17 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。

5.17 GNU 、Clang 和 ACLE 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 5.17.6)
- __has_attribute

此外，编译器还支持 ARM C 语言扩展 (ACLE) 规范中描述的许多功能。这些功能适用于 Cortex-M 和 Cortex-R 处理器变体。ACLE 支持会影响您可能在 C/C++ 代码中使用的预定义宏命令 (表 2-31)、函数属性 (节 5.17.2) 和内在函数 (节 5.14)。实现这些功能以支持源代码开发，这些源代码可以使用多个供应商提供的 ACLE 兼容编译器为各种 ARM 处理器进行编译。

5.17.1 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 5-5 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 5-5. 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

表 5-5. GCC 语言扩展 (continued)

扩展	说明
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
函数原型	原型声明和旧式定义
C++ 注释	系统会识别 C++ 注释。
美元符号	标识符中允许使用美元符号。
字符转义	字符 ESC 表示为 \e
变量属性	指定变量的属性
类型属性	指定类型的属性
对齐	查询类型或变量的对齐情况
内联	定义内联函数（和宏命令一样快）
汇编标签	指定要用于 C 符号的汇编器名称
扩展的 asm	带有 C 操作数的汇编器指令
约束条件	asm 操作数的约束条件
包装器头文件	包装器头文件可以使用 #include_next 包含另一个版本的头文件
替代关键字	头文件可以使用 __const__、__asm__ 等
显式寄存器变量	定义驻留在指定寄存器中的变量
不完整的枚举类型	定义枚举标签而不指定其可能的值
函数名称	作为当前函数名称的可打印字符串
返回地址	获取函数的返回地址或帧地址（有限支持）
其他内置	其他内置函数（请参见节 5.17.6）
矢量扩展	通过内置函数使用矢量指令
目标内置	专用于特定目标的内置函数
Pragma	GCC 接受的 pragma
未命名字段	结构体/联合体中的未命名结构体/联合体字段
线程本地	每线程变量
二进制常量	使用“0b”前缀的二进制常量。

(1) 为 GCC 3.0 定义的功能；请访问 http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/C-Extensions.html#C-Extensions 查看定义和示例

5.17.2 函数属性

支持以下 GCC 函数属性：

- alias

- aligned
- always_inline
- calls
- const
- constructor
- deprecated
- format
- format_arg
- interrupt
- malloc
- naked
- noinline
- noreturn
- pure
- section
- target
- unused
- used
- warn_unused_result
- weak

支持以下其他 TI 特定函数属性：

- retain
- ramfunc

例如，此函数声明使用 **alias** 属性使“my_alias”成为“myFunc”函数的别名：

```
void my_alias() __attribute__((alias("myFunc")));
```

aligned 函数属性会使用指定的对齐方式来对齐函数。该对齐必须为 2 的幂。

always_inline 函数属性与 FUNC_ALWAYS_INLINE pragma 具有相同的效果。请参阅 节 5.11.12

calls 属性与 CALLS pragma 具有相同的效果，相关描述请参阅 节 5.11.1。

format 属性应用于 stdio.h 中 printf、fprintf、sprintf、snprintf、vprintf、vfprintf、vsprintf、vsnprintf、scanf、fscanf、vfscanf、vscanf、vsscanf 和 sscanf 的声明。因此，当启用 GCC 扩展时，系统会根据格式字符串参数中的格式说明符对这些函数的数据参数进行类型检查，并在不匹配时发出警告。如果不需要这些警告，可以通过常见方式抑制这些警告。

有关如何使用 **interrupt** 函数属性的更多信息，请参阅节 5.11.16。

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

naked 属性标识了使用 `__asm` 语句编写为嵌入式汇编函数的函数。编译器不会为此类函数生成序言和结语序列。请参阅节 5.10。

noinline 函数属性与 FUNC_CANNOT_INLINE pragma 具有相同的效果。请参阅 节 5.11.13

ramfunc 属性指定一个函数将被放置在 RAM 中并从中执行。ramfunc 属性允许编译器优化 RAM 执行的函数，以及自动将函数复制到基于闪存的器件上的 RAM 上。例如：

```
__attribute__((ramfunc))
void f(void) {
   ...
}
```

--ramfunc=on 选项指定使用此选项编译的所有函数都放置在 RAM 中并从中执行，即使未使用此函数属性也是如此。

较新的 TI 链接器命令文件通过将具有此属性的函数放置在 .TI.ramfunc 段中来自动支持 ramfunc 属性。如果您的链接器命令文件不包含 .TI.ramfunc 段的段规格，您可以修改链接器命令文件以将此段放在 RAM 中。有关段放置的详细信息，请参阅 *ARM 汇编语言工具用户指南*。

target 属性使函数在 ARM（32 位）或 Thumb（16 位）模式下编译。target 属性与 CODE_STATE pragma 的效果相同。下述示例使用 target 属性。

```
__attribute__((target("arm"))) void foo(int arg1, int arg2)
__attribute__((target("thumb"))) void foo(int arg1, int arg2)
```

请注意，ACLE 规格中所述的“pcs”属性不受支持。

retain 属性与 RETAIN pragma (节 5.11.27) 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 5.11.4

weak 属性与 WEAK pragma (节 5.11.32) 具有相同的效果。

5.17.3 For 循环属性

如果您使用的是 C++，则有几个特定于 TI 的属性可应用于循环。C 中没有相应的语法。以下 TI 特定属性与其相应程序具有相同的功能：

- TI::must_iterate
- TI::unroll

有关使用 for 循环属性的示例，请参阅节 5.11.18.1。

5.17.4 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 5.11.6

location 属性与 LOCATION pragma 的效果相同。请参阅节 5.11.17。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《*ARM 汇编语言工具用户指南*》中的“RAM 模型与 ROM 模型”章节及其小节。

noinit 属性可用在未初始化的变量上；可防止这些变量在重置期间被设置为 0。**persistent** 属性可用在初始化的变量上；可防止这些变量在重置期间被初始化。默认情况下，标记为 noinit 或 persistent 的变量将分别置于 .TI.noinit 和 .TI.persistent 段。这些段的位置由链接器命令文件控制。通常对于支持 FRAM 的器件，.TI.persistent 段置于 FRAM 中，.TI.noinit 段置于 RAM 中。也请参见 节 5.11.19。

packed 属性可应用于结构体或联合体中的单个字段。所有 ARM 目标均支持 packed 属性。有关编译器如何访问未对齐数据的更多信息，请参阅 --unaligned_access 选项说明。

retain 属性与 RETAIN pragma (节 5.11.27) 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 5.11.7

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 5.11.32) 的效果相同。

5.17.5 类型属性

编译器支持以下类型属性：

- aligned
- deprecated
- packed
- transparent_union
- unused

结构体和联合体类型都支持 **packed** 属性。如果使用了 --relaxed_ansi 选项，则在所有 ARM 目标上都受支持。请参阅 --unaligned_access 选项说明，了解更多有关编译器如何访问未对齐数据的信息。

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。

压缩结构的位字段是位对齐的。不是位字段的相邻结构成员的子节对齐方式不变。不过，相邻位字段之间没有填充位。

“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*。

5.17.6 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_llabs()
- __builtin_sqrt()
- __builtin_sqrtf()
- __builtin_memcpy()
- __builtin_return_address()

__builtin_frame_address() 函数始终返回 0，除非参数是常数零。

仅当启用了硬件浮点支持时，才支持 __builtin_sqrt() 和 __builtin_sqrtf() 函数。此外，如果 --float_support 设置为 fpv4spd16，则不支持 __builtin_sqrt() 函数。

调用运行时可能不可用的内置函数时，请使用以下示例中所示的 Clang __has_builtin 宏命令，以确保该函数受支持：

```
#if __has_builtin(__builtin_sqrt)
double estimate = __builtin_sqrt(x);
#else
double estimate = fast_approximate_sqrt(x);
#endif
```

如果支持内置函数，且设备具有适当的硬件支持，则内置函数将调用硬件支持。

如果支持内置函数，但设备未启用相应的硬件，则内置函数通常会成为对 RTS 库函数的调用。例如，__builtin_sqrt() 将成为对库函数 sqrt() 的调用。

__builtin_return_address() 函数始终返回零。

5.18 AUTOSAR

ARM 编译器通过提供以下头文件来支持 AUTOSAR 3.1 标准：

- Compiler.h
- Platform_Types.h
- Std_Types.h
- Compiler_Cfg.h

Compiler_Cfg.h 是一个空文件，其内容应由最终用户提供。提供的文件包含文件应包括哪些内容的信息。它包含在 Compiler.h 中。如果用户提供了新的 Compiler_Cfg.h 文件，则其包含路径必须位于运行时支持头文件的路径之前。

有关 AUTOSAR 的更多信息，请访问 http://www.autosar.org。

5.19 编译器限制

由于 C/C++ 编译器支持的主机系统的多样性，以及其中一些系统的局限性，编译器可能无法成功编译过大或过于复杂的源文件。通常，超过此系统限制会阻止继续编译，因此编译器会在打印错误消息后立即中止。简化程序以避免超过系统限制。

某些系统不允许文件名长度超过 500 个字符。确保您的文件名短于 500 个字符。

编译器并非任意限制，但受主机系统上可用内存量的限制。在较小的主机系统（如 PC）上，优化器可能会耗尽内存。如果出现这种情况，优化器将终止，shell 将继续使用代码生成器编译文件。这会导致编译文件时没有进行优化。优化器一次编译一个函数，因此更可能的原因是源模块中的函数太大或非常复杂。若要更正此问题，您可以进行如下选择：

- 不要优化有问题的模块。
- 确定导致问题的函数，并将其分解为较小的函数。
- 从模块中提取函数，并将其放在一个单独的模块中，该模块可在不进行优化的情况下进行编译，以便对其余函数进行优化。

章节 6

运行时环境

本章介绍 ARM C/C++ 运行时环境。为确保 C/C++ 程序的成功执行，所有运行时代码维护这一环境是至关重要的。如果要编写与 C/C++ 代码交互的汇编语言函数，那么遵循本章中的指导原则也是很重要的。

6.1 存储器模型 134
6.2 对象表示 136
6.3 寄存器惯例 143
6.4 函数结构和调用惯例 145
6.5 访问 C 和 C++ 中的链接器符号 148
6.6 将 C 和 C++ 与汇编语言相连 148
6.7 中断处理 151
6.8 固有运行时支持算术和转换例程 155
6.9 内置函数 156
6.10 系统初始化 156
6.11 TIABI 下的双状态交互工作（已弃用） 165

6.1 存储器模型

ARM 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。编译器假设目标内存中有完整的 32 位地址空间可用。

备注

链接器定义内存映射

由链接器而不是编译器定义内存映射并将代码和数据分配到目标内存中。编译器不考虑可用内存的类型、不考虑代码或数据（漏洞）的任何不可用的位置，也不考虑为 I/O 或控制目的保留的任何位置。编译器生成可重定位代码，允许链接器将代码和数据分配到合适的内存空间中。例如，可以使用链接器将全局变量分配到片上 RAM 中或将可执行代码分配到外部 ROM 中。可以将每个代码块或数据块单独分配到内存中，但这不是通用做法（一个例外是内存映射 I/O ，尽管可以使用 C/C++ 指针类型访问物理存储器位置）。

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 *ARM 汇编语言工具用户指南*中介绍的目标文件信息。。

段有两种基本的类型：

- **初始化段**包含数据或可执行代码。初始化段通常是只读的；例外情况如下所示。C/C++ 编译器会创建以下初始化段：
  - **.binit 段**包含引导时复制表。有关 BINIT 的详细信息，请参阅 *ARM 汇编语言工具用户指南*。
  - **.init_array 段**包含全局构造函数表。
  - **.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**包含初始化的全局变量和静态变量。此段是可写的。
  - **.const 段**包含只读数据，通常为字符串常量和使用 C/C++ 限定符 *const* 定义的静态作用域对象。请注意，并非所有标记为“const”的静态作用域对象都被放置在 .const 段中（请参阅节 5.7.1）。
  - **.text 段**包含所有可执行代码。该段还包含字符串文字、开关表和由编译器生成的常量。此段通常是只读的。请注意，某些字符串文字可能会放置在 .const:.string 中。字符串文字的位置取决于字符串的大小以及是否使用 --embedded_constants 选项。
  - **.TI.crctab 段**包含 CRC 检查表。
- **未初始化的段**会在存储器中保留空间（通常为 RAM）。程序可以在运行时使用此空间来创建和存储变量。编译器会创建以下未初始化的段：
  - 仅对于 EABI，**.bss 段**为未初始化的全局变量和静态变量保留空间。未初始化且也未使用的变量通常创建为通用符号（除非指定 --common=off），而不是放置在 .bss 中，以便将该变量从生成的应用中排除。
  - **.stack 段**为保留空间。C/C++ 软件栈。
  - **.sysmem 段**为动态存储器分配保留空间。此空间由动态存储器分配例程使用，如 malloc()、calloc()、realloc() 或 new()。如果 C/C++ 程序不使用这些函数，编译器不会创建 .sysmem 段。

汇编器会创建默认段 .text、.bss 和 .data。您可以指示编译器使用 CODE_SECTION 和 DATA_SECTION pragma 创建其他段（请参阅节 5.11.4 和节 5.11.7）。

链接器从不同的目标文件中获取各个段，并合并具有相同名称的段。表 6-1 中列出了生成的输出段，以及每个段在存储器中的适当位置。您可以根据需要将这些输出段放置在地址空间中的任何位置，以满足系统要求。

表 6-1. 段和存储器位置摘要

段	存储器类型	段	存储器类型
.bss	RAM	.pinit	ROM 或 RAM
.cinit	ROM 或 RAM	.stack	RAM
.const	ROM 或 RAM	.sysmem	RAM
.data	RAM	.text	ROM 或 RAM

表 6-1. 段和存储器位置摘要 (continued)

段	存储器类型	段	存储器类型
.init_array	ROM 或 RAM		

可以使用链接器命令文件中的 SECTIONS 指令来自定义段分配过程。有关将段分配到存储器中的更多信息，请参阅 *ARM 汇编语言工具用户指南* 中的链接器说明一章。

6.1.2 C/C++ 系统堆栈

C/C++ 编译器使用堆栈来：

- 分配局部变量
- 传递参数给函数
- 保存寄存器内容

运行时堆栈从高位地址增长到低位地址。编译器使用 R13 寄存器来管理此堆栈。R13 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 2048 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《*ARM 汇编语言工具用户指南*》中的链接器描述章节。

在系统初始化时，SP 被设置为堆栈顶部的指定地址。此地址是 .stack 段末尾之后的第一个位置。由于堆栈的位置取决于 .stack 段的分配位置，因此堆栈的实际地址是在链接时确定的。

C/C++ 环境在函数输入时自动递减 SP，以保留执行该函数所需的所有空间。堆栈指针在函数出口处递增，以将堆栈恢复到函数输入之前的状态。如果将汇编语言例程连接到 C/C++ 程序，请确保将堆栈指针恢复到函数输入之前的相同状态。

更多有关使用堆栈指针的信息，请参阅节 6.3；更多有关堆栈的信息，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 2.15。

6.1.3 动态存储器分配

ARM 编译器随附的运行时支持库包含几个函数（例如 malloc、calloc 和 realloc），这些函数允许您在运行时为变量动态地分配存储器。

内存是从 .sysmem 段中定义的全局池（或堆）分配的。可以在链接器命令中使用 heap_size=*size* 选项来更改 .sysmem 段的大小。链接器还会创建一个全局符号 __TI_SYSMEM_SIZE，并为其分配一个等于堆大小的值（以字节为单位）。默认大小为 2048 字节。有关 --heap_size 选项的更多信息，请参阅 *ARM 汇编语言工具用户指南* 中的链接器说明一章。

如果您使用任何 C I/O 函数，RTS 库会为您访问的每个文件分配一个 I/O 缓冲区。这个缓冲区将比 BUFSIZ 大一点，BUFSIZ 在 stdio.h 中定义，默认为 256）。确保为这些缓冲区分配了足够大的堆或使用 setvbuf 将缓冲区更改为静态分配的缓冲区。

动态分配的对象并非采用直接寻址方式（始终使用指针访问），并且存储器池位于单独的段（.sysmem）中。因此，动态存储器池的大小仅受系统中可用存储器大小的限制。为了节省 .bss 段的空间，可以从堆中分配大型数组，而不是将它们定义为全局或静态数组。例如，不是定义如下：

```
struct big table[100];
```

而是改用指针并调用 malloc 函数：

```
struct big *table
table = (struct big *)malloc(100*sizeof(struct big));
```

当从堆进行分配时，请确保堆的大小足够满足分配要求。在分配可变长度数组时，这一点尤为重要。

6.2 对象表示

有关数据类型的基本信息，请参阅节 5.5。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

表 6-2 列出了各种数据类型的寄存器和内存存储空间：

表 6-2. 寄存器和内存中的数据表示

数据类型	寄存器存储	内存存储
char、signed char	寄存器的 0-7 位 (1)	8 位，与 8 位边界对齐
unsigned char、bool	寄存器的 0-7 位	8 位，与 8 位边界对齐
short、signed short	寄存器的 0-15 位 (1)	16 位，与 16 位（半字）边界对齐
unsigned short、wchar_t	寄存器的 0-15 位	16 位，与 16 位（半字）边界对齐
int、signed int	寄存器的 0-31 位	32 位，与 32 位（字）边界对齐
unsigned int	寄存器的 0-31 位	32 位，与 32 位（字）边界对齐
long、signed long	寄存器的 0-31 位	32 位，与 32 位（字）边界对齐
unsigned long	寄存器的 0-31 位	32 位，与 32 位（字）边界对齐
long long	偶数/奇数寄存器对	64 位，与 32 位（字）边界对齐(2)
unsigned long long	偶数/奇数寄存器对	64 位，与 32 位（字）边界对齐(2)
float	寄存器的 0-31 位	32 位，与 32 位（字）边界对齐
double	寄存器对	64 位，与 32 位（字）边界对齐(2)
long double	寄存器对	64 位，与 32 位（字）边界对齐(2)
结构体	成员按其各自类型的要求存储。	成员按其各自类型的要求存储；根据具有最严格的对齐要求对成员对齐。
数组	成员按其各自类型的要求存储。	成员按其各自类型的要求存储；与 32 位（字）边界对齐。结构中的所有数组都根据数组中每个元素的类型对齐。
数据成员指针	寄存器的 0-31 位	32 位，与 32 位（字）边界对齐
成员函数指针	组件按其各自类型的要求存储	64 位，与 32 位（字）边界对齐

(1) 负值符号扩展到 31 位.。
(2) 64 位数据在 64 位边界上对齐。

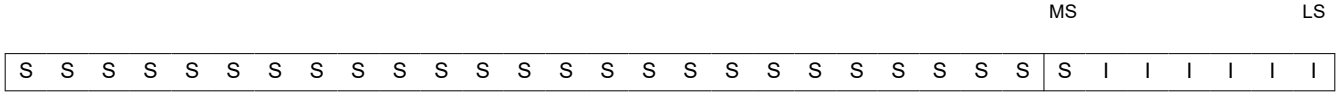
有关枚举类型大小的详细信息，请参阅表 5-2。

6.2.1.1 char 和 short 数据类型（有符号和无符号）

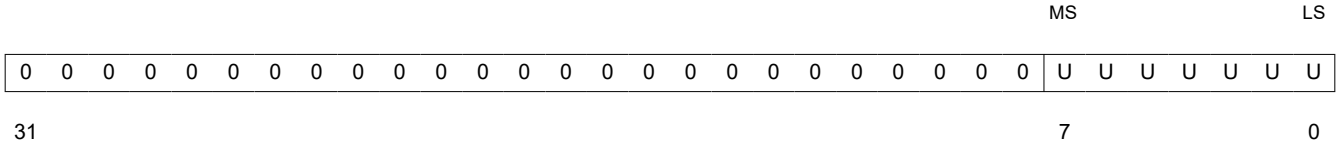
char 和 unsigned char 数据类型作为单个字节存储在内存中，数据类型加载到寄存器的 0-7 位上，并从这些位进行存储（请参阅图 6-1）。定义为 short 或 unsigned short 的对象作为两个字节存储在内存中，位于半字（2 字节）对齐的地址上；这些对象加载到寄存器的 0-15 位上，并从这些位进行存储（请参阅图 6-1）。

图 6-1. Char 和 Short 数据存储格式

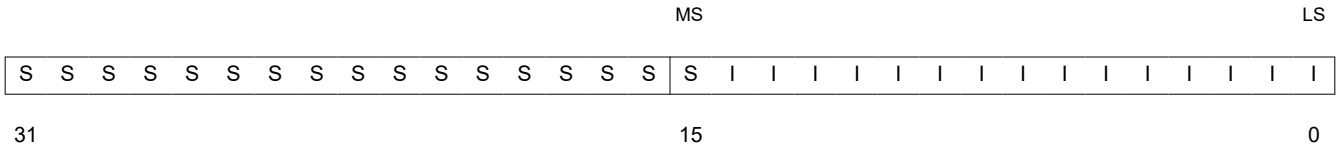
有符号 8 位 char



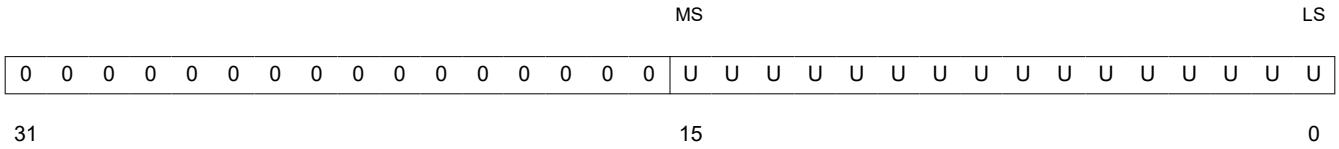
无符号 8 位 char



有符号 16 位 short



无符号 16 位 short



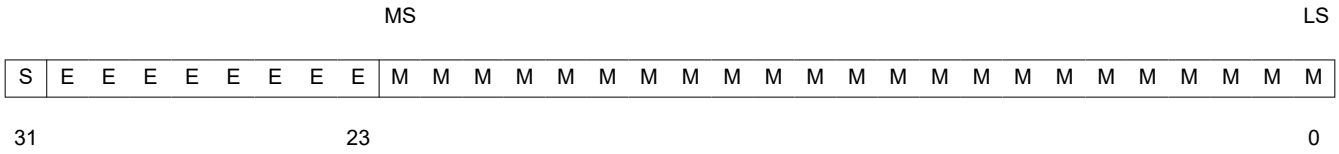
图例: S = 符号，I = 有符号整数，U = 无符号整数，MS = 最高有效，LS = 最低有效

6.2.1.2 float、int 和 long 数据类型（有符号和无符号）

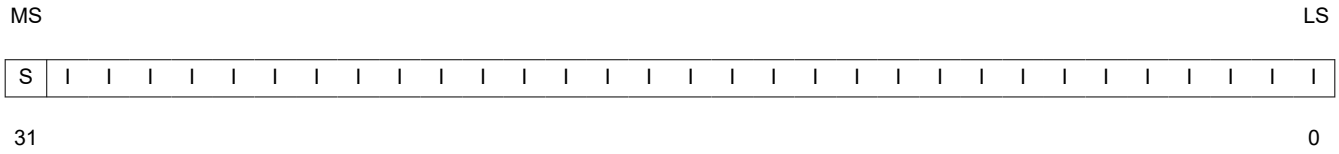
int、unsigned int、float、long 和 unsigned long 数据类型作为 32 位对象存储在存储器中，位于字（4 字节）对齐的地址处。将这些类型的对象加载到并存储在寄存器的位 0-31 ，如图 6-2 所示。在大端字节序模式下，通过以下方式将 4 字节对象加载到寄存器中：将存储器的第一个字节（即低位地址）移动到寄存器的位 24-31，将存储器的第二个字节移动到位 16-23，将第三个字节移动到位 8-15，并将第四个字节移到位 0-7。在小端字节序模式下，通过以下方式将 4 字节对象加载到寄存器中：将存储器的第一个字节（即低位地址）移动到寄存器的位 0-7，将第二个字节移动到位 8-15，将第三个字节移动到位 16-23，并将第四个字节移到位 24-31。

图 6-2. 32 位数据存储格式

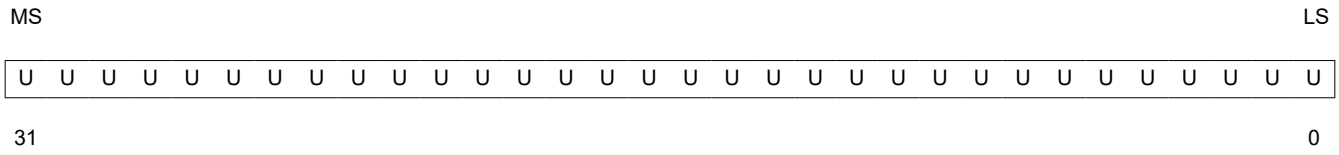
单精度浮点字符



带符号 32 位整数或 long 字符



无符号 32 位整数或 long



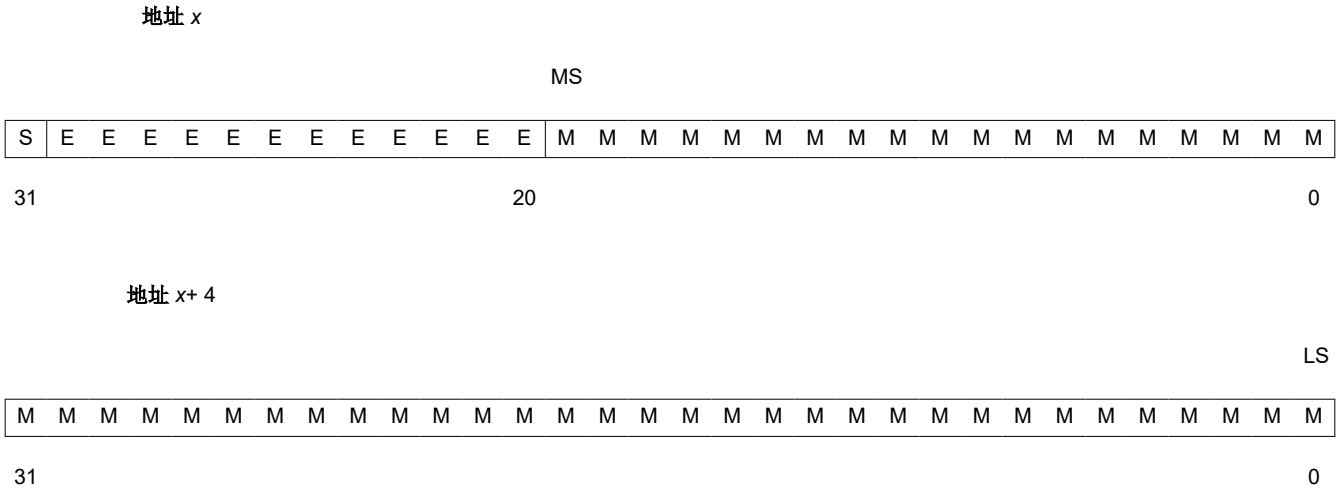
图例: S = 符号，M = 尾数，U = 无符号整数，E = 指数，I = 带符号整数，MS = 最高有效，LS = 最低有效

6.2.1.3 double、long double 和 long long 数据类型（有符号和无符号）

Double、long double、long long 和 unsigned long long 数据类型存储在内存中的一对寄存器中，并且始终被成对引用。这些类型作为 64 位对象存储在字（4 字节）对齐的地址上。对于 FPA 模式，位于最低地址上的字包含符号位、指数以及尾数的最高有效部分。位于最高地址上的字包含尾数的最低有效部分。无论目标的字节序如何，都是如此。对于 VFP 模式，字是根据目标的字节序排列的。

此类型的对象会加载到并存储在寄存器对中，如下图所示。最高有效内存字包含符号位、指数以及尾数的最高有效部分。最低有效内存字包含尾数的最低有效部分。

图 6-3. 双精度浮点数据存储格式



图例：S = 符号，M = 尾数，E = 指数，MS = 最高有效，LS = 最低有效

6.2.1.4 指向数据成员类型的指针

指向数据成员对象的指针像 unsigned int（32 位）整数类型一样存储在存储器中。它的值是类中数据成员的字节偏移加上 1。零值是保留值，表示指向数据成员的 NULL 指针。

6.2.1.5 指向成员函数类型的指针

指向成员函数对象的指针存储为具有三个成员的结构，布局相当：

```
struct {
   short int d;
   short int i;
   union {
      void (f) ();
      long 0; }
   };
```

参数 d 是要添加到该指针的类对象开头的偏移量。参数 I 是虚函数表的索引，偏移 1。索引允许表示 NULL 指针。如果函数是非虚函数，则索引的值为 -1。当 I 为 0 时，如果参数 f 是非虚函数，则为指向成员函数的指针。0 是类对象中虚函数指针的偏移量。

6.2.1.6 结构和数组对齐

结构根据具有最严格对齐要求的成员进行对齐。结构被填充，以便结构的大小是其对齐大小的倍数。数组始终为字对齐。数组元素的存储方式与它们作为单个对象的存储方式相同。

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。位字段的大小可以从 1 位到 32 位不等，但它们从不跨越 4 字节边界。

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。

以下是有关如何处理位字段的一些详细信息：

- 纯文本 int 位字段是无符号的。考虑以下 C 代码：

```
struct st
{
   int a:5;
} S;
foo()
{
   if (S.a < 0)
      bar();
}
```

  在此示例中， bar()永远不会被调用，因为位字段“a”是无符号的。如果需要有符号位字段，请使用 signed int。
- 支持 long long 类型的位字段。
- 位字段被视为声明的类型。
- 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 6-4 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 6-4. 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 6.10。
- 在表达式中。例如：

```
strcpy (s, "abc");
```

  在表达式中使用字符串时，字符串本身是在 .const 段中使用 .string 汇编器指令定义的，并带有指向该字符串的唯一标签；包括终止 0 字节。例如，以下行定义了字符串 abc 和终止 0 字节（标签 SL5 指向该字符串）：

```
      .sect   ".const"
SL5: .string "abc",0
```

  字符串标签的形式为 SL*n*，其中 *n* 是编译器分配的数字，用于使标签唯一。该数字从 0 开始，每定义一个字符串就增加 1。源模块中使用的所有字符串都在编译后的汇编语言模块的末尾定义。

  标签 SL*n* 表示字符串常量的地址。编译器使用此标签引用字符串表达式。

  由于字符串存储在 .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。如计划在 C/C++ 程序中使用汇编语言例程，则必须理解并遵循这些寄存器惯例。

寄存器惯例规定了编译器如何使用寄存器以及如何在函数调用之间保留值。下表显示了受这些惯例影响的寄存器类型。“寄存器使用”表总结了编译器如何使用寄存器以及是否在各调用中保留寄存器的值。有关如何在各调用中保留这些值的信息，请参阅节 6.4。

表 6-3. 惯例对寄存器类型的影响

寄存器类型	说明
参数寄存器	在函数调用期间传递参数
返回寄存器	保留函数调用的返回值
表达式寄存器	保留值
参数指针	用作访问函数参数（传入参数）的基值
栈指针	保留软件栈顶的地址
链接寄存器	包含函数调用的返回地址
程序计数器	包含正在执行的代码的当前地址

表 6-4. 寄存器的使用

寄存器	别名	使用	由函数保留 [1]
R0	A1	参数寄存器、返回寄存器、表达式寄存器	父级
R1	A2	参数寄存器、返回寄存器、表达式寄存器	父级
R2	A3	参数寄存器、表达式寄存器	父级
R3	A4	参数寄存器、表达式寄存器	父级
R4	V1	表达式寄存器	子级
R5	V2	表达式寄存器	子级
R6	V3	表达式寄存器	子级
R7	V4，AP	表达式寄存器、参数指针	子级
R8	V5	表达式寄存器	子级
R9	V6	表达式寄存器	子级
R10	V7	表达式寄存器	子级
R11	V8	表达式寄存器	子级
R12	V9、1P	表达式寄存器、指令指针	父级
R13	SP	栈指针	子函数 [2]
R14	LR	链接寄存器、表达式寄存器	子级
R15	PC	程序计数器	不适用
CPSR		当前程序状态寄存器	子级
SPSR		已保存的程序状态寄存器	子级

表 6-5. VFP 寄存器使用

32 位寄存器	64 位寄存器	使用	由函数保留 [1]
FPSCR		状态寄存器	不适用
S0 S1	D0	浮点表达式、返回值、传递参数	不适用
S2 S3	D1	浮点表达式、返回值、传递参数	不适用
S4 S5	D2	浮点表达式、返回值、传递参数	不适用
S6 S7	D3	浮点表达式、返回值、传递参数	不适用
S8 S9	D4	浮点表达式、传递参数	不适用
S10 S11	D5	浮点表达式、传递参数	不适用
S12 S13	D6	浮点表达式、传递参数	不适用
S14 S15	D7	浮点表达式、传递参数	不适用
S16 S17	D8	浮点表达式	子级
S18 S19	D9	浮点表达式	子级
S20 S21	D10	浮点表达式	子级
S22 S23	D11	浮点表达式	子级
S24 S25	D12	浮点表达式	子级
S26 S27	D13	浮点表达式	子级
S28 S29	D14	浮点表达式	子级
S30 S31	D15	浮点表达式	子级
	D16-D31	浮点表达式	

表 6-6. Neon 寄存器使用

64 位寄存器	四倍寄存器	使用	由函数保留 [1]
D0 D1	Q0	SIMD 寄存器	不适用
D2 D3	Q1	SIMD 寄存器	不适用
D4 D5	Q2	SIMD 寄存器	不适用
D6 D7	Q3	SIMD 寄存器	不适用
D8 D9	Q4	SIMD 寄存器	子级
D10 D11	Q5	SIMD 寄存器	子级
D12 D13	Q6	SIMD 寄存器	子级
D14 D15	Q7：	SIMD 寄存器	子级
D16 D17	Q8	SIMD 寄存器	不适用
D18 D19	Q9	SIMD 寄存器	不适用
D20 D21	Q10	SIMD 寄存器	不适用
D22 D23	Q11	SIMD 寄存器	不适用
D24 D25	Q12	SIMD 寄存器	不适用
D26 D27	Q13	SIMD 寄存器	不适用
D28 D29	Q14	SIMD 寄存器	不适用
D30 D31	Q15	SIMD 寄存器	不适用
FPSCR		状态寄存器	不适用

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。

以下各节使用此术语来描述 C/C++ 编译器的函数调用惯例：

- **参数块**。本地帧的一部分，用于将参数传递给其他函数。采用将参数移至参数块而不是将其压入堆栈的方式来将这些参数传递给函数。本地帧和参数块同时被分配。
- **寄存器保存区域**。本地帧的一部分，用于在程序调用函数时保存寄存器，并在程序退出函数时恢复寄存器。
- **调用保存寄存器**。寄存器 R0-R3 和 R12（备用名称是 A1-A4 和 V9）。被调用的函数不会将这些值保留在寄存器中；因此，如果需要保留它们的值，调用函数必须保存它们。

- **入口保存寄存器**。寄存器 R4 至 R11 和 R14（备用名称是 V1 至 V8 和 LR）。被调用的函数负责将这些值保留在寄存器中。如果被调用函数修改这些寄存器，它会在获得控制权时保存它们，并在将控制权返回给调用函数时保留它们。

有关 EABI 模式下或使用 VFP 协处理器时的调用惯例的详细信息，请参阅位于 ARM 信息中心的 EABI 文档。

图 6-5 演示了典型的函数调用。在此示例中，参数被传递给函数，该函数使用局部变量并调用另一个函数。前四个参数会传递给寄存器 R0-R3。此示例还显示了为被调用函数分配本地帧和参数块。没有局部变量且不需要参数块的函数不分配本地帧。

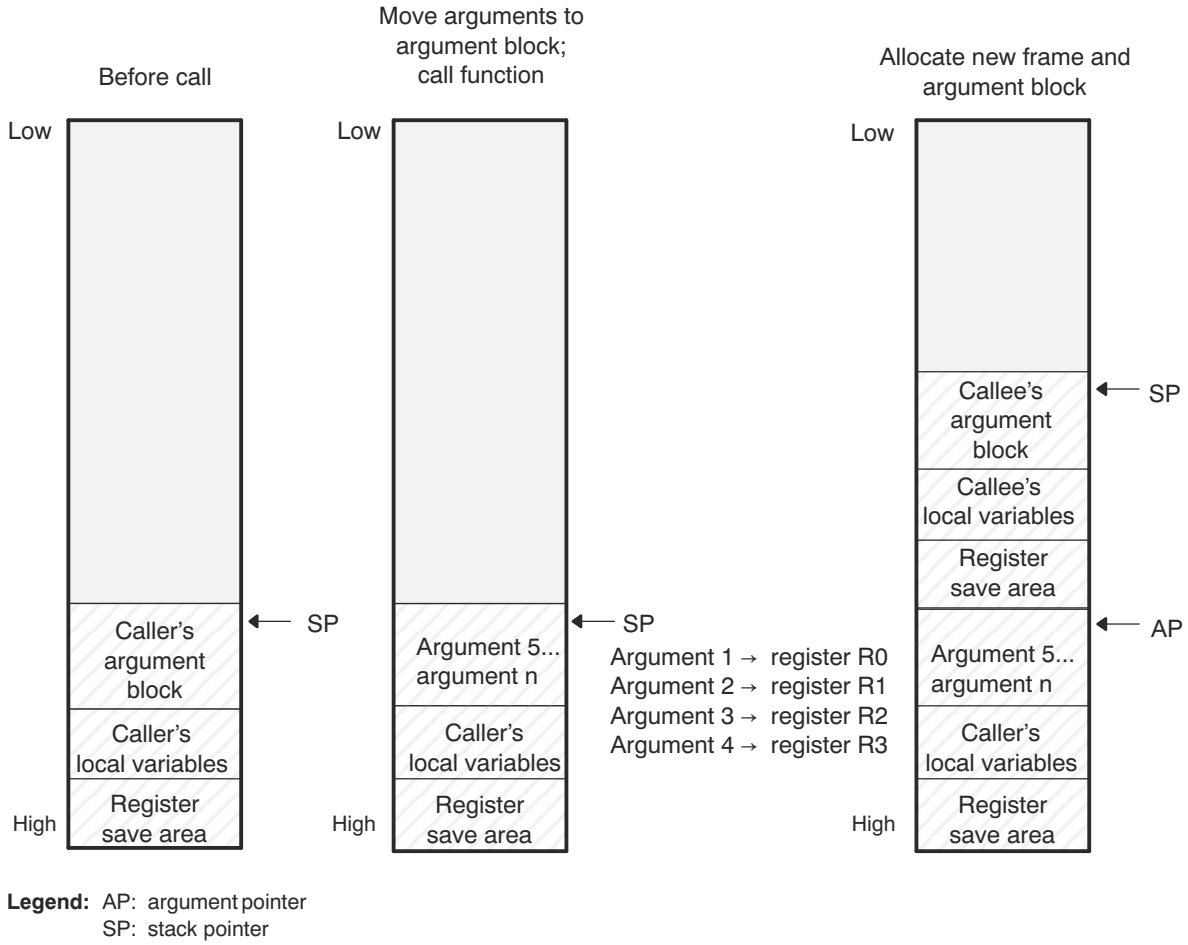


图 6-5. 在函数调用期间使用堆栈

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。

1. 调用函数（父级函数）负责保存在整个调用中活跃的调用保存寄存器。（调用保存寄存器是 R0-R3 和 R12（备用名称是 A1-A4 和 V9）。）
2. 如果被调用函数（子级函数）返回一个结构体，则调用方会为该结构体分配空间并将该空间的地址作为第一个参数传递给被调用函数。
3. 调用方将第一个参数按顺序放在寄存器 R0-R3 中。调用方以相反的顺序将剩余的参数移动到参数块，并将剩余的最左侧参数放在最低位地址。因此，剩余的最左侧参数被放置在栈顶。
4. 如果参数在步骤 3 中存储到参数块中，则调用方会在参数块中保留一个字用于双状态支持。（更多信息请参阅节 6.11。）

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. 如果函数是用省略号声明的，则可以使用可变数量的参数调用该函数。如果这些参数满足这两个条件，则被调用函数将这些参数推入堆栈中：
   - 该参数包含或跟随最后一个显式声明的参数。
   - 该参数在寄存器中进行传递。
2. 被调用函数将所有被该函数修改的寄存器的并在函数退出时必须保留的寄存器值推入堆栈中。通常，如果该函数包含调用，这些寄存器是入口保存寄存器（R4-R11 和 R14，备用名称为 V1 到 V8 和 LR）和链接寄存器 (R14)。如果该函数是中断函数，可能需要保留额外的寄存器。有关更多信息，请参阅 节 6.7。
3. 被调用函数通过从 SP 中减去一个常量来为局部变量和参数块分配内存。此常量通过以下公式计算：

   size of all local variables + max = constant

   max 参数用于指定放置在每次调用的参数块中的所有参数的大小。
4. 被调用函数会执行函数的代码。
5. 如果被调用函数返回一个值，则将该值放入 R0（或 R0 和 R1 值）中。
6. 如果被调用函数返回结构体，则将该结构体复制到第一个参数 R0 指向的内存块中。如果调用方不使用返回值，则将 R0 设置为 0。这会指示被调用函数不要复制返回结构体。

   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，调用方只需将 s 的地址作为第一个参数传递并调用 f。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。

   无论是在调用函数时（使调用方正确设置第一个参数）还是声明函数时（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. 被调用函数通过添加在步骤 3 中计算的常量来取消分配帧和参数块。
8. 被调用函数会恢复步骤 2 中保存的所有寄存器。
9. 被调用函数 (_f) 将返回地址加载到程序计数器 (PC)。

下述示例是被调用函数如何响应调用的典型示例：

```
                                ; called function entry point
STMFD   SP!, {V1, V2, V3, LR} ; save V1, V2, V3, and LR
SUB     SP, SP, #16           ; allocate frame
...                           ; body of the function
ADD     SP, SP, #16           ; deallocate frame
LDMFD   SP!, {V1, V2, V3, PC} ; restore V1, V2, V3, and store LR
                              ; in the PC, causing a return
```

6.4.3 C 异常处理程序调用惯例

如果 C 异常处理程序调用其他函数，必须满足以下条件：

- 处理程序必须设置自己的栈指针。
- 处理程序保存调用未保留的所有寄存器：R0-R3、R-12、LR（硬件为 FIQ 而保存的 R8-R12）
- 可重入异常处理程序必须保存 SPSR 和 LR。

6.4.4 访问参数和局部变量

函数通过栈指针（SP 或 R13）间接访问其本地非寄存器变量，并通过参数指针 (AP) 间接访问其栈参数。如果 SP 可以引用所有栈参数，则会引用这些参数，并且不会保留 AP。SP 始终指向栈顶（最新压入的值），AP 指向最左侧的栈参数（最接近栈顶的参数）。例如：

```
   LDR  A2 [SP, #4]  ; load local var from stack
   LDR  A1 [AP, #0]  ; load argument from stack
```

栈向较小的地址增长，因此使用 SP 或 AP 寄存器的正偏移量访问 C/C++ 函数栈上的本地数据和参数数据。

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 *ARM 汇编语言工具用户指南* 中关于“链接器符号”的部分。

6.6 将 C 和 C++ 与汇编语言相连

以下是在 C/C++ 代码中使用汇编语言的方法：

- 使用汇编代码的单独模块并将它们与编译的 C/C++ 模块链接（请参阅节 6.6.1）。
- 在 C/C++ 源代码中使用汇编语言变量和常量（请参阅节 6.6.3）。
- 使用直接嵌入 C/C++ 源代码中的内联汇编语言（请参阅节 6.6.5）。
- 修改编译器产生的汇编语言代码（请参阅节 6.6.6）。

6.6.1 使用汇编语言模块与 C/C++ 代码

只要遵循 节 6.4 中定义的调用惯例，以及 节 6.3 中定义的寄存器惯例，即可同时使用 C/C++ 与汇编语言函数。C/C++ 代码可访问变量并调用使用汇编语言定义的函数，汇编代码也可访问 C/C++ 变量并调用 C/C++ 函数。

结合使用汇编语言和 C 时，请遵循以下指南：

- 您必须保留由某函数修改的所有专用寄存器。专用寄存器包括：
  - 入口保存寄存器（R4-R11，又名 V1 至 V8 和 LR）
  - 栈指针（SP 或 R13）

  如果通常使用 SP，则无需显式保留。换而言之，汇编函数可自由使用栈，只要在函数返回前将压入栈的全部内容弹出即可（因此要保留 SP）。

  非专用寄存器可自由使用，无需首先保存。
- 中断例程必须保存所使用的*所有*寄存器。如需更多信息，请参阅 节 6.7。
- 如果通过汇编语言调用 C/C++ 函数，请加载指定的寄存器与参数，并将其余参数压入栈，如 节 6.4.1 中所述。

  请记住，函数可改变未保留的任何寄存器，而且不必恢复该寄存器。如果这些寄存器中有任何内容需要在不同调用间保留，则必须显式保存它们。
- 函数必须根据其 C/C++ 声明正确地返回值。双精度值会返回 R0 和 R1，结构也会返回，如 节 6.4.1 的第 2 步所述。任何其他值都会返回 R0。
- 任何汇编模块均不应出于任何目的使用 .cinit 段，除非进行全局变量的自动初始化。C/C++ 启动例程假设 .cinit 段包含*完整的* 初始化表。将其他信息放入 .cinit 会将表打乱，导致无法预测的结果。
- 编译器会为所有外部对象指定链接名称。因此您在编写汇编语言代码时，使用的链接名称必须与编译器指定的相同。相关详细信息，请参阅节 5.15。
- 在汇编语言中声明、并从 C/C++ 访问或调用的任何对象或函数，必须在汇编语言修饰符中使用 .def 或 .global 指令进行声明。这样可将符号声明为外部引用，并允许链接器解析对它的引用。

  同理，若要从汇编语言中访问 C/C++ 函数或对象，应在汇编语言模块中使用 .ref 或 .global 指令来声明 C/C++ 对象。这样可创建未声明的外部引用，并由链接器解析。

6.6.2 从 C/C++ 访问汇编语言函数

对于将从汇编语言中调用的已在 C++ 中定义的函数，应在 C++ 文件中定义为 extern "C"。对于将从 C++ 中调用的已在汇编语言中定义的函数，必须在 C++ 中原型设计为 extern "C"。

示例 6-1 列举了一个名为 main 的 C++ 函数，此函数调用一个名为 asmfunc 的汇编语言函数，如示例 6-2 中所示。asmfunc 函数采用其单个参数，将其添加到名为 gvar 的 C++ 全局变量，并返回结果。

示例 6-1. 从 C/C++ 程序调用汇编语言函数

```
extern "C" {
extern int asmfunc(int a); /* 申明外部 asm 函数 */
int gvar = 0;              /* 定义全局变量        */
}
void main()
{
   int I = 5;
   I = asmfunc(I);         /* 正常调用函数        */
```

示例 6-2. 由 示例 6-1 调用的汇编语言程序

```
          .global asmfunc
          .global gvar
asmfunc:
          LDR      r1, gvar_a
          LDR      r2, [r1, #0]
          ADD      r0, r0, r2
          STR      r0, [r1, #0]
          MOV      pc, lr
gvar_a    .field   gvar, 32
```

在示例 6-1 的 C++ 程序中，extern “C” 声明会告知编译器使用 C 命名规则 (即，不进行名称改编)。当链接器解析 .global _asmfunc 引用时，程序集文件中的相应定义将匹配。

参数 i 在 R0 中传递，并在 R0 中返回结果。R1 用于保存全局 gvar 的地址。R2 会保存 gvar 的值，然后再向其添加值 i。

6.6.3 从 C/C++ 访问汇编语言变量

有时，C/C++ 程序访问汇编语言中定义的变量或常量会很有用。根据项目定义的位置和方式，您可以使用以下方式完成此任务：在 .bss 段中定义的变量、未在 .bss 段中定义的变量，或者链接器符号。

6.6.3.1 访问汇编语言全局变量

从 .bss 段或以 .usect 命名的段访问变量非常简单：

1. 使用 .bss 或 .usect 指令来定义变量。
2. 使用 .def 或 .global 指令将定义设置为外部引用。
3. 在汇编语言中使用适当的链接名。
4. 在 C/C++ 语言中，将变量声明为 *extern* 并正常访问它。

示例 6-4 和示例 6-3 说明了如何访问 .bss 中定义的变量。

示例 6-3. 汇编语言变量程序

```
        .bss     var,4,4  ; Define the variable
        .global  var      ; Declare the variable as external
```

示例 6-4. C 程序从示例 6-3 中访问汇编语言

```
extern int var;        /* 外部变量 */
var = 1;               /* 使用该变量  */
```

6.6.3.2 访问汇编语言常量

您可以通过将 .set 指令与 .def 或 .global 指令结合使用，在汇编语言中定义全局常量，也可以使用链接器赋值语句在链接器命令文件中定义它们。这些常量只能通过使用特殊运算符从 C/C++ 中访问。

对于 C/C++ 或汇编语言中定义的**变量**，符号表包含变量所包含的*值的地址*。从 C/C++ 按名称访问程序集变量时，编译器使用符号表中的地址获取值。

但是，对于**汇编语言常量**，符号表包含常量的实际*值*。编译器无法分辨符号表中的哪些项是地址，哪些是值。如果按名称访问汇编语言（或链接器）常量，编译器将尝试使用符号表中的值作为地址来获取值。若要防止这种行为，必须使用 & (address of) 运算符来获取值 (_symval)。换而言之，如果 x 是汇编语言常量，那么它在 C/C++ 中的值是 &x。请参阅《*ARM 汇编语言工具用户指南*》中的“在 C/C++ 应用程序中使用链接器符号”，了解更多使用 _symval 的示例。

有关符号和符号表的更多信息，请参阅《*ARM 汇编语言工具用户指南*》中的“符号”部分。

您可以使用 cast 和 #define 来简化这些符号在程序中的使用，如示例 6-5 和示例 6-6 中所示。

示例 6-5. 从 C 语言访问汇编语言常量

```
extern int table_size;          /*外部引用 */
#define TABLE_SIZE ((int) (&table_size))
      .                         /* 通过强制转换来隐藏地址 */
      .
      .
for (I=0; i<TABLE_SIZE; ++I)   /* 像普通符号一样使用 */
```

示例 6-6. 示例 6-5 的汇编语言程序

```
_table_size  .set10000      ; define the constant
             .global _table_size  ; make it global
```

由于您只引用存储在符号表中的符号值，符号的声明类型并不重要。在示例 6-5 中，使用了 int。您能够以类似的方式引用链接器定义的符号。

6.6.4 与汇编源代码共享 C/C++ 头文件

您可以使用 .cdecls 汇编器指令在 C 和汇编代码之间共享包含声明和原型的 C 头文件。任何合法的 C/C++ 都可以在 .cdecls 块中使用，C/C++ 声明将导致自动生成合适的汇编语言，从而允许您在汇编语言代码中引用 C/C++ 构造。更多相关信息，请参阅《*ARM 汇编语言工具用户指南*》中的 C/C++ 头文件章节。

6.6.5 使用内联汇编语言

在 C/C++ 程序中，您可以使用 asm 语句，在编译器创建的汇编语言文件中插入一行汇编语言。一系列 asm 语句可在编译器输出中放置多行连续的汇编语言，之间没有代码。如需更多信息，请参阅 节 5.10。

asm 语句可用于在编译器输出中插入注释。只需在汇编代码字符串前添加分号 (;)，如下所示：

```
asm(";*** this is an assembly language comment");
```

备注

使用 asm 语句：在使用 asm 语句时应注意以下要点：

- 要极为小心，不要干扰 C/C++ 环境。编译器不会检查或分析插入的指令。
- 避免在 C/C++ 代码中插入跳跃指令或标签，因为它们会迷惑代码生成器使用的寄存器跟踪算法，导致无法预测的结果。
- 使用 asm 语句时不要改变 C/C++ 变量的值。原因是编译器不会验证此类语句。它们会原样插入汇编代码中，如果您不确定它们的效果，可能会造成问题。
- 不要使用 asm 语句插入可改变汇编环境的汇编器指令。
- 避免在 C 代码中创建汇编宏，并使用 --symdebug:dwarf (或 -g) 选项进行编译。C 环境的调试信息和汇编宏扩展不兼容。

6.6.6 修改编译器输出

您可以通过以下方法检查和更改编译器的汇编语言输出：编译源代码，然后在汇编之前编辑汇编输出文件。C/C++ interlist 功能可以帮助您检查编译器输出。请参阅节 2.12。

6.7 中断处理

只要您遵循本节中的准则，就可以中断并返回至 C/C++ 代码，而不会中断 C/C++ 环境。在 C/C++ 环境初始化完成后，启动例程不会启用或禁用中断。如果系统通过硬件复位进行初始化，则中断被禁用。如果您的系统使用中断，您必须处理任何所需的中断启用或屏蔽。此类操作对 C/C++ 环境没有影响，并且很容易与 asm 语句或调用汇编语言函数合并。

6.7.1 在中断期间保存寄存器

当 C/C++ 代码被中断时，中断例程必须保存例程或例程所调用任何函数使用的所有机器寄存器的内容。除了分组寄存器之外，寄存器保留必须由中断例程显式处理。

硬件会自动保留所有分组寄存器 (可重入的中断除外) 。如果编写可重入的中断例程，则必须添加代码来保留中断的分组寄存器。) 每种中断类型都有一组分组寄存器。有关中断类型的信息，请参阅节 5.11.16。

6.7.2 使用 C/C++ 中断例程

当 C/C++ 代码被中断时，中断例程必须保存例程或例程所调用任何函数使用的所有机器寄存器的内容。寄存器保存必须由中断例程明确处理。

```
__interrupt void example (void)
{
...
}
```

如果 C/C++ 中断例程未调用任何其他函数，则只会保存和恢复中断处理程序使用的寄存器。但如果 C/C++ 中断例程调用了其他函数，这些函数可修改中断处理程序未使用的未知寄存器。因此，如果调用了任何其他函数，例程会保存所有调用保存寄存器。（这不包括影子寄存器。）请勿直接调用中断处理函数。

可使用 INTERRUPT pragma 或 __interrupt 关键字，直接通过 C/C++ 函数处理中断。相关信息，请参见节 5.11.16 和节 5.7.2。

6.7.3 使用汇编语言中断例程

只要您遵循编译器的寄存器惯例，就可以利用汇编语言代码处理中断。就像所有汇编函数一样，中断例程可使用栈、访问全局 C/C++ 变量并正常调用 C/C++ 函数。调用 C/C++ 函数时，请确保在调用前保存所有调用保存寄存器，因为 C/C++ 函数可以修改所有这类寄存器。您无需保存入口保存寄存器，因为它们由调用的 C/C++ 函数保存。

6.7.4 如何将中断例程映射到中断向量

备注

本节不适用于 Cortex-M 器件。

若要将 Cortex-A 中断例程映射到中断向量，您需要添加一个 intvecs.asm 文件。该文件将包含汇编语言指令，可用于设置带有中断例程分支的 ARM 中断向量。按如下步骤使用该文件：

1. 以示例 6-7 为指导，创建 intvecs.asm 并添加您的中断例程。对于每个例程：
   a. 在文件的开头，添加一个命名例程的 .global 指令。
   b. 修改相应的 .word 指令以创建指向例程名称的分支。
2. 将 intvecs.asm 与您的应用代码和编译器的链接器控制文件（lnk16.cmd 或 lnk32.cmd）组装并链接在一起。控制文件包含一个 SECTIONS 指令，该指令将 .intvecs 段映射到存储器位置 0x00-0x1F。

例如，在 ARM v4 目标上，如果您已经为名为 c_intIRQ 的 IRQ 中断编写了一个 C 中断例程，并为名为 tim1_int 的 FIQ 中断编写了一个汇编语言例程，那么您应该创建 intvecs.asm，如示例 6-7 所示。

示例 6-7. intvecs.asm 文件示例

```
        .if __TI_EABI_ASSEMBLER
        .asg c_intIRQ, C_INTIRQ
        .else
        .asg _c_intIRQ, C_INTIRQ
        .endif
.global _c_int00
.global C_INTIRQ
.global tim1_int
.sect ".intvecs"
B _c_int00 ; reset interrupt
.word 0 ; undefined instruction interrupt
.word 0 ; software interrupt
.word 0 ; abort (prefetch) interrupt
.word 0 ; abort (data) interrupt
.word 0 ; reserved
B C_INTIRQ ; IRQ interrupt
B tim1_int ; FIQ interrupt
```

6.7.5 使用软件中断功能

软件中断 (SWI) 是由执行特定指令生成的同步例外。应用使用软件中断功能向受保护的系统（例如操作系统）请求提供服务，该系统只能在监控模式下执行服务。一些 ARM 文档使用“监控调用”(SVC) 这一术语，而不是“软件中断”。

C/C++ 应用可使用 SWI_ALIAS pragma 将一个软件中断号与一个函数名相关联，然后像调用函数一样来调用软件中断。如需更多信息，请参阅 节 5.11.29。

调用软件中断函数就代表着调用软件中断功能，因此向软件中断传递数据并从中返回数据就可指定为普通函数参数，传递具有以下限制：

传递到软件中断的所有参数必须驻留在四个参数寄存器中 (R0-R3)。不能通过软件栈来传递参数。因此只能传递四个参数，但以下情况除外：

- 传递 Floating-point doubles，在这种情况下每个 double 格式占用两个寄存器。
- 返回结构，在这种情况下返回的结构地址占用第一个参数寄存器。

对于 Cortex-M 架构，C SWI 处理程序无法返回值。其他架构中的 SWI 处理程序可能会返回值。

对于寄存器的使用，C/C++ 编译器将调用软件中断与调用函数视为等同。它假设所有入口保存寄存器均由软件中断保留，save-on-call 寄存器（其余寄存器）可由软件中断更改。

6.7.6 其他中断信息

中断例程可以执行由任何其他函数执行的任何任务，包括访问全局变量、分配局部变量和调用其他函数。

编写中断例程时，请谨记以下要点：

- 需要处理任何特殊的中断屏蔽。
- 不能直接从 C/C++ 代码调用 C/C++ 中断例程。
- 在系统复位中断（例如 c_int00）中，不能假设运行时环境已设置；因此，*不能分配局部变量，也不能在运行时栈上保存任何信息*。
- 在汇编语言中，请记住在 C/C++ 中断名称之前加上适当的链接名。例如，将 c_int00 表示为 _c_int00。
- 当中断发生时，处理器的状态（ARM 或 Thumb 模式）取决于使用的器件。编译器允许在 ARM 或 Thumb-2 模式下定义中断处理程序。应该确保中断处理程序让器件使用正确的模式。
- FIQ、监控器、中止、IRQ 和未定义模式具有独立的栈，C/C++ 运行时环境不会自动设置这些栈。如果这些模式中的任何一个有中断例程，则必须为该模式设置软件栈。但是，ARM Cortex-M 处理器有两个栈，IRQ（Cortex-M 的唯一中断类型）使用的主栈 (MSP) 由编译器自动处理。
- 中断例程不可重入。如果中断例程允许其类型的中断，则必须在执行此操作之前保存返回地址和 SPSR（已保存的程序状态寄存器）的副本。
- 软件中断是同步的，只要针对此目的设计了系统，节 6.7.1 中讨论的寄存器保存惯例就可以减少限制。然而，编译器生成的软件中断例程遵循 节 6.7.1 中的惯例。

6.8 固有运行时支持算术和转换例程

内在运行时支持库包含许多汇编语言例程，它们为 C/C++ 操作提供算术和转换功能，而 32 位和 16 位指令集不提供这些功能。这些例程包括整数除法、整数取模和浮点运算。

每个例程都有两个版本：

- 只能从 16-BIS (位指令集) 状态调用的 16 位版本
- 只能从 32-BIS 状态调用的 32 位版本

这些例程不遵循标准的 C/C++ 调用惯例，因为不支持命名和寄存器惯例。有关 EABI 命名惯例的信息，请参阅 ARM 信息中心。

6.8.1 CPSR 寄存器和中断内在函数

通过表 6-7 中的内在函数，您能够获取/设置 CPSR 寄存器并启用/禁用中断。除 _call_swi() 函数外，所有函数仅在 ARM 模式下编译时可用。节 5.14 中提供了用于 ARM 汇编指令的其他内在函数。

表 6-7. CPSR 和中断 C/C++ 编译器内在函数

C/C++ 编译器内在函数	汇编 指令	说明
void **_call_swi(**unsigned int *src* **);**	**SWI $** *src*	调用软件中断。*src* 必须是立即数。
unsigned int *dst* = **_disable_FIQ( ) ;**	Cortex-R4/A8: **MRS** *dst* **, FAULTMASK** **CPSID** *f*	禁用 FIQ 中断并返回以前的 FAULTMASK 或 CPSR 设置。
unsigned int *dst* = **_disable_interrupts( ) ;**	Cortex-M0: **MRS** *dst* **, PRIMASK** **CPSID** *i* Cortex-M3/M4/R4/A8: **MRS** *dst* **, FAULTMASK** **CPSID** *f*	禁用所有中断并返回以前的 PRIMASK 或 FAULTMASK 设置。汇编指令取决于架构。
unsigned int *dst* = **_disable_IRQ( ) ;**	**MRS** *dst* **, PRIMASK** **CPSID** *i*	禁用 IRQ 中断并返回以前的 PRIMASK 设置。
unsigned int *dst* = **_enable_FIQ( ) ;**	Cortex-R4/A8: **MRS** *dst* **, FAULTMASK** **CPSIE** *f*	启用 FIQ 中断并返回以前的 FAULTMASK 或 CPSR 设置。
unsigned int *dst* = **_enable_interrupts( ) ;**	Cortex-M0: **MRS** *dst* **, PRIMASK** **CPSIE** *i* Cortex-M3/M4/R4/A8: **MRS** *dst* **, FAULTMASK** **CPSIE** *f*	启用所有中断并返回以前的 PRIMASK 或 FAULTMASK 设置。汇编指令取决于架构。
unsigned int *dst* = **_enable_IRQ( ) ;**	**MRS** *dst* **, PRIMASK** **CPSIE** *i*	启用 IRQ 中断并返回以前的 PRIMASK 设置。
unsigned int *dst* = **_get_CPSR( ) ;**	**MRS** *dst* **, CPSR**	获取 CPSR 寄存器。
void **_restore_interupts(**unsigned int *src* **);**	Cortex-M0: **MSR PRIMASK** *src* Cortex-M3/M4: **MSR FAULTMASK** *src* Cortex-R4: **MSR CPSR_cf ,** *src*	将中断恢复到由 _disable_interrupts 返回的值指示的状态。汇编指令取决于架构。
void **_set_CPSR(**unsigned int *src* **);**	**MSR CPSR ,** *src*	设置 CPSR 寄存器。
void **_set_CPSR_flg(**unsigned int *src* **);**	**MSR** *dst* **, CPSR**	设置 CPSR 标志位。*src* 由内在函数旋转。
unsigned int *dst* = **_set_interrupt_priority(** unsigned int *src* **) ;**	仅 Cortex-M0/M3/M4 : **MRS** *dst* **, BASEPRI** **MSR BASEPRI,** *src*	设置中断优先级并返回以前的设置。

6.9 内置函数

内置函数由编译器预定义。可以像调用常规函数一样调用它们，但它们不需要原型或定义。编译器提供正确的原型和定义。

ARM 编译器支持以下内置函数：

- *__curpc* 函数，它返回调用它的程序计数器的值。函数的语法为：

```
void *__curpc(void);
```

- *__run_address_check* 函数，如果执行调用的代码位于由链接器分配的运行时地址处，则返回 TRUE。否则，返回 FALSE。函数的语法为：

```
int __run_address_check(void);
```

6.10 系统初始化

您必须先创建 C/C++ 运行时环境，才能运行 C/C++ 程序。C/C++ 启动例程使用被称为 c_int00 (or _c_int00) 的函数来执行此任务。运行时支持源码库 rts.src 在名为 boot.c（或 boot.asm）的模块中包含此例程的源码。

若要开始运行该系统，可以分支到或调用 c_int00 函数由复位硬件调用。您必须将 c_int00 函数与其他目标文件链接。当您使用 --rom_model or --ram_model 链接选项并将标准运行时支持库作为其中一个链接器输入文件时，此操作会自动发生。

链接 C/C++ 程序时，链接器会将可执行输出文件中的入口点值设置为符号 c_int00。

c_int00 函数会执行以下任务来对环境进行初始化：

1. 切换到相应的模式，为运行时堆栈保留空间，并设置堆栈指针 (SP) 的初始值。该堆栈在 64 位边界上对齐。
2. 调用函数 __TI_auto_init 来执行 C/C++ 自动初始化。

   __TI_auto_init 函数会执行以下任务：
   - 处理二进制符号复制表（若存在）。
   - 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 6.10.3。
   - 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 6.10.3.6。
3. 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。

6.10.1 用于系统预初始化的引导挂钩函数

引导挂钩是可将应用程序函数插入 C/C++ 引导进程的点。默认引导挂钩函数随运行时支持 (RTS) 库一同提供。但是，您可以实现这些引导挂钩函数的自定义版本，如果在运行时库之前链接了 RTS 库中的默认引导挂钩函数，则自定义版本将覆盖 RTS 库中的默认引导挂钩函数。在继续进行 C/C++ 环境设置之前，这些函数可以执行任何应用特定的初始化。

请注意，TI-RTOS 操作系统使用自定义版本的引导挂钩函数进行系统设置，因此，如果使用 TI-RTOS，则应小心覆盖这些函数。

以下引导挂钩函数可用：

__mpu_init(): 如果支持 MPU ，此函数提供用于初始化 MPU 的接口。在初始化栈指针之后，但在执行任何 C/C++ 环境设置之前，会调用 __mpu_init() 函数。此函数不应返回值。

_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。对于支持 MPU 的目标，此函数在 __mpu_init() 之后调用。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。

_c_int00( ) 初始化例程还提供了一种机制，让应用程序在初始化 C/C++ 环境之前执行设置操作（设置 I/O 寄存器、启用/禁用计时器等）。

6.10.2 运行时栈

运行时栈是在单个连续存储器块中分配的，并从高位地址向下增长到低位地址。 SP 指向栈顶。

代码不会检查运行时栈是否溢出。当栈增长超出为其分配的内存空间限值时，就会发生栈溢出。确保为栈分配足够的存储器空间。

通过在链接器命令行上使用 --stack_size 链接选项并在选项后直接将栈大小指定为常量，可以在链接时更改栈大小。

编译器附带的 C/C++ 引导例程设置用户/线程模式运行时栈。如果程序在其他运算模式下使用运行时栈，则还必须分配空间并设置与这些模式对应的运行时栈。

EABI 要求 64 位数据（long long 和 long double 类型）按 64 位对齐。这要求栈在函数入口处以 64 位边界对齐，以便局部 64 位变量以正确对齐方式分配到栈中。引导例程在 64 位边界对齐栈。

6.10.3 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。

6.10.3.1 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。

6.10.3.2 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 6.10.3.5。

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 6.10.3.3。

6.10.3.3 运行时变量自动初始化

在运行时自动初始化变量是自动初始化的默认方法。如需使用此方法，请使用 --rom_model 选项调用链接器。

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

图 6-6 演示了运行时的自动初始化。可在任何系统中使用此方法，其中，您的应用程序会运行刻录到 ROM 中的代码。

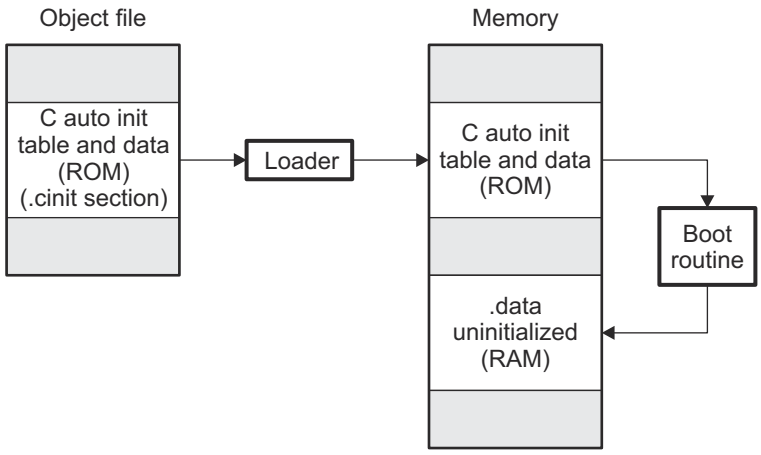
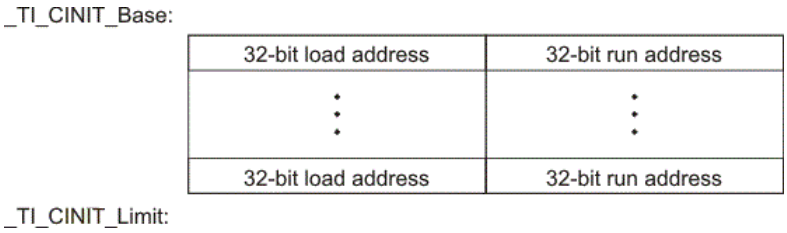


图 6-6. 运行时自动初始化

6.10.3.4 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：



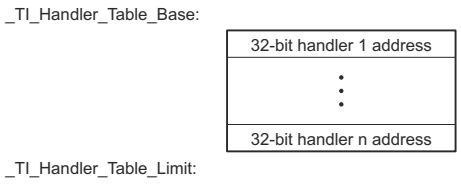
链接器定义的符号 __TI_CINIT_Base 和 __TI_CINIT_Limit 分别指向表的开头和结尾。此表中的每个条目对应一个需要初始化的输出段。可以使用不同的编码对每个输出段的初始化数据进行编码。

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。

6.10.3.4.1 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。

6.10.3.4.2 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。

6.10.3.4.3 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。

6.10.3.4.4 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。

6.10.3.4.5 用于处理 C 自动初始化表的 C 代码示例

运行时支持引导例程具有处理 C 自动初始化表的代码。以下 C 代码说明了如何在目标上处理自动初始化表。

```
typedef void (*handler_fptr)(const unsigned char *in,
unsigned char *out);
#define HANDLER_TABLE   __TI_Handler_Table_Base
#pragma WEAK(HANDLER_TABLE)
extern  unsigned int   HANDLER_TABLE;
extern  unsigned char *__TI_CINIT_Base;
extern  unsigned char *__TI_CINIT_Limit;
void auto_initialize()
{
   unsigned char **table_ptr;
   unsigned char **table_limit;
   /*-------------------------------------------------------------*/
   /* Check if Handler table has entries.                         */
   /*-------------------------------------------------------------*/
   if (&__TI_Handler_Table_Base >= &__TI_Handler_Table_Limit)
      return;
   /*-------------------------------------------------------------*/
   /* Get the Start and End of the CINIT Table.                   */
   /*-------------------------------------------------------------*/
   table_ptr   = (unsigned char **)&__TI_CINIT_Base;
   table_limit = (unsigned char **)&__TI_CINIT_Limit;
   while (table_ptr < table_limit)
   {
      /*----------------------------------------------------------*/
      /* 1.Get the Load and Run address.                          */
      /* 2.Read the 8-bit index from the load address.            */
      /* 3.Get the handler function pointer using the index from  */
      /*    handler table.                                        */
      /*----------------------------------------------------------*/
      unsigned char *load_addr   = *table_ptr++;
      unsigned char *run_addr    = *table_ptr++;
      unsigned char  handler_idx = *load_addr++;
      handler_fptr   handler     =
                  (handler_fptr)(&HANDLER_TABLE)[handler_idx];
      /*----------------------------------------------------------*/
      /* 4.Call the handler and pass the pointer to the load data */
      /*    after index and the run address.                      */
      /*----------------------------------------------------------*/
      (*handler)((const unsigned char *)load_addr, run_addr);
   }
}
```

6.10.3.5 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 6-7 演示了加载时变量的初始化。

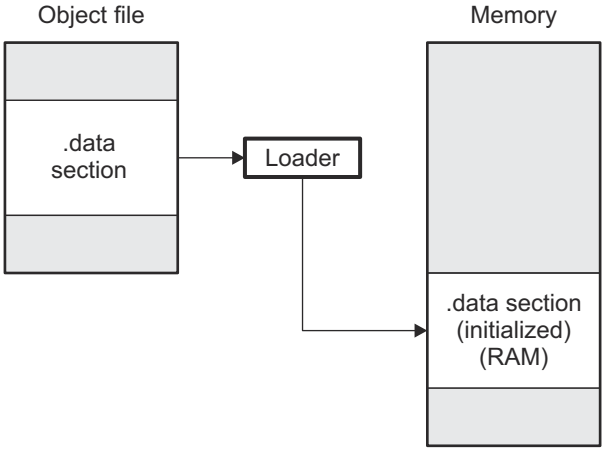


图 6-7. 加载时初始化

6.10.3.6 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。

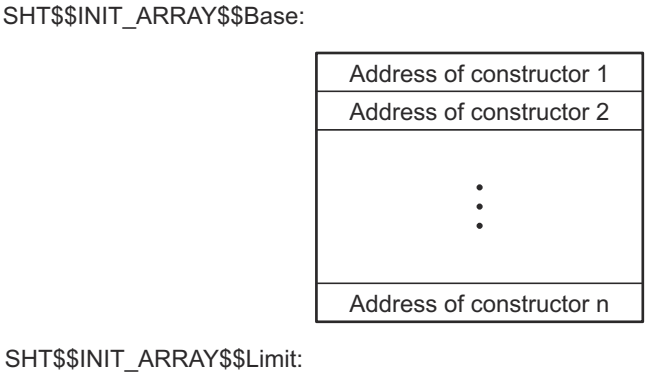


图 6-8. 构造函数表

6.10.4 初始化表

.cinit 段中的表由可变大小的初始化记录组成。每个必须自动初始化的变量在 .cinit 段中都有一条记录。图 6-9 显示了 .cinit 段的格式和初始化记录。

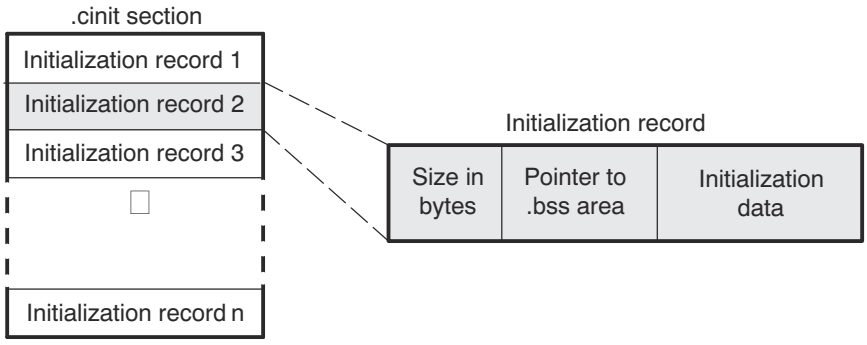


图 6-9. .cinit 段中初始化记录的格式

初始化记录的字段包含以下信息：

- 初始化记录的第一个字段包含初始化数据的大小（以字节为单位）。该字段的宽度为一个字（32 位）。
- 第二个字段包含 .bss 段中必须复制初始化数据的区域的起始地址。该字段的宽度为一个字。
- 第三个字段包含被复制到 .bss 段以初始化变量的数据。该字段的宽度为变量。

每个必须自动初始化的变量在 .cinit 段中都有一条初始化记录。

以下示例显示了在 C 中定义的初始化全局变量。

```
int   i = 23;
int   a[5] = { 1, 2, 3, 4, 5 };
```

相应的初始化表如下所示。.cinit:c 段包含所有标量数据，是 .cinit 段的子段。该子段在初始化期间作为一条记录处理，从而最大限度地减少了 .cinit 段的整体大小。

```
     .sect     ".cinit"          ; Initialization section
* Initialization record for variable i
     .align    4                 ; align on word boundary
     .field    4,32              ; length of data (1 word)
     .field    _i+0,32           ; address of i
     .field    23,32             ; _i @ 0
* Initialization record for variable a
     .sect     ".cinit"
     .align    4                 ; align on word boundary
     .field    IR1,32            ; Length of data (5 words)
     .field    _a+0,32           ; Address of 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
IR1: .set      20                ; set length symbol
```

.cinit 段必须只包含这种格式的初始化表。连接汇编语言模块时，请勿将 .cinit 段用于任何其他目的。

.pinit 段中的表仅包含要调用的构造函数的地址列表（如下图所示）。构造函数显示在表中的 .cinit 初始化后。

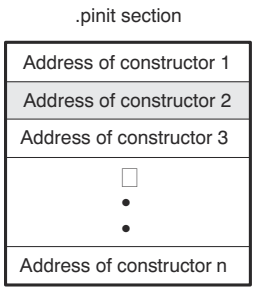


图 6-10. .pinit 段中初始化记录的格式

当您使用 --rom_model 或 --ram_model 选项时，链接器会组合来自所有 C/C++ 模块的 .cinit 段，并在 .cinit 复合段的末尾附加一个空字。此终止记录显示为字段大小为 0 的记录，并标记初始化表的末尾。

同样，--rom_model 或 --ram_model 链接选项会使链接器组合来自所有 C/C++ 模块的所有 .pinit 段，并在 .pinit 复合段的末尾附加一个空字。启动例程在遇到空构造函数地址时知道全局构造函数表的末尾。

符合 const 条件的变量的初始化方式有所不同；请参阅节 5.7.1。

6.11 TIABI 下的双状态交互工作（已弃用）

ARM 是一种独特的处理器，它提供 32 位架构的性能和 16 位架构的代码密度。它支持 16 位指令集和 32 位指令集，允许在这两个指令集之间动态切换。

ARM 处理器使用的指令集由处理器的状态决定。处理器可以在任何给定时间处于 32 BIS（位指令集）状态或 16 BIS 状态。编译器使您能够指定模块应在 32 BIS 还是 16 BIS 状态下编译，并允许在一种状态下编译的函数调用在另一种状态下编译的函数。

6.11.1 双状态支持级别

默认情况下，编译器允许函数之间的双状态交互工作。但是，编译器允许您更改支持级别以满足您的特定需求。

在双状态交互工作中，被调用函数负责处理调用函数所需的正确状态更改。调用函数负责处理间接调用函数（通过地址调用）所需的正确状态更改。因此，如果某个函数提供了需要状态改变的函数直接调用该函数（按名称调用）的能力，并提供间接调用涉及状态改变的函数的机制，那么该函数支持双状态交互工作。

如果某个函数不支持双状态交互工作，则不能被需要状态改变的函数调用，也不能间接调用支持双状态交互工作的函数。无论某个函数是否支持双状态交互工作，它都可以直接或间接调用某些函数：

- 直接调用相同状态的函数
- 直接调用处于不同状态的函数（如果该函数支持双状态交互工作）
- 间接调用处于相同状态的函数（如果该函数不支持双状态交互工作）

鉴于支持双状态的这种定义，ARM C/C++ 编译器提供了三个级别的支持。使用表 6-8 来确定用于代码的最佳支持级别。

表 6-8. 选择双状态支持级别

如果代码...	使用这种支持级别...
需要少量状态更改	默认值
需要很多状态更改	经优化的
不需要状态更改并具有频繁间接调用	无

以下是有关每个支持级别的详细信息：

- **默认值**。支持完全双状态交互工作。对于每个支持完全双状态交互工作的函数，编译器都会生成代码，允许需要状态更改的函数调用该函数，无论该函数是否被使用过。该代码所在的段与实际函数所在的段不同。如果链接器确定从未引用该代码，则不会将其链接到最终的可执行映像中。但是，用于支持双状态交互工作的间接调用的机制已集成到函数中，并且无法被链接器删除，即使链接器确定不需要该机制也是如此。
- **经优化的**。经优化的双状态交互工作在默认级别上不提供额外的功能，但针对需要状态更改的情况优化了双态支持代码（在代码大小和执行速度方面）。它通过将支持功能集成到函数中来进行此优化。仅当您知道对该函数的大部分调用都需要状态更改时，才使用优化级别的支持。即使从未使用过双状态支持代码，链接器也无法删除该代码，因为它已集成到函数中。若要指定此级别的支持，请使用 DUAL_STATE pragma。请参阅节 5.11.9，了解详情。

- **无**。双状态交互工作被禁用。此级别使用 -md shell 选项来调用。包含此支持功能的函数可以直接调用以下函数：
  - 以相同状态编译的函数
  - 支持双状态交互工作且处于不同状态的函数

  具有此支持级别的函数只能间接调用不需要状态更改且不支持双状态交互工作的函数。具有此支持级别的函数不提供双状态交互工作，因此需要状态更改的函数无法调用它们。

  如果您不需要双状态交互工作，间接调用频繁，并且不能容忍支持双状态交互工作的间接调用所产生的额外代码大小或速度，请使用此支持级别。

当程序不需要任何状态更改时，指定不支持和默认支持之间的唯一区别是默认支持级别中的间接调用更复杂。

6.11.2 实现

通过为函数提供备用入口点来实现双状态支持。此备用入口点供需要状态更改的函数使用。双状态支持可处理对正确状态的更改，并在返回函数时将函数更改回调用方的状态。此外，间接调用会设置返回地址，以便在被调用函数返回后，状态可以可靠地改回调用方的状态。

6.11.2.1 入口点的命名规则

ARM 编译器会保存所有以下划线 (_) 或美元符号 ($) 开头的标识符的命名空间。在这个双状态支持方案中，所有 32-BIS 状态入口点都以下划线开头，所有 16-BIS 状态入口点都以美元符号开头。编译器生成的所有其他标识符（与处理器状态无关）都以下划线开头。按照此命名规则，16 位函数内的所有直接调用都引用以美元符号开头的入口点，而 32 位函数内的所有直接调用都引用以下划线开头的入口点。

6.11.2.2 间接调用

在 16-BIS 状态下获取的函数地址使用相应函数的 16-BIS 状态入口点的地址（设置第 0 位地址）。同样，在 32-BIS 状态下获取的函数地址使用相应函数的 32-BIS 状态入口点的地址（清除第 0 位地址）。然后，所有间接调用都是通过将被调用函数的地址加载到寄存器中并执行分支和交换（BX）指令来完成的。这会自动更改状态并确保代码正常运行，无论地址在获取时处于何种状态。

还必须设置返回地址，以便处理器的状态在返回时保持一致且已知。测试第 0 位地址以确定 BX 指令是否调用状态更改。如果它不调用状态更改，则为函数的状态设置返回地址。如果它调用状态更改，则为备用状态设置返回地址并执行代码以返回到函数的状态。

函数的入口点取决于获取地址的函数的状态，因此在与函数处于相同状态时获取该函数的地址更有效。这可确保使用实际函数的地址，而不是其备用入口点。间接调用本身可以调用状态更改，因此即使从不同的状态调用它，也不需要通过其备用入口点进入函数。

示例 6-8 显示了调用 max() 的 sum()，具有针对 16-BIS 状态编译的代码并支持双状态交互工作。sum( ) 函数使用 -code_state=16 选项进行了编译，该选项为 UAL 之前的汇编代码创建 16 位指令。（有关 UAL 语法的信息，请参阅《*ARM 汇编语言工具用户指南*》。）示例 6-11 显示了相同的函数调用，具有针对 32-BIS 状态编译的代码并支持双状态交互工作。函数 max( ) 在编译时未使用 -code_state=16 选项，并创建 32 位指令。

示例 6-8. 针对 16-BIS 状态编译的 C 代码：sum( )

```
int total = 0;
sum(int val1, int val2)
{
   int val = max(val1, val2);
   total += val;
}
```

示例 6-9. 示例 6-8 的 16 位汇编语言程序

```
;***************************************************************
;* FUNCTION VENEER: _sum                                       *
;***************************************************************
_sum:
      .state32
      STMFD  sp!, {lr}
      ADD    lr, pc, #1
      BX     lr
      .state16
      BL     $sum
      BX     pc
      NOP
      .state32
      LDMFD  sp!, {pc}
      .state16
      .sect   ".text"
      .global sum
;***************************************************************
;* FUNCTION DEF: $sum                                          *
;***************************************************************
$sum:
      PUSH   {LR}
      BL     $max
      LDR    A2, CON1 ; {_total+0}
      LDR    A3, [A2, #0]
      ADD    A1, A1, A3
      STR    A1, [A2, #0]
      POP    {PC}
;***************************************************************
;* CONSTANT TABLE                                              *
;***************************************************************
      .sect  ".text"
      .align 4
CON1: .field _total, 32
```

示例 6-10. 针对 32-BIS 状态编译的 C 代码：sum( )

```
int max(int a, int b)
{
   return a < b ? b : a;
}
```

示例 6-11. 示例 6-10 的 32 位汇编语言程序

```
;*************************************************************
;* FUNCTION VENEER: $max                                     *
;*************************************************************
$max:
      .state16
      BX     pc
      NOP
      .state32
      B      _max
      .text
      .global _max
;*************************************************************
;* FUNCTION DEF: _max                                        *
;*************************************************************
_max:
      CMP    A1, A2
      MOVLE  A1, A2
      BX     LR
```

sum() 是一个 16 位函数，因此它的入口点是 $sum。它是为双状态交互工作而编译的，因此包含了位于不同段的备用入口点 _sum。所有需要更改状态的 sum() 调用都使用 _sum 入口点。

sum() 中对 max() 的调用引用了 $max，因为 sum() 是一个 16 位函数。如果 max( ) 是一个 16 位函数，sum( ) 将调用 max( ) 的实际入口点。但是，max( ) 是一个 32 位函数，因此 $max 是 max( ) 的替代入口点并处理 sum( ) 所需的状态更改。

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，可处理例外情况、信号和区域问题（取决于当地语言、民族和文化的属性）的库功能除外。使用 ANSI/ISO 标准库可确保提供一组一致的函数，可移植性更高。

除了 ANSI/ISO 指定函数，运行时支持库还包括其他例程，提供处理器特定命令和 C 语言 I/O 直接请求。这些内容在节 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 170
7.2 C I/O 函数 174
7.3 处理可重入性（_register_lock() 和 _register_unlock() 函数） 186
7.4 库构建流程 187

7.1 C 和 C++ 运行时支持库

ARM 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。为每种模式、大端字节序和小端字节序、每个 ABI（编译器版本 4.1.0 及更高版本）、各种架构以及 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 7.1.9。

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 4.3.1。

链接库后，链接器仅包含解析未定义的引用所需的那些库成员。有关链接的更多信息，请参阅《*ARM 汇编语言工具用户指南*》。

C、C++ 以及混合的 C 和 C++ 程序可以使用相同的运行时支持库。可以从 C 和 C++ 调用和引用的运行时支持函数和变量将具有相同的链接。

如果您想将使用 TI CodeGen 工具创建的目标文件与其他编译器工具链生成的目标文件链接起来，根据 ARM 标准的要求，您应先定义 _AEABI_PORTABILITY_LEVEL 预处理器符号（如下所示），然后再包含任何标准头文件，如 <stdlib.h>。

```
#define _AEABI_PORTABILITY_LEVEL 1
```

此定义可实现完全可移植性。将符号定义为 0 指定将使用“C 标准”可移植性级别。

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 TI_ARM_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：

- 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。
- file.h -- 声明由 RTS 库中的低级别 I/O 函数使用的函数。
- _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()。

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\arm_#.#.#\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 rtsv4_A_be_eabi.lib 库，或者构建新库。有关库命名的详细信息，请参阅节 7.1.9；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 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 5.6，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);`
- `time_t time(time_t *timer);`

time() 函数会返回挂钟时间。clock() 函数会返回自程序开始执行以来经过的时钟周期数；它与挂钟时间完全无关。

这些函数的默认实现要求程序在 CCS 或支持 CIO 系统调用协议的类似工具下运行。如果 CIO 不可用，而您需要使用这些函数中的其中一个，则您必须提供针对相应函数的自有定义。

clock() 函数会返回自程序开始执行以来经过的时钟周期数。这类信息可能存在于器件内的寄存器中，但位置会因平台而异。编译器的 RTS 库提供了采用 CIO 系统调用协议来与 CCS 进行通信的实现方案，这将确定如何为此器件计算正确的值。

如果 CCS 不可用，您必须提供一种有关 clock() 函数的实现方案来从器件中的相应位置收集时钟周期信息。

time() 函数会返回从 epoch 到现在的真实时间（以秒为单位）。

很多嵌入式系统中没有内部现实时钟，因此程序需要通过外部来源发现时间。编译器的 RTS 库提供了一种实现方案，利用 CIO 系统调用协议来与 CCS 进行通信，从而提供真实时间。

如果 CCS 不可用，您必须提供一种有关 time() 函数的实现方案来从一些其他来源查找时间。如果程序在操作系统中运行，该操作系统应该提供一种有关 time() 的实现方案。

time() 函数会返回从 *epoch* 到现在的秒数。在 POSIX 系统中，epoch 定义为自 1970 年 1 月 1 日 UTC 午夜零点到现在的秒数。不过，C 标准不需要任何特定的 epoch，并且 time() 的默认 TI 版本使用不同的 epoch：1900 年 1 月 1 日 UTC-6 (CST) 午夜零点。例外，默认的 TI time_t 类型为 32 位类型，而 POSIX 系统通常使用 64 位 time_t 类型。

RTS 库提供了一种有关 time() 函数的非默认实现方案，在该实现中，epoch 为 1970 年 1 月 1 日 UTC 午夜零点，time_t 类型为 64 位，也就是 __type64_t 的 typedef。

如果您的代码采用原始时间值，则您可以通过以下方式之一来处理 epoch 问题：

- 使用 epoch 为 1900 且 time_t 类型为 32 位的默认 time() 函数。本例中提供了一个单独的 __time64_t 类型。
- 定义宏命令 __TI_TIME_USES_64。time() 函数将使用 1970 epoch 和 64 位 time_t 类型，其中 time_t 为 __type64_t 的 typedef。

表 7-1. __time32_t 和 __time64_t 之间的区别

	__time32_t	__time64_t
Epoch (start)	1900 年 1 月 1 日 CST-0600	1970 年 1 月 1 日 UTC-0000
结束日期	2036 年 2 月 7 日 06:28:14	292277026596 年
符号	无符号，因此不能表示 epoch 之前的日期。	带符号，因此可以表示 epoch 之前的日期。

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 源码树中的非标准头文件

编译器安装 lib/src 子目录中的源代码包含以下用于构建库的非 ANSI 头文件：

- *values.h* 文件包含重新编译三角和超越数学函数所必需的定义。如有必要，您可以在 values.h 中对函数进行定制。
- *file.h* 文件包含用于低级 I/O 函数的宏命令和定义。
- *format.h* 文件包含 printf 和 scanf 中使用的结构和宏命令。
- *470cio.h* 文件包含特定于目标的低级 C I/O 宏定义。如有必要，您可以对 470cio.h 进行定制。
- *rtti.h* 文件包含执行运行时类型识别所必需的内部函数原型。
- *vtbl.h* 文件包含类的虚拟函数表格格式的定义。

7.1.9 库命名规则

默认情况下，链接器使用自动库选择功能来为您的应用程序选择正确的运行时支持库（请参阅节 4.3.1.1）。如果您手动选择库，则必须使用类似如下的命名方案来选择匹配的库：

rtsArchVersion_mode_endian[_n][_v*n*]_*abi*[_eh].lib

ArchVersion	该库用于构建的 ARM 架构版本。这可以是以下情况之一：v4、v5、v6、v6M0、v7A8、v7R4、v7R5 或 v7M3。	
mode	指示编译模式：	
	T	Thumb 模式
	A	ARM 模式
字节序	指示字节序：	
	le	小端字节序库
	be	大端字节序库
n	指示该库包含对 NEON 的支持。	
v*n*	指示该库支持 VFP 。*n* 指定版本。当前值为：	
	2	VFPv2
	3	VFPv3
	3D16	VFPv3D16
abi	指示使用的应用程序二进制接口 (ABI)。尽管不再支持 TI_ARM9_ABI 和 TIARM ABI，但库文件名仍然包含“_eabi”，以便与较旧库中的 EABI 库区分开来。	
	eh	指示该库支持异常处理

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。

超长型数据类型的格式规则要求格式字符串中使用 ll（小写 LL）。例如：

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

备注

默认主机所需的调试器：若要让默认主机器件正常工作，必须使用调试器来处理 C I/O 请求；默认的主机器件无法在嵌入式系统中自行工作。若要在嵌入式系统中工作，您需要为系统提供适当的驱动程序。

备注

C I/O 函数莫名失败：如果堆上没有足够的空间用于 C I/O 缓冲区，文件上的操作将会以静默方式失败。如果 printf() 调用莫名失败，那么原因可能就是这个。堆必须足够大，至少应足以分配执行 I/O 的每个文件所需的块大小 BUFSIZ（在 stdio.h 中定义），包括 stdout、stdin 和 stderr，以及用户代码执行的分配和分配记账开销。也可以声明一个大小字符数组 BUFSIZ，并将其传递给 setvbuf 来避免动态分配。要设置堆大小，请在链接时使用 --heap_size 选项（请参阅 *ARM 汇编语言工具用户指南* 中的*链接器说明*一章。

备注

Open 函数莫名失败：运行时支持会将打开的文件总数限制为相对于通用处理器的较小数字。如果您尝试打开的文件数量超过最大值，您可能会发现 open 函数将会莫名失败。您可以通过从 rts.src 提取源代码并编辑控制一些 C I/O 数据结构大小的常量，增加可打开文件的数量。宏命令 _NFILE 能控制一次可打开的 FILE (fopen) 对象数量（stdin、stdout 和 stderr 均计入此总数）。（另请参阅 FOPEN_MAX。）宏命令 _NSTREAM 能控制一次可打开的低级别文件描述符数量（stdin、stdout 和 stderr 下的低级别文件计入此总数）。宏命令 _NDEVICE 能控制一次可安装的器件驱动程序数量（主机器件计入此总数）。

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。

便携式应用只应使用高级别 I/O 函数。

若要使用高级别 I/O 函数，请进行以下操作：

- 为引用函数的每个模块加上头文件 stdio.h。
- 为程序中使用的每个 I/O 流实现 320 个字节的堆空间。流是与终端或键盘灯外设关联的数据源或数据目的地。流使用从堆中获取的动态分配内存进行缓冲。可能需要更多堆空间来支持需要额外动态分配内存的程序（调用 malloc()）。若要设置堆大小，请在链接时使用--heap_size 选项；请参阅表 2-22。

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：

```
armcl main.c --run_linker --heap_size=400 --library=rtsv4_A_be_eabi.lib --output_file=main.out
```

执行 main.out 会得到

```
Hello, world
```

输出到文件以及

```
Hello again, world
```

输出到主机的 stdout 窗口。

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 ARM 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：

- 将器件 *mydevice* 添加至器件表

add_device (continued)

向器件表添加器件

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

示例 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，例如从内部中断调用运行时支持函数；这仍然是您的责任。

运行时支持环境提供了钩子程序来安装临界区基元。默认情况下，假定使用单线程模型，并且不采用临界区基元。在 SYS/BIOS 等多线程系统中，内核会安排在这些钩子程序中安装信号量锁基元函数，然后在运行时支持输入需要由临界区加以保护的代码时调用这些函数。

在整个运行时支持环境中，当因访问全局状态而需要由临界区加以保护时，会调用函数 _lock()。此操作会调用提供的基元 (若已安装) 并获取信号量，然后再继续。在临界区完成后，会调用 _unlock() 来释放信号量。

通常，SYS/BIOS 负责创建和安装基元，因此您无需采取任何操作。不过，这种机制可以在不使用 SYS/BIOS 锁定机制的多线程应用程序中使用。

您不应直接定义 _lock() 和 _unlock() 函数；相反，通过调用安装函数来指示运行时支持环境使用以下基元：

```
void _register_lock  (void (  *lock)());
void _register_unlock(void (*unlock)());
```

_register_lock() 和 _register_unlock() 的参数应为无参数且不返回任何值的函数，此类函数会实现某种全局信号量锁定：

```
    extern volatile sig_atomic_t *sema = SHARED_SEMAPHORE_LOCATION;
    static int sema_depth = 0;
    static void my_lock(void)
    {
        while (ATOMIC_TEST_AND_SET(sema, MY_UNIQUE_ID) != MY_UNIQUE_ID);
        sema_depth++;
    }
    static void my_unlock(void)
    {
        if (!--sema_depth) ATOMIC_CLEAR(sema);
    }
```

运行时支持会对 _lock() 的调用进行嵌套，因此基元必须跟踪嵌套级别。

7.4 库构建流程

使用 C/C++ 编译器时，您可使用 大量彼此不一定兼容的不同配置和选项来编译代码。由于囊括所有可能的运行时支持库变体并不实际，各个编译器版本只会预构建少量最常用的库， 。

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 TI_ARM_C_DIR 环境变量查找运行时支持库。通常， TI_ARM_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 TI_ARM_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=rtsv4_A_be_eabi），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/rtsv4_A_be_eabi），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 *ARM 汇编语言工具用户指南* 中的归档器一章。

现在链接器已经决定了要使用的库，接下来它会检查 TI_ARM_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 TI_ARM_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=rtsv4_A_be_eabi.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 rtsv4_A_be_eabi 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=rtsv4_A_be_eabi.lib --name=rtsv4_A_be_eabi_debug.lib
      --install_to=$Project/Debug --extra_options="-g"
```

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 7-2 中所述。

```
mklib --help
```

表 7-2. mklib 程序选项

选项	效果
--index= *filename*	此版本的索引库 (libc.a)。用于查找定制构建的模板库，以及查找源文件（位于编译器安装程序的 lib/src 子目录中）。必备选项。
--pattern= *filename*	用于构建库的模式。如果既未指定 --extra_options，也未指定 --options，那么该库将为具有对应标准选项的标准库。如果指定了 --extra_options 或 --options，那么该库为具有自定义选项的自定义库。除非使用了 --all，否则为必备选项。
--all	一次性构建所有标准库。
--install_to= *directory*	要将库写入的目录。对于标准库，这个默认为与索引库 (libc.a) 相同的目录。对于自定义库，这个选项为必备选项。
--compiler_bin_dir= *directory*	编译器可执行文件所在的目录。直接调用 mklib 时，可执行文件应位于路径中，但如果不在那里，则必须使用这个选项来告知 mklib 这些文件的位置。这个选项主要是在链接器调用 mklib 时使用。
--name= *filename*	库的文件名且没有目录部分。仅用于自定义库。
--options=' *str* **'**	构建库时使用的选项。默认选项（见下文）会由此字符串所*取代*。如果使用此选项，则库将为自定义库。
--extra_options=' *str* **'**	构建库时使用的选项。也会使用默认选项（见下文）。如果使用此选项，则库将为自定义库。
--list_libraries	列出此脚本能够构建的库并退出。普通系统特有目录。
--log= *filename*	将构建日志另存为 *filename*。
--tmpdir= *directory*	使用 *directory* 作为暂存空间，而不是普通系统特有目录。
--gmake= *filename*	要调用的兼容 Gmake 的程序，而不是“gmake”
--parallel= *N*	一次性编译 *N* 个文件 ("gmake -j N")。
--query= *filename*	此脚本是否知道如何构建 FILENAME？
--help 或 **--h**	显示此帮助。
--quiet 或 **--q**	以静默方式运行。
--verbose 或 **--v**	用于调试此可执行文件的额外信息。

示例：

构建所有标准库并将它们放入编译器的库目录：

```
mklib --all --index=$C_DIR/lib
```

构建一个标准库并将其放入编译器的库目录：

```
mklib --pattern=rtsv4_A_be_eabi.lib --index=$C_DIR/lib
```

构建类似 rtsv4_A_be_eabi.lib 的自定义库，但启用符号调试支持：

```
mklib --pattern=rts16.lib --extra_options="-g" --index=$C_DIR/lib --install_to=$Project/Debug
     --name=rtsv4_A_be_eabi_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 TI_ARM_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 7-2 中列出的所有选项而不出现错误，即使这些选项不发挥任何作用也是如此。

章节 8

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。

8.1 调用 C++ 名称还原器 192
8.2 C++ 名称还原器的示例用法 193

8.1 调用 C++ 名称还原器

调用 C++ 名称还原器的语法如下：

armdem [*options*] [*filenames*]

armdem	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 armdem 使用标准输入。

默认情况下，C++ 名称还原器输出到标准输出。如果要输出到文件，可以使用 -o 文件选项。

以下选项仅适用于 C++ 名称还原器：

--debug (--d)	打印调试消息。
--diag_wrap[=on,off]	将诊断消息设置为在 79 列换行（on，这是默认值）或不换行 (off)。
--help (-h)	打印帮助屏幕，该帮助屏幕提供 C++ 名称还原器选项的在线汇总。
--output= *file* (-o)	输出到指定的文件而不是标准输出。
--quiet (-q)	减少执行期间生成的消息数量。
-u	指定外部名称没有 C++ 前缀。（已弃用）

8.2 C++ 名称还原器的示例用法

本节中的示例说明名称还原过程。

该示例显示了示例 C++ 程序。在该示例中，所有函数的链接名称都已改编；也就是说，函数的签名信息已编码到函数的名称中。

```
class banana {
public:
    int calories(void);
    banana();
    ~banana();
};
int calories_in_a_banana(void)
{
    banana x;
    return x.calories();
}
```

编译器输出的汇编代码结果如下。

```
_Z20calories_in_a_bananav:
        STMFD     SP!, {A3, A4, V1, LR}
        MOV       A1, SP
        BL        _ZN6bananaC1Ev
        BL        _ZN6banana8caloriesEv
        MOV       V1, A1
        MOV       A1, SP
        BL        _ZN6bananaD1Ev
        MOV       A1, V1
        LDMFD     SP!, {A3, A4, V1, LR}
        BX        LR
```

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：

```
armdem 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():
        STMFD     SP!, {A3, A4, V1, LR}
        MOV       A1, SP
        BL        banana::banana()
        BL        banana::calories()
        MOV       V1, A1
        MOV       A1, SP
        BL        banana::~banana()
        MOV       A1, V1
        LDMFD     SP!, {A3, A4, V1, LR}
        BX        LR
```

This page intentionally left blank.

A.1 术语

绝对列表器 一种调试工具，允许创建包含绝对地址的汇编器列表。

别名消歧 一种决定两个指针表达式何时不能指向同一位置的技术，从而允许编译器自由地优化此类表达式。

别名使用 以多种方式访问单个对象的能力，例如当两个指针指向单个对象时。其会破坏优化，这是因为任何间接引用都可能引用任何其它对象。

分配 链接器计算输出段最终存储器地址的过程。

ANSI 美国国家标准协会；一个建立行业自愿遵循的标准的组织。

应用程序二进制接口 (ABI) 一项指定两个目标模块之间接口的标准。ABI 规定了如何调用函数以及如何将信息从一个程序组件传递到另一个程序组件。

存档库 由归档器将单独文件组合成单个文件的集合。

归档器 将多个单独文件集合成一个单个文件（称为存档库）的软件程序。借助归档器，可以添加、删除、提取或替换存档库的成员。

汇编器 根据包含汇编语言指令、指示和宏定义的源文件创建机器语言程序的软件程序。汇编器将绝对操作码替换为符号操作码，并将绝对地址或可重定位地址替换为符号地址。

赋值语句 用值来初始化变量的语句。

自动初始化 在程序开始执行之前，初始化全局 C 变量（包含在 .cinit 段中）的过程。

运行时的自动初始化 链接器在链接 C 代码时使用的自动初始化方法。在使用 --rom_model 链接选项调用链接器时，链接器会使用此方法。链接器将数据表的 .cinit 段加载到内存中，并在运行时初始化变量。

大端 一种寻址协议，字中的字节从左至右进行编号。字中较高的有效字节存放在低地址处。字节序视硬件而定，并在复位时确定。另请参阅 *小端*

BIS 位指令集。

块 一组在大括号内组合在一起并被视为实体的语句。

.bss 段[.bss section] 默认的目标文件段之一。使用汇编器 .bss 指令在存储器映射中保留指定量的空间，以便稍后用于存储数据。.bss 段未被初始化。

字节	根据 ANSI/ISO C，可容纳一个字符的最小可寻址单元。
C/C++ 编译器	一种将 C 源语句转换成汇编语言源语句的软件程序。
代码生成器	一种编译器工具，采用解析器和优化器生成的文件并生成汇编语言源文件。
COFF	通用目标文件格式；根据 AT&T 开发的标准配置的目标文件系统。不再支持该 ABI。
命令文件	包含链接器或十六进制转换实用程序的选项、文件名、指令或命令的文件。
注释	用于记录或提高源文件可读性的源语句（或源语句的一部分）。不对注释进行编译、汇编或链接；不会影响对象文件。
编译器程序	一种实用工具，可以一步完成编辑、汇编和选择性链接操作。通过编译器（包括解析器、优化器和代码生成器）、汇编器和链接器，编译器可以运行一个或多个源代码模块。
配置内存	链接器指定用于分配的存储器。
常量	其值不能改变的类型。
交叉引用列表	由汇编器创建的输出文件，其中列出了定义的符号、定义符号的行、引用符号的行以及符号的最终值。
.data 段[.data section]	默认的目标文件段之一。.data 段是包含初始化数据的初始化段。可以使用 .data 指令将代码汇编到 .data 段中。
直接调用	一种函数调用，其中一个函数使用函数名称调用另一函数。
指令	用于控制软件工具操作和功能的专用命令（与用于控制器件操作的汇编语言指令相反）。
消歧	请参阅*别名消歧*
动态内存分配	几个函数（如 malloc，calloc 和 realloc）在运行时为变量动态分配内存所使用的技术。这是通过定义较大的内存池（堆）并使用函数分配堆中的内存来实现。
ELF	可执行和可链接格式；根据系统 V 应用程序二进制接口规范配置的目标文件系统。
仿真器	复制 ARM 运行的硬件开发系统。
入口点	目标存储器中的执行起点。
环境变量	由用户定义并分配给字符串的系统符号。环境变量通常包含在 Windows 批处理文件或 UNIX shell 脚本（例如 .cshrc 或 .profile）中。
收尾程序	函数中恢复堆栈并返回的代码部分。
可执行目标文件	在目标系统上下载并执行的可执行链接目标文件。
表达式	一个常量、一个符号或由算术运算符分隔的一系列常量和符号。
外部符号	一种在当前程序模块中使用但在其他程序模块中定义或声明的符号。

文件级优化	一种优化级别，编译程序使用其具有的有关整个文件的信息来优化代码（与程序级优化相反，编译程序使用其具有的有关整个程序的信息来优化代码）。
函数内联	在调用点为函数插入代码的过程。这节省了函数调用的开销，并允许优化器在周围代码的上下文中优化函数。
全局符号	一种在当前模块中定义并在另一模块中访问或者在当前模块中访问但在另一模块中定义的符号。
高级别语言调试	编译程序保留符号和高级别语言信息（如类型和函数定义）的能力，这样调试工具就可以使用此类信息。
间接调用	一种函数调用，其中一个函数通过给出被调用函数的地址来调用另一个函数。
加载时初始化	链接 C/C++ 代码时由链接器使用的自动初始化方法。在使用 --ram_model 链接选项调用时，链接器会使用此方法。此方法在加载时而不是运行时初始化变量。
初始化段	从目标文件中链接到可执行目标文件中的段。
输入段	从目标文件中链接到可执行目标文件中的段。
集成预处理器	与解析器合并的 C/C++ 预处理器，以允许更快的编译。也可以使用独立的预处理或已预处理的列表。
交叠特征	一种将原始 C/C++ 源语句作为注释插入到汇编器的汇编语言输出中的特征。C/C++ 语句被会插入到等效汇编指令的旁边。
内联函数	像函数一样使用的运算符，可生成在 C 中无法表达或者需要更多时间和精力才能编写代码的汇编语言代码。
ISO	国际标准化组织；一个由国家标准机构组成的全球联合会，其制定了行业自愿遵循的国际标准。
K&R C	Kernighan 和 Ritchie C，在 *C 程序设计语言* (K&R) 第一版中定义的事实标准。大多数为早期非 ISO C 编译器编写的 K&R C 程序应该无需修改即可正确编译和运行。
标签	从汇编器源语句第 1 列开始并与该语句的地址相对应的符号。标签是唯一可以从第 1 列开始的汇编器语句。
链接器	一种将目标文件组合成可执行目标文件的软件程序，该文件可分配到系统内存中并由器件执行。
列表文件	由汇编器创建的输出文件，其中列出源语句、源语句的行号以及源语句对段程序计数器 (SPC) 的影响。
小端	一种寻址协议，字中的字节从右至左进行编号。字中较高的有效字节存放在高地址处。字节序视硬件而定，并在复位时确定。另请参阅*大端字节序*
加载器	一种将可执行目标文件放入系统内存的器件。
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。
pop	从堆栈中检索数据对象的操作。
pragma	一种指示编译器如何处理特殊语句的预处理器指令。
预处理器	一种解释宏定义、扩展宏、解释头文件、解释有条件编译以及对预处理器指令起作用的软件工具。
程序级优化	一种将所有源文件编译成一个中间文件的积极的优化级别。由于编译器可以看到整个程序，因此在程序级优化中执行了一些很少在文件级优化中应用的优化。
序言	函数中设置堆栈的代码部分。
推入	将数据对象放在堆栈上以进行临时存储的操作。
无声运行	用于抑制正常横幅和进度信息的选项。
原始数据	输出段中的可执行代码或初始化数据。
重定位	一种当符号的地址改变时由链接器调整对符号的所有引用的过程。

运行时环境 程序必须在其中运行的运行时参数。这些参数由内存和寄存器约定、堆栈组织、函数调用约定及系统初始化定义。

运行时支持函数 标准的 ISO 函数，执行不属于 C 语言的任务（比如内存分配、字符串转换和字符串搜索等）。

运行时支持库 库文件 rts.src，其包含运行时支持函数的源代码。

段 一个可重定位的代码块或数据块，最终将与内存映射中的其他段接续。

符号扩展 用值的符号位来填充该值未使用的 MSB 的过程。

模拟器 一种模拟 ARM 运行的软件开发系统。

源文件 一种包含 C/C++ 代码或汇编语言代码的文件，该代码经编译或汇编后形成目标文件。

独立预处理器 一种将宏、#include 文件和条件编译扩展为独立程序的软件工具。其还执行集成预处理，包括解析指令。

静态变量 范围局限在一个函数或程序内的一种变量。当函数或程序退出时，静态变量的值不会被丢弃；当重新输入函数或程序时，将恢复其之前的值。

存储类 符号表中指示如何访问符号的条目。

字符串表 存储长度超过八个字符的符号名称的表（长度为八个字符或更长的符号名称不能存储在符号表中，而是存储在字符串表中）。符号入口点的名称部分指向字符串表中字符串的位置。

结构 一个或者多个变量组合在单个名称下的集合。

子段 一个可重定址的代码块或数据块，最终将占用存储器映射中的连续空间。子段为较大段中的小段。子段使用户能够更严格地控制存储器映射。

符号 表示地址或值的字母数字字符串。

符号调试 软件工具的能力，用于保留可供仿真器或模拟器等调试工具使用的符号信息。

目标系统 执行其上开发了目标代码的系统。

.text 段 默认的目标文件段之一。.text 段被初始化并包含可执行代码。可以使用 .text 指令将代码汇编到 .text 段中。

三字符序列 具有某种含义的 3 字符序列（由 ISO 646-1983 不变代码集定义）。这些字符不能在 C 字符集中表示，而是扩展为一个字符。例如，三个字符 ??' 扩展为 ^。

循环计数 循环结束前执行的次数。

未配置的内存 未定义为存储器映射的一部分，且无法加载代码或数据的存储器。

未初始化段 在存储器映射中保留空间但没有实际内容的目标文件段。这些段是使用.bss 和 .usect 指令创建的。

无符号值 无论实际符号如何都会被当作非负数的值。

变量 表示可以假设一组值中的任何一个数的符号。

合成 一种指令序列，当需要改变状态时，其作为例程的备用入口点。

字 目标内存中的 32 位可寻址位置。

TEXAS INSTRUMENTS

Changes from MARCH 11, 2020 to MARCH 31, 2023 (from Revision V (March 2020) to Revision W (March 2023)) **Page**

- 更新了整个文档中的表格、图和交叉参考的编号格式。 9
- 删除了整个文档中对处理器 Wiki 的引用。 9
- --strict_compatibility 链接器选项不再起任何作用，已从文档中删除。 25
- 记录了 ptrdiff_t 和 size_t 类型的预定义宏。 35
- 更正了出现在其中的 --gen_cross_reference_listing 和 --asm_cross_reference_listing 选项的名称。 44
- 阐明了 --opt_level=4 必须位于 --run_linker 选项之前 58
- 更正了有关 --gen_data_subsections 选项默认值及其与 SET_DATA_SECTION pragma 交互的信息。 74
- 更新了有关枚举类型大小的信息。 88
- 删除了不受支持的 CODE_ALIGN pragma 文档，改用对齐的函数属性 95
- 阐明 --opt_level 和 FUNCTION_OPTIONS pragma 之间的交互。 102
- 记录了与 MUST_ITERATE pragma 对应的各个属性的 C++ 属性语法。 104
- 添加了 PROB_ITERATE pragma 的文档。 109
- 记录了与 UNROLL pragma 对应的各个属性的 C++ 属性语法。 113
- 增加了使用位置属性的示例。 129
- 阐明了有关字符串处理函数的信息。 171
- 添加了关于时间和时钟 RTS 函数的信息。 171

下表列出了更改文档编号格式前对此文档做出的改动。左列标识了本文档出现该特定改动的首个版本。

早期修订版本

添加内容的版本	章节	位置	添加/修改/删除
SPNU151V	链接	节 4.3.5	阐明了如果只有链接器在运行，则需要 --rom_model 或 --ram_model，但如果编译器在同一命令行中的 C/C++ 文件上运行，则 --rom_model 是默认选项。
SPNU151V	C/C++ 语言	节 5.11.22	`#pragma once` 现记录在头文件中使用。
SPNU151V	运行时环境	节 6.10.3.1	阐明了只有使用 --rom_model 链接器选项时，才发生零初始化，使用 --ram_model 选项则不发生。
SPNU151U		-- 全文 --	更改了由编译器创建的目标文件的默认文件扩展名，以防止在 C 和 C++ 文件具有相同名称时发生冲突。从 C 源文件生成的目标文件具有 .c.obj 扩展名。从 C++ 源文件生成的目标文件具有 .cpp.obj 扩展名。
SPNU151T	使用编译器	节 2.3.1	添加了 --emit_references:file 链接器选项。
SPNU151T	使用编译器	节 2.5.1	记录了支持 C 标准宏命令，例如 __STDC_VERSION__。
SPNU151T	C/C++ 语言	节 5.11	添加了 CODE_ALIGN pragma 的文档。
SPNU151T	C/C++ 语言	节 5.11.19	阐明了 NOINIT 和 PERSISTENT pragma 的段放置。
SPNU151T	C/C++ 语言	节 5.14	更正了 _norm 内在函数的语法。
SPNU151T	C/C++ 语言	节 5.16.1	更新了 C99 不受支持的运行时函数列表。
SPNU151T	C/C++ 语言	节 5.17.2	添加了 aligned、calls、naked 和 weak 函数属性的文档。
SPNU151T	C/C++ 语言	节 5.17.4	添加了 location 和 packed 变量属性的文档。

早期修订版本 (continued)

添加内容的版本	章节	位置	添加/修改/删除
SPNU151T	运行时支持函数	DEV_lseek 主题	更正了 DEV_lseek 函数的语法记录。
SPNU151S	简介， 使用编译器， C/C++ 语言	节 1.3、节 2.3、节 5.1 和节 5.16.2	添加了对 C11 的支持。
SPNU151S	使用编译器	节 2.3.1	添加了 --ecc=on 链接器选项，支持生成 ECC。请注意，现在 ECC 生成默认关闭。
SPNU151S	使用编译器	节 2.5.1	__TI_STRICT_ANSI_MODE__ 和 __TI_STRICT_FP_MODE__ 宏命令定义为在条件为 false 时为 0。
SPNU151S	使用编译器， C/C++ 语言	节 2.11 和节 5.11	修订了有关内联函数扩展的段及子段，以包括新的 pragma 并更改了编译器关于内联哪些函数的决策。添加了 FORCEINLINE、FORCEINLINE_RECURSIVE 和 NOINLINE pragma。
SPNU151S	C/C++ 语言	节 5.2	现在支持与原子相关的 C++11 功能。此外，从例外列表中删除了几个 C++ 功能，因为有多个版本已支持这些功能。
SPNU151S	C/C++ 语言	节 5.6	添加了有关字符集和文件编码的信息。
SPNU151S	C/C++ 语言	节 5.14	更正了 _smac 内在函数的语法。
SPNU151S	C/C++ 语言	节 5.17.2 和节 5.17.4	添加了 "retain" 作为函数属性和变量属性。
SPNU151S	C/C++ 语言	节 5.17.6	阐明了 __builtin_sqrt() 和 __builtin_sqrtf() 函数的可用性。
SPNU151R	使用编译器， C/C++ 语言	节 2.3 和节 5.2	编译器现在遵循 C++14 标准。
SPNU151R	C/C++ 语言	节 5.17	编译器现在支持多个 Clang __has_ 宏命令扩展。
SPNU151R	C/C++ 语言	节 5.17.1	现在支持包装器头文件 GCC 扩展 (#include_next)。
SPNU151Q	使用编译器， C/C++ 语言	表 2-31、节 5.1、节 5.14、节 5.17.2	支持 ARM C 语言扩展 (ACLE)。
SPNU151Q	使用编译器	节 2.14	更新了 --float_support 选项的设置列表。
SPNU151Q	C/C++ 语言	节 5.2	为了在将来的版本中支持 C++14，进行了初步更改。这些更改可能会导致链接时错误。重新编译目标文件以解决这些错误。
SPNU151Q	C/C++ 语言	节 5.7.1	阐明了由 const 关键字设置的常量数据存储的例外情况。
SPNU151Q	C/C++ 语言	节 5.14	删除了 _smuad、_smuadx、_smusd 和 _smusdx 内在函数的不正确的第三个参数。
SPNU151P	优化	节 3.7.1.4	更正了处理配置文件数据的命令中的错误。
SPNU151O	使用编译器， C/C++ 语言	节 2.3.3	修改为指明：即使使用 CHECK_MISRA pragma 也需要 --check_misra 选项。
SPNU151O	使用编译器、 C/C++ 语言和 运行时支持函数	节 2.5.1、节 5.16 和节 7.1.1	可定义 _AEABI_PORTABILITY_LEVEL，在包含头文件时实现目标文件全面可移植。
SPNU151O	使用编译器	节 2.10	更正了文档以描述 ---gen_preprocessor_listing 选项。名称 --gen_parser_listing 不正确。
SPNU151N	优化	节 3.7.3	更正了 _TI_start_pprof_collection() 和 _TI_stop_pprof_collection() 的函数名称。
SPNU151M	使用编译器	节 2.3	--cinit_compression 和 --copy_compression 的默认值已从 RLE 更改为 LZSS。
SPNU151M	使用编译器	--	几个编译器选项已被弃用、删除或重命名。编译器仍然接受一些已弃用的选项，但不建议使用它们。
SPNU151M	使用编译器	节 2.5.1	__little_endian__ 和 __big_endian__ 宏命令前面有两个下划线。
SPNU151M	C/C++ 语言	节 5.14	Cortex-M3 支持以下内在函数：__ldrex、__ldrexb、__ldrexh、__strex、__strexb 和 __strexh。
SPNU151M	运行时环境	节 6.8.1	Cortex-R4 和 Cortex-A8 的 _enable_interrupts、_enable_IRQ、_enable_FIQ、_disable_interrupts、_disable_IRQ 和 _disable_FIQ 内在函数现在使用 CPSIE 和 CPSID 指令。

早期修订版本 (continued)

添加内容的版本	章节	位置	添加/修改/删除
SPNU151L	使用编译器	节 2.3 和节 4.2.2	添加了 --gen_data_subsections 选项。
SPNU151L	使用编译器	节 2.3.5	--symdebug:dwarf_version 选项可以设置为 4 以支持使用 DWARF 调试格式版本 4。
SPNU151L	优化	节 3.7 和节 3.8	描述了反馈导向优化。该技术可用于代码覆盖分析。
SPNU151L	C/C++ 语言	节 5.11.1	添加了 CALLS pragma 以指定一组可从指定调用函数间接调用的函数。使用此 pragma 能够将此类间接调用包含在函数的 inclusive 栈大小的计算中。
SPNU151L	C/C++ 语言	节 5.14	在文档中添加了以下内在函数：__MCR、__MRC。
SPNU151L	运行时环境	节 6.10.1	提供了额外的引导挂钩函数。这些可以定制以在系统初始化期间使用。
SPNU151K	引言	节 1.4	不再支持 COFF 目标文件格式以及 TI_ARM9_ABI 和 TIARM ABI。ARM 代码生成工具现在仅支持嵌入式应用二进制接口 (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。已删除或简化了本文档中提及 COFF 格式的各段。如果希望生成 COFF 输出文件，请使用 v5.2 的 ARM 代码生成工具，并参考 SPNU151J 文档。 弃用了 --abi=coff、--symdebug:profile_coff、--no_sym_merge 和 --diable_clink 选项。
SPNU151K	使用编译器	节 2.3.4	添加了 --ramfunc 选项。如果设置此选项，则将所有函数放在 RAM 中。
SPNU151K	C/C++ 语言	节 5.14	在文档中添加了以下内在函数：__nop、__sqrt、__sqrtf、__wfi、__wfe
SPNU151K	C/C++ 语言	节 5.17.2	添加了 ramfunc 函数属性。它规定了一个函数应该放置在 RAM 中。
SPNU151K	运行时支持函数	节 7.1.2	添加了有关头文件扩展的信息。
SPNU151J	引言	节 1.3	添加了对 C99 和 C++03 的支持。
SPNU151J	使用编译器	表 2-1	添加了 --endian=[big | little] 选项。
SPNU151J	使用编译器	表 2-6、节 2.7 和节 2.3.3	添加了可与 ULP Advisor 搭配使用的 --advice:power 和 --advice:power_severity 选项。
SPNU151J	使用编译器	表 2-8	添加了对 C99 和 C++03 的支持。弃用了 -gcc 选项。--relaxed_ansi 现在是默认选项。
SPNU151J	使用编译器	表 2-8	删除了已弃用的预编译标头的文档。
SPNU151J	使用编译器	表 2-11 和节 2.7.1	添加了--section_sizes 选项，用于段大小的诊断报告。
SPNU151J	使用编译器	表 2-28 和节 4.3.3	添加了 –cinit_hold_wdt 链接器选项。
SPNU151J	使用编译器	节 2.5.1	为 Cortex-M4 添加了 __TI_ ARM_V7M4__ 预定义宏名称。
SPNU151J	使用编译器	节 2.5.3	记录了对 #warning 和 #warn 预处理器指令的支持。
SPNU151J	使用编译器	节 2.6	添加了有关向 main() 传递参数的技术的段。
SPNU151J	使用编译器	节 2.11	记录了 inline 关键字现在在除 C89 严格 ANSI 模式之外的所有模式中都启用。
SPNU151J	C/C++ 语言	节 5.1.1	添加了记录实现定义行为的段。
SPNU151J	C/C++ 语言	节 5.4	添加了对 ULP Advisor 的支持
SPNU151J	C/C++ 语言	节 5.5.1	添加了有关枚举类型大小的文档。
SPNU151J	C/C++ 语言	节 5.11.3、节 5.11.12、节 5.11.13、节 5.11.19 和节 5.11.26	添加了 CHECK_ULP、FUNC_ALWAYS_INLINE、FUNC_CANNOT_INLINE、NOINIT、PERSISTENT 和 RESET_ULP pragma。
SPNU151J	C/C++ 语言	节 5.11.16、节 5.11.27 和节 5.17.2	为 INTERRUPT 和 RETAIN pragma 添加了 C++ 语法。还从 #pragma 语法规范中删除了不必要的分号。现在还支持 GCC 中断和别名函数属性。
SPNU151J	C/C++ 语言	节 5.11.8	添加了 diag_push 和 diag_pop 诊断消息 pragma。
SPNU151J	C/C++ 语言	节 5.14	添加了 __delay_cycles、__get_PRIMASK、__set_PRIMASK、__get_MSP 和 __set_MSP 内在函数。
SPNU151J	C/C++ 语言	节 5.14	更正了 smlalbb、smlalbt、smlaltb、smlaltt、smlabb、smlabt、smlatb 和 smlatt 内在函数的参数。

早期修订版本 (continued)

添加内容的版本	章节	位置	添加/修改/删除
SPNU151J	C/C++ 语言	节 5.16、节 5.16.1 和节 5.16.3	添加了对 C99 和 C++03 的支持。--relaxed_ansi 现在是默认选项，--strict_ansi 是另一个选项；标准违反严格性的“正常模式”不再可用。
SPNU151J	运行时环境	节 6.5	添加了对*汇编语言工具用户指南*中有关在 C 和 C++ 语言中访问链接器符号一节的引用。
SPNU151J	运行时环境	节 6.7.5	添加了有关来自 SWI 处理程序的允许返回值的信息。
SPNU151J	运行时环境	节 6.8.1	为 _disable_interrupts、_enable_interrupts 和 _restore_interrupts 内在函数添加了几个器件系列的说明。添加了对 _enable_IRQ、_disable_IRQ 和 _set_interrupt_priority 内在函数的 Cortex-M 支持。
SPNU151J	运行时环境	节 6.10.1	添加了对系统预初始化的支持。
SPNU151J	运行时支持函数	节 7.1.3	rtssrc.zip 文件中不再提供 RTS 源代码。相反，它位于编译器安装程序 lib/src 子目录内的单独文件中。
SPNU151J	C++ 名称还原器	节 8.1	更正了有关名称还原器选项的信息。
SPNU151J	C++ 名称还原器	节 8.2	更正了生成的汇编输出的示例。

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