

MSP430 优化 C/C++ 编译器 **v21.6.0.LTS**

User' s Guide



Literature Number: ZHCU820Y
SEPTEMBER 2004 - REVISED JUNE 2021

请先阅读.....	9
关于本手册.....	9
标记规则.....	10
相关文档.....	11
德州仪器 (TI) 提供的相关文档.....	12
商标.....	12
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.....	24
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 符号调试选项.....	30
2.3.6 指定文件名.....	30
2.3.7 更改编译器解释文件名的方式.....	31
2.3.8 更改编译器处理 C 文件的方式.....	31
2.3.9 更改编译器解释和命名扩展名的方式.....	31
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.....	33
2.4.1 设置默认编译器选项 (MSP430_C_OPTION).....	34
2.4.2 命名一个或多个备用目录 (MSP430_C_DIR).....	35
2.5 控制预处理器.....	35
2.5.1 预先定义的宏名称.....	36
2.5.2 #include 文件的搜索路径.....	37
2.5.3 支持#warning 和 #warn 指令.....	38
2.5.4 生成预处理列表文件 (--preproc_only 选项).....	38
2.5.5 预处理后继续编译 (--preproc_with_compile 选项).....	38
2.5.6 生成带有注释的预处理列表文件 (--preproc_with_comment 选项).....	38
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).....	38
2.5.8 为 Make 实用程序生成预处理输出 (--preproc_dependency 选项).....	39
2.5.9 生成包含#include 在内的文件列表 (--preproc_includes 选项).....	39
2.5.10 在文件中生成宏列表 (--preproc_macros 选项).....	39
2.6 将参数传递给 main().....	39
2.7 了解诊断消息.....	40
2.7.1 控制诊断消息.....	41
2.7.2 如何使用诊断抑制选项.....	42

2.8 其他消息.....	42
2.9 生成交叉引用列表信息 (--gen_cross_reference 选项)	42
2.10 生成原始列表文件 (--gen_preprocessor_listing 选项)	44
2.11 使用内联函数扩展.....	45
2.11.1 内联内在函数运算符.....	45
2.11.2 内联限制.....	46
2.12 使用交叉列出功能.....	47
2.13 控制应用程序二进制接口.....	47
2.14 启用入口挂钩和出口挂钩函数.....	49
3 优化您的代码.....	51
3.1 调用优化.....	52
3.2 控制代码大小与速度.....	52
3.3 执行文件级优化 (--opt_level=3 选项)	53
3.3.1 创建优化信息文件 (--gen_opt_info 选项)	53
3.4 程序级优化 (--program_level_compile 和 --opt_level=3 选项)	53
3.4.1 控制程序级优化 (--call_assumptions 选项)	54
3.4.2 混合 C/C++ 和汇编代码时的优化注意事项.....	54
3.5 自动内联扩展 (--auto_inline 选项)	55
3.6 链接时优化 (--opt_level=4 选项)	56
3.6.1 选项处理.....	56
3.6.2 不兼容的类型.....	56
3.7 使用反馈制导优化.....	56
3.7.1 反馈向导优化.....	57
3.7.2 分析数据解码器.....	59
3.7.3 反馈制导优化 API.....	60
3.7.4 反馈制导优化总结.....	60
3.8 使用配置文件信息分析代码覆盖率.....	60
3.8.1 代码覆盖.....	61
3.8.2 相关的特征和功能.....	61
3.9 访问优化代码中的别名变量.....	62
3.10 在优化代码中谨慎使用 asm 语句.....	62
3.11 将交叉列出功能与优化一起使用.....	62
3.12 调试优化代码.....	63
3.13 正在执行什么类型的优化 ?	63
3.13.1 基于成本的寄存器分配.....	64
3.13.2 别名消歧.....	64
3.13.3 分支优化和控制流简化.....	64
3.13.4 数据流优化.....	64
3.13.5 表达式简化.....	65
3.13.6 函数的内联扩展.....	65
3.13.7 函数符号别名.....	65
3.13.8 归纳变量和强度降低.....	65
3.13.9 循环不变量代码运动.....	65
3.13.10 循环旋转.....	65
3.13.11 指令排程.....	65
3.13.12 尾部合并.....	65
3.13.13 用常数除数进行整数除法.....	65
4 链接 C/C++ 代码.....	67
4.1 通过编译器调用链接器 (-z 选项)	68
4.1.1 单独调用链接器.....	68
4.1.2 调用链接器作为编译步骤的一部分.....	69
4.1.3 禁用链接器 (--compile_only 编译器选项)	69
4.2 链接器代码优化.....	70
4.2.1 条件链接.....	70
4.2.2 生成聚合数据子段 (--gen_data_subsections 编译器选项)	70
4.3 控制链接过程.....	70

4.3.1 包含运行时支持库.....	70
4.3.2 运行时初始化.....	71
4.3.3 通过中断向量进行初始化.....	71
4.3.4 FRAM 内存保护单元的初始化.....	71
4.3.5 Cinit 的初始化和看门狗计时器保持.....	72
4.3.6 全局对象构造函数.....	72
4.3.7 指定全局变量初始化类型.....	72
4.3.8 指定在内存中分配段的位置.....	72
4.3.9 链接器命令文件示例.....	74
5 C/C++ 语言实现.....	75
5.1 MSP430 C 的特征.....	76
5.1.1 实现定义的行为.....	76
5.2 MSP430 C++ 的特征.....	80
5.3 使用 ULP Advisor.....	81
5.4 关于硬件配置的建议.....	81
5.5 数据类型.....	82
5.5.1 枚举类型大小.....	83
5.6 文件编码和字符集.....	84
5.7 关键字.....	85
5.7.1 const 关键字.....	85
5.7.2 __interrupt 关键字.....	85
5.7.3 restrict 关键字.....	86
5.7.4 volatile 关键字.....	86
5.8 C++ 异常处理.....	88
5.9 寄存器变量和参数.....	88
5.10 __asm 语句.....	89
5.11 pragma 指令.....	90
5.11.1 BIS_IE1_INTERRUPT.....	91
5.11.2 CALLS Pragma.....	91
5.11.3 CHECK_ULP Pragma.....	91
5.11.4 CODE_ALIGN Pragma.....	91
5.11.5 CODE_SECTION Pragma.....	92
5.11.6 DATA_ALIGN Pragma.....	94
5.11.7 DATA_SECTION Pragma.....	94
5.11.8 诊断消息 Pragma.....	94
5.11.9 FORCEINLINE Pragma.....	95
5.11.10 FORCEINLINE_RECURSIVE Pragma.....	95
5.11.11 FUNC_ALWAYS_INLINE Pragma.....	96
5.11.12 FUNC_CANNOT_INLINE Pragma.....	97
5.11.13 FUNC_EXT_CALLED Pragma.....	97
5.11.14 FUNC_IS_PURE Pragma.....	98
5.11.15 FUNC_NEVER_RETURNS Pragma.....	98
5.11.16 FUNC_NO_GLOBAL_ASG Pragma.....	98
5.11.17 FUNC_NO_IND_ASG Pragma.....	99
5.11.18 FUNCTION_OPTIONS Pragma.....	99
5.11.19 INTERRUPT Pragma.....	100
5.11.20 LOCATION Pragma.....	101
5.11.21 MUST_ITERATE Pragma.....	101
5.11.22 NOINIT 和 PERSISTENT Pragma.....	103
5.11.23 NOINLINE Pragma.....	104
5.11.24 NO_HOOKS Pragma.....	104
5.11.25 once Pragma.....	104
5.11.26 pack Pragma.....	105
5.11.27 PROB_ITERATE Pragma.....	106
5.11.28 RESET_ULP Pragma.....	106
5.11.29 RETAIN Pragma.....	106
5.11.30 SET_CODE_SECTION 和 SET_DATA_SECTION Pragma.....	107
5.11.31 UNROLL Pragma.....	107

5.11.32	vector Pragma.....	109
5.11.33	WEAK Pragma.....	109
5.12	_Pragma 运算符.....	110
5.13	应用程序二进制接口.....	111
5.14	目标文件符号命名规则 (链接名)	111
5.15	更改 ANSI/ISO C/C++ 语言模式.....	111
5.15.1	C99 支持 (--c99).....	112
5.15.2	C11 支持 (--c11).....	112
5.15.3	严格 ANSI 模式和宽松 ANSI 模式 (--strict_ansi 和 --relaxed_ansi)	112
5.16	GNU 和 Clang 语言扩展.....	114
5.16.1	扩展.....	114
5.16.2	函数属性.....	115
5.16.3	For 循环属性.....	117
5.16.4	变量属性.....	118
5.16.5	类型属性.....	119
5.16.6	内置函数.....	120
5.17	编译器限制.....	120
6	运行时环境	121
6.1	内存模型	122
6.1.1	代码内存模型.....	122
6.1.2	数据存储模式.....	122
6.1.3	支持近数据.....	122
6.1.4	段.....	123
6.1.5	C/C++ 软件堆栈.....	124
6.1.6	动态内存分配.....	124
6.2	对象表示.....	126
6.2.1	数据类型存储.....	126
6.2.2	字符串常量.....	127
6.3	寄存器惯例.....	128
6.4	函数结构和调用惯例.....	128
6.4.1	函数如何进行调用.....	130
6.4.2	被调用函数如何响应.....	131
6.4.3	访问参数和局部变量.....	131
6.5	访问 C 和 C++ 中的链接器符号.....	131
6.6	将 C 和 C++ 与汇编语言相连.....	131
6.6.1	使用汇编语言模块与 C/C++ 代码.....	132
6.6.2	从 C/C++ 访问汇编语言函数.....	132
6.6.3	从 C/C++ 访问汇编语言变量.....	133
6.6.4	与汇编源代码共享 C/C++ 头文件.....	134
6.6.5	使用内联汇编语言.....	134
6.7	中断处理.....	134
6.7.1	在中断期间保存寄存器.....	135
6.7.2	使用 C/C++ 中断例程.....	135
6.7.3	使用汇编语言中断例程.....	135
6.7.4	中断向量	135
6.7.5	其他中断信息.....	135
6.8	使用内在函数访问汇编语言语句.....	136
6.8.1	MSP430 内在函数.....	136
6.8.2	Floating Point Conversion Intrinsics.....	138
6.8.3	弃用的内在函数.....	138
6.8.4	__delay_cycle 内在函数.....	138
6.8.5	__never_executed 内在函数.....	138
6.9	系统初始化.....	140
6.9.1	用于系统预初始化的引导挂钩函数.....	140
6.9.2	运行时栈.....	140
6.9.3	变量的自动初始化.....	141

6.9.4 初始化表.....	146
6.10 对 20 位 MSP430X 器件的编译.....	147
7 使用运行时支持函数并构建库.....	149
7.1 C 和 C++ 运行时支持库.....	150
7.1.1 将代码与对象库链接.....	150
7.1.2 头文件.....	150
7.1.3 修改库函数.....	150
7.1.4 支持字符串处理.....	151
7.1.5 极少支持国际化.....	151
7.1.6 时间和时钟函数支持.....	151
7.1.7 允许打开的文件数量.....	152
7.1.8 源码树中的非标准头文件.....	152
7.1.9 库命名规则.....	153
7.2 C I/O 函数.....	153
7.2.1 高级 I/O 函数.....	154
7.2.2 低级 I/O 实现概述.....	154
7.2.3 器件驱动程序级别 I/O 函数.....	158
7.2.4 为 C I/O 添加用户定义的器件驱动程序.....	162
7.2.5 器件前缀.....	163
7.3 处理可重入性 (_register_lock() 和 _register_unlock() 函数)	165
7.4 库构建流程.....	166
7.4.1 所需的非德州仪器 (TI) 软件.....	166
7.4.2 使用库构建流程.....	166
7.4.3 扩展 mklib.....	169
8 C++ 名称还原器.....	171
8.1 调用 C++ 名称还原器.....	172
8.2 C++ 名称还原器的示例用法.....	172
A 术语表.....	175
A.1 术语.....	175
B 修订历史记录.....	180

插图清单

图 1-1. MSP430 软件开发流程.....	14
图 6-1. var 的内存布局.....	127
图 6-2. 在函数调用期间使用栈.....	129
图 6-3. 运行时自动初始化.....	142
图 6-4. 加载时初始化.....	145
图 6-5. 构造函数表.....	145
图 6-6. .cinit 段中初始化记录的格式.....	146
图 6-7. .pinit 段中初始化记录的格式.....	147

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⁾	19
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. 硬件配置顾问选项.....	20
表 2-8. 控制选项.....	20
表 2-9. 语言选项.....	21
表 2-10. 解析器预处理选项.....	21
表 2-11. 预定义宏选项.....	21
表 2-12. 诊断消息选项.....	22
表 2-13. 补充信息选项.....	22
表 2-14. 运行时模型选项.....	22

表 2-15. 入口/出口挂钩选项.....	23
表 2-16. 反馈选项.....	23
表 2-17. 汇编器选项.....	23
表 2-18. 文件类型说明符选项.....	23
表 2-19. 目录说明符选项.....	24
表 2-20. 默认文件扩展名选项.....	24
表 2-21. 命令文件选项.....	24
表 2-22. 链接器基本选项.....	24
表 2-23. 文件搜索路径选项.....	24
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.....	25
表 2-28. 运行时环境选项.....	26
表 2-29. 其他选项.....	26
表 2-30. 预定义的 MSP430 宏名称.....	36
表 2-31. 原始列表文件标识符.....	44
表 2-32. 原始列表文件诊断标识符.....	44
表 3-1. 可与 --opt_level=3 结合使用的选项.....	53
表 3-2. 为 --gen_opt_info 选项选择一个级别.....	53
表 3-3. 为 --call_assumptions 选项选择一个级别.....	54
表 3-4. 使用 --call_assumptions 选项时的特殊注意事项.....	54
表 4-1. 由编译器创建的初始化段.....	72
表 4-2. 由编译器创建的未初始化段.....	73
表 5-1. MSP430 C/C++ 数据类型.....	82
表 5-2. MSP430 指针的数据大小.....	83
表 5-3. GCC 语言扩展.....	114
表 6-1. 段和存储器位置摘要.....	123
表 6-2. 寄存器和存储器中的数据表示.....	126
表 6-3. 惯例对寄存器类型的影响.....	128
表 6-4. 寄存器使用和保留惯例.....	128
表 6-5. MSP430 内在函数.....	136
表 6-6. 弃用的内在函数.....	138
表 7-1. __time32_t 和 __time64_t 之间的区别.....	152
表 7-2. mklib 程序选项.....	168

关于本手册

《MSP430 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 (相当于 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：

```
cl430 [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：

```
cl430 --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***]。

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/>)

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：

- SLAU131** 《*MSP430 汇编语言工具用户指南*》。介绍了用于 MSP430 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、目标文件格式和符号调试指令。
- SLAA534** 《*MSP430 嵌入式应用二进制接口应用报告*》。为德州仪器 (TI) 的 MSP430 系列处理器提供基于 ELF 的嵌入式应用二进制接口 (EABI) 的规范。EABI 定义了程序、程序组件和执行环境 (如果存在操作系统，还包括操作系统) 之间的低级接口。
- SLAU049** 《*MSP430x1xx 系列用户指南*》。介绍了 MSP430x1xx™ 这些超低功耗微控制器的 CPU 架构、指令集、流水线和中断。
- SLAU144** 《*MSP430x2xx 系列用户指南*》。介绍了 MSP430x2xx™ 这些超低功耗微控制器的 CPU 架构、指令集、流水线和中断。
- SLAU012** 《*MSP430x3xx 系列用户指南*》。介绍了 MSP430x3xx™ 这些超低功耗微控制器的 CPU 架构、指令集、流水线和中断。
- SLAU056** 《*MSP430x4xx 系列用户指南*》。介绍了 MSP430x4xx™ 这些超低功耗微控制器的 CPU 架构、指令集、流水线和中断。
- SLAU208** 《*MSP430x5xx 系列用户指南*》。介绍了 MSP430x5xx™ 这些超低功耗微控制器的 CPU 架构、指令集、流水线和中断。
- SLAU134** 《*MSP430FE42x ESP30CE1 外设模块用户指南*》。介绍了 MSP430FE42x 和 ESP430CE1 超低功耗微控制器上可用的常见外设。本书包含有关 ESP430CE1 的设置、操作和寄存器的信息。
- 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

MSP430x1xx™, MSP430x2xx™, MSP430x3xx™, MSP430x4xx™, MSP430x5xx™, MSP430™, and Code Composer Studio™ are trademarks of Texas Instruments.

所有商标均为其各自所有者的财产。

MSP430™ 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《MSP430 汇编语言工具用户指南》中详细讨论了汇编器和链接器。

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。

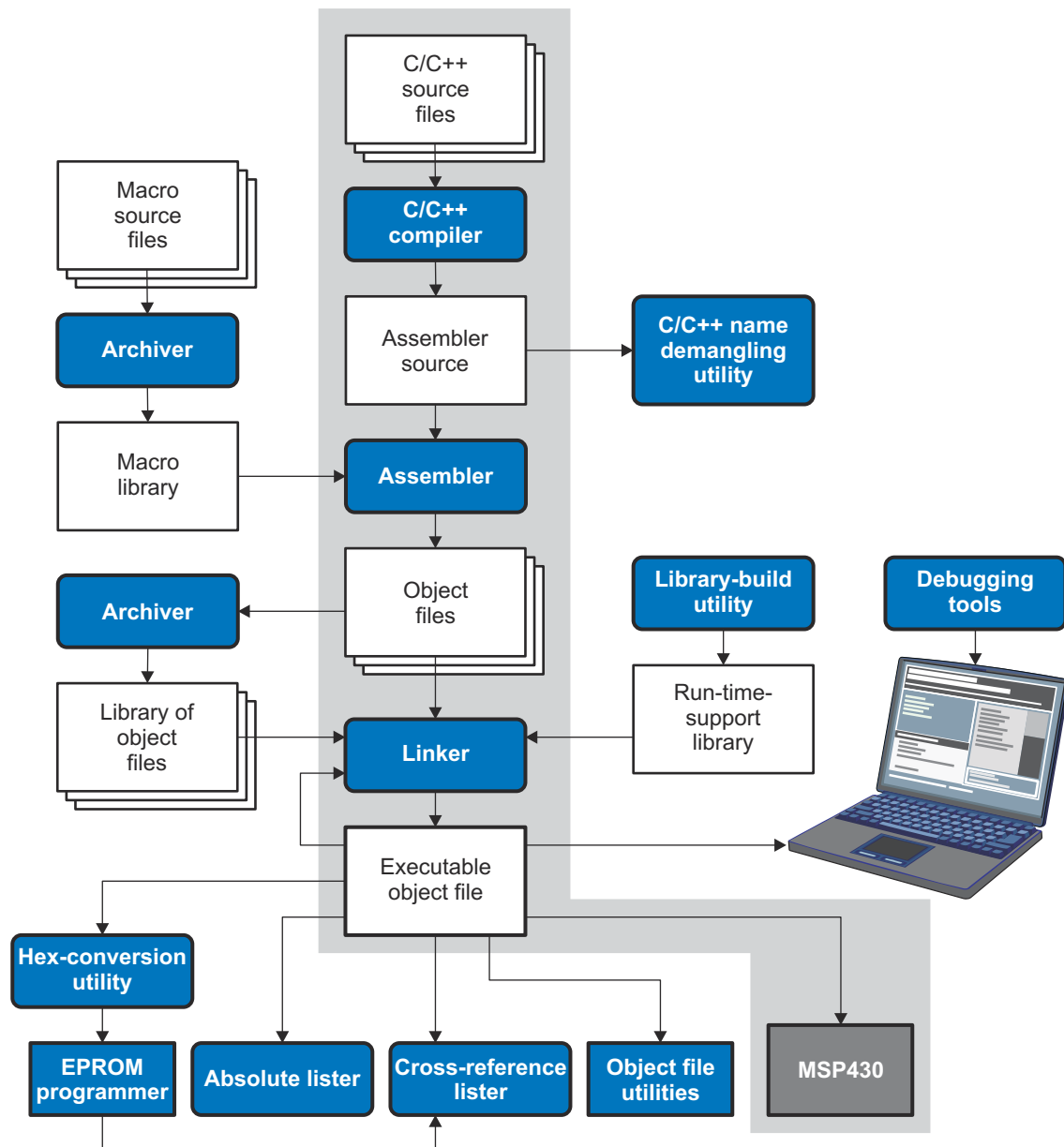


图 1-1. MSP430 软件开发流程

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 MSP430 汇编语言源代码。请参阅 [章节](#)。
- **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《[MSP430 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](#)。请参阅《[MSP430 汇编语言工具用户指南](#)》。
- **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《[MSP430 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](#)。

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 [《MSP430 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 [《MSP430 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 [《MSP430 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](#)。
- **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 [《MSP430 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 **MSP430** 器件上执行。

1.2 编译器接口

编译器是名为 cl430 的命令程序。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](#)。

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++ 语言特征符合下述 ISO 标准：

- **ISO 标准 C**

C 编译器支持 1989、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++ 功能的说明，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](#)。

请参阅 [节 5.15](#)，了解命令行选项以选择代码所使用的 C 或 C++ 标准。

1.4 输出文件

以下类型的输出文件由编译器创建：

- **ELF 目标文件。**可执行和可链接格式 (ELF) 支持早期模板实例化和内联函数导出等现代语言功能。ELF 是 [System V 应用程序二进制接口 \(ABI\)](#) 的一部分。用于 MSP430 的 ELF 格式由 MSP430 嵌入式应用程序二进制接口 (EABI) 扩展，相关信息请参阅 [SLAA534](#) 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。如果希望生成 COFF 输出文件，请使用 v4.4 的 MSP430 代码生成工具，并参考 [SLAU132J](#) 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 (dem430) 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](#)。

- **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《[MSP430 汇编语言工具用户指南](#)》所述。

编译器将您的源程序转换成 MSP430 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.....	33
2.5 控制预处理器.....	35
2.6 将参数传递给 main().....	39
2.7 了解诊断消息.....	40
2.8 其他消息.....	42
2.9 生成交叉引用列表信息 (--gen_cross_reference 选项)	42
2.10 生成原始列表文件 (--gen_preprocessor_listing 选项)	44
2.11 使用内联函数扩展.....	45
2.12 使用交叉列出功能.....	47
2.13 控制应用程序二进制接口.....	47
2.14 启用入口挂钩和出口挂钩函数.....	49

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文件或可重新链接可执行文件。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](#)。

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《*MSP430 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：

```
cl430 [options] [filenames] [--run_linker [link_options] object files]
```

cl430	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 2-8 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。 <code>--run_linker</code> 选项的缩写形式为 <code>-z</code> 。有关更多信息，请参阅 章节 。
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 `syntab.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：

```
cl430 syntab.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，还提供最常用选项（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 **cl430**。

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 `MSP430_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 2-8](#) 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
<code>--silicon_version={msp msp x}</code>	<code>-v</code>	选择指令集。	节 2.3.4
<code>--code_model={small large}</code>		指定代码内存模型。	节 6.1.1
<code>--data_model={small large restricted}</code>		指定数据内存模型。	节 6.1.2
<code>--near_data={globals none}</code>		指定必须靠近的数据。默认为 <code>globals</code> 。	节 6.1.3

表 2-2. 优化选项⁽¹⁾

选项	别名	效果	段
<code>--opt_level=off</code>		禁用所有优化。	节 3.1
<code>--opt_level=n</code>	<code>-On</code>	级别 0 (<code>-O0</code>) 仅优化寄存器使用情况（如果未使用选项，则为默认值）。 级别 1 (<code>-O1</code>) 使用级别 0 优化并在本地进行优化。 级别 2 (<code>-O2</code>) 使用级别 1 优化并在本地进行优化。 级别 3 (<code>-O3</code>) 使用级别 2 优化并对文件进行优化。 级别 4 (<code>-O4</code>) 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
<code>--opt_for_speed=[n]</code>	<code>-mf</code>	控制大小和速度之间的权衡（0-5 范围）。如果此选项未指定 <code>n</code> ，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 2-14](#)）也会影响优化。

表 2-3. 高级优化选项⁽¹⁾

选项	别名	效果	段
<code>--align_for_power</code>		将所有函数和循环与 4 字节边界对齐。如果小函数或循环与 32 位缓冲区对齐以从闪存中获取代码，则可以实现节能。对于较大的函数和循环，节能就不太重要。	<code>--</code>
<code>--auto_inline=[size]</code>	<code>-oi</code>	设置自动内联大小（仅限 <code>--opt_level=3</code> ）。如果未指定 <code>size</code> ，则默认值为 1。	节 3.5

表 2-3. 高级优化选项⁽¹⁾ (continued)

选项	别名	效果	段
--call_assumptions= <i>n</i>	-op <i>n</i>	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= <i>n</i>	-on <i>n</i>	级别 0 (-on0) 禁用优化信息文件。 级别 1 (-on1) 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
--aliased_variables	-ma	假设变量被别名	节 3.9

(1) 注意：机器专用选项 (参阅表 2-14) 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= <i>directory</i>	-I	将指定的目录添加到 #include 搜索路径。	节 2.5.2.1
--preinclude= <i>filename</i>		在编译开始时包含 <i>filename</i> 。	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. 硬件配置顾问选项

选项	别名	效果	段
--advice:hw_config[={all none rulespec}]		启用检查硬件配置设置。(如果此选项未包含在命令行中，则默认为 none。如果使用此选项但未指定值，则默认为 all。)	节 2.3.3

表 2-8. 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 2-8. 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 2-9. 语言选项

选项	别名	效果	段
--c89		根据 ISO C89 标准处理 C 文件。	节 5.15
--c99		根据 ISO C99 标准处理 C 文件。	节 5.15
--c11		根据 ISO C11 标准处理 C 文件。	节 5.15
--c++14		根据 ISO C++14 标准处理 C++ 文件。 已弃用 --c++03 选项。	节 5.15
--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={packed unpacked}		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	-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	将字符类型的变量从无符号更改为有符号。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 5.15.3
--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 5.15.3

表 2-10. 解析器预处理选项

选项	别名	效果	段
--preproc_dependency[= <i>filename</i>]	-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 实用程序的依赖列表。	节 2.5.8
--preproc_includes[= <i>filename</i>]	-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[= <i>filename</i>]	-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 2-11. 预定义宏选项

选项	别名	效果	段
--define= <i>name</i> [= <i>def</i>]	-D	预定义 <i>name</i> 。	节 2.3.2

表 2-11. 预定义宏选项 (continued)

选项	别名	效果	段
--undefine= <i>name</i>	-U	未定义 <i>name</i> 。	节 2.3.2

表 2-12. 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= <i>num</i>	-pdse	将 <i>num</i> 标识的诊断分类为错误。	节 2.7.1
--diag_remark= <i>num</i>	-pdsr	将 <i>num</i> 标识的诊断分类为备注。	节 2.7.1
--diag_suppress= <i>num</i>	-pds	抑制 <i>num</i> 标识的诊断。	节 2.7.1
--diag_warning= <i>num</i>	-pds	将 <i>num</i> 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= <i>num</i>	-pdel	将错误限制设置为 <i>num</i> 。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 2-13. 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 2-14. 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
--disable_interrupts_around_hw_mpy=off		将此选项设置为 off 会在内联的硬件乘法运算期间使中断启用。仅当所有 ISR 不使用乘法或调用其他函数时才能使用此选项。	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={r4 r5}		保留寄存器供用户使用。	
--large_memory_model	-ml	在编译 MSP430X 时使用大型内存模型。(已弃用)	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

表 2-14. 运行时模型选项 (continued)

选项	别名	效果	段
--silicon_errata={ <i>errata</i> }		生成代码以处理指定的芯片勘误表。 <i>errata</i> 值可以是 CPU12、CPU13、CPU15、CPU18、CPU19、CPU21、CPU22、CPU23 或 CPU40。有关详细信息，请参阅相应的器件勘误表。	
--use_hw_mpy[={16 32 F5 none}]		如果 --opt_for_speed 设置为 1 或更高，则将所有对默认整数/长型乘法例程的引用替换为采用硬件乘法器支持的乘法例程版本。此外，如果 --opt_for_speed 设置为 4 或 5，则在代码生成期间内联硬件乘法。	节 2.3.4

表 2-15. 入口/出口挂钩选项

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

表 2-16. 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= <i>file1</i> [, <i>file2</i> ,...]		指定配置文件信息文件。	节 3.7.1.3

表 2-17. 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_define= <i>name</i> [= <i>def</i>]	-ad	设置 <i>name</i> 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= <i>name</i>	-au	不对预定义的常量 <i>name</i> 进行定义。	节 2.3.11
--asm_listing_cross_reference	-ax	生成交叉引用文件。	节 2.3.11
--include_file= <i>filename</i>	-ahi	包含汇编模块的指定文件。	节 2.3.11

表 2-18. 文件类型说明符选项

选项	别名	效果	段
--asm_file= <i>filename</i>	-fa	不管其扩展名为何，都将 <i>filename</i> 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= <i>filename</i>	-fc	不管其扩展名为何，都将 <i>filename</i> 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= <i>filename</i>	-fp	不管其扩展名为何，都将 <i>filename</i> 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

表 2-18. 文件类型说明符选项 (continued)

选项	别名	效果	段
--obj_file= <i>filename</i>	-fo	不管其扩展名为何, 都将 <i>filename</i> 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 2-19. 目录说明符选项

选项	别名	效果	段
--abs_directory= <i>directory</i>	-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= <i>directory</i>	-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= <i>directory</i>	-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= <i>directory</i>	-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= <i>filename</i>	-fe	指定编译输出文件名; 可以覆盖 --obj_directory。	节 2.3.10
--pp_directory= <i>dir</i>		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= <i>directory</i>	-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 2-20. 默认文件扩展名选项

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

表 2-21. 命令文件选项

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

2.3.1 链接器选项

以下各表列出了链接器选项。有关这些选项的详细信息, 请参阅本文档的[章节](#)以及《MSP430 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= <i>file</i>	-o	为可执行输出文件命名。默认文件名为 .out file。
--map_file= <i>file</i>	-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 <i>file</i> 中。
--stack_size= <i>size</i>	[-]-stack	将 C 系统栈大小设为 <i>size</i> 字节, 并定义全局符号来指定栈大小。默认值 = 128 字节。
--heap_size= <i>size</i>	[-]-heap	将堆大小 (对于 C 中的动态存储器分配) 设为 <i>size</i> 字节, 并定义全局符号来指定栈大小。默认值 = 128 字节。

表 2-23. 文件搜索路径选项

选项	别名	说明
--library= <i>file</i>	-l	将存档库或链接命令 <i>file</i> 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。

表 2-23. 文件搜索路径选项 (continued)

选项	别名	说明
--search_path= <i>pathname</i>	-I	在查找默认位置之前，更改库搜索算法以查找用 <i>pathname</i> 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= <i>name=value</i>		将 <i>name</i> 预定义为预处理器宏命令。
--undefine= <i>name</i>		删除预处理器宏命令 <i>name</i> 。
--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= <i>num</i>		将由 <i>num</i> 标识的诊断分类为错误。
--diag_remark= <i>num</i>		将由 <i>num</i> 标识的诊断分类为备注。
--diag_suppress= <i>num</i>		抑制由 <i>num</i> 标识的诊断。
--diag_warning= <i>num</i>		将由 <i>num</i> 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[= <i>file</i>]		发射包含段信息的文件。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= <i>count</i>		将错误限制设置为 <i>count</i> 。在出现此数量的错误后，链接器将放弃链接。（默认为 100。）
--verbose_diagnostics		提供详细的诊断消息，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 }		启用链接器生成的错误校正码 (ECC)。默认关闭。
--ecc:data_error		将指定的错误注入到输出文件中进行测试。
--ecc:ecc_error		将指定的错误注入到错误校正码 (ECC) 中进行测试。
--mapfile_contents= <i>attribute</i>		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= <i>file</i>		生成结构良好的 XML <i>file</i> ，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= <i>symbol</i>	-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= <i>pattern</i>		将与 <i>pattern</i> 匹配的符号的符号链接更改为全局型。
--hide= <i>pattern</i>		隐藏与指定 <i>pattern</i> 匹配的符号。
--localize= <i>pattern</i>		将与指定 <i>pattern</i> 匹配的符号设为局部型。
--make_global= <i>symbol</i>	-g	将 <i>symbol</i> 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。

表 2-27. 符号管理选项 (continued)

选项	别名	说明
--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 <i>refname</i> 符号的引用被替换为对 <i>defname</i> 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 <i>symbol</i> 作为未解析符号添加到符号表中。
--unhide=pattern		排除与指定 <i>pattern</i> 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 <i>size</i> 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 4.3.5。
--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 <i>type</i> ，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 <i>type</i> ，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。相关详细信息，请参阅节 4.3.7。
--rom_model	-c	在运行时自动初始化变量。相关详细信息，请参阅节 4.3.7。
--use_hw_mpy		此选项是编译器选项。它应放置在命令行上的 --run_linker 或 -z 选项之前。请参阅节 2.3.4。

表 2-29. 其他选项

选项	别名	说明
--linker_help	[-]help	显示有关语法和可用选项的信息。
--preferred_order=function		为函数放置设定优先级。
--strict_compatibility[=off on]		对输入目标文件执行更保守和更严格的兼容性检查。默认为 on。
--unused_section_elimination[=off on]		消除可执行模块中不需要的段。默认为 on。

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 <code>--optimizer_interlist</code> 和 <code>--c_src_interlist</code> 选项，将交叉列出功能与优化器结合使用。请参阅节 3.11。 <code>--c_src_interlist</code> 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 <code>#</code> 或在命令文件中的一行的开头包含注释。可以用 <code>/*</code> 和 <code>*/</code> 括起来添加注释。要指定选项，请用引号将连字符括起来。例如， <code>"--quiet</code> 。可以多次使用 <code>--cmd_file</code> 选项来指定多个文件。例如，以下代码表示 <code>file3</code> 应编译为源文件，而 <code>file1</code> 和 <code>file2</code> 是 <code>--cmd_file</code> 文件： <pre>cl430 --cmd_file=file1 --cmd_file=file2 file3</pre>
--compile_only	抑制链接器并覆盖用于指定链接的 <code>--run_linker</code> 选项。 <code>--compile_only</code> 选项的缩写形式为 <code>-c</code> 。在 <code>MSP430_C_OPTION</code> 环境变量中指定了 <code>--run_linker</code> 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 <code>name</code> 。这相当于在每个 C 源文件的顶部插入 <code>#define name def</code> 。如果省略可选的 <code>[=def]</code> ，则 <code>name</code> 设置为 1。 <code>--define</code> 选项的缩写形式为 <code>-D</code> 。 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： <ul style="list-style-type: none"> • 对于 Windows，请使用 <code>--define=name="string def"</code>。例如，<code>--define=car="sedan"</code> • 对于 UNIX，请使用 <code>--define=name='string def'</code>。例如，<code>--define=car='sedan'</code> • 对于 CCS，请在文件中输入定义并使用 <code>--cmd_file</code> 选项包含该文件。
--gen_func_info_listing	生成带 <code>.aux</code> 文件扩展名的用户信息文件。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 <code>--help</code> 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 <code>--help debug</code> 。
--include_path=directory	将 <code>directory</code> 添加到目标编译器搜索 <code>#include</code> 文件的目录列表中。 <code>--include_path</code> 选项的缩写形式为 <code>-I</code> 。可以多次使用此选项来定义几个目录；请确保用空格分隔 <code>--include_path</code> 选项。如果未指定目录名称，预处理器将忽略 <code>--include_path</code> 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。 <code>--keep_asm</code> 选项的缩写形式为 <code>-k</code> 。
--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。 <code>--quiet</code> 选项的缩写形式为 <code>-q</code> 。
--run_linker	在指定的目标文件上运行链接器。 <code>--run_linker</code> 选项及其参数紧跟在命令行上的所有其他选项之后。 <code>--run_linker</code> 后面的所有参数都传递给链接器。 <code>--run_linker</code> 选项的缩写形式为 <code>-z</code> 。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。 <code>--skip_assembler</code> 选项的缩写形式为 <code>-n</code> 。此选项覆盖了 <code>--run_linker</code> 。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（ <code>--opt_level=n</code> 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。 <code>--src_interlist</code> 选项意味着 <code>--keep_asm</code> 选项。 <code>--src_interlist</code> 选项的缩写形式为 <code>-s</code> 。
--tool_version	打印编译器中每个工具的版本号。没有编译。
--undefine=name	对预定义的常量 <code>name</code> 不定义。此选项覆盖指定常量的任何 <code>--define</code> 选项。 <code>--undefine</code> 选项的缩写形式为 <code>-U</code> 。
--verbose	编译时显示进度信息和工具集版本。重置 <code>--quiet</code> 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:hw_config={all none rulespec}	启用对器件配置的检查。例如，检查 FRAM 等待状态配置。当其中一项检查产生警告时，单击 Code Composer Studio 中提供的链接可以找到有关解决器件配置问题的信息。有关详细信息，请参阅节 5.4。
---	--

--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 5.3。
--advice:power_severity={error warning remark suppress}	设置 ULP Advisor 规则的诊断严重程度。
--float_operations_allowed= {none all 32 64}	限制应用程序中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}	<p>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：</p> <ul style="list-style-type: none"> • 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，则表达式中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，那么它们也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。 <p>在以下示例中，iN=整数变量，fN=浮点变量，dN=双精度变量：</p> <pre style="border: 1px solid black; padding: 5px;">i1 = f1 + f2 * 5.0 -> +, * are float, 5.0 is converted to 5.0f i1 = d1 + d2 * d3 -> +, * are float f1 = f2 + f3 * 1.1; -> +, * are float, 1.1 is converted to 1.1f</pre> <p>要启用宽松浮点模式，请使用 --fp_mode=relaxed 选项，这也会设置 --fp_reassoc=on。要禁用宽松浮点模式，请使用 --fp_mode=relaxed 选项，这也会设置 --fp_reassoc=off。</p> <p>如果指定了 --strict_ansi，则会自动设置 --fp_mode=strict。可以通过在 --strict_ansi 之后指定 --fp_mode=relaxed 以采用严格的 ANSI 模式来启用宽松浮点模式。</p>
--fp_reassoc={on off}	<p>启用或禁用浮点算术的重新组合。如果设置了 --strict_ansi，则设置 --fp_reassoc=off，因为浮点算术的重新关联是违反 ANSI 要求的。</p> <p>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。</p>
--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}	<p>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：</p> <ul style="list-style-type: none"> • 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 <p>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。</p>

2.3.4 运行时模型选项

这些选项对 MSP430 工具集来说是特定的。有关更多信息，请参阅参考的章节。节 2.3.11 中列出了 MSP430 专用汇编器选项。

MSP430 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 MSP430 v4.4 代码生成工具，并参阅 SLAU132J 和 SLAU131J 以查看相关文档。

--common={on off}	<p>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, 则变量不能作为通用符号。</p>
--code_model={large small}	<p>指定代码内存模型: 小型 (16 位函数指针和低 64K 内存) 或大型 (20 位函数指针和 1MB 地址空间)。有关详细信息, 请参阅节 6.1.1。</p>
--data_model={restricted large small}	<p>指定数据内存模型: 小型 (16 位数据指针和低 64K 内存)、受限 (32 位数据指针、限制为 64K 的对象和 1MB 内存) 和大型 (32 位数据指针和 1MB 内存)。有关详细信息, 请参阅节 6.1.2。</p>
--disable_interrupts_around_hw_mpy=off	<p>将此选项设置为 off 会在内联的硬件乘法期间启用中断。(内联硬件乘法由 --use_hw_mpy 和 --opt_for_speed 选项控制。) 仅当应用程序中的所有 ISR 不使用乘法并且不调用其他函数时才应使用此选项。默认情况下, 如果硬件乘法被内联, 则在硬件乘法期间禁用中断, 因为此类运算不可重入。</p> <p>如果此选项设置为 off 并且 ISR 使用乘法, 则链接器会发出警告, 指出 ISR 中的乘法将使用 RTS 软件乘法例程而不是 RTS 硬件乘法例程。</p> <p>如果此选项设置为 off 并且 ISR 调用其他函数, 则链接器生成非致命错误诊断, 指示如果应用程序包含调用其他函数的 ISR, 则不能使用选项 --disable_interrupts_around_hw_mpy=off。这样做的原因是, 如果 ISR 调用使用硬件乘法器进行乘法的例程, 那么启用中断的内联硬件乘法器就不再安全了。</p>
--enum_type={packed unpacked}	<p>默认情况下, 编译器使用 --enum_type=unpacked, 这意味着编译器使用不小于 16 位的整数类型 (int、long 或 long long) 来存储枚举类型变量。如果枚举常量值适合 8 位整数类型, 则 --enum_type=packed 选项会使编译器使用 8 位整数类型。有关枚举类型大小的详细信息, 请参阅节 5.5.1。</p> <p>不要将使用 --enum_type=packed 选项编译的目标文件与没有使用该选项编译的目标文件一起链接。如果正在使用 --enum_type=packed 选项, 则必须始终使用该选项来编译所有 C/C++ 文件; 否则, 您将遇到直到运行时才能检测到的错误。</p>
--large_memory_model	<p>已弃用此选项。使用 --data_model=large。</p>
--near_data={globals none}	<p>指定全局读取/写入数据必须位于内存的前 64K 中。有关详细信息, 请参阅节 6.1.3。</p>
--pending_instantiations=#	<p>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 指定无限数量。</p>
--plain_char={unsigned signed}	<p>指定如何处理 C/C++ 普通字符变量。默认为无符号。</p>
--ramfunc={on off}	<p>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这种方式被处理。请参阅节 5.16.2。</p> <p>较新的 TI 链接器命令文件通过在 .TI.ramfunc 段中旋转函数来自动支持 --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《MSP430 汇编语言工具用户指南》。</p>
--silicon_version	<p>选择指令集版本。使用 --silicon_version=mbsp 为 MSP430X 器件生成代码 (20 位代码寻址)。使用 --silicon_version=msp 为 16 位 MSP430 器件生成代码。</p> <p>为 16 位 MSP430 器件汇编/编译的模块与为 20 位 MSPx 器件汇编/编译的模块不兼容。如果试图组合不兼容的目标文件, 链接器将生成错误。</p>

--use_hw_mpy[= 16,32,F5,none]

此选项是编译器选项。该选项应放置在命令行上 **-z** 选项之前。

MSP430 器件没有乘法指令，而有些器件根本没有硬件乘法。乘法由库函数执行。该库函数要么使用硬件乘法功能，要么在软件中模拟该功能。默认情况下，编译器生成相应版本的引用但该软件版本不使用某些 MSP430 器件版本上可用的硬件乘法器外设。如果 MSP430 器件具有多个硬件，则可以选择使用与硬件匹配的库例程以获得最佳性能。在为硬件乘法器可用的器件进行编译时，请使用 **--use_hw_mpy** 编译器选项。

在编译时，如果 **--opt_for_speed** 设置为 4 或 5 并且 **--opt_level** 设置为除“off”以外的任何值，则使用此选项会导致在代码生成期间内联硬件乘法调用。如果要控制在内联硬件乘法期间是否启用中断，请参阅 **--disable_interrupts_around_hw_mpy=off** 选项。

如果未启用内联但启用了其他优化（**--opt_for_speed** 设置为 1、2 或 3），则使用此选项会使链接器将所有对默认乘法例程的引用替换为支持硬件乘法器的乘法例程版本。

可选参数指示正在使用哪个版本的硬件乘法，并且必须为以下之一：

- 16 使用 F1xx/2xx/4xx 系列 16 位硬件乘法器（默认值）
- 32 使用 F4xx 32 位硬件乘法器
- F5 使用 F5xx/6xx 系列 32 位硬件乘法器
- none = 不使用硬件乘法器

有关硬件乘法器的更多信息，请参阅 MSP430x1xx、MSP430x3xx、MSP430x4xx 和 MSP430x5xx 系列用户指南。

2.3.5 符号调试选项

使用以下选项选择符号调试：

--symdebug:dwarf

（默认）生成 C/C++ 源代码级调试器使用的指令，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}

指定在指定 **--symdebug:dwarf**（默认值）的情况下生成的 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。DWARF 版本 2、3 和 4 可以安全地混合使用。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。这种类型合并方法优于 DWARF 2 或 3，并且可以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。无效。

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：

```
cl1430 *.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：

<code>--asm_file=filename</code>	用于汇编语言源文件
<code>--c_file=filename</code>	用于 C 源文件
<code>--cpp_file=filename</code>	用于 C++ 源文件
<code>--obj_file=filename</code>	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：

```
cl1430 --c_file=file.s --asm_file=assy
```

不能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更改，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。

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：

<code>--asm_extension=new extension</code>	用于汇编语言文件
<code>--c_extension=new extension</code>	用于 C 源文件
<code>--cpp_extension=new extension</code>	用于 C++ 源文件
<code>--listing_extension=new extension</code>	设置列表文件的默认扩展名
<code>--obj_extension=new extension</code>	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：

```
cl430 --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：

```
cl430 --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： <code>cl430 --abs_directory=d:\abso_list</code>
--asm_directory=directory	指定汇编文件的目录。例如： <code>cl430 --asm_directory=d:\assembly</code>
--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： <code>cl430 --list_directory=d:\listing</code>
--obj_directory=directory	指定目标文件的目录。例如： <code>cl430 --obj_directory=d:\object</code>
--output_file=filename	指定编译输出文件名；可以覆盖 --obj_directory 。例如： <code>cl430 --output_file=transfer</code>
--pp_directory=directory	指定目标文件的预处理器文件目录（默认为.）。例如： <code>cl430 --pp_directory=d:\preproc</code>
--temp_directory=directory	指定临时中间文件的目录。例如： <code>cl430 --temp_directory=d:\temp</code>

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《MSP430 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 <i>name</i> ，为常量生成 <code>.set</code> 指令，或为字符串生成 <code>.arg</code> 指令。如果省略可选的 <code>[=def]</code> ，则 <i>name</i> 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： <ul style="list-style-type: none"> 对于 Windows，请使用 <code>--asm_define=name="string def"</code>。例如： <code>asm_define=car "\"sedan\""</code> 对于 UNIX，请使用 <code>--asm_define=name="string def"</code>。例如： <code>asm_define=car "'sedan'"</code> 对于 Code Composer Studio，请在文件中输入定义，并使用 <code>--cmd_file</code> 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 <code>make</code> 实用程序。该列表将写入与源文件同名但扩展名为 <code>.ppa</code> 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 <code>#include</code> 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 <code>.ppa</code> 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 <i>name</i> 进行定义。此选项覆盖指定名称的任何 <code>--asm_define</code> 选项。
--asm_listing_cross_reference	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 <code>.include</code> 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

环境变量是由您定义并为其分配字符串的系统符号。当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时，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 (MSP430_C_OPTION)

您可能会发现，使用 **MSP430_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 **MSP430_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 **MSP430_C_OPTION** 环境变量并进行处理。

下表展示了如何设置 **MSP430_C_OPTION** 环境变量。为操作系统选择命令：

操作系统	输入
UNIX (Bourne shell)	MSP430_C_OPTION=" option₁ [option₂ ...]"; export MSP430_C_OPTION
Windows	set MSP430_C_OPTION= option₁ [option₂ ...]

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 **MSP430_C_OPTION** 环境变量，如下所示：

```
set MSP430_C_OPTION=--quiet --src_interlist --run_linker
```

命令行或 **MSP430_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 **MSP430_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 **MSP430_C_OPTION** 设置如上所示：

```
cl430 *.c ; compiles and links
cl430 --compile_only *.c ; only compiles
cl430 *.c --run_linker lnk.cmd ; compiles and links using a command file
cl430 --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 *MSP430 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 (MSP430_C_DIR)

链接器使用 MSP430_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)	MSP430_C_DIR=" pathname₁ ; pathname₂ ;..."; export MSP430_C_DIR
Windows	set MSP430_C_DIR= pathname₁ ; pathname₂ ;...

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 MSP430_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 MSP430_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)	<code>unset MSP430_C_DIR</code>
Windows	<code>set MSP430_C_DIR=</code>

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. 预定义的 MSP430 宏名称

宏名称	说明
<code>__DATE__</code> ⁽¹⁾	以 <i>mmm dd yyyy</i> 形式扩展到编译日期
<code>__FILE__</code> ⁽¹⁾	扩展到当前源文件名
<code>__LARGE_CODE_MODEL__</code>	如果指定了 <code>--code_model=large</code> ，则已定义
<code>__LARGE_DATA_MODEL__</code>	如果指定了 <code>--data_model=large</code> 或 <code>--data_model=restricted</code> ，则已定义
<code>__LINE__</code> ⁽¹⁾	扩展到当前行号
<code>__LONG_PTRDIFF_T__</code>	如果指定了 <code>--data_model=large</code> 时，则已定义。指示 <code>ptrdiff_t</code> 为 long。
<code>__MSP430__</code>	始终已定义
<code>__MSP430X__</code>	如果指定了 <code>--silicon_version=mspx</code> ，则已定义
<code>__MSP430X461X__</code>	如果指定了 <code>--silicon_version=mspx</code> ，则已定义
<code>__PTRDIFF_T_TYPE__</code>	设置为 <code>ptrdiff_t</code> 类型。由 <code>--data_model</code> 选项确定。
<code>__signed_chars__</code>	如果 <code>char</code> 类型默认为有符号类型 (<code>--plain_char=signed</code>)，则已定义
<code>__SIZE_T_TYPE__</code>	设置为 <code>size_t</code> 类型。由 <code>--data_model</code> 选项确定。
<code>__STDC__</code> ⁽¹⁾	定义为 1 以表示编译器符合 ISO C 标准。有关 ISO C 一致性的例外情况，请参阅节 5.1。
<code>__STDC_VERSION__</code>	C 标准宏。
<code>__STDC_HOSTED__</code>	C 标准宏。始终定义为 1。
<code>__STDC_NO_THREADS__</code>	C 标准宏。始终定义为 1。
<code>__TI_COMPILER_VERSION__</code>	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。
<code>__TI_EABI__</code>	如果启用了 EABI (请参阅节 2.13)，则定义为 1；否则未定义。
<code>__TI_GNU_ATTRIBUTE_SUPPORT__</code>	如果启用了 GCC 扩展 (这是默认设置)，则定义为 1
<code>__TI_STRICT_ANSI_MODE__</code>	如果启用了严格的 ANSI/ISO 模式 (使用了 <code>--strict_ansi</code> 选项)，则定义为 1；否则定义为 0。
<code>__TI_STRICT_FP_MODE__</code>	如果使用了 <code>--fp_mode=strict</code> (默认设置)，则定义为 1；否则定义为 0。
<code>__TIME__</code> ⁽¹⁾	以 “ <i>hh:mm:ss</i> ” 形式扩展到编译时间
<code>__unsigned_chars__</code>	如果 <code>char</code> 类型默认为无符号类型 (默认设置或 <code>--plain_char=unsigned</code>)，则已定义
<code>__UNSIGNED_LONG_SIZE_T__</code>	如果指定了 <code>--data_model=large</code> 时，则已定义。指示 <code>size_t</code> 为无符号长类型。
<code>__INLINE</code>	如果使用了优化 (<code>--opt_level</code> 或 <code>-O</code> 选项)，则扩展为 1；否则未定义。

(1) 由 ISO 标准指定

可以按照与任何其他已定义名称相同的方式使用表 2-30 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");
```

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。文件名可以是完整路径名、部分路径信息或没有路径信息文件名。

- 如果将文件名括在双引号 (" ") 中，编译器将按以下顺序搜索文件：
 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 MSP430_C_DIR 环境变量设置的目录。
- 如果将文件名括入尖括号 (< >) 中，编译器将按以下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 MSP430_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 将目录添加到 #include 文件搜索路径 (--include_path 选项)

--include_path 选项命名包含 #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 指令而无需为文件指定任何目录信息；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。请根据操作系统选择命令：

操作系统	输入
UNIX	cl430 --include_path=/tools/files source.c
Windows	cl430 --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 `MSP430_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 `MSP430_C_DIR`：

```
MSP430_C_DIR "/usr/include;/usr/ucb"; export MSP430_C_DIR
```

或使用以下命令调用编译器：

```
cl430 --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。

输出仅包括源文件直接包含的那些文件。首先列出预定义宏，并由注释 /* Predefined */ 标示。接着列出用户定义宏，并由源文件名标示。

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的内容以供程序使用。

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
    xxxxxx;
    ^
```

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。

<code>--diag_error=num</code>	将由 <code>num</code> 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 <code>--display_error_number</code> 选项。然后使用 <code>--diag_error=num</code> 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。
<code>--diag_remark=num</code>	将由 <code>num</code> 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 <code>--display_error_number</code> 选项。然后使用 <code>--diag_remark=num</code> 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。
<code>--diag_suppress=num</code>	抑制由 <code>num</code> 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 <code>--display_error_number</code> 选项。然后使用 <code>--diag_suppress=num</code> 来抑制诊断。您只能抑制任意诊断消息。
<code>--diag_warning=num</code>	将由 <code>num</code> 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 <code>--display_error_number</code> 选项。然后使用 <code>--diag_warning=num</code> 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。
<code>--display_error_number</code>	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（ <code>--diag_suppress</code> 、 <code>--diag_error</code> 、 <code>--diag_remark</code> 和 <code>--diag_warning</code> ）。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 <code>-D</code> ；否则，不存在后缀。请参阅 节 2.7 。
<code>--emit_warnings_as_errors</code>	将所有警告视为错误。此选项不能与 <code>--no_warnings</code> 选项一同使用。 <code>--diag_remark</code> 选项优先于此选项。此选项优先于 <code>--diag_warning</code> 选项。
<code>--issue_remarks</code>	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 <code>--run_linker</code> 或 <code>--z</code> 选项之前）。
--set_error_limit=num	将错误限制设置为 <i>num</i> ，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 <code>.err</code> 的诊断消息信息文件。（链接器不支持 <code>--write_diagnostics_file</code> 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 生成交叉引用列表信息（`--gen_cross_reference` 选项）

`--gen_cross_reference` 选项生成一个交叉引用列表文件，其包含源文件中每个标识符的引用信息。（`--gen_cross_reference` 选项与 `--asm_listing_cross_reference` 是分开的，后者是一个汇编器选项而不是编译器选项。）交叉引用列表文件与扩展名为 `.crl` 的源文件具有相同的名称。

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

<i>sym-id</i>	唯一分配给每个标识符的整数
<i>name</i>	标识符名称
<i>X</i>	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
<i>filename</i>	源文件
<i>line number</i>	源文件中的行号
<i>column number</i>	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) 显示预处理版本的源文件，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 2-31 中列出的标识符之一开头。

表 2-31. 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，该会立即出现在正常源代码行之后。
S	跳过的源代码行 (false #if 子句)
L	源代码位置变化，按以下格式给出： L <i>line number filename key</i>
	其中， <i>line number</i> 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， <i>key</i> 才存在。可能的 <i>key</i> 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 2-32 中定义的诊断标识符。

表 2-32. 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

诊断原始列表信息使用以下格式显示：

```
S filename line number column number diagnostic
```

S	表 2-32 中的标识符之一表示诊断的严重程度
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，每个后续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 5.11.11）和等效的 `always_inline` 属性（节 5.16.2）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 5.11.9）会强制函数内联到带注释的语句中。也就是说，它通常对这些函数没有影响，只对单个语句中的函数调用产生影响。`FORCEINLINE_RECURSIVE pragma` 不仅强制内联在语句中可见的调用，而且还强制内联该语句内联的调用体。
- `--disable_inlining` 选项阻止任何内联。`pragma FUNC_CANNOT_INLINE` 阻止函数被内联。`NOINLINE pragma` 阻止单个语句中的调用被内联。（`NOINLINE` 与 `FORCEINLINE pragma` 相反。）

备注

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

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

`inline` 关键字在所有 C++ 模式中、所有 C 标准的宽松的 ANSI 模式中以及 C99 和 C11 的严格的 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。如果要在严格的 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 6.8.1。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，编译器会将调用保持原样：

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

如果函数具有会给编译器带来困难情形的特性，编译器也不会内联调用：

- 有一个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非注释的 `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。

使用 `FORCEINLINE` pragma 注释的语句中的调用将始终内联，如果它没有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。

换句话说，语句级 pragma 会覆盖函数级 pragma 或属性。如果 `NOINLINE` 和 `FORCEINLINE` 都适用于同一个语句，则将使用第一个出现的语句，其余的将被忽略。

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：

```
cl430 --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。

```

;*****
;      .compiler_opts --mem model:code=flat --mem_model:data=flat --symdebug:none
;      acp430 -@/var/tmp/TI764/AAAv0aGVG
;      .sect      ".text"
;      .align    2
;      .clink
;      .global   main
;-----
;      3 | int main()
;-----
;*****
;* FUNCTION NAME: main                                     *
;* *                                                       *
;* Regs Modified      : SP,SR,r11,r12,r13,r14,r15         *
;* Regs Used          : SP,SR,r11,r12,r13,r14,r15         *
;* Local Frame Size   : 2 Args + 0 Auto + 0 Save = 2 byte *
;*****
main:
;* -----*
;      SUB.W      #2,SP
;-----
;      5 | printf("Hello, world\n");
;-----
;      MOV.W     #0,SP+0(SP)      ; |5|
;      CALL     #printf          ; |5|
;                               ; |5|
;-----
;      7 | return 0;
;-----
;      MOV.W     #0,r12           ; |7|
;      ADD.W     #2,SP           ; |7|
;      RET      ; |7|
;      ; |7|
;*****
;* STRINGS *
;*****
;      .sect      ".const"
;      .align    2
;      $CSSL1: .string "Hello, world",10,0
;*****
;* UNDEFINED EXTERNAL REFERENCES *
;*****
;      .global   printf
    
```

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。

MSP430 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 COFF ABI，请使用 MSP430 v4.4 代码生成工具，并参阅 [SLAU132J](#) 和 [SLAU131J](#) 文档。

EABI 应用程序中的所有代码都必须为 EABI 构建。在将 COFF ABI 系统迁移到 MSP430 EABI 之前，请确保所有库都在 EABI 模式下可用。

更多有关 ABI 的详细信息，请参阅 [节 5.13](#)。

2.14 启用入口挂钩和出口挂钩函数

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

--entry_hook[=<i>name</i>]	启用入口挂钩。若已指定，挂钩函数被称为 <i>name</i> 。否则，默认的入口挂钩函数名称为 <code>__entry_hook</code> 。
--entry_parm={<i>name</i> address none}	指定挂钩函数的参数。 <i>name</i> 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： <code>void hook(const char *name);</code> <i>address</i> 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： <code>void hook(void (*addr)());</code> <i>none</i> 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： <code>void hook(void);</code>
--exit_hook[=<i>name</i>]	启用出口挂钩。若已指定，挂钩函数被称为 <i>name</i> 。否则，默认的出口挂钩函数名称为 <code>__exit_hook</code> 。
--exit_parm={<i>name</i> address none}	指定挂钩函数的参数。 <i>name</i> 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： <code>void hook(const char *name);</code> <i>address</i> 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： <code>void hook(void (*addr)());</code> <i>none</i> 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： <code>void hook(void);</code>

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 5.11.24。

This page intentionally left blank.

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，以及如何调试优化的代码。

3.1 调用优化.....	52
3.2 控制代码大小与速度.....	52
3.3 执行文件级优化 (--opt_level=3 选项)	53
3.4 程序级优化 (--program_level_compile 和 --opt_level=3 选项)	53
3.5 自动内联扩展 (--auto_inline 选项)	55
3.6 链接时优化 (--opt_level=4 选项)	56
3.7 使用反馈制导优化.....	56
3.8 使用配置文件信息分析代码覆盖率.....	60
3.9 访问优化代码中的别名变量.....	62
3.10 在优化代码中谨慎使用 asm 语句.....	62
3.11 将交叉列出功能与优化一起使用.....	62
3.12 调试优化代码.....	63
3.13 正在执行什么类型的优化 ?	63

3.1 调用优化

C/C++ 编译器能够执行各种优化。高级别优化在优化器中执行，目标专用的低级别优化在代码生成器中执行。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。

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 `-On` 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
 - 将变量分配给寄存器
 - 执行循环旋转
 - 消除未使用的代码
 - 简化表达式和语句
 - 扩展对内联声明的函数的调用
- `--opt_level=1` 或 `-O1`

执行所有 `--opt_level=0` (`-O0`) 优化，并且：

- 执行本地复制/常量传播
- 删除未使用的赋值
- 消除局部公用表达式

- `--opt_level=2` 或 `-O2`

执行所有 `--opt_level=1` (`-O1`) 优化，并且：

- 执行循环优化
- 消除全局公用子表达式
- 消除全局未使用的赋值
- 执行循环展开

- `--opt_level=3` 或 `-O3`

执行所有 `--opt_level=2` (`-O2`) 优化，并且：

- 删除所有从未调用过的函数
- 简化返回值从未使用过的函数
- 内联对小函数的调用
- 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

如果使用 `--opt_level=3` (`-O3`)，请参阅[节 3.3](#)和[节 3.4](#)以了解更多信息。

- `--opt_level=4` 或 `-O4`

执行链接时优化。有关详细信息，请参阅[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。

优化由独立优化通道执行。代码生成器执行几个额外的优化，特别是处理器专用优化。无论您是否调用优化器，都会进行上述操作。这些优化始终处于启用状态，但在使用优化器时，这些优化的效率会更高。

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

3.3 执行文件级优化 (`--opt_level=3` 选项)

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

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	<code>--gen_opt_level=n</code>	节 3.3.1
希望编译多个源文件	<code>--program_level_compile</code>	节 3.4

3.3.1 创建优化信息文件 (`--gen_opt_info` 选项)

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

表 3-2. 为 `--gen_opt_info` 选项选择一个级别

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

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。该模块会移动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，因此它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。

为了执行这些优化，`--program_level_compile` 选项需要使用 `--opt_level=3` 或更高版本。

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如声明为静态的一样。请根据表 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它模块中修改的全局变量	<code>--call_assumptions=0</code>
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	<code>--call_assumptions=1</code>
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	<code>--call_assumptions=2</code>
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	<code>--call_assumptions=3</code>

在某些情况下，编译器会将指定的级别恢复为不同的 `--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 <code>--call_assumptions</code> 为...	在以下条件下...	则 <code>--call_assumptions</code> 级别...
未指定	指定了 <code>--opt_level=3</code> 优化级别	默认为 <code>--call_assumptions=2</code>
未指定	编译器在 <code>--opt_level=3</code> 优化级别下发现对外部函数的调用	恢复为 <code>--call_assumptions=0</code>
未指定	未定义 <code>main</code>	恢复为 <code>--call_assumptions=0</code>
<code>--call_assumptions=1</code> 或 <code>--call_assumptions=2</code>	没有将 <code>main</code> 定义为入口点的函数，也没有定义中断函数，也没有由 <code>FUNC_EXT_CALLED</code> pragma 标识的函数	恢复为 <code>--call_assumptions=0</code>
<code>--call_assumptions=1</code> 或 <code>--call_assumptions=2</code>	定义了 <code>main</code> 函数，或定义了中断函数，或者用 <code>FUNC_EXT_CALLED</code> pragma 标识了函数	保留 <code>--call_assumptions=1</code> 或 <code>--call_assumptions=2</code>
<code>--call_assumptions=3</code>	任何条件下	保留 <code>--call_assumptions=3</code>

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 5.11.13) 放在对要保留的函数的任何声明或引用之前。

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

一般而言，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：外部函数不调用 C/C++ 函数，也不修改 C/C++ 变量。

如果仅使用 `--program_level_compile --opt_level=3` 选项进行编译，编译器将从默认优化级别 (`--call_assumptions=2`) 恢复为 `--call_assumptions=0`。编译器使用 `--call_assumptions=0` 是因为编译器假定对在 C/C++ 中有定义的汇编语言函数的调用可以调用其他 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。汇编语言函数不调用 C/C++ 函数，但会修改 C/C++ 变量。

解决方案：尝试以下两种解决方案，然后选择最适合您的代码的一种解决方案：

- 使用 `--program_level_compile --opt_level=3 --call_assumptions=1` 进行编译。
- 将 `volatile` 关键字添加到可能被汇编函数修改的变量中，并使用 `--program_level_compile --opt_level=3 --call_assumptions=2` 进行编译。

- **情况：**您的应用程序由 C/C++ 源代码和汇编源代码组成。汇编函数是调用 C/C++ 函数的中断服务例程；汇编函数调用的 C/C++ 函数永远不会从 C/C++ 调用。这些 C/C++ 函数的作用类似于 `main`：充当 C/C++ 的入口点。

解决方案：将 `volatile` 关键字添加到可能被中断修改的 C/C++ 变量中。然后，可以通过以下方式之一优化代码：

- 通过将 `FUNC_EXT_CALLED pragma` 应用于从汇编语言中断调用的所有入口点函数，然后使用 `--program_level_compile --opt_level=3 --call_assumptions=2` 进行编译，可以实现最佳优化。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`，编译器将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，编译器会自动内联小函数。命令行选项 `--auto_inline=size` 指定大小阈值。任何大于 `size` 阈值的函数都不会自动被内联。可以通过以下方式使用 `--auto_inline=size` 选项：

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

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

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

有关影响内联的命令行选项、`pragma` 和关键字之间的交互信息，请参阅 [节 2.11](#)。

备注

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

备注

内联和代码大小：内联扩展函数会增加代码大小，尤其是内联在多个地方调用的函数。对于仅在少数地方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它要求一次性将所有源文件传递给编译器。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的，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。在链接时优化中，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，则发出错误。

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。对于应用地程序级的选项，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`，会影响广泛的优化。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，链接器不会检查在不同文件中是否已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 pdd430 的工具对生成的 PDAT 文件进行后处理。pdd430 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 <code>run-time-support exit()</code> 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 <code>TI_PROFDATA</code> ，则将数据写入指定的文件中。否则，它使用默认文件名： <code>pprofout.pdat</code> 。可以使用 <code>TI_PROFDATA</code> 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 <code>--gen_profile_info</code> 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 pdd430 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： <code>--use_profile_info==file1[, file2, ..., filen]</code> 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 <code>pprofout.prf</code> 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。

```
cl430 --opt_level=2 --gen_profile_info foo.c --run_linker --output_file=foo.out
--library=lnk.cmd --library=rts430.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。

使用多个数据集运行应用程序后，在 PDAT 文件上运行 pdd430 以创建与 `--use_profile_info` 一起使用的分析信息 (PRF) 文件。

```
pdd430 -e foo.out -o pprofout.prf pprofout.pdat
```

4. 使用分析反馈文件重新编译。

```
cl430 --opt_level=2 --use_profile_info=pprofout.prf foo.c --run_linker  
--output_file=foo.out --library=lnk.cmd --library=rts430.lib
```

3.7.1.5 .ppdata 段

第 1 阶段收集的分析信息存储在 `.ppdata` 段中，且该段必须分配到目标内存中。`.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` 段放置在数据内存中的指令。如果链接命令文件中没有用于分配 `.ppdata` 段的段指令，则链接步骤将 `.ppdata` 段放置在可写内存范围内。

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。由于分析仪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。
- 在 `gen-profile` 之后，源代码被修改了。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 `pdd430` 的工具，该工具用于对分析数据 (PDAT) 文件进行后处理。`pdd430` 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 `pdd430` 的讨论，请参阅节 3.7.1。使用以下语法调用 `pdd430` 工具：

```
pdd430 -e exec.out -o application.prf filename .pdatt
```

-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 <code>exec.out</code> 是应用程序可执行文件的名称。
-o application.prf	指定 <code>application.prf</code> 是格式化的分析反馈文件，在重新编译期间用作 <code>--use_profile_info</code> 的参数。如果未指定输出文件，则默认输出文件名为 <code>pprofout.prf</code> 。
filename .pdatt	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 <code>TI_PROFDATA</code> 将其覆盖。

运行时支持函数和 `pdd430` 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

分析数据解码器要求

您的应用程序必须至少使用 DWARF 调试支持进行编译，才能启用反馈制导优化。当为反馈制导优化进行编译时，pdd430 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 pdd430 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()`

此接口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

<code>--gen_profile_info</code>	将检测添加到已编译的代码中。执行代码时会将分析数据发送到 PDAT 文件。
<code>--use_profile_info=file.prf</code>	使用分析信息进行优化和/或生成代码覆盖率信息。
<code>--analyze=codecov</code>	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 <code>--use_profile_info</code> 一起使用。
<code>--analyze_only</code>	仅生成代码覆盖率信息文件。必须与 <code>--use_profile_info</code> 一起使用。必须同时指定 <code>--analyze=codecov</code> 和 <code>--analyze_only</code> 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

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

API

<code>_TI_start_pprof_collection()</code>	清除要归档的分析计数器
<code>_TI_stop_pprof_collection()</code>	将所有分析计数器写入文件
<code>PPHDNL</code>	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 C I/O 驱动程序。

创建的文件

<code>*.pdat</code>	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
<code>*.prf</code>	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 pdd430 的工具对生成的 PDAT 文件进行后处理。pdd430 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：

```
cl430 --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/msp430/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 `pprof430`，该程序是从编译器 `cl430` 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 `pprof430` 实用程序：

```
cl430 --gen_profile ... file.c
```

```
cl430 --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 `pprof430` 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 `pprof430` 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，`pprof430` 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 将交叉列出功能与优化一起使用

使用 `--optimizer_interlist` 和 `--c_src_interlist` 选项进行优化 (`--opt_level=n` 或 `-On` 选项) 编译时，可以控制交叉列出功能的输出。

- `--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，因为它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您的程序。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，假设以下 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:
;* -----*
SUB.W    #2,SP
;** 5  -----   printf((const unsigned char *)"Hello, world\n");
MOV.W    #CSSL1+0,0(SP)    ; |5|
CALL    #printf            ; |5|
;* 6  -----   return 0;
MOV.W    #0,r12            ; |6|
ADD.W    #2,SP
RET
```

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:
;* -----*
SUB.W    #2,SP
;** 5  -----   printf((const unsigned char *)"Hello, world\n");
;-----*
; 5 | printf ("Hello, world\n");
;-----*
MOV.W    #CSSL1+0,0(SP)    ; |5|
CALL    #printf            ; |5|
;* 6  -----   return 0;
;-----*
; 6 | return 0;
;-----*
MOV.W    #0,r12            ; |6|
ADD.W    #2,SP
RET
```

3.12 调试优化代码

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

默认的优化级别为 `off`。

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

3.13 正在执行什么类型的优化？

MSP430 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 3.13.1
分支优化和控制流简化	节 3.13.3
数据流优化	节 3.13.4
<ul style="list-style-type: none"> • 复制传播 • 通用子表达式消除 • 冗余分配消除 	
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
归纳变量优化和强度降低	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
<hr/>	
MSP430 专用优化	请参阅
尾部合并	节 3.13.12
带常数除数的整数除法	节 3.13.13

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。

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。
- **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。
- **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。

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 $a = b - c + 3$ 。

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](#)。

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 5.16.2](#)。

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。

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。

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 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。

This page intentionally left blank.

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《MSP430 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)	68
4.2 链接器代码优化.....	70
4.3 控制链接过程.....	70

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：

```
cl430 --run_linker {--rom_model | --ram_model} filenames
      [options] [--output_file= name.out] --library= library [lnk.cmd]
```

cl430 --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 <code>cl430 --run_linker</code> 而不列出要在命令行编译的任何 C/C++ 文件时， <i>必须</i> 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 4.3.7。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： <div style="border: 1px solid black; padding: 2px; margin-top: 5px;">warning: no suitable entry-point found; setting to 0</div>
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 <code>.c.obj</code> （用于 C 源文件）和 <code>.cpp.obj</code> （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 <code>a.out</code> 。
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。（《MSP430 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 <code>-l</code> 。
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《MSP430 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：

```
cl430 --run_linker --rom_model prog1 prog2 prog3 --output_file=prog.out
      --library=rts430.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：

```
cl430 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：

```
cl430 prog1.c prog2.c prog3.c --run_linker --rom_model --output_file=prog.out --library=rts430.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 **cl430 --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 4.3.7。

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. MSP430_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 MSP430_C_OPTION 环境变量中指定了 **--run_linker** 选项，并希望有选择地禁用命令行上的 **--compile_only** 选项，那么 **--compile_only** 选项特别有用。

4.2 链接器代码优化

4.2.1 条件链接

使用 ELF 条件链接时，除 `.reset` 段外，所有的段都可以在条件链接过程中予以删除。如果段被引用或用 `.retain` 指令进行了标记，则该段不会被删除。

程序中从未被任何其他段引用的段都将自动从程序中删除，除非其标记为 `.retain` 或者是 `.reset` 段。当执行部分链接时或重定位信息与链接的输出一起保存时，将禁用条件链接。

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《MSP430 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 4.3.7](#)。

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

```
cl430 --issue_remarks main.c --run_linker --rom_model
<Linking>
```

```
remark: linking in "libc.a"
remark: linking in "rts430.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 `MSP430_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。

```
cl430 --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 **引导程序** 将所有 C/C++ 程序链接起来，以便将初始化 C/C++ 环境并开始执行程序。引导程序负责执行以下任务：

1. 设置堆栈
2. 处理 `.cinit` 运行时初始化表以自动初始化全局变量（使用 `--rom_model` 选项时）
3. 调用 C++ 的所有全局构造函数 (`.init_array`)
4. 调用 `main()` 函数
5. 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 4.3.7](#)。

4.3.3 通过中断向量进行初始化

如果 C/C++ 程序在 RESET 处开始运行，必须设置 RESET 向量以分支到合适的引导程序，例如 `_c_int00`。此外，还必须确保项目中包含中断向量，通常是使用 `--undef_sym` 链接器选项将中断向量开头的符号作为根链接器对象。运行时支持库中的 `boot.c.obj` 对象提供了一个名为 `.reset` 的段，其中包含对 `_c_int00` 的引用，该引用很适合放置在链接器命令文件中的 RESET 向量中。

4.3.4 FRAM 内存保护单元的初始化

链接器支持 FRAM 内存保护单元 (MPU) 的初始化。链接器使用引导程序，该程序根据某些符号的定义执行 MPU 初始化。默认情况下用于不同器件的 TI 链接器命令文件定义了必要的符号，使 MPU 初始化自动进行。代码段和数据段会自动地获得正确的访问权限。如果要手动调整 MPU 的初始化方式，可以修改链接器命令文件中的 `__mpuseg` 和 `__mpusam` 定义。MPU 专用引导程序是在定义这两个符号时使用，其根据这些值设置 MPUSEG 和 MPUSAM 寄存器的值。如果不想初始化 MPU，可以从链接器命令文件中删除这些定义。

4.3.5 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。

4.3.6 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 6.9.3.6 讨论了 EABI 模式的全局构造函数表的格式。

4.3.7 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 6.9.3.4 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 6.9.3.3）。
- 在加载时初始化全局变量。使用 `--ram_model` 链接器选项（请参阅节 6.9.3.5）。

如果不编译任何 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 6.9.3.3 和节 6.9.3.5。

备注

引导加载器

加载器不包含在 C/C++ 编译器工具中。有关启动加载的更多信息，请参阅《MSP430 汇编语言工具用户指南》中“程序加载和运行”一章。

4.3.8 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，块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 6.1.4。

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《MSP430 汇编语言工具用户指南》）
.mspabi.exidx	异常处理的索引表；只读（请参阅 <code>--exceptions</code> 选项）。
.mspabi.exstab	异常处理的展开指令；只读（请参阅 <code>--exceptions</code> 选项）。
.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。
.name.load	段名称的压缩图像；只读（有关复制表的信息，请参阅《MSP430 汇编语言工具用户指南》。）
.ovly	复制除引导时间 (.binit) 复制表以外的表。只读数据。
.rodata	具有常量说明符的全局和静态变量。

表 4-1. 由编译器创建的初始化段 (continued)

名称	内容
.TI.crctab	生成的 CRC 校验表。只读数据。
.TI.noinit	pragma NOINIT 使未初始化变量放置在 .TI.noinit 段中。默认链接器命令文件使用 .bss 放置该段。有关 NOINIT 和 PERSISTENT pragma 的详细信息，请参阅节 5.11.22。
.TI.persistent	pragma PERSISTENT 使初始化变量放置在 .TI.persistent 段中。默认链接器命令文件使用 .data 放置该段。

表 4-2. 由编译器创建的未初始化段

名称	内容
.args	链接器创建的段，用于将参数从加载器的命令行传递到程序
.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	函数调用帧栈
.system	用于动态内存分配 (malloc 等) 的内存池 (堆)

链接程序时，必须指定在段分配内存中的位置。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 指令来分配段。有关将段分配到内存中的更多信息，请参阅《MSP430 汇编语言工具用户指南》。

4.3.9 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 `Ink32.cmd`，且命令文件列出了几个链接选项：

<code>--rom_model</code>	通知链接器在运行时使用自动初始化功能
<code>--stack_size</code>	通知链接器将 C 栈大小设置为 0x140 字节
<code>--heap_size</code>	通知链接器将堆大小设置为 0x120 字节
<code>--library</code>	通知链接器使用存档库文件 <code>rts430.lib</code>

要链接该程序，请输入：

```
cl430 --run_linker object_file(s) --output_file= file --map_file= file Ink.cmd
```

链接器命令文件

```
--rom_model
--stack_size=0x0140
--heap_size=0x120
--library=rts430.lib
/*****
/* SPECIFY THE SYSTEM MEMORY MAP
*****/
MEMORY
{
    SFR(R)          : origin = 0x0000, length = 0x0010
    PERIPHERALS_8BIT : origin = 0x0010, length = 0x00F0
    PERIPHERALS_16BIT: origin = 0x0100, length = 0x0100
    RAM(RW)         : origin = 0x0200, length = 0x0800
    INFOA           : origin = 0x1080, length = 0x0080
    INFOB           : origin = 0x1000, length = 0x0080
    FLASH           : origin = 0x1100, length = 0xEEE0
    VECTORS(R)      : origin = 0xFFE0, length = 0x001E
    RESET           : origin = 0xFFFF, length = 0x0002
}
/*****
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY
*****/
SECTIONS
{
    .bss          : {} > RAM           /* UNINITIALIZED GLOBAL, STATIC */
    .system      : {} > RAM           /* DYNAMIC MEMORY ALLOCATION AREA */
    .stack       : {} > RAM           /* SOFTWARE SYSTEM STACK */
    .cio         : {} > RAM           /* C I/O BUFFER */
    .text        : {} > FLASH         /* PROGRAM CODE */
    .data        : {} > FLASH         /* INITIALIZED GLOBAL, STATIC */
    .const       : {} > FLASH         /* CONSTANT DATA */
    .args        : {} > FLASH         /* PROGRAM ARGUMENTS */
    .cinit       : {} > FLASH         /* GLOBAL INITIALIZATION DATA */
    .init_array  : {} > FLASH         /* EABI C++ GLOBAL CONSTRUCTOR TABLE */
    .mspabi.exidx : {} > FLASH         /* EABI TDEH METADATA */
    .mspabi.extab : {} > FLASH         /* EABI TDEH METADATA */
    .intvecs     : {} > VECTORS      /* INTERRUPT VECTORS */
    .reset       : > RESET
}
```

受 MSP430 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 MSP430 支持的 C++ 语言由 ANSI/ISO/IEC 14882:2014 标准定义，但有一些例外。

5.1 MSP430 C 的特征.....	76
5.2 MSP430 C++ 的特征.....	80
5.3 使用 ULP Advisor.....	81
5.4 关于硬件配置的建议.....	81
5.5 数据类型.....	82
5.6 文件编码和字符集.....	84
5.7 关键字.....	85
5.8 C++ 异常处理.....	88
5.9 寄存器变量和参数.....	88
5.10 <code>__asm</code> 语句.....	89
5.11 <code>pragma</code> 指令.....	90
5.12 <code>_Pragma</code> 运算符.....	110
5.13 应用程序二进制接口.....	111
5.14 目标文件符号命名规则 (链接名).....	111
5.15 更改 ANSI/ISO C/C++ 语言模式.....	111
5.16 GNU 和 Clang 语言扩展.....	114
5.17 编译器限制.....	120

5.1 MSP430 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。名称“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 5.16）。

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 5.15。

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。原子操作是通过禁用运算中的中断来实现的。

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

不受支持的 C 库功能包括：

- 运行时库对宽字符的支持很少。类型 `wchar_t` 实现为 `unsigned int`（16 位）。宽字符集相当于 `char` 类型的值集。该库包含头文件 `<wchar.h>` 和 `<wctype.h>`，但不包含标准中指定的所有函数。请参阅节 5.6，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 5.15。

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 编译器可为独立环境提供正式的支持。C 标准不需要一个独立的环境来提供每个 C 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。

随后列表中的节编号对应于 C99 标准附录 J 中的节编号。每项末尾括号中的数字是 C99 标准中讨论相应主题的各节编号。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假定为诊断信息。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。相关详细信息，请参阅节 5.6。(5.1.1.2)
- 程序启动时调用的函数名称为“main”(5.1.2.1)
- 程序终止不会影响环境；没有办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 的替代定义。在严格 ANSI 模式下，替代定义会被拒绝。(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 调用并将结果传递回程序。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。相关详细信息，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。请参阅节 5.5，了解有关数据类型的详细信息。(3.6)
- 执行字符集与基本执行字符集相同，为纯 ASCII，(5.2.1)
- 为标准字母转义序列生成的值如下：(5.2.2)

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

- char 对象（其中存储了除基本执行字符集成员之外的任何其他字符）的值是该字符的 ASCII 值。(6.2.5)
- Plain char 等同于 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对一的。编译器在注释中接受多字节字符。相关详细信息，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

- 未提供任何扩展整数类型。(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)

- 当一个整数转换为一个不能代表该值的有符号整数类型时，该值会被截断（不发出信号），丢弃不能存储在目标类型中的位；最低位不会被修改。(6.3.1.3)
- 有符号整数值右移执行算术（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按相反方向转换时，指针被视为相同大小的无符号整数，并且适用正常的整数转换规则。一些指针的大小与任何整数类型的大小不同，但转换过程就像这样的类型确实存在一样，其规则由正常整数转换隐含。
- 在将指针转换为整数或按相反方向转换时，如果目标的按位表示可以容纳源代码的按位表示中的所有位，则这些位会被完全复制。(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，寄存器存储类说明符将被忽略。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。编译器保存将任何寄存器存储类对象放置在寄存器以外位置的权利。(6.7.1)
- 除非使用优化器，否则内联函数说明符会被忽略。有关内联的其他限制，请参阅节 2.11.2。(6.7.4)

J.3.9 结构体、联合体、枚举和位字段

- “plain” 整数位字段会被视为有符号整数位字段。(6.7.2, 6.7.2.1)
- 除了 `_Bool`、`signed int` 和 `unsigned int`，编译器还允许将 `char`、`signed char`、`unsigned char`、`signed short`、`unsigned short`、`signed long`、`unsigned long`、`signed long long`、`unsigned long long` 和枚举类型作为位字段类型。(6.7.2.1)
- 位字段不能跨越存储单元边界。(6.7.2.1)
- 位字段在一个单元内按字节序分配。(6.7.2.1)
- 结构体的非位字段成员按照节 6.2 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大易失性访问。用户有责任确保访问大小适合仅允许访问特定宽度的设备。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-修改-写入表达式很重要；对于没有相应的读取-修改-写入指令的体系结构，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体系结构，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，但它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)
- 用于控制条件包含的常量表达式中字符常量的值与执行字符集中相同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 “” 分隔头文件。请参阅节 2.5.2。(6.10.2)(6.10.2)
- `#include` 处理没有任意嵌套限制。(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 5.15.1 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)
- `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](#)。(6.5.3.4)

5.2 MSP430 C++ 的特征

根据 ANSI/ISO/IEC 14882:2014 标准 (C++14) 中的定义，MSP430 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。
- 如果两个类不相关，则 `reinterpret_cast` 类型不允许将一个类的 `pointer-to-member` 强制转换为另一个类的 `pointer-to-member`。
- 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 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 5.11.28。

`--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}")
```

5.4 关于硬件配置的建议

`--advice:hw_config` 选项允许您启用、禁用或配置待执行的与设备配置设置相关的检查。例如，检查 `FRAM waitstate` 配置。当其中一项检查发出警告时，单击 `Code Composer Studio` 中提供的链接能够找到有关解决设备配置问题的信息。

语法为：

```
--advice:hw_config={all|none|rulespec}
```

rulespec 参数是待启用的以逗号分隔的一系列规则编号。

5.5 数据类型

表 5-1 列出了 MSP430 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 中介绍了数据类型的存储和对齐。

表 5-1. MSP430 C/C++ 数据类型

类型	大小	对齐	表示	范围	
				最小值	最大值
signed char	8 位	8	二进制	-128	127
char	8 位	8	ASCII	0 或 -128 ⁽¹⁾	255 或 127 ⁽¹⁾
unsigned char	8 位	8	二进制	0	255
bool (C99)	8 位	8	二进制	0 (false)	1 (true)
_Bool (C99)	8 位	8	二进制	0 (false)	1 (true)
bool (C++)	8 位	8	二进制	0 (false)	1 (true)
short、signed short	16 位	16	二进制	-32 768	32 767
unsigned short	16 位	16	二进制	0	65 535
int、signed int	16 位	16	二进制	-32 768	32 767
unsigned int	16 位	16	二进制	0	65 535
long、signed long	32 位	16	二进制	-2 147 483 648	2 147 483 647
unsigned long	32 位	16	二进制	0	4 294 967 295
long long、signed long long	64 位	16	二进制	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 位	16	二进制	0	18 446 744 073 709 551 615
enum	不尽相同 ⁽²⁾	16	二进制	不尽相同	不尽相同
float	32 位	16	IEEE 32 位	1.175 494e-38 ⁽³⁾	3.40 282 346e+38
double	64 位	16	IEEE 64 位	2.22 507 385e-308 ⁽³⁾	1.79 769 313e+308
long double	64 位	16	IEEE 64 位	2.22 507 385e-308 ⁽³⁾	1.79 769 313e+308
函数和数据指针	不尽相同 (请参阅表 5-2)	16			

- (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) 有关枚举类型大小的详细信息，请参阅节 5.5.1。
- (3) 数字是最低精度。

默认情况下，char 类型是无符号型。这与“signed char”和“unsigned char”类型不同，后者指定了它们的符号行为。您可以使用 --plain_char=signed 编译器选项更改“char”类型的默认设置。

有符号类型的负值用 2 的补码表示。

C、C99 和 C++ 中的其他类型被定义为标准类型的同义词：

```
typedef unsigned int    wchar_t;
typedef unsigned int    wint_t;
```

MSP 设备支持多种数据和代码存储器模式。代码和数据模式会影响大小、对齐，以及用于函数指针、数据指针、size_t 类型、和 ptrdiff_t 类型的存储空间。始终在大小为 2 位幂的容器中存储大小不是 2 的幂的指针。也就是说，20 位类型存储在 32 位中。

表 5-2. MSP430 指针的数据大小

代码或数据模式	类型	大小	存储	对齐
小型代码模式	函数指针	16	16	16
大型代码模式	函数指针	20	32	16
小型数据模式	数据指针	16	16	16
小型数据模式	size_t	16	16	16
小型数据模式	ptrdiff_t	16	16	16
受限数据模式	数据指针	20	32	16
受限数据模式	size_t	16	16	16
受限数据模式	ptrdiff_t	16	16	16
大型数据模式 (1)	数据指针	20	32	16
大型数据模式	size_t	32	32	16
大型数据模式	ptrdiff_t	32	32	16

(1) MSP430X 大型数据模式由 `--silicon_version=msp430 --data_model=large` 指定

5.5.1 枚举类型大小

在以下声明中，`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是针对 C 还是 C++ 进行编译。C++11 允许您为枚举类型指定具体的类型；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，所有采用严格符合 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，您甚至可以在 C 模式下使用大于“int”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，控制挑选枚举类型的策略。如果您使用 `--enum_type=packed` 选项（默认），则编译器会使用可用作枚举类型的最小类型。如果您使用 `--enum_type=unpacked` 选项（默认），则编译器会使用不小于“int”的最小类型。您可以使用“packed”或“unpacked”属性来针对单个枚举类型设置策略；这会覆盖命令行选项的值。

例如，即使在命令行中指定了 `--enum_type=unpacked`，这个枚举类型也将会是无符号字符型。

```
enum { a, b, c } __attribute__((packed));
```

即使在命令行中指定了 `--enum_type=packed`，这个枚举类型也将会是无符号整型。

```
enum { a, b, c } __attribute__((unpacked));
```

您可以在头文件中使用枚举类型的 `packed/unpacked` 属性来确保枚举类型的大小始终相同，即使在命令行中更改了 `--enum_type` 选项也是如此。

对于枚举类型，编译器会选择以下列表中足够大且符号正确、能够表示所有枚举常量值的第一个类型。不使用“short”类型，因为其与“int”类型大小相同。

- unsigned char (仅在使用 `--enum_type=packed` 时)
- signed char (仅在使用 `--enum_type=packed` 时)
- unsigned int
- signed int

- unsigned long
- signed long
- unsigned long long
- signed long long

例如，默认情况 (`--enum_type=unpacked`) 下，下面这个枚举类型将会以 “unsigned int” 作为其基础类型：

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

但下面这个将会以 “signed int” 作为其基础类型：

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

而下面这个将会以 “signed long” 作为其基础类型：

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

对于 C++，枚举常量全都具有与枚举类型相同的类型。

对于 C，则会根据它们的值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。在所有其他上下文中 (包括字符串常量、标识符、汇编文件和链接器命令文件) 中，仅支持 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

MSP430 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 5.11.19 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。

备注

中断服务例程 (ISR) 警告：链接器会针对任何未关联中断服务例程的器件专有中断发出警告。不过，默认向量处理程序现在由运行时支持 (RTS) 库提供，因此如果与提供的 `RTS` 库链接，那么应该不会看到此错误。

5.7.3 restrict 关键字

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

“`restrict`”关键字是一个 `C99` 关键字，在严格的 `ANSI C89` 模式下不被接受。如果必须使用严格的 `ANSI C89` 模式，请使用 “`__restrict`” 关键字。请参阅节 5.15。

以下示例使用 `restrict` 关键字来告诉编译器，永远不会使用指向存储器中重叠对象的指针 `a` 和 `b` 来调用函数 `func1`。您承诺通过 `a` 和 `b` 进行访问永远不会发生冲突；因此，通过一个指针进行的写入操作不能影响通过任何其他指针进行的读取操作。1999 版的 `ANSI/ISO C` 标准中描述了 `restrict` 关键字的精确语义。

```
void func1(int * restrict a, int * restrict b)
{
    /* func1's code here */
}
```

以下示例在将数组传递给函数时使用 `restrict` 关键字。在这里，数组 `c` 和 `d` 不得重叠，`c` 和 `d` 也不得指向同一数组。

```
void func2(int c[restrict], int d[restrict])
{
    int i;
    for(i = 0; i < 64; i++)
    {
        c[i] += d[i];
        d[i] += 1;
    }
}
```

5.7.4 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` 模式中支持 `__volatile` 关键字。

`volatile` 关键字向编译器指示该变量的访问方式存在一些要求，要求编译器不要对涉及该变量的表达式使用过于巧妙的优化。例如，外部程序、中断、其他线程或外围设备也可能会访问该变量。

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不明白的某些方面比较特殊，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` 读取和写入的数量将会与 `C/C++` 代码中所示的完全相同，不多不少而且顺序也完全相同。

如果任何变量可能由明显程序控制流程外部的东西（例如中断服务例程）进行修改，则必须声明为 **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。这是 **volatile** 关键字的主要用途。在以下示例中，循环旨在等待位置作为 **0xFF** 被读取：

```
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、一些其他存储器位置被访问时或者出现某些信号时发生改变。

在会调用 **setjmp** 的函数中，如果局部变量的值需要在发生 **longjmp** 时保持有效，则对于局部变量也必须使用 **volatile**。

```
#include <stdlib.h>
jmp_buf context;
void function()
{
    volatile int x = 3;
    switch(setjmp(context))
    {
        case 0: setup(); break;
        default:
        {
            /* 我们只有在 longjmp 发生时才到达这里。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 的实现需要通过将信息移动到 `const` 数据段来减少代码大小开销。

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。

有关寄存器惯例的更多信息，请参阅节 6.3。

5.10 __asm 语句

C/C++ 编译器可以将汇编语言指令直接嵌入到编译器的汇编语言输出中。此功能是通过 `__asm` 关键字实现的对 C/C++ 语言的扩展。`__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”。它可以在宽松 C 和 C++ 模式下使用。

使用 `__asm` 在语法上是作为对名为 `__asm` 的函数的调用执行的，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通过用 `__asm` 语句被写为嵌入式汇编函数的函数。请参阅节 5.16.2。

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《MSP430 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

小心操作，不要使用 `__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造成很多麻烦。

在对 `__asm` 语句进行优化时要特别小心。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。

- `BIS_IE1_INTERRUPT` (请参阅节 5.11.1)
- `CALLS` (请参阅节 5.11.2)
- `CHECK_ULP` (请参阅节 5.11.3)
- `CODE_ALIGN` (请参阅节 5.11.4)
- `CODE_SECTION` (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)
- `FORCEINLINE` (请参阅节 5.11.9)
- `FORCEINLINE_RECURSIVE` (请参阅节 5.11.10)
- `FUNC_ALWAYS_INLINE` (请参阅节 5.11.11)
- `FUNC_CANNOT_INLINE` (请参阅节 5.11.12)
- `FUNC_EXT_CALLED` (请参阅节 5.11.13)
- `FUNC_IS_PURE` (请参阅节 5.11.14)
- `FUNC_NEVER_RETURNS` (请参阅节 5.11.15)
- `FUNC_NO_GLOBAL_ASG` (请参阅节 5.11.16)
- `FUNC_NO_IND_ASG` (请参阅节 5.11.17)
- `FUNCTION_OPTIONS` (请参阅节 5.11.18)
- `INTERRUPT` (请参阅节 5.11.19)
- `LOCATION` (请参阅节 5.11.20)
- `MUST_ITERATE` (请参阅节 5.11.21)
- `NOINIT` (请参阅节 5.11.22)
- `NOINLINE` (请参阅节 5.11.23)
- `NO_HOOKS` (请参阅节 5.11.24)
- `once` (请参阅节 5.11.25)
- `pack` (请参阅节 5.11.26)
- `PERSISTENT` (请参阅节 5.11.22)
- `PROB_ITERATE` (请参阅节 5.11.27)
- `RESET_ULP` (请参阅节 5.11.28)
- `RETAIN` (请参阅节 5.11.29)
- `SET_CODE_SECTION` (请参阅节 5.11.30)
- `SET_DATA_SECTION` (请参阅节 5.11.30)
- `UNROLL` (请参阅节 5.11.31)
- `vector` (请参阅节 5.11.32)
- `WEAK` (请参阅节 5.11.33)

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 BIS_IE1_INTERRUPT

`BIS_IE1_INTERRUPT` pragma 将已命名函数视为中断例程。此外，编译器还在函数退出时在 IE1 特殊函数寄存器上生成 BIS 操作。掩码值必须是 8 位常量文字，在 RETI 指令之前，与 IE1 SFR 进行逻辑或运算。编译器假定 IE1 SFR 映射至地址 0x0000。

C 中 pragma 的语法为：

```
#pragma BIS_IE1_INTERRUPT ( func , mask )
```

C++ 中 pragma 的语法为：

```
#pragma BIS_IE1_INTERRUPT ( mask )
```

在 C 中，参数 `func` 是中断函数的名称。在 C++ 中，`pragma` 应用于所声明的下一个函数。

5.11.2 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《MSP430 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 5.16.2）具有与 `CALLS` pragma 相同的效果，语法如下：

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

5.11.3 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 5.3 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 5.11.28。

5.11.4 CODE_ALIGN Pragma

`CODE_ALIGN` pragma 沿指定的对齐方式对齐 `func`。对齐 `constant` 必须是 2 的幂。如果要在某个边界上启动函数，`CODE_ALIGN` pragma 会非常有用。

CODE_ALIGN pragma 具有与使用 GCC 样式 aligned 函数属性相同的效果。请参阅节 5.16.2。

C 中 pragma 的语法为：

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

C++ 中 pragma 的语法为：

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

5.11.5 CODE_SECTION Pragma

CODE_SECTION pragma 在一个名为 *section name* 的段中，为 C 中的 *symbol* 或在 C++ 中声明的下一个 symbol 分配空间。如果要将代码对象链接到与 .text 段分开的区域，CODE_SECTION pragma 会非常有用。CODE_SECTION pragma 具有与使用 GCC 样式 section 函数属性相同的效果。请参阅节 5.16.2。

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(funcA, "codeA")
int funcA(int a)
{
    int i;
    return (i = a);
}
```

以下示例 C 代码将生成下列汇编代码：

```

    .sect "codeA"
    .align 2
    .global funcA
;*****
;* FUNCTION NAME: funcA
;*
;* Regs Modified      : SP,SR,r12
;* Regs Used          : SP,SR,r12
;* Local Frame Size   : 0 Args + 4 Auto + 0 Save = 4 byte
;*****
funcA:
;* -----*
    SUB.W    #4,SP
    MOV.W    r12,0(SP)          ; |4|
    MOV.W    0(SP),2(SP)       ; |6|
    MOV.W    2(SP),r12         ; |6|
    ADD.W    #4,SP
    RET
    
```

在 C++ 中使用 `CODE_SECTION` Pragma

```

#pragma CODE_SECTION("codeB")
int i_arg(int x) { return 1; }
int f_arg(float x) { return 2; }
    
```

以下示例 C++ 代码将生成下列汇编代码：

```

    .sect "codeB"
    .align 2
    .global i_arg_Fi
;*****
;* FUNCTION NAME: i_arg(int)
;*
;* Regs Modified      : SP,SR,r12
;* Regs Used          : SP,SR,r12
;* Local Frame Size   : 0 Args + 2 Auto + 0 Save = 2 byte
;*****
i_arg_Fi:
;* -----*
    SUB.W    #2,SP
    MOV.W    r12,0(SP)          ; |2|
    MOV.W    #1,r12            ; |2|
    ADD.W    #2,SP
    RET
    .sect ".text"
    .align 2
    .global f_arg_Ff
;*****
;* FUNCTION NAME: f_arg(float)
;*
;* Regs Modified      : SP,SR,r12
;* Regs Used          : SP,SR,r12,r13
;* Local Frame Size   : 0 Args + 4 Auto + 0 Save = 4 byte
;*****
f_arg_Ff:
;* -----*
    SUB.W    #4,SP
    MOV.W    r12,0(SP)          ; |3|
    MOV.W    r13,2(SP)         ; |3|
    MOV.W    #2,r12            ; |3|
    ADD.W    #4,SP
    RET
    
```

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 5.16.4。

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 `section` 变量属性具有相同的效果。请参阅节 5.16.4。

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,512,2
.global bufferB
bufferB: .usect "my_sect",512,2
```

5.11.8 诊断消息 Pragma

以下 pragma 可用于控制诊断消息，方法与相应的命令行选项相同：

Pragma	选项	说明
diag_suppress <i>num</i>	-pds= <i>num</i> [, <i>num</i> ₂ , <i>num</i> ₃ ...]	抑制诊断 <i>num</i>
diag_remark <i>num</i>	-pdsr= <i>num</i> [, <i>num</i> ₂ , <i>num</i> ₃ ...]	将诊断 <i>num</i> 视为备注
diag_warning <i>num</i>	-pds= <i>num</i> [, <i>num</i> ₂ , <i>num</i> ₃ ...]	将诊断 <i>num</i> 视为警告
diag_error <i>num</i>	-pdse= <i>num</i> [, <i>num</i> ₂ , <i>num</i> ₃ ...]	将诊断 <i>num</i> 视为错误
diag_default <i>num</i>	不适用	使用诊断的默认严重性
diag_push	不适用	按下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[, num2, num3...]
```

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 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。

5.11.10 FORCEINLINE_RECURSIVE Pragma

FORCEINLINE_RECURSIVE 可以放置在语句前，以强制在该语句中进行的任何函数调用与从这些函数进行的任何调用一起内联。也就是说，在语句中不可见但作为语句结果被调用的调用将被内联。

此 pragma 在 C/C++ 中的语法为：

```
#pragma FORCEINLINE_RECURSIVE
```

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。

5.11.11 FUNC_ALWAYS_INLINE Pragma

FUNC_ALWAYS_INLINE pragma 指示编译器始终内联命名函数。

编译器仅在内联函数是合法的情况下内联函数。如果使用 `--opt_level=off` 选项来调用编译器，则绝不会内联函数。即使未使用 `inline` 关键字来声明函数，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。在 C++ 中，pragma 应用于所声明的下一个函数。

此 pragma 可用于在链接时强制进行跨 C 文件内联。

FUNC_ALWAYS_INLINE pragma 具有与使用 GCC 样式 `always_inline` 函数属性相同的效果。请参阅节 5.16.2。

C 中 pragma 的语法为：

```
#pragma FUNC_ALWAYS_INLINE ( func )
```

C++ 中 pragma 的语法为：

```
#pragma FUNC_ALWAYS_INLINE
```

以下示例使用此 pragma：

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

备注

使用 FUNC_ALWAYS_INLINE Pragma 时要小心

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

5.11.12 FUNC_CANNOT_INLINE Pragma

FUNC_CANNOT_INLINE pragma 指示编译器命名函数不能内联展开。使用此 pragma 命名的任何函数都会覆盖您以任何其他方式指定的任何内联，例如使用内联关键字。自动内联也会被此 pragma 覆盖；请参阅节 2.11。

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

FUNC_CANNOT_INLINE pragma 具有与使用 GCC 样式 `noinline` 函数属性相同的效果。请参阅节 5.16.2。

C 中 pragma 的语法为：

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

C++ 中 pragma 的语法为：

```
#pragma FUNC_CANNOT_INLINE
```

5.11.13 FUNC_EXT_CALLED Pragma

使用 `--program_level_compile` 选项时，编译器使用程序级优化。使用这种类型的优化时，编译器将删除 `main()` 未直接或间接调用的任何函数。您的 C/C++ 函数可能而不是通过 `main()` 调用。

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

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。

5.11.14 FUNC_IS_PURE Pragma

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

- 如果不需要函数的值，则删除对该函数的调用
- 删除重复的函数

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

C 中 pragma 的语法为：

```
#pragma FUNC_IS_PURE ( func )
```

C++ 中 pragma 的语法为：

```
#pragma FUNC_IS_PURE
```

5.11.15 FUNC_NEVER_RETURNS Pragma

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

此 pragma 必须出现在针对要保留的函数进行的任何声明或引用之前。在 C 语言中，参数 *func* 是不返回的函数的名称。在 C++ 中，pragma 应用于所声明的下一个函数。

C 中 pragma 的语法为：

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

C++ 中 pragma 的语法为：

```
#pragma FUNC_NEVER_RETURNS
```

5.11.16 FUNC_NO_GLOBAL_ASG Pragma

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

此 pragma 必须出现在要保留的函数的任何声明或引用之前。在 C 语言中，参数 *func* 是不进行赋值的函数的名称。在 C++ 中，pragma 应用于所声明的下一个函数。

C 中 pragma 的语法为：

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

C++ 中 pragma 的语法为：

```
#pragma FUNC_NO_GLOBAL_ASG
```

5.11.17 FUNC_NO_IND_ASG Pragma

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

此 pragma 必须出现在针对要保留的函数进行的任何声明或引用之前。在 C 语言中，参数 *func* 是不进行赋值的函数的名称。在 C++ 中，pragma 应用于所声明的下一个函数。

C 中 pragma 的语法为：

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

C++ 中 pragma 的语法为：

```
#pragma FUNC_NO_IND_ASG
```

5.11.18 FUNCTION_OPTIONS Pragma

FUNCTION_OPTIONS pragma 使您能够在 C 或 C++ 文件中编译特定的函数，并使用附加的命令行编译器选项。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 不能用于完全禁用用于编译函数的优化器；可以指定的最低优化级别是 --opt_level=0。

5.11.19 INTERRUPT Pragma

借助 `INTERRUPT pragma`，您可以使用 C 代码来直接处理中断。在 C 语言中，参数 *func* 是函数的名称。在 C++ 中，`pragma` 应用于所声明的下一个函数。

C 中 `pragma` 的语法为：

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

C++ 中 `pragma` 的语法为：

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

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

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

该中断函数的返回地址将保存在堆栈中。

借助该中断属性语法，您可以选择指定数字参数来指示为 `vector pragma` 指定的向量位置。例如：

```
#define TIMER_A0 20  
#define TIMER_A1 10  
__attribute__((interrupt(TIMER_A0))) void foo1() {  
    int i;  
    for (i=1; i < 1000; i++)  
        global_ptr[i] = i;  
}
```

```
#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。

备注

中断服务例程 (ISR) 警告

链接器会针对任何未关联中断服务例程的器件专有中断发出警告。不过，默认向量处理程序由运行时支持 (RTS) 库提供，因此如果与提供的 RTS 库链接，那么应该不会看到此错误。

5.11.20 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 的位置属性来实现。`LOCATION pragma` 与使用 GCC 的 `location` 函数属性具有相同的效果。请参阅节 5.16.2。

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
```

该 `pragma` 在 IAR 中的语法为：

```
#pragma location= address
int x
```

GCC 相应属性 (请参阅节 5.16.4) 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 5.11.22。

5.11.21 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` 和 `PROB_ITERATE` 等其他 `pragma`。

5.11.21.1 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。不存在 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。行程计数是指循环的迭代次数。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) /* 请注意 max 的空白字段 */
```

有时必须提供 *min* 和 *multiple*，编译器才能执行展开。尤其是在编译器无法轻松确定循环要执行的迭代次数（即循环具有复杂的退出条件）的情况下。

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*。

以下示例使用 `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];
    }
    ...
}
```

5.11.2.1.2 使用 `MUST_ITERATE` 扩展编译器对循环的了解

通过使用 `MUST_ITERATE`，可以保证循环执行特定次数。以下示例会告知编译器，循环保证可以精确地运行 10 次：

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

在此示例中，编译器尝试生成循环（即使没有 `pragma`）。但如果未为循环指定 `MUST_ITERATE`（例如本例），编译器会生成代码绕过循环，因此可能出现 0 次迭代。利用 `pragma` 规范，编译器可以知道循环至少会迭代一次，可消除循环旁路代码。

`MUST_ITERATE` 可用于指定循环计数的范围，以及循环计数的系数。以下示例会告知编译器，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++) { ...
```

对于具有复杂条件的循环，您应考虑使用 `MUST_ITERATE`。在以下示例中，编译器必须生成一个除法函数调用，以便在运行时确定执行的迭代次数。

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

编译器不会执行以上操作。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) { ...
```

5.11.22 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 5.16.4。

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** 只能与不会被初始化的变量搭配使用。它用于防止在复位时将此类变量设置为 0。它可以与 **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，而不会出现意外写入。
- **PERSISTENT pragma** 只能与会被静态初始化的变量搭配使用。它用于防止在复位时将此类变量初始化。**Persistent** 变量禁用启动初始化功能；它们会在代码加载时被赋予一个初始值，但再也不会进行初始化。

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，从而防止出现意外写入。一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，则可以定义会将初始值 0 载入 RAM 的 persistent 变量。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它初始化为 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;
```

5.11.23 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。它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
```

如需了解影响内联的命令行选项、**pragma** 和关键字之间的交互，请参阅 [节 2.11](#)。

5.11.24 NO_HOOKS Pragma

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

C 中 **pragma** 的语法为：

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

C++ 中 **pragma** 的语法为：

```
#pragma NO_HOOKS
```

有关入口和出口钩子程序的详细信息，请参阅 [节 2.14](#)。

5.11.25 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。

5.11.26 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 5.16.5](#)。

5.11.27 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 5.11.21.1。

```
[[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 */
```

5.11.28 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 5.3。有关规则列表，请访问 www.ti.com/ulpadvisor。

5.11.29 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，不管要编译和链接的应用程序中的其他段是否引用了该段，它都会包含在链接的输出文件结果中。

RETAIN pragma 具有与使用 retain 函数或变量属性相同的效果。请分别参阅节 5.16.2 和节 5.16.4。

C 中 pragma 的语法为：

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

C++ 中 pragma 的语法为：

```
#pragma RETAIN
```

5.11.30 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。

5.11.31 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 和 PROB_ITERATE 等其他 pragma。

C 和 C++ 中该 pragma 语法为：

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

对应属性的 C++ 语法如下所示。不存在 C 属性语法。有关使用类似语法的示例，请参阅[节 5.11.21.1](#)。

```
[[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 vector Pragma

vector pragma 指示后跟的函数将用作所列矢量的中断矢量例程。该 pragma 的语法为：

```
#pragma vector = vec1[, vec2 , vec3, ...]
```

vector pragma 需要链接器命令文件支持。该命令文件必须为表单 .intxx 的每个中断矢量指定输出节，其中 xx 是中断矢量的编号。输出节必须映射到相应中断矢量的物理存储器位置。标准链接器命令文件用于处理 vector pragma。请参阅节 6.7.4。

如果您没有为某些中断矢量指定 ISR 例程，则将为这些来自 RTS 库的矢量提供 ISR，并且 RTS 库将自动与您的应用链接。默认 ISR 例程会将器件置于低功耗模式。您可以按如下所示使用 unused_interrupts 关键字来覆盖由 RTS 提供的 ISR：

```
#pragma vector=unused_interrupts
interrupt void user_trap_function(void)
{
    // code for handling all interrupts that do not have
    // specific ISRs
}
```

__even_in_range 固有函数给编译器提供了一个提示，指示何时会为中断矢量例程生成 switch 语句。该固有函数通常按如下方式使用：

```
switch (__even_in_range( x , NUM ))
{
    ...
}
```

__even_in_range 固有函数会返回值 x 来控制 switch 语句，但也会告诉编译器 x 必须为 0 至 NUM (含) 范围内的偶数。

备注

中断服务例程 (ISR) 警告

链接器会针对任何未关联中断服务例程的器件专有中断发出警告。不过，默认矢量处理程序现在由运行时支持 (RTS) 库提供，因此如果与提供的 RTS 库链接，那么应该不会看到此错误。

更多相关详情，请参阅《MSP430 汇编语言工具用户指南》的 .intvec 指令段。

5.11.33 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 5.16.2 和节 5.16.4。

5.12 `_Pragma` 运算符

MSP430 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) 定义了单独编译或汇编 (可能使用不同供应商的编译器进行编译) 的函数如何协同工作。这涉及到数据类型表示、寄存器惯例、函数结构和调用惯例的标准化。ABI 定义了从 C 符号名称生成的链接名称。它定义了目标文件格式和调试格式。它定义了系统初始化的方式。如果是 C++，它则定义了对 C++ 名称的处理和异常处理支持。

v15.6.0.STS 和更高版本的 TI 代码生成工具不支持 COFF ABI。如果您想生成 COFF 输出文件，请使用 4.4 版本 of MSP430 代码生成工具，并参考 [SPRU132J](#) 以查看相关文档。

MSP430 使用 EABI ABI，它需要 ELF 目标文件格式。此格式支持现代语言功能，如早期模板实例化和导出内联函数。

节 6.9.3 中描述了有关 EABI 模式的 TI 特定信息。

若要生成与 EABI 兼容的目标文件，必须使用 MSP430 编译器版本 4.0 或更高版本；请参阅节 2.13。

如果为 EABI 编译，则定义 `__TI_EABI__` 预定义符号并将其设置为 1。

有关 MSP430 EABI 的低层面细节，请参阅《MSP430 嵌入式应用程序二进制接口》(SLAA534)。

5.14 目标文件符号命名规则 (链接名)

每个外部可见的标识符都会分配一个用于目标文件的唯一符号名，即所谓的 *链接名*。该名称由编译器根据一种算法分配，该算法取决于符号的名称、类型和源语言。该算法可能会向标识符添加前缀 (通常是下划线)，并且可能会 *改编* 名称。

用户定义的符号 (使用 C 代码和汇编代码) 存储在同一个命名空间中，这意味着需要确保 C 标识符不与汇编代码标识符相冲突。标识符可能与汇编关键字 (例如寄存器名称) 相冲突；在这种情况下，编译器会自动使用转义序列来防止冲突。编译器使用双平行线对标识符进行转义，这指示汇编器不要将标识符视为关键字。需要确保 C 标识符不与用户定义的汇编代码标识符相冲突。

名称改编会对函数链接名中函数参数的类型进行编码，仅发生在未声明为 `extern "C"` 的 C++ 函数中。改编会实现函数重载、运算符重载和类型安全链接。请注意，函数的返回值未在改编后的名称中编码，因为无法根据返回值重载 C++ 函数。

例如，名为 `func` 的函数的 C++ 链接名的一般形式如下：

`func__F parmcodes`

其中，`parmcodes` 是对 `func` 参数类型进行编码的字母序列。

对于这个简单的 C++ 源文件：

```
int foo(int i){ } //global C++ function
```

生成的汇编代码如下：

```
foo__Fi
```

`foo` 的链接名是 `foo__Fi`，表示 `foo` 是一个仅接受整数类型参数的函数。为了帮助检查和调试，提供了一个名称还原实用程序，可将名称还原为 C++ 源代码中的名称。请参阅 [章节](#)，了解详情。

改编算法遵循 Itanium C++ ABI (<http://www.codesourcery.com/cxx-abi/abi.html>) 中的描述。

`int foo(int i) { }` 将改编为 `_Z3fooi`

5.15 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 5.15.1。)
- ANSI/ISO C11 (`--c11` , 请参阅节 5.15.2)
- 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 5.15.3。)

5.15.1 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

5.15.2 C11 支持 (`--c11`)

编译器支持 ISO 标准化的 2011 年标准 C。但是，除了节 5.15.1 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`

5.15.3 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 5.16 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。

5.16 GNU 和 Clang 语言扩展

GNU 编译器集合 (GCC) 定义了许多在 ANSI/ISO C 和 C++ 标准中没有的语言特性。这些扩展的定义和示例 (针对 GCC 4.7 版) 可以在以下 GNU 网站上找到: <http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/C-Extensions.html#C-Extensions>。其中大多数扩展也可用于 C++ 源代码。

编译器还支持以下 Clang 宏命令扩展, 这些扩展在 [Clang 6 文档](#) 中进行了描述:

- `__has_feature` (直到为 Clang 3.5 描述的测试)
- `__has_extension` (直到为 Clang 3.5 描述的测试)
- `__has_include`
- `__has_include_next`
- `__has_builtin` (请参阅节 5.16.6)
- `__has_attribute`

5.16.1 扩展

在宽松 ANSI 模式 (`--relaxed_ansi`) 下进行编译时, 大多数 GCC 语言扩展可在 TI 编译器中使用。

表 5-3 中列出了 TI 编译器支持的扩展, 基于在 GNU 网站上找到的扩展列表。阴影行描述了不受支持的扩展。

表 5-3. GCC 语言扩展

扩展	说明
语句表达式	将语句和声明放在表达式中 (用于创建智能的“安全”宏命令)
局部标签	语句表达式的局部标签
标签作为值	指向标签和计算得到的 <code>goto</code> 的指针
嵌套函数	就像在 <code>Algol</code> 和 <code>Pascal</code> 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
<code>typeof</code> 运算符	表示表达式类型的 <code>typeof</code>
广义左值	在左值中使用问号 (?)、逗号 (,) 和 <code>cast</code>
条件语句	省略 ?: 表达式的中间操作数
十六进制浮点值	十六进制浮点常量
复数	复数的数据类型
零长度	零长度数组
可变参数宏命令	具有可变数量参数的宏命令
可变长度	在运行时计算长度的数组
空结构	没有任何成员的结构
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
指定初始化程序	初始化程序的标签元素
强制转换为 <code>union</code>	从 <code>union</code> 的任何成员强制转换为 <code>union</code> 类型
<code>Case</code> (强制转换) 范围	“ <code>Case 1 ...9</code> ”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, 或者它们根本绝不会返回
属性语法	属性正式语法
函数原型	原型声明和旧式定义

表 5-3. GCC 语言扩展 (continued)

扩展	说明
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	有关类型或变量对齐方式的询问
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 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 5.16.6)
矢量扩展	通过内置函数使用矢量指令
目标内置	特定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

5.16.2 函数属性

支持以下 GCC 函数属性：

- alias
- aligned
- always_inline
- call_conv
- calls
- const
- constructor
- deprecated
- format
- format_arg
- interrupt
- malloc
- naked
- noline
- noreturn
- pure
- section
- 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` 函数属性与 `CODE_ALIGN` pragma 具有相同的效果。请参阅 [节 5.11.4](#)

`always_inline` 函数属性与 `FUNC_ALWAYS_INLINE` pragma 具有相同的效果。请参阅 [节 5.11.11](#)

`call_conv` 属性可用于修改调用惯例，以使 IAR 和 TI 编译器能够链接同一个 ROM 映像。此函数属性将使用 TI 编译器编译的函数与 IAR 编译器生成的 ROM 映像链接在一起。请注意，TI 编译器不会生成 ROM 映像。

`call_conv` 属性可通过 “`cc_rom`”（可实现 IAR/TI 兼容性）或 “`cc_norm`”（默认调用惯例）指定。如果要共享使用 IAR 编译的 ROM 映像，请使用 “`cc_rom`”。以下示例以多种方式使用 `call_conv` 属性：

```
#define __cc_rom __attribute__((call_conv("cc_rom")))

__cc_rom void rom_func(void)
{
    ...
}
typedef __cc_rom void (rom_func_t)(void);

int main()
{
    rom_func();

    rom_func_t *fp = (rom_func_t*)0x1234;
    fp();

    ((void (__cc_rom *) (void))0x2468)();

    void (__cc_rom *rom_func_ptr)(void);
    rom_func_ptr = &rom_func;
    rom_func_ptr();
}
```

如果您希望 IAR/TI 与您的调用惯例兼容，请注意以下有关参数传递的限制。

- 组合的所有参数必须适合寄存器。寄存器不能在栈上传递。
- 仅允许标量参数。不要传递结构体。
- 不要传递函数指针、枚举或双精度值。
- 对于单个寄存器，按顺序使用 R12、13、14、15。
- 对于寄存器对，按顺序使用 12:R13 或 R14:R15。
- 对于四倍字寄存器，请使用 R12:R13:R14:R15。
- 对于 MSP430，请始终使用 CALL/RET。
- 对于 MSP430x，请始终使用 CALLA/RETA。
- 对于调用保存寄存器，请使用 R11-R15。
- 对于入口保存寄存器，请使用 R4-R10。

`calls` 属性的效果与 `CALLS` pragma 相同，相关信息请参阅 [节 5.11.2](#)。

`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它们。

请参阅 [节 5.11.19](#)，详细了解如何使用 **interrupt** 函数属性。

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

naked 属性用于标识使用 `__asm` 语句编写为嵌入式程序集函数的函数。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 函数属性具有与 `FUNC_CANNOT_INLINE` pragma 相同的效果。请参阅 [节 5.11.12](#)

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

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

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

较新的 TI 链接器命令文件通过将具有此属性的函数放置在 `.TI.ramfunc` 段中来自动支持 **ramfunc** 属性。如果您的链接器命令文件不包含 `.TI.ramfunc` 段的段规格，您可以修改链接器命令文件以将此段放在 RAM 中。如需详细了解段放置，请参阅《*MSP430 汇编语言工具用户指南*》。

retain 属性与 `RETAIN` pragma ([节 5.11.29](#)) 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。

在函数中使用的 **section** 属性具有与 `CODE_SECTION` pragma 相同的效果。请参阅 [节 5.11.5](#)

weak 属性与 `WEAK` pragma ([节 5.11.33](#)) 具有相同的效果。

5.16.3 For 循环属性

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

- `TI::must_iterate`
- `TI::prob_iterate`
- `TI::unroll`

有关使用 `for` 循环属性的示例，请参阅 [节 5.11.21.1](#)。

5.16.4 变量属性

支持以下变量属性：

- aligned
- deprecated
- location
- mode
- noinit
- packed
- persistent
- retain
- section
- transparent_union
- unused
- used
- weak

在变量中使用的 **aligned** 属性具有与 DATA_ALIGN pragma 相同的效果。请参阅 节 5.11.6

location 属性具有与 LOCATION pragma 相同的效果。请参阅节 5.11.20。例如：

```
__attribute__((location(0x100))) extern struct PERIPH peripheral;
```

noinit 和 **persistent** 属性适用于 ROM 初始化模式，应用可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《MSP430 汇编语言工具用户指南》中的“RAM 模型与 ROM 模型”一节及其字节。

noinit 属性可用于未初始化的变量；可防止这些变量在重置期间设为 0。**persistent** 属性可用于初始化的变量；可防止这些变量在重置期间初始化。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 5.11.22。

packed 属性可应用于结构体或联合体中的单独字段。节 5.16.5 中介绍了用于结构体或联合体字段的 **packed** 属性的行为。

retain 属性具有与 RETAIN pragma (节 5.11.29) 相同的效果。也就是说，包含该变量的段即使在应用程序的其他地方没有被引用，也不会从条件链接输出中省略。

变量中使用的 **section** 属性具有与 DATA_SECTION pragma 相同的效果。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 属性具有与 WEAK pragma (节 5.11.33) 相同的效果。

5.16.5 类型属性

编译器支持以下类型属性：

- aligned
- deprecated
- packed
- transparent_union
- unused

如果使用了 `--relaxed_ansi` 选项，则结构体和联合体类型都支持 **packed** 属性。

压缩结构的成员在存储时会尽可能靠近彼此，并会忽略通常为保持字对齐而添加的额外填充字节。例如，假定一个 4 个字节的字大小通常在成员 `c1` 和 `i` 之间具有 3 个填充字节，在成员 `c2` 后具有另外 3 个填充字节，因此总大小为 12 个字节：

```
struct unpacked_struct { char c1; int i; char c2;};
```

不过，压缩结构的成员是字节对齐的。因此，以下示例中成员之间或之后没有任何填充字节，总共为 6 个字节：

```
struct __attribute__((__packed__)) packed_struct { char c1; int i; char c2;};
```

因此，数组中的压缩结构会压缩在一起，数组元素之间没有填充字节。

在位字段上使用“**packed**”会覆盖位字段的 **EABI** 要求。对于**非压缩位字段**，位字段的声明类型会用于容器类型。对于**打包位字段**，不管声明类型如何，都会使用最小整型（`bool` 除外）。对于**非压缩易失性位字段**，位字段必须使用大小与声明类型相同的访问来进行访问。对于**压缩易失性位字段**，访问必须与实际容器类型具有相同的大小，并可能与声明类型的大小不同；另外，实际容器可能不对齐，并且可能跨多个对齐的容器边界，因此访问压缩易失性位字段时可能需要多次进行存储器存取。这还可能会影响结构体的总体大小；例如，如果该结构体仅包含位字段，它可能与位字段的声明类型不一样大。对于**压缩和非压缩位字段**，位字段都是位对齐，并与以下相邻位字段压缩在一起：没有填充字节，完全包含在至少是字节对齐的整数容器中；并且不会改变相邻非位字段结构成员的对齐方式。

“**packed**”属性只能应用于结构体或联合体类型的原始定义。它不能通过 `typedef` 用于已定义的非压缩结构，也不能用于结构体或联合体对象的声明。因此，任意给定结构体或联合体类型都只能是压缩或非压缩，并且该类型的所有对象都会继承其 **packed** 或 **non-packed** 属性。

“**packed**”属性不能递归应用到压缩结构体中包含的结构体类型。因此，在以下示例中，成员会保留与上方第一个示例相同的内部布局。`c` 和 `s` 之间没有填充字节，因此 `s` 在未对齐的边界上：

```
struct __attribute__((__packed__)) outer_packed_struct { char c; struct unpacked_struct s;};
```

以隐式或显式方式将压缩结构体成员的地址作为指针投射到除无符号字符以外的任意非压缩类型都是非法的。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;
```

“packed” 属性也可以应用到枚举类型。在枚举中，“packed” 表示应该使用最小整型。

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*。

5.16.6 内置函数

支持以下内置函数：

- `__builtin_abs()`
- `__builtin_constant_p()`
- `__builtin_expect()`
- `__builtin_fabs()`
- `__builtin_fabsf()`
- `__builtin_frame_address()`
- `__builtin_labs()`
- `__builtin_memcpy()`
- `__builtin_return_address()`

`__builtin_frame_address()` 函数始终返回 0，。

`__builtin_return_address()` 函数始终返回零。

5.17 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。

本章介绍 MSP430 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 内存模型	122
6.2 对象表示	126
6.3 寄存器惯例	128
6.4 函数结构和调用惯例	128
6.5 访问 C 和 C++ 中的链接器符号	131
6.6 将 C 和 C++ 与汇编语言相连	131
6.7 中断处理	134
6.8 使用内在函数访问汇编语言语句	136
6.9 系统初始化	140
6.10 对 20 位 MSP430X 器件的编译	147

6.1 内存模型

MSP430 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 16 位（对于 MSP430）或 20 位（对于 MSP430X）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，但这不是一般的做法（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 代码内存模型

MSP430 编译器支持两种不同的代码内存模型：小型的和大型的，都由 `--code_model` 选项控制。

- 小型代码模型使用 16 位函数指针，并要求将所有代码和所有文本段都放置在低 64K 的内存中。这是 16 位 MSP430 器件的唯一有效代码模型。
- 大型代码模型为代码提供 1MB 地址空间并使用 20 位函数指针。这是 MSP430X 器件（包括 MSP430F5xxx 和 MSP430F6xxx 系列）的默认代码模型。中断服务例程仍必须放置在低 64KB 的内存中（请参阅节 6.7.5）。

与大型代码模型相比，小型代码模型在运行时性能和内存使用方面的效率略高。因此，当所有代码和文本段都能适应低 64K 的内存时，使用小型代码模型更有利。使用小型代码模型进行汇编/编译的模块与使用大型代码模型进行汇编/编译的模块不兼容。如果尝试对使用不同代码内存模型的目标文件进行组合，链接器将生成错误。此外，还必须使用合适的运行时库。

6.1.2 数据存储器模式

MSP430 编译器支持三种不同的数据存储器模式：小型、受限型和大型。使用的数据模式由 `--data_model` 选项控制。16 位 MSP430 器件始终使用小型数据存储器模式。20 位 MSP430X 器件可以使用任何数据存储器模式，默认情况下使用小型数据模式。

- 小型数据模式要求所有数据段都位于低 64K 存储器中。数据指针的大小为 16 位。就性能和应用大小而言，这是更有效的数据模式。
- 受限数据模式允许在 MSP430X 器件上可用的整个 1MB 地址空间中定位数据，与小型数据模式相比，效率损失更小。它受到限制是因为单个对象（结构、数组等）的大小不能超过 64K。数据指针的大小为 32 位。
- 大数据模式还允许在整个 1MB 地址空间中定位数据，并且对单个对象的最大大小没有限制。允许单个对象的大小大于 64K 会导致为大数据模式生成的代码的效率低于为受限数据模式生成的代码的效率。

数据存储器还受 `--near_data` 选项的影响，如节 6.1.3 中所述。

在大数据模式中，对象的最大大小 (`size_t`) 和两个指针之间的最大差值 (`ptrdiff_t`) 从 16 位增加到 32 位。依赖于 `size_t` 或 `ptrdiff_t` 的应用程序可能需要更新。

使用不同数据模式生成的目标文件不兼容。应用程序中的所有文件必须使用相同的数据模式生成。此外，必须使用与该数据模式匹配的运行时支持库。使用自动库选择（默认）时，链接器将自动选择正确的库，如节 4.3.1.1 中所述。如果手动选择库，则必须根据节 7.1.9 选择匹配的库。

6.1.3 支持近数据

大多数 MSP430X 器件在 64K 边界以上没有任何可写内存。但是，大多数 FRAM MSP430x 器件和少数非 FRAM 器件在 64k 边界以上具有可写内存。

对于那些具有高 RAM (64K 以上) 的少数非 FRAM 器件，通过链接器命令文件可以将非常量数据放置在 64K 边界以上。

对于所有其他器件，即使使用受限或大型数据模型，64K 边界以上也没有可写内存，只有常量数据将放置在 64K 以上。编译器可以利用这一知识生成更高效的代码。--near_data 选项控制此优化。

指定 --near_data=globals 时，此选项告诉编译器所有全局读取/写入数据必须位于前 64K 的内存中。这是默认行为。全局读取/写入数据默认放置在 .bss 和 .data 段中。

如果指定了 --near_data=none，此选项告诉编译器它不能依赖这个假设来生成更高效的代码。

6.1.4 段

编译器产生被称为段的可重定位代码块和数据块。这些段以各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《MSP430 汇编语言工具用户指南》介绍的目标文件信息。

有两种基本类型的段：

- **初始化段**中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《MSP430 汇编语言工具用户指南》。
 - **.cinit** 段包含用于初始化变量和常量的表。编译器不创建此段；相反，链接器会创建此段。
 - **.init_array** 段包含指向全局 C++ 对象初始化例程的指针的表。
 - **.ovly** 段包含联合的复制表，其中的不同段具有相同的运行地址。
 - **.data** 段包含初始化的全局变量和静态变量。此段是可写的。
 - 仅对于 EABI，**.mspabi.exidx** 段包含用于异常处理的索引表。**.mspabi.exstab** 段包含用于异常处理的展开指令。相关详细信息，请参阅 --exceptions 选项。
 - **.const** 段包含字符串常量、字符串文字、开关表和使用 C/C++ 限定符 const 定义的数据（前提是该常量未定义为 volatile 或节 5.7.1 中描述的异常之一）。字符串文字放置在 .const:string 子段中，以实现更大力度的链接时放置控制。
 - **.text** 段包含所有可执行代码和由编译器生成的常量。此段通常是只读的。
 - **.TI.crctab** 段包含 CRC 检查表。
- **未初始化的段**会在存储器中保留空间（通常为 RAM）。程序可以在运行时使用此空间来创建和存储变量。编译器会创建以下未初始化的段：
 - **.bss** 段为全局变量和静态变量保留空间。在引导或加载时，C/C++ 引导例程或加载程序从 .cinit 段（可能在 ROM 中）复制数据，并将其存储在 .bss 段中。
 - 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。未初始化且未使用的变量通常创建为公共符号（除非您指定 --common=off），而不是放置在 .bss 中，以便将它们从生成的应用中排除。
 - **.stack** 段为保留空间。C/C++ 软件栈。
 - **.systemem** 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 .systemem 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 5.11.5 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 或 .init_array	ROM 或 RAM
.cinit	ROM 或 RAM	.stack	RAM
.const	ROM 或 RAM	.systemem	RAM
.data	RAM	.text	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《*MSP430 汇编语言工具用户指南*》中的链接器说明一章。

6.1.5 C/C++ 软件堆栈

C/C++ 编译器使用函数帧堆栈来：

- 分配局部变量
- 传递参数给函数
- 保存寄存器内容

运行时堆栈从高地址增长到低地址。编译器使用 **R1** 寄存器来管理此堆栈。**R1** 是 *堆栈指针 (SP)*，指向堆栈上的下一个未使用位置。

链接器设置堆栈大小，创建全局符号 `__TI_STACK_SIZE`，并为其分配一个等于堆栈大小的值（以字节为单位）。默认的堆栈大小为 80 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《*MSP430 汇编语言工具用户指南*》中的链接器说明一章。

备注

Save-On-Entry 寄存器和 C/C+ 栈大小

由于 MSP430X 器件的寄存器大小增加（用 `--silicon_version=mbsp` 指定），因此保存和恢复 `save-on-entry` 寄存器需要为堆栈上保存的每个寄存器提供 32 位栈空间。在移植最初为 16 位 MSP430 器件编写的代码时，可能需要在以前的值的基础上增加 C 栈大小。

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++ 环境在函数入口处自动递减 **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

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。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栈溢出；请参阅节 2.14。

6.1.6 动态内存分配

MSP430 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内存。

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_SYSTEMEM_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 128 字节。有关 `--heap_size` 选项的更多信息，请参阅《*MSP430 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。

动态分配的对象没有被直接寻址（始终使用指针访问），并且内存池位于单独的段（.system）中；因此，动态内存池的大小仅受系统中可用内存量的限制。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。这在分配可变长度数组时尤为重要。

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存储空间和存储器存储空间：

表 6-2. 寄存器和存储器中的数据表示

数据类型	寄存器存储	存储器存储
char	寄存器的位 0-7	8 位，与 8 位边界对齐
unsigned char	寄存器的位 0-7	8 位，与 8 位边界对齐
short	寄存器的位 0-15	16 位，与 16 位边界对齐
unsigned short	寄存器的位 0-15	16 位，与 16 位边界对齐
int	整个寄存器	16 位，与 16 位边界对齐
unsigned int	整个寄存器	16 位，与 16 位边界对齐
enum ⁽¹⁾	整个寄存器	16 位，与 16 位边界对齐
float	两个寄存器，不需要相邻	32 位，与 16 位边界对齐
long	两个寄存器，不需要相邻	32 位，与 16 位边界对齐
unsigned long	两个寄存器，不需要相邻	32 位，与 16 位边界对齐
long long	四个寄存器，不需要相邻	64 位，与 16 位边界对齐
unsigned long long	四个寄存器，不需要相邻	64 位，与 16 位边界对齐
double	四个寄存器，不需要相邻	64 位，与 16 位边界对齐
long double	四个寄存器，不需要相邻	64 位，与 16 位边界对齐
结构体	成员按其各自类型的需要存储。	与最大成员类型边界对齐的 8 位的倍数；成员根据其各自类型的需要进行存储和对齐。
数组	成员按其各自类型的需要存储。	成员按其各自类型的需要存储。结构中的所有数组都根据数组中每个元素的类型对齐。
指向数据成员的指针	MSP430 寄存器的位 0-15。MSP430X 寄存器的位 0-19。	不尽相同，请参阅表 5-2
指向成员函数的指针	组件按其各自类型的要求存储	64 位，与 32 位边界对齐

(1) 枚举的大小因最大枚举值的大小以及是否打包而异。有关枚举类型大小的详细信息，请参阅节 5.5.1。

6.2.1.1 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，布局相当于：

```
struct {
    short int d;
    short int i;
    union {
        void (f) ();
        long 0;
    }
};
```

参数 d 是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，偏移量为 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。

6.2.1.2 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。结构在最后一个成员之后不包含直译。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。

数组的最小对齐是其元素类型所需的对齐方式。

6.2.1.3 字段/结构对齐

当编译器为结构分配空间时，它会根据需要分配尽可能多的字来容纳结构的所有成员，并遵守每个成员的对齐约束。

当结构包含 32 位 (long) 成员时，long 与 1 字 (16 位) 边界对齐。这可能需要在结构的前面、内部或末尾进行填充，以确保 long 相应地对齐并且结构的大小值是偶数。

所有非字段类型都在字或字节边界上对齐。字段按请求的位数分配。相邻字段被打包成一个字的相邻位，但它们不会与字重叠。如果一个字段与下一个字重叠，则会将整个字段放入下一个字中。

字段相遇时就会被打包；首先填充结构字的最低有效位。示例 6-1 显示了 var 的 C 代码定义，而图 6-1 显示了 var 的内存布局。

示例 6-1. var 的 C 代码定义

```
struct example { char c; long l; int bf1:1; int bf2:2; int bf3:3; int bf4:4; int bf5:5; int bf6:6; };
```

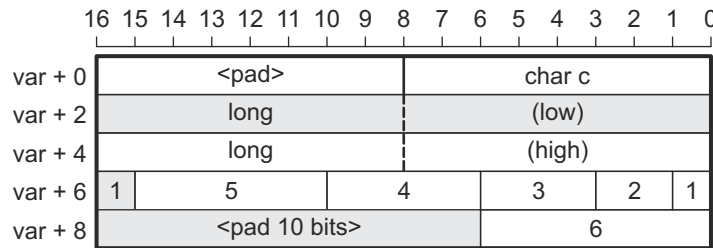


图 6-1. var 的内存布局

6.2.2 字符串常量

在 C 语言中，字符串常量用于下述方式之一：

- 初始化字符数组。例如：

```
char s[] = "abc";
```

当字符串用作初始化值时，其被简单地视为初始化数组；每个字符都是一个单独的初始化值。更多有关初始化的信息，请参阅节 6.9。

- 在表达式中。例如：

```
strcpy (s, "abc");
```

在表达式中使用字符串时，字符串本身是在 .const 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 SLn，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 SLn 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。表 6-3 显示了受这些惯例影响的寄存器类型。表 6-4 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
程序计数器	包含正在执行的代码的当前地址

表 6-4. 寄存器使用和保留惯例

寄存器	别名	使用	保留函数 ⁽¹⁾
R0	PC	程序计数器	不适用
R1	SP	栈指针	不适用 ⁽²⁾
R2	SR	状态寄存器	不适用
R3		常量发生器	不适用
R4-R10		表达式寄存器	子级
R11		表达式寄存器	父级
R12		表达式寄存器、参数指针、返回寄存器	父级
R13		表达式寄存器、参数指针、返回寄存器	父级
R14		表达式寄存器、参数指针、返回寄存器	父级
R15		表达式寄存器、参数指针、返回寄存器	父级

(1) 父函数指进行函数调用的函数。子函数指被调用的函数。

(2) SP 根据惯例保留, 即压入栈的所有内容都在返回之前弹出。

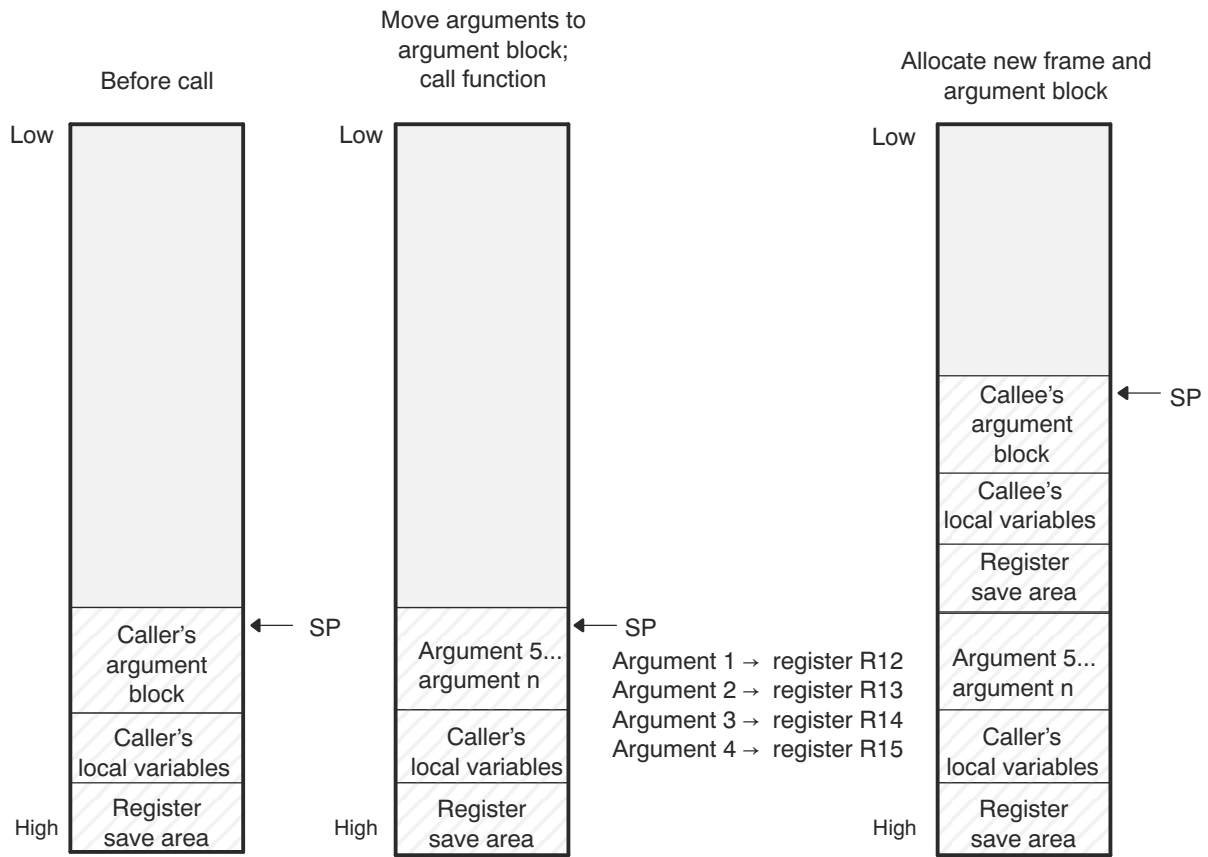
6.4 函数结构和调用惯例

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。不遵循这些规则可能会破坏 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它们压入栈的方式将这些参数传递给函数。本地帧和参数块是同时分配的。
- **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 R11-R15。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-R10。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。

图 6-2 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 R12-R15。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会分配本地帧。



Legend: SP: stack pointer

图 6-2. 在函数调用期间使用栈

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。

<pre>struct big { long x[10]; }; struct small { int x; };</pre>					
T0	T0	AC0	AR0		
<pre>int fn(int i1, long l2, int *p3);</pre>					
AC0	AR0	T0	T1	AR1	
<pre>long fn(int *p1, int i2, int i3, int i4);</pre>					
AR0	AR1				
<pre>struct big fn(int *p1);</pre>					
T0	AR0	AR1			
<pre>int fn(struct big b, int *p1);</pre>					
AC0	AR0				
<pre>struct small fn(int *p1);</pre>					
T0	AC0	AR0			
<pre>int fn(struct small b, int *p1);</pre>					
T0	stack	stack...			
<pre>int printf(char *fmt, ...);</pre>					
AC0	AC2	AC2	stack	T0	
<pre>void fn(long l1, long l2, long l3, long l4, int i5);</pre>					
AC0	AC1	AC2	AR0	AR1	
<pre>void fn(long l1, long l2, long l3, int *p4, int *p5,</pre>					
AR2	AR3	AR4	T0	T1	
<pre>int *p6, int *p7, int *p8, int i9, int i10);</pre>					

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。 (调用保存寄存器为 R11-R15。)
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. 调用方会将第一批参数按顺序放在寄存器 R12-R15 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。类型大于 16 位且在调用保存寄存器中开始的参数可能会在 R15 和栈之间被拆分。
4. 调用方会调用函数。

C++ 中定义的函数若必须在 asm 中调用，则必须定义为 extern "C"，asm 中定义的函数若必须在 C++ 中调用，则必须在 C++ 文件中原型化为 extern "C"。

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. 如果用省略号声明了函数，则可以使用可变数量的参数调用它。如果这些参数满足这两个条件，则被调用函数会将这些参数压入栈：
 - 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. 被调用函数将被该函数修改的所有寄存器的寄存器值压入栈，这些值必须在函数退出时保留。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-R10)。如果该函数是中断，可能需要保留额外的寄存器。如需了解更多信息，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：

$$\text{size of all local variables} + \text{max} = \text{constant}$$

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值，它会将该值放入 R12、R12 和 R13 中或 R12 到 R15 中，具体取决于返回类型的大小。
6. 如果被调用函数返回结构体，它会将该结构体复制到第一个参数 R12 指向的内存块。如果调用方不使用返回值，则将 R12 设置为 0。这会指示被调用函数不要复制返回结构体。

结构体和联合体始终通过引用传递。

通过这种方式，调用方可以用聪明的方式告知被调用函数从哪里返回结构体。例如，在语句 `s = func(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。

您必须小心操作以正确地声明返回结构体的函数，无论是在调用它们的位置 (使调用方正确设置第一个参数) 还是声明它们的位置 (使函数知道复制结果)。

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 (`func`) 将返回。

以下示例是被调用函数如何响应调用的典型示例：

```
func:                                ; Called function entry point
    PUSH.W    r10
    PUSH.W    r9                    ; Save SOE registers
    SUB.W     #2,SP                 ; Allocate the frame
    :
    :                                ; Body of function
    :
    ADD.W     #2,SP                 ; Deallocate the frame
    POP      r9                    ; Restore SOE registers
    POP      r10
    RET
```

6.4.3 访问参数和局部变量

函数通过栈指针 (SP 或 R1) 间接访问其本地非寄存器变量和栈参数。SP 始终指向栈顶 (最新压入的值)。

栈向较小的地址增长，因此使用 SP 寄存器的正偏移量访问 C/C++ 函数栈上的本地数据。

6.5 访问 C 和 C++ 中的链接器符号

请参阅《MSP430 汇编语言工具用户指南》中关于“在 C/C++ 应用程序中使用链接器符号”的部分，了解有关在 C/C++ 代码中引用链接器符号的信息。

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) 。

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-R10)
 - 栈指针 (SP 或 R1)

如果通常使用 SP, 则无需显式保留。换言之, 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 R12 和 R13, 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 R12。
- 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 5.14。
- 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。

- 任何与 MSP430x C 程序搭界的汇编例程都应符合大型代码模型:
 - 使用 CALLA/RETA 而不使用 CALL/RET
 - 使用 PUSHM.A/POPM.A 保存并恢复任何未使用的入口保存寄存器。必须保存/恢复整个 20 位寄存器。
 - 操作函数指针需要 20 位运算 (OP.A)
 - 如果与 C 代码 (针对大型数据模型编译) 搭界, 则数据指针操作必须使用 20 位运算 (OP.A) 执行。

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 6-2 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 6-3 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 6-2. 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 var = 5;
    var = asmfunc(var);          /* 正常调用函数    */
}
```

示例 6-3. 示例 6-2 调用的汇编语言程序

```
.global asmfunc
.global gvar
asmfunc:
    MOV    &gvar,R11
    ADD   R11,R12
    RET
```

在示例 6-2 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 `var` 在 R12 中传递，并在 R12 中返回结果。

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 6-5 和示例 6-4 说明了如何访问 `.bss` 中定义的变量。

示例 6-4. 汇编语言变量程序

```
.bss    var,4,4      ; Define the variable
.global var         ; Declare it as external
```

示例 6-5. C 程序从示例 6-4 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《MSP430 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《MSP430 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 6-6 和示例 6-7 中所示。

示例 6-6. 从 C 语言访问汇编语言常量

```
extern int table_size;          /*外部引用 */
#define TABLE_SIZE ((int) _symval(&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 6-7. 示例 6-6 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 6-6 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《MSP430 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。

6.7 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。寄存器保存必须由中断例程明确处理。

6.7.2 使用 C/C++ 中断例程

C/C++ 中断例程就像任何其他 C/C++ 函数一样，它可以有局部变量和寄存器变量。除软件中断例程外，中断例程必须声明为不带参数且必须返回 `void`。例如：

```
__interrupt void example (void)
{
    ...
}
```

如果 C/C++ 中断例程未调用任何其他函数，则只会保存和恢复中断处理程序使用的寄存器。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。（这不包括分组寄存器。）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 5.11.19](#) 和 [节 5.7.2](#)。

6.7.3 使用汇编语言中断例程

只要您遵守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（限制为 16 位）、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 中断向量

MSP430 和 MSP430X 器件的中断向量是 16 位。因此，中断服务例程 (ISR) 必须放置在存储器的低 64K 中。MSP430X 器件头文件中提供了便利宏命令来声明中断，以确保链接时的 16 位放置。

或者，使用 `CODE_SECTIONS pragma` 将 ISR 的代码放入与默认 `.text` 段分开的段中。使用链接器命令文件和 `SECTIONS` 指令，确保将与 ISR 关联的代码段放入低速存储器。

如果您没有为某些中断向量指定 ISR，将为这些来自 RTS 库的向量提供 ISR，并且 RTS 库将自动与您的应用链接。默认 ISR 会将器件置于低功耗模式。您可以使用 `pragma` 向量和未使用的中断关键字覆盖由 RTS 提供的 ISR，如 [节 5.11.32](#) 中所示。

6.7.5 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。需要安排中断信号的发生。
- 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。

6.8 使用内在函数访问汇编语言语句

编译器可识别若干内在函数运算符。内在函数的用法与函数类似，可生成在 C/C++ 中无法以其他方式表达的汇编语言语句。您可以将 C/C++ 变量与这些内在函数结合使用，就像在任何普通函数中一样。内在函数通过双先导下划线指定，可像调用函数一样进行访问。（内在函数可使用单先导下划线和双先导下划线，但首选双下划线。例如：

```
short state;
:
state = __get_SR_register();
```

内在函数无需声明。

6.8.1 MSP430 内在函数

表 6-5 列出了 MSP430 C/C++ 编译器中的所有内在函数运算符。每个内在函数都有一个函数式原型，显示每个参数的预期类型。如果参数类型与参数不匹配，则对参数执行类型转换。编译器的“包含”目录中的 `intrinsics.h` 文件包含所有受支持内在函数的函数式原型。

更多有关生成的汇编语言助记符的信息，请参阅《MSP430x1xx 系列用户指南》、《MSP430x3xx 系列用户指南》和《MSP430x4xx 系列用户指南》。

表 6-5. MSP430 内在函数

内在函数	生成的汇编
unsigned short __bcd_add_short(unsigned short op1, unsigned short op2);	MOV op1、dst CLRC DADD op2、dst
unsigned long __bcd_add_long(unsigned long op1, unsigned long op2);	MOV op1_low、dst_low MOV op1_hi、dst_hi CLRC DADD op2_low、dst_low DADD op2_hi、dst_hi
unsigned short __bic_SR_register(unsigned short mask);	BIC mask、SR
unsigned short __bic_SR_register_on_exit(unsigned short mask);	BIC mask、saved_SR
unsigned short __bis_SR_register(unsigned short mask);	BIS mask、SR
unsigned short __bis_SR_register_on_exit(unsigned short mask);	BIS mask、saved_SR
unsigned long __data16_read_addr(unsigned short addr);	MOV.W addr、Rx MOVA 0(Rx)、dst
void __data16_write_addr (unsigned short addr, unsigned long src);	MOV.W addr、Rx MOVA src、0(Rx)
unsigned char __data20_read_char(unsigned long addr); ⁽¹⁾	MOVA addr、Rx MOVX.B 0(Rx)、dst
unsigned long __data20_read_long(unsigned long addr); ⁽¹⁾	MOVA addr、Rx MOVX.W 0(Rx)、dst.lo MOVX.W 2(Rx)、dst.hi
unsigned short __data20_read_short(unsigned long addr); ⁽¹⁾	MOVA addr、Rx MOVX.W 0(Rx)、dst
void __data20_write_char(unsigned long addr, unsigned char src); ⁽¹⁾	MOVA addr、Rx MOVX.B src、0(Rx)
void __data20_write_long(unsigned long addr, unsigned long src); ⁽¹⁾	MOVA addr、Rx MOVX.W src.lo、0(Rx) MOVX.W src.hi、2(Rx)
void __data20_write_short(unsigned long addr, unsigned short src); ⁽¹⁾	MOVA addr、Rx MOVX.W src、0(Rx)
void __delay_cycles(unsigned long);	请参阅节 6.8.4。

表 6-5. MSP430 内在函数 (continued)

内在函数	生成的汇编
void __disable_interrupt(void); 或 __disable_interrupts(void);	DINT
void __enable_interrupt(void); 或 __enable_interrupts(void);	EINT
unsigned int __even_in_range(unsigned int, unsigned int);	请参阅节 5.11.32。
uint32_t __f32_bits_as_u32(float src)	将浮点型中的位提取为 32 位寄存器。请参阅节 6.8.2。
uint64_t __f64_bits_as_u64(double src)	将双精度型中的位提取为 64 位寄存器。请参阅节 6.8.2。
unsigned short __get_interrupt_state(void);	MOV SR, dst
unsigned short __get_R4_register(void); /* 已弃用 */	MOV.W R4, dst
unsigned short __get_R5_register(void); /* 已弃用 */	MOV.W R5, dst
unsigned short __get_SP_register(void);	MOV SP, dst
unsigned short __get_SR_register(void);	MOV SR, dst
unsigned short __get_SR_register_on_exit(void);	MOV saved_SR, dst
void __low_power_mode_0(void);	BIS.W #0x18, SR
void __low_power_mode_1(void);	BIS.W #0x58, SR
void __low_power_mode_2(void);	BIS.W #0x98, SR
void __low_power_mode_3(void);	BIS.W #0xD8, SR
void __low_power_mode_4(void);	BIS.W #0xF8, SR
void __low_power_mode_off_on_exit(void);	BIC.W #0xF0, saved_SR
void __never_executed(void);	请参阅节 6.8.5。
void __no_operation(void);	NOP
void __op_code(unsigned short);	对与参数对应的任何指令进行编码。
short __saturated_add_signed_short(short src1, short src2);	对短整型值执行饱和和加法的指令。超出有符号短整型范围的值将视情况设置为最大值或最小值。
long __saturated_add_signed_long(long src1, long src2);	对长整型值执行饱和和加法的指令。超出有符号长整型范围的值将视情况设置为最大值或最小值。
short __saturated_sub_signed_short(short src1, short src2);	对短整型值执行饱和和减法的指令。超出有符号短整型范围的值将视情况设置为最大值或最小值。
long __saturated_sub_signed_long(long src1, long src2);	对长整型值执行饱和和减法的指令。超出有符号长整型范围的值将视情况设置为最大值或最小值。
void __set_interrupt_state(unsigned short src);	MOV src, SR
void __set_R4_register(unsigned short src); /* deprecated */	MOV.W src, R4
void __set_R5_register(unsigned short src); /* deprecated */	MOV.W src, R5
void __set_SP_register(unsigned short src);	MOV src, SP
unsigned short __swap_bytes(unsigned short src);	MOV src, dst SWPB dst
float __u32_bits_as_f32(uint32_t src)	将 32 位寄存器打包为浮点型。请参阅节 6.8.2。

表 6-5. MSP430 内在函数 (continued)

内在函数	生成的汇编
double __u64_bits_as_f64(uint64_t src)	将 64 位寄存器打包为双精度型。请参阅节 6.8.2。

(1) 内在函数会根据代码对多个指令进行编码。此处提供了常见的指令。

6.8.2 Floating Point Conversion Intrinsics

The following intrinsics extract the bits in a floating value or pack bits into a floating value. No instructions or movement of data is performed; these intrinsics tell the compiler how to interpret the bits in place. See 节 6.8.1 for calling syntax.

- `__f32_bits_as_u32`: Extract the bits in a float as a 32-bit register.
- `__f64_bits_as_u64`: Extracts the bits in a double as a 64-bit register.
- `__u32_bits_as_f32`: Packs a 32-bit register as a float.
- `__u64_bits_as_f64`: Packs a 64-bit register as a double.

The following function extracts the sign bit of a float. It does this by using the `__f32_bits_as_u32` intrinsic to interpret a float as a 32-bit register. This intrinsic does not move the bits stored in `x`; it tells the compiler to change how it interprets the bits in place.

```
bool sign_bit_set(float x) {
    return __f32_bits_as_u32(x) >> 31;
}
```

Another way to use these intrinsics is for converting a float constant that has more precision than needed to a value that has all zero bits in the least significant mantissa bits. This enables the compiler to use one instruction instead of two to load a 32-bit constant into a register. The following code truncates a constant:

```
__u32_bits_as_f32(__f32_bits_as_u32(3.14159f) & 0xffff0000)
```

6.8.3 弃用的内在函数

以下内在函数已被弃用。不建议使用这些函数，因为将来的版本可能不支持它们。

表 6-6. 弃用的内在函数

内在函数
<code>__get_R4_register</code>
<code>__get_R5_register</code>
<code>__set_R4_register</code>
<code>__set_R5_register</code>

6.8.4 __delay_cycle 内在函数

`__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。

6.8.5 __never_executed 内在函数

MSP430 C/C++ 编译器支持一个 `__never_executed()` 内在函数，此内在函数可用于断言开关块中的默认标签从未执行。如果断言从不执行默认标签，编译器可以根据开关块内的 `case` 标签中指定的值生成更高效的代码。

6.8.5.1 将 `__never_executed` 与矢量发生器搭配使用

内在函数 `__never_executed()` 对于测试 MSP430 中断矢量发生器的值特别有用，例如定时器 A (TAIV) 的矢量发生器。MSP430 矢量发生器值映射到一个中断源，其特征是处于特定范围内，只能取偶数值。处理矢量发生器中呈现的特定中断源的常用方法是使用 `switch` 语句。但 C 语言对编译器的限制在于，它无法假设 `switch` 表达式具有的值。编译器必须生成代码，以处理每个可能的值，导致出现低效代码。

内在函数 `__never_executed()` 可用于向编译器断言，`switch` 表达式只能处理 `switch` 代码块中的 `case` 标签代表的值。利用此断言，编译器就可避免生成测试代码来处理 `switch case` 标签未指定的值。此断言特别适用于处理以矢量发生器为特征的值。

示例 6-8 展示了 `switch` 代码块，用于处理计时器 B (TBIV) 矢量发生器的值。

示例 6-8. TBIV 矢量发生器

```
__interrupt void Timer_B1 (void)
{
    switch( TBIV )
    {
        case 0: break; /* Do nothing */
        case 2: TBCCR1 += 255;
                state +=1;
                break;
        case 4: TBCCR0 = 254;
                TBCCR1 = 159;
                state =200;
                break;
        case 6: break;
        case 8: break;
        case 10: break;
        case 12: break;
        case 14: break;
        default: __never_executed();
    }
}
```

在示例 6-8 中，使用 `__never_executed()` 内在函数断言 TBIV 的值只能采用 `case` 标签指定的值，即从 0 到 14 的偶数值。通常，编译器必须生成代码来处理任何会导致额外范围检查的值。但对于此示例，编译器将生成一个开关表，其中 TBIV 的值被简单地添加到 PC 以跳转到会处理由 `case` 标签表示的每个值的适当代码块。

6.8.5.2 将 `__never_executed` 与通用 `Switch` 表达式搭配使用

在默认标签上使用内在函数 `__never_executed()`，也能改善更加常用的不涉及矢量发生器类型值的由 `switch` 表达式生成的 `switch` 代码。

示例 6-9. 通用 `Switch` 语句

```
switch( val)
{
    case 0:
    case 5: action(a); break;
    case 14: action(b); break;
    default: __never_executed();
}
```

通常，对于 `switch` 表达式值 0 和 5，编译器会生成代码来测试 0 和 5，因为编译器必须处理可能的值 1-4。示例 6-9 中的 `__never_executed()` 内在函数断言 `val` 不能取值 1-4，因此编译器只需要生成一个测试 (`val < 6`) 即可处理两个 `case` 标签。

此外，使用 `__never_executed()` 内在函数会导致断言：如果 `val` 不是 0 或 5，那么它必须是 14，并且编译器不需要生成代码来测试 `val == 14`。

`__never_executed()` 内在函数仅在指定为采用默认 `case` 标签的单个语句时才被定义。编译器会在任何其他上下文中忽略对内在函数的使用。

6.9 系统初始化

您必须先创建 C/C++ 运行时环境，才能运行 C/C++ 程序。C/C++ 启动例程使用被称为 `_c_int00` 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` 或 `--ram_model` 链接选项并将标准运行时支持库作为其中一个链接器输入文件时，此操作会自动发生。

链接 C/C++ 程序时，链接器会将可执行输出文件中的入口点值设置为符号 `_c_int00`。

`_c_int00` 函数会执行以下任务来对环境进行初始化：

1. 为用户模式运行时堆栈保留空间，并设置堆栈指针 (SP) 的初始值
2. 它通过将数据从初始化表复制到 `.bss` 段中为对应变量的存储空间，对全局变量进行初始化。如果您在加载时对变量进行初始化 (`--ram_model` 选项)，加载程序会在程序运行前执行此步骤 (并非由启动例程执行)。如需更多信息，请参阅 [节 6.9.3](#)。
3. 执行在全局构造函数表中找到的全局构造函数。如需更多信息，请参阅 [节 6.9.3.6](#)。
4. 调用 `main()` 函数来运行 C/C++ 程序

您可以更换或修改启动例程以满足系统要求。不过，启动例程必须执行上面列出的操作来正确地初始化 C/C++ 环境。

6.9.1 用于系统预初始化的引导挂钩函数

引导挂钩是可将应用程序函数插入 C/C++ 引导进程的点。默认引导挂钩函数随运行时支持 (RTS) 库一同提供。但是，您可以实现这些引导挂钩函数的自定义版本，如果在运行时库之前链接了 RTS 库中的默认引导挂钩函数，则自定义版本将覆盖 RTS 库中的默认引导挂钩函数。在继续进行 C/C++ 环境设置之前，这些函数可以执行任何应用特定的初始化。

请注意，TI-RTOS 操作系统使用自定义版本的引导挂钩函数进行系统设置，因此，如果使用 TI-RTOS，则应小心覆盖这些函数。

以下引导挂钩函数可用：

`_system_pre_init()`: 此函数提供执行应用特定的初始化的位置。它在初始化栈指针之后，但在执行任何 C/C++ 环境设置之前被调用。默认情况下，`_system_pre_init()` 应返回非零值。如果 `_system_pre_init()` 返回 0，则会绕过默认的 C/C++ 环境设置。

`_system_post_cinit()`: 在 C/C++ 环境设置过程中，在 C/C++ 全局数据被初始化，但在调用任何 C++ 构造函数之前调用此函数。此函数不应返回值。

`_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)。

6.9.2 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。

6.9.3 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。

6.9.3.1 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。

6.9.3.2 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 6.9.3.5。

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 6.9.3.3。

6.9.3.3 在运行时自动初始化变量

在运行时自动初始化变量是自动初始化的默认方法。若要使用此方法，请使用 `--rom_model` 选项调用链接器。

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

图 6-3 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。

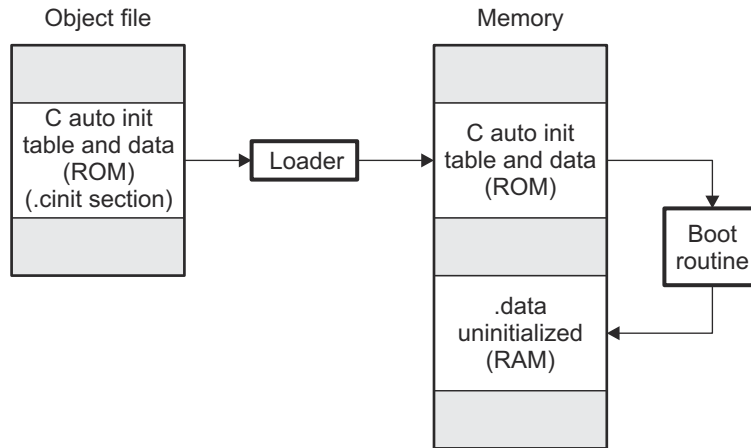


图 6-3. 运行时自动初始化

6.9.3.4 的自动初始表

编译的目标文件没有初始表。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。

自动初始表的格式如下：

_TI_CINIT_Base:

32-bit load address	32-bit run address
⋮	⋮
32-bit load address	32-bit run address

_TI_CINIT_Limit:

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

C 自动初始化记录中的加载地址指向以下格式的初始化数据：

8 位索引	编码数据
-------	------

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

处理程序表是 32 位函数指针的列表。

_TI_Handler_Table_Base:

32-bit handler 1 address
⋮
32-bit handler n address

_TI_Handler_Table_Limit:

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

6.9.3.4.1 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。

6.9.3.4.2 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。

6.9.3.4.3 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 $B \neq D$ ，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 $L \leq 16$ ，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 $L \leq 8$ ，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。

6.9.3.4.4 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。

6.9.3.5 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 6-4 演示了加载时变量的初始化。

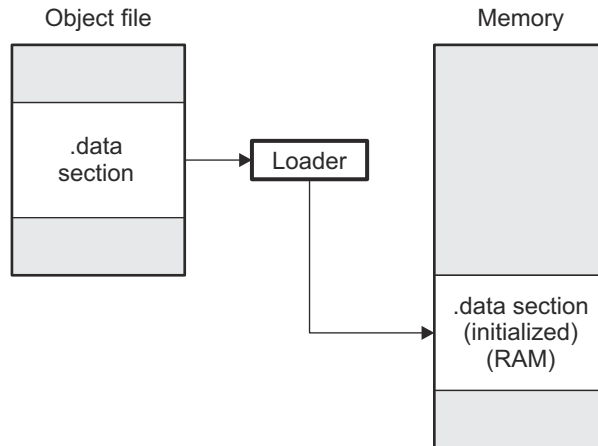


图 6-4. 加载时初始化

6.9.3.6 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。

`__TI_INITARRAY_Base:`

Address of constructor 1
Address of constructor 2
⋮
Address of constructor n

`__TI_INITARRAY_Limit:`

图 6-5. 构造函数表

6.9.4 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 6-6 显示了 .cinit 段的格式和初始化记录。

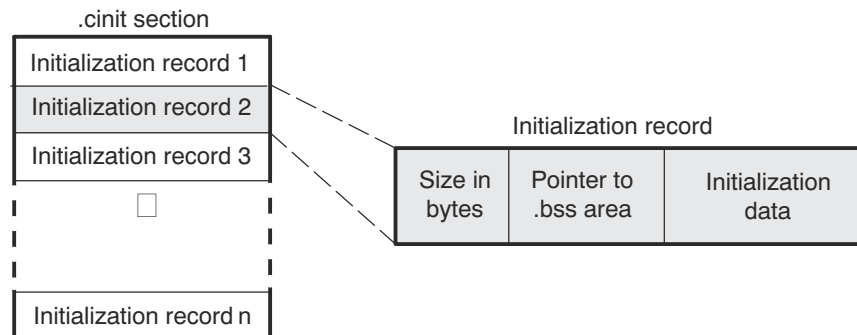


图 6-6. .cinit 段中初始化记录的格式

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）。
- 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。
- 第三个字段包含被复制到 .bss 段以初始化变量的数据。

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 x;
short i = 23;
int *p =
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。

```

.global _x
.bss _x,4,4
.sect ".cinit:c"
.align 8
.field (CIR - $) - 8, 32
.field I+0,32
.field 23,16 ; _I @ 0
.sect ".text"
.global _I
_I: .usect ".bss:c",2,2
.sect ".cinit:c"
.align 4
.field _x,32 ; _p @ 0
.sect ".text"
.global _p
_p: .usect ".bss:c",4,4
.sect ".cinit"
.align 8
.field IR_1,32
.field a+0,32
.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
IR_1: .set 20
.sect ".text"
.global _a
.bss _a,20,4
;*****
;* 标记 CINIT:C 中标量初始化记录的结尾 *
;*****
CIR: .sect ".cinit:c"

```

`.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。

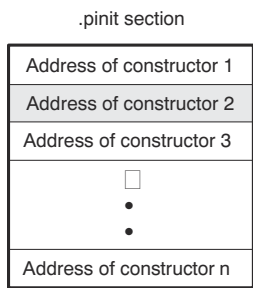


图 6-7. `.pinit` 段中初始化记录的格式

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。

6.10 对 20 位 MSP430X 器件的编译

MSP430 工具支持为 MSP430 和 MSP430X (MSP430X) 器件编译和链接代码。有关适用于 MSP430X 器件编译的选项和主题的更多信息，请参阅以下内容：

- 使用 `--silicon_version=msp_x` 选项编译 MSP430X 器件。请参阅节 2.3.4。

- 函数指针是 20 位的。请参阅节 5.5 和表 6-2。
- 在为 MSP430X 器件生成代码时，编译器支持大型代码内存模型。请参阅节 6.1.1。
- 在为 MSP430X 器件生成代码时，编译器支持大型数据内存模型。请参阅节 6.1.2。
- 任何与 MSP430X C 程序接口的汇编例程都必须适合大型代码模型。请参阅节 6.6.1。
- 中断服务例程必须放在低内存中。请参阅节 6.7.4。
- 链接到 `rts430x.lib` 或 `rts430x_eh.lib` 运行时支持库。

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 对此进程进行了介绍。

7.1 C 和 C++ 运行时支持库.....	150
7.2 C I/O 函数.....	153
7.3 处理可重入性 (_register_lock() 和 _register_unlock() 函数)	165
7.4 库构建流程.....	166

7.1 C 和 C++ 运行时支持库

MSP430 编译器版本包括可提供所有标准功能的预构建运行时支持 (RTS) 库。为每个 ABI 和 C++ 异常支持提供了单独的库。有关库命名规则的信息，请参阅节 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 `<wchar>` 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《MSP430 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。

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 `MSP_C_DIR` 环境变量设为安装相关工具的编译器版本中“包含”目录的绝对路径。

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\msp430_#.##\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 `rts430.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` 实现为 `unsigned int`。宽字符集相当于 `char` 类型的值集。该库包含头文件 `<wchar.h>` 和 `<wctype.h>`，但不包含标准中指定的所有函数。请参阅节 5.6，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

	<code>__time32_t</code>	<code>__time64_t</code>
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 <code>epoch</code> 之前的日期。	带符号，因此可以表示 <code>epoch</code> 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一些 ISO/ANSI 标准未指定但构建库时需要的文件：

- `file.h` 文件包含用于低级 I/O 函数的宏命令和定义。用户可以选择包含此文件。尽管它不是标准文件，但却是构建库所需要的。这个同一类别中还有其他头文件，例如 `cpy_tbl.h`。
- `format.h` 文件包含 `printf` 和 `scanf` 中使用的结构和宏命令。
- `430cio.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 `430cio.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：

rts430[x][l][abi][_eh].lib

rts430	指示 16 位 MSP430 库。
x	可选的 x 指示 20 位 MSP430X 库。
l	x 后可选的 l 指示 20 位大代码、大数据模型 MSP430X 库。（这是一种旧格式。）
abi	指示使用的应用程序二进制接口 (ABI)。尽管不再支持 COFF 文件格式，但库文件名仍然包含 “_eabi”，以便与较旧 COFF 库中的 EABI 库区分开来。
_eabi	EABI
_eh	指示该库支持异常处理
_lc	指示该库支持大代码
_sc	指示该库支持小代码
_ld	指示该库支持大数据
_rd	指示该库支持受限数据
_sd	指示该库支持小数据

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、低级和器件驱动程序级。

借助恰当编写的器件驱动程序，C 标准高级 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 I/O 函数的复杂缓冲技术的简易方法。

备注

默认主机所需的调试器：若要让默认主机器件正常工作，必须使用调试器来处理 C I/O 请求；默认的主机器件无法在嵌入式系统中自行工作。若要在嵌入式系统中工作，您需要为系统提供适当的驱动程序。

备注

C I/O 函数莫名失败：如果堆上没有足够的空间用于 C I/O 缓冲区，文件上的操作将会以静默方式失败。如果 printf() 调用莫名失败，那么原因可能就是这个。堆必须足够大，至少应足以分配执行 I/O 的每个文件所需的块大小 BUFSIZ（在 stdio.h 中定义），包括 stdout、stdin 和 stderr，以及用户代码执行的分配和分配记账开销。也可以声明一个大小字符数组 BUFSIZ，并将其传递给 setvbuf 来避免动态分配。若要设置堆大小，请在链接时使用 --heap_size 选项（请参阅《MSP430 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文件描述符数量 (stdin、stdout 和 stderr 下的低级文件计入此总数)。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 I/O 函数

高级函数是流 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 I/O 函数来执行高级 I/O 请求。高级 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 I/O 函数。

若要使用高级 I/O 函数, 请为引用 C I/O 函数的每个模块加上头文件 `stdio.h` (对于 C++ 代码, 则为 `cstdio`)。例如, 在名为 `main.c` 的文件中提供以下 C 程序:

```
#include <stdio.h>
void main()
{
    FILE *fid;
    fid = fopen("myfile","w");
    fprintf(fid,"Hello, world\n");
    fclose(fid);
    printf("Hello again, world\n");
}
```

通过发出以下编译器命令, 从运行时支持库编译、链接和创建 `main.out` 文件:

```
cl430 main.c --run_linker --heap_size=400 --library=rts430_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 100 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。请参阅节 7.2.3 了解更多详细信息。

这些低级函数从相同名称的 POSIX 函数获得灵感, 但与那些函数并不相同。

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：

<code>O_RDONLY</code>	<code>(0x0000)</code>	<code>/* 打开以进行读取 */</code>
<code>O_WRONLY</code>	<code>(0x0001)</code>	<code>/* 打开以进行写入 */</code>
<code>O_RDWR</code>	<code>(0x0002)</code>	<code>/* 打开以进行读写 */</code>
<code>O_APPEND</code>	<code>(0x0008)</code>	<code>/* 在每次写入时添加 */</code>
<code>O_CREAT</code>	<code>(0x0200)</code>	<code>/* 打开并创建文件 */</code>
<code>O_TRUNC</code>	<code>(0x0400)</code>	<code>/* 打开并截断 */</code>
<code>O_BINARY</code>	<code>(0x8000)</code>	<code>/* 以二进制模式打开 */</code>

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 <i>count</i>)
-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 <i>count</i>)
-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);
    /*-----*/
    /* 将 stderr 作为 MYDEVICE 文件重新打开 */
    /*-----*/
    if (!freopen("mydevice:stderrfile", "w", stderr))
    {
        puts("Failed to freopen stderr");
        exit(EXIT_FAILURE);
    }
    /*-----*/
    /* 不应完全缓冲 stderr: 我们希望错误尽快显现出来。 */
    /* 通常对 stderr 进行缓冲, 但本例中 */
    /* 完全未缓冲 stderr。 这意味着对于消息中的每个字符都会调用一次 */
    /* write() 。 */
    /*-----*/
    if (setvbuf(stderr, NULL, _IONBF, 0))
    {
        puts("Failed to setvbuf stderr");
        exit(EXIT_FAILURE);
    }
    /*-----*/
    /* 尝试一下! */
    /*-----*/
    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 MSP430 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

add_device (continued)

向器件表添加器件
示例

示例 7-2 将执行以下操作：

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

示例 7-2 显示了为 C I/O 添加和使用器件：

示例 7-2. 为 C I/O 器件编程

```
#include <file.h>
#include <stdio.h>
/*****
/* Declarations of the user-defined device drivers          */
*****/
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，例如 `rts430_eabi.lib`。

为了尽可能提高灵活性，各个编译器版本中都提供了运行时支持源代码。您可根据需要构建缺少的库。链接器也可以自动构建缺少的库。这通过新库构建流程来完成的，其核心是从 CCS 5.1 开始提供的可执行的 `mklib`。

7.4.1 所需的非德州仪器 (TI) 软件

若要使用自包含运行时支持构建流程来使用自定义选项重建库，需要以下工具：

- `sh` (Bourne shell)
- `gmake` (GNU make 3.81 或更高版本)

更多相关信息，请访问 GNU 网站 (<http://www.gnu.org/software/make>)。早期版本的 Code Composer Studio 中也提供了 GNU make (`gmake`)。一些适用于 Windows 的 UNIX 支持包中也包含 GNU make，例如 MKS Toolkit、Cygwin 和 Interix。从命令提示窗口执行以下命令时，Windows 平台上使用的 GNU make 应该会明确报告“此程序是为 Windows32 构建的”：

```
gmake -h
```

所有这三个程序都作为 CCS 5.1 的非可选功能提供。如果您使用的是早期版本的 CCS，它们也可以作为可选 XDC 工具功能的一部分获得。

`mklib` 程序会按照以下顺序查找这些可执行文件：

1. 在您的路径中
2. 在 `getenv("CCS_UTILS_DIR")/cygwin` 目录中
3. 在 `getenv("CCS_UTILS_DIR")/bin` 目录中
4. 在 `getenv("XDCROOT")` 目录中
5. 在 `getenv("XDCROOT")/bin` 目录中

如果您从命令行调用 `mklib` 程序，并且这些可执行文件不在您的路径中，则您必须对环境变量 `CCS_UTILS_DIR` 进行设置，以使 `getenv("CCS_UTILS_DIR")/bin` 包含正确的程序。

7.4.2 使用库构建流程

您通常应该让链接器根据需要自动重建库。如有必要，您可以直接调用 `mklib` 来填充库。请参阅节 7.4.2.2，以了解可能需要您这样做的情形。

7.4.2.1 通过链接器自动重建标准库

链接器主要通过 `MSP430_C_DIR` 环境变量查找运行时支持库。通常，`MSP430_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 `MSP430_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=rts430_eabi.lib`），运行时支持函数会精确地查找对应的库。如果没有指定库名称，链接器会使用索引库 `libc.a` 来挑选合适的库。如果通过路径指定了库（例如 `-library=/foo/rts430_eabi.lib`），则会假定对应库已经存在，而不会自动进行构建。

索引库描述了一组具有不同构建属性的库。链接器将会比较每个潜在库的构建属性与应用程序的构建属性，然后挑选最合适的库。有关索引库的详细信息，请参阅《MSP430 汇编语言工具用户指南》中的归档器一章。

现在链接器已经选择好了要使用的库，接下来它会检查 `MSP430_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 `MSP430_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=rts430_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 **rts430_eabi.lib** 库的调试版本，请将工作目录更改为“**lib**”目录并运行以下命令：

```
mklib --pattern=rts430_eabi.lib --name=rts430_dbg.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
<code>--index= filename</code>	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。
<code>--pattern= filename</code>	用于构建库的模式。如果既未指定 <code>--extra_options</code> ，也未指定 <code>--options</code> ，那么该库将为具有对应标准选项的标准库。如果指定了 <code>--extra_options</code> 或 <code>--options</code> ，那么该库为具有自定义选项的自定义库。除非使用了 <code>--all</code> ，否则为必备选项。
<code>--all</code>	一次性构建所有标准库。
<code>--install_to= directory</code>	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。
<code>--compiler_bin_dir= directory</code>	编译器可执行文件所在的目录。直接调用 <code>mklib</code> 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 <code>mklib</code> 这些文件的位置。这个选项主要是在链接器调用 <code>mklib</code> 时使用。
<code>--name= filename</code>	库的文件名且没有目录部分。仅用于自定义库。
<code>--options=' str '</code>	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。
<code>--extra_options=' str '</code>	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。
<code>--list_libraries</code>	列出此脚本能够构建的库并退出。普通系统特有目录。
<code>--log= filename</code>	将构建日志另存为 <code>filename</code> 。
<code>--tmpdir= directory</code>	使用 <code>directory</code> 作为暂存空间，而不是普通系统特有目录。
<code>--gmake= filename</code>	要调用的兼容 Gmake 的程序，而不是 “gmake”
<code>--parallel= N</code>	一次性编译 <code>N</code> 个文件 (“gmake -j N”)。
<code>--query= filename</code>	此脚本是否知道如何构建 FILENAME ?
<code>--help</code> 或 <code>--h</code>	显示此帮助。
<code>--quiet</code> 或 <code>--q</code>	以静默方式运行。
<code>--verbose</code> 或 <code>--v</code>	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=rts430_eabi.lib --index=$C_DIR/lib
```

构建类似 `rts430_eabi.lib` 的自定义库，但启用符号调试支持：

```
mklib --pattern=rts430_eabi.lib --extra_options="-g" --index=$C_DIR/lib --install_to=$Project/Debug  
--name=rts430_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 MSP430_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。

This page intentionally left blank.

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.....	172
8.2 C++ 名称还原器的示例用法.....	172

8.1 调用 C++ 名称还原器

调用 C++ 名称还原器的语法如下：

```
dem430 [options] [filenames]
```

dem430	调用 C++ 名称还原器的命令。
options	影响名称还原器的行为的选项。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 dem430 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，它们的签名信息已编码到它们的名称中。

```
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。

```

PUSH.W   r10
SUB.W    #2,SP
MOV.W    SP,r12
CALL     #_ZN6bananaC1Ev

MOV.W    SP,r12
CALL     #_ZN6banana8caloriesEv

MOV.W    r12,r10
MOV.W    SP,r12
CALL     #_ZN6bananaD1Ev

MOV.W    r10,r12
ADD.W    #2,SP
POP      r10
RET

```

执行 C++ 名称还原器将会还原它认为已改编的所有名称。输入：

```
dem430 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():
;* -----*
    PUSH.W   r10
    SUB.W    #2,SP
    MOV.W    SP,r12
    CALL     #banana::banana()

    MOV.W    SP,r12
    CALL     #banana::calories()

    MOV.W    r12,r10
    MOV.W    SP,r12
    CALL     #banana::~~banana()

    MOV.W    r10,r12
    ADD.W    #2,SP
    POP      r10
    RET

```

This page intentionally left blank.

A.1 术语

绝对列表器	一调试工具，允许创建包含绝对地址的汇编器列表。
别名消歧	一种当两个指针表达式不能指向同一位置的技术，允许编译器自由地优化此类表达式的技术。
别名化	以多种方式访问单个对象的能力，如两个指针指向单个对象时。它会破坏优化，这是因为任何间接引用都可能引用任何其它对象。
分配	链接器计算输出段的最终内存地址的过程。
ANSI	美国国家标准协会；这是一家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 指定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	一种将几个单独文件集合到称为存档库的单个文件的软件程序。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	一种从包含汇编语言指令、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 <code>.cinit</code> 段中) 的过程。
运行时的自动初始化	链接器在链接 C 代码时使用的初始化方法。在使用 <code>--rom_model</code> 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 <code>.cinit</code> 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编号。字中最高有效字节具有较低编号的地址。端排序是特定于硬件的，并在复位时确定。另请参阅 <i>小端</i>
BIS	位指令集。
块	一组用大括号组合在一起并作为实体的语句。
.bss 段	默认目标文件段之一。使用汇编器 <code>.bss</code> 指令在内存映射中保留指定量的空间，用于以后存储数据。 <code>.bss</code> 段未初始化。
字节	根据 ANSI/ISO C，可以容纳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。
代码发生器	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	记录或提高源文件可读性的源语句（或源语句的一部分）。注释不进行编译、汇编或链接；对目标文件没有影响。
编译器程序	一种实用工具，可以一步编辑、汇编和选择性链接。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	链接器已指定用于分配的内存。
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、符号定义在哪行、哪些行引用了符号以及符号的最终值。
.data 段	默认目标文件段之一。 .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 <i>别名消歧</i>
动态内存分配	一种在运行时为变量动态分配内存的几个函数（如 malloc ， calloc 和 realloc ）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 MSP430 运行的硬件开发系统。
入口点	目标内存中开始执行的点。
环境变量	由您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。
结语	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，汇编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，汇编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。
高级语言调试	编译程序保留符号和高级语言信息（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	链接器在链接 C/C++ 代码时使用的自动初始化方法。在使用 <code>--ram_model</code> 链接选项调用链接器时，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++ 语句会被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 <i>C 程序设计语言 (K&R)</i> 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	一种从汇编程序源语句第 1 列开始并对应于该语句的地址的符号。标签是唯一能从第 1 列开始的汇编程序语句。
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编号。字中最高有效字节具有较高编号的地址。端排序是特定于硬件，并在复位时确定。另请参阅大端
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，以便循环的每次迭代出现在您的代码中。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	将源语句插入代码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	一种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、汇编器指示或宏指令的参数，为指令或指示执行的操作提供信息。
优化器	一种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	可执行的链接模块中最终分配段。
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、解释头文件和有条件编译，并对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。
推送	将数据对象放在栈上以进行临时存储的操作。
无声运行	抑制正常横幅和进度信息的选项。
原始数据	输出段中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 <code>rts.src</code> ，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	一种用某个值的符号位填充该值未使用的 MSB 的过程。
模拟器	一种模拟 MSP430 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，该代码被编译或汇编成目标文件。
独立预处理器	一种将宏、 <code>#include</code> 文件和条件编译作为独立程序扩展的软件工具。它还执行集成预处理，包括解析指令。
静态变量	一种其作用域局限于函数或程序的变量。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 8 个字符的符号名称表（八个字符或更长字符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成在单个名称下的集合。
子段	一个可重定位的代码块或数据块，最终将占据内存映射中的连续空间。子段为较大段中的较小段。子段让您可以更严格地控制内存映射。
符号	表示地址或值的字母数字字符串。
符号调试	一种软件工具保留可由仿真器或模拟器等调试工具使用的符号信息的能力。
目标系统	执行开发的目标代码的系统。
.text 段	默认目标文件段之一。 <code>.text</code> 段被初始化并包含可执行代码。可以使用 <code>.text</code> 指令将代码汇编到 <code>.text</code> 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 <code>??'</code> 扩展为 <code>^</code> 。
未配置的内存	未定义为内存映射的一部分且无法加载代码或数据的内存。
未初始化段	在内存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 <code>.bss</code> 和 <code>.usect</code> 指令创建的。
无符号值	无论实际符号是什么都会当作非负数的值。
变量	一种表示数量的符号，该数量可假设任何一组值。
字	目标内存中的 16 位可寻址位置。

Changes from DECEMBER 16, 2020 to JUNE 15, 2021 (from Revision X (December 2020) to Revision Y (June 2021))
Page

• SET_DATA_SECTION pragma 优先于 --gen_data_subsections=on 选项。.....	70
• 添加了 MUST_ITERATE、PROB_ITERATE 和 UNROLL pragma.....	90
• 添加了 MUST_ITERATE pragma。.....	101
• 添加了 PROB_ITERATE pragma。.....	106
• SET_DATA_SECTION pragma 优先于 --gen_data_subsections=on 选项。.....	107
• 添加了 UNROLL pragma。.....	107
• 添加了与 MUST_ITERATE、PROB_ITERATE 和 UNROLL pragma 等效的“for”循环属性。.....	117
• 记录了 __f32_bits_as_u32、__f64_bits_as_u64、__u32_bits_as_f32 和 __u64_bits_as_f64 内在函数。...	136
• Documented the __f32_bits_as_u32, __f64_bits_as_u64, __u32_bits_as_f32, and __u64_bits_as_f64 intrinsics.....	138
• 阐明了有关字符串处理函数的信息。.....	151
• 添加了关于时间和时钟 RTS 函数的信息。.....	151

Changes from SEPTEMBER 1, 2020 to DECEMBER 15, 2020 (from Revision W (September 2020) to Revision X (December 2020))
Page

• 不再支持 MISRA-C 检查.....	19
• 不再支持 MISRA-C 检查.....	27
• 阐明了 --opt_level=4 必须位于 --run_linker 选项之前.....	56
• 不再支持 MISRA-C 检查的 pragma.....	90

Changes from FEBRUARY 28, 2020 to AUGUST 31, 2020 (from Revision V (February 2020) to Revision W (August 2020))
Page

• 更新了整个文档中的表格、图和交叉参考的编号格式.....	9
• 删除了整个文档中对处理器 wiki 的引用.....	9
• 编译器默认使用 --enum_type=unpacked.....	28
• 更正了有关 --gen_data_subsections 选项的默认值的信息。.....	70
• 更新了有关枚举类型大小的信息。.....	83
• 阐明 --opt_level 和 FUNCTION_OPTIONS pragma 之间的交互。.....	99
• 增加了使用位置属性的示例。.....	118
• 结构体和联合体通过引用传递，无论其大小如何。.....	131

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
SLAU132V	链接	节 4.3.7	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。

早期修订版本 (continued)

添加内容的版本	章节	位置	添加/修改/删除
SLAU132V	C/C++ 语言	节 5.11.25	#pragma once 现记录在头文件中使用。
SLAU132V	运行时环境	节 6.9.3.1	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。
SLAU132U		-- 全文 --	更改了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。
SLAU132U	使用编译器	节 2.3 和 节 2.3.2	现在记录了 --gen_func_info_listing 编译器选项。
SLAU132T	使用编译器	节 2.3.1	添加了 --emit_references:file 链接器选项。
SLAU132T	使用编译器	节 2.5.1	记录了支持 C 标准宏命令,例如 __STDC_VERSION__。
SLAU132T	C/C++ 语言	节 5.11.4	添加了 CODE_ALIGN pragma 的文档。
SLAU132T	C/C++ 语言	节 5.11.22	阐明了 NOINIT 和 PERSISTENT pragma 的段放置。
SLAU132T	C/C++ 语言	节 5.11.33	添加了 WEAK pragma 的文档。
SLAU132T	C/C++ 语言	节 5.15.1	更新了 C99 不受支持的运行时函数列表。
SLAU132T	C/C++ 语言	节 5.16.2	添加了 aligned、calls、naked 和 weak 函数属性的文档。
SLAU132T	C/C++ 语言	节 5.16.4	添加了 location 和 packed 变量属性的文档。
SLAU132T	运行时支持函数	DEV_lseek 主题	更正了 DEV_lseek 函数的语法记录。
SLAU132S	简介, 使用编译器, C/C++ 语言	节 1.3 、 节 2.3 、 节 5.1 和 节 5.15.2	添加了对 C11 的支持。
SLAU132S	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。
SLAU132S	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。
SLAU132S	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。
SLAU132S	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,因为有多版本已支持这些功能。
SLAU132S	C/C++ 语言	节 5.6	添加了有关字符集和文件编码的信息。
SLAU132S	C/C++ 语言	节 5.16.2 和 节 5.16.4	添加了 "retain" 作为函数属性和变量属性。
SLAU132R	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。
SLAU132R	C/C++ 语言	节 5.16	编译器现在支持多个 Clang __has_ 宏命令扩展。
SLAU132R	C/C++ 语言	节 5.16.1	现在支持包装器头文件 GCC 扩展 (#include_next)。
SLAU132Q	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。
SLAU132Q	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。
SLAU132P	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。
SLAU132O	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。
SLAU132O	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。
SLAU132O	运行时环境	节 6.1.3	指明某些 MSP430 器件确实具有超出 64K 边界的可写存储器。
SLAU132N	优化	节 3.7.3	更正了 _TI_start_pprof_collection() 和 _TI_stop_pprof_collection() 的函数名称。
SLAU132N	C/C++ 语言	节 5.16.2	添加了 call_conv 函数属性以允许 IAR 和 TI 编译器链接到同一个 ROM 映像。

早期修订版本 (continued)

添加内容的版本	章节	位置	添加/修改/删除
SLAU132M	使用编译器	节 2.3	添加了 <code>--align_for_power</code> 选项。
SLAU132M	使用编译器	节 2.3	<code>--cinit_compression</code> 和 <code>--copy_compression</code> 的默认值已从 RLE 更改为 LZSS。
SLAU132M	使用编译器	节 2.3	现在记录了可用于 <code>--silicon_errata</code> 选项的设置。
SLAU132M	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。
SLAU132L	使用编译器	节 2.3 和节 4.2.2	添加了 <code>--gen_data_subsections</code> 选项。
SLAU132L	使用编译器	节 2.3.5	添加了 <code>--symdebug:dwarf_version</code> 编译器选项。此选项设置使用的 DWARF 调试格式版本。
SLAU132L	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。
SLAU132L	C/C++ 语言	节 5.11.2	添加了 <code>CALLS pragma</code> 以指定一组可从指定调用函数间接调用的函数。使用此 <code>pragma</code> 能够将此类间接调用包含在函数的 <code>inclusive</code> 栈大小的计算中。
SLAU132L	运行时环境	节 6.8	为短整数和长整数的饱和加减添加了内在函数。
SLAU132L	运行时环境	节 6.9.1	提供了额外的引导挂钩函数。这些可以定制以在系统初始化期间使用。
SLAU132K	引言	节 1.4	不再支持 COFF 目标文件格式。MSP430 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中提及 COFF 格式的各段。如果希望生成 COFF 输出文件，请使用 v4.4 的 MSP430 代码生成工具，并参考 SLAU132J 文档。 弃用了 <code>--abi=coff</code> 、 <code>--symdebug:profile_coff</code> 、 <code>--no_sym_merge</code> 和 <code>--diablib_clink</code> 选项。
SLAU132K	使用编译器	节 2.3.4	添加了 <code>--ramfunc</code> 选项。如果设置此选项，则将所有函数放在 RAM 中。
SLAU132K	C/C++ 语言	节 5.16.2	添加了 <code>ramfunc</code> 函数属性。它规定了一个函数应该放置在 RAM 中。
SLAU132K	运行时环境	节 6.1.5	更正了文档以说明 R1 是栈指针。
SLAU132I	引言	节 1.3	添加了对 C99 和 C++03 的支持。
SLAU132I	使用编译器	表 2-7 和节 5.4	添加了 <code>--advice:hw_config</code> 选项以便 ULP Advisor 检查器件配置。
SLAU132I	使用编译器	表 2-9	添加了对 C99 和 C++03 的支持。弃用了 <code>-gcc</code> 选项。 <code>--relaxed_ansi</code> 现在是默认选项。
SLAU132I	使用编译器	表 2-9	删除了 MSP430 不支持的预编译标头的文档。
SLAU132I	使用编译器	表 2-12 和节 2.7.1	添加了 <code>--section_sizes</code> 选项，用于段大小的诊断报告。
SLAU132J	使用编译器	表 2-14 和节 2.3.4	添加了 <code>--disable_interrupts_around_hw_mpy</code> 选项。
SLAU132J	使用编译器	表 2-14 和节 2.3.4	<code>--use_hw_mpy</code> 选项现在是一个编译器选项，并在编译时执行其他操作。
SLAU132J	使用编译器	节 2.5.3	记录了对 <code>#warning</code> 和 <code>#warn</code> 预处理器指令的支持。
SLAU132J	使用编译器	节 2.6	添加了有关向 <code>main()</code> 传递参数的技术的段。
SLAU132I	使用编译器	节 2.11	记录了 <code>inline</code> 关键字现在在除 C89 严格 ANSI 模式之外的所有模式中都启用。
SLAU132J	C/C++ 语言	节 5.1.1	添加了记录实现定义行为的段。
SLAU132I	C/C++ 语言	节 5.3、节 5.11.3 和节 5.11.28	添加了用于启用/重置 ULP Advisor 规则检查的 <code>pragma</code> 。
SLAU132J	C/C++ 语言	节 5.5.1	添加了有关枚举类型大小的文档。
SLAU132J	C/C++ 语言	节 5.11.19、节 5.11.29 和节 5.16.2	为 <code>INTERRUPT</code> 和 <code>RETAIN pragma</code> 添加了 C++ 语法。还从 <code>#pragma</code> 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。
SLAU132J	C/C++ 语言	节 5.11.8	添加了 <code>diag_push</code> 和 <code>diag_pop</code> 诊断消息 <code>pragma</code> 。
SLAU132I	C/C++ 语言	节 5.15、节 5.15.1 和节 5.15.3	添加了对 C99 和 C++03 的支持。 <code>--relaxed_ansi</code> 现在是默认选项， <code>--strict_ansi</code> 是另一个选项；标准违反严格性的“正常模式”不再可用。
SLAU132J	运行时环境	节 6.5	添加了对 <i>汇编语言工具用户指南</i> 中有关在 C 和 C++ 语言中访问链接器符号一节的引用。

早期修订版本 (continued)

添加内容的版本	章节	位置	添加/修改/删除
SLAU132I	运行时环境	节 6.8.1 和 节 6.8.5	更正了：__never_executed 内在函数有两个下划线作为前缀。
SLAU132J	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。
SLAU132J	C++ 名称还原器	节 8.1	更正了有关名称还原器选项的信息。

This page intentionally left blank.

重要声明和免责声明

TI 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安全、安保或其他要求。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 (<https://www.ti.com/legal/termsofsale.html>) 或 [ti.com](https://www.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。

邮寄地址: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2021, 德州仪器 (TI) 公司

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

Copyright © 2022，德州仪器 (TI) 公司