ARM 汇编语言工具 v20.2.0.LTS

User’s Guide

DRAFT ONLY
TI Confidential – NDA Restrictions

Literature Number: ZHCUAV7Z
SEPTEMBER 1995 – REVISED MARCH 2023

DRAFT ONLY
TI Confidential – NDA Restrictions

目录

TEXAS INSTRUMENTS

请先阅读......11
关于本手册......11
如何使用本手册......11
命名规则......12
德州仪器 (TI) 提供的相关文档......13
商标......13
1 软件开发工具简介......15
1.1 软件开发工具概述......16
1.2 工具说明......17
2 目标模块简介......19
2.1 目标文件格式规范......20
2.2 可执行目标文件......20
2.3 段简介......20
2.3.1 特殊段名......21
2.4 汇编器如何处理段......21
2.4.1 未初始化的段......21
2.4.2 已初始化段......22
2.4.3 用户命名的段......23
2.4.4 当前段......23
2.4.5 段程序计数器......23
2.4.6 子段......23
2.4.7 使用 Sections 指令......24
2.5 链接器如何处理段......26
2.5.1 合并输入段......26
2.5.2 放置段......27
2.6 符号......27
2.6.1 全局（外部）符号......28
2.6.2 局部符号......28
2.6.3 弱符号......28
2.6.4 符号表......29
2.7 符号重定位......29
2.8 加载程序......30
3 程序加载和运行......31
3.1 负载......32
3.1.1 加载和运行地址......32
3.1.2 引导加载......33
3.2 入口点......36
3.3 运行时初始化......37
3.3.1 _c_int00 函数......37
3.3.2 RAM 模型与 ROM 模型......37
3.3.3 关于链接器生成的复制表......39
3.4 main 的参数......39
3.5 运行时重定位......39
3.6 其他信息......39
4 汇编器说明......41
4.1 汇编器概述......42

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

4.2 汇编器在软件开发流程中的作用....42
4.3 调用汇编器....43
4.4 控制应用二进制接口....43
4.5 指定用于汇编器输入的备用目录....44
4.5.1 使用 --include_path 汇编器选项....44
4.5.2 使用 TI_ARM_A_DIR 环境变量....45
4.6 源语句格式....46
4.6.1 标签字段....46
4.6.2 助记符字段....47
4.6.3 操作数字段....47
4.6.4 注释字段....49
4.7 字面常量....49
4.7.1 整数字面量....49
4.7.2 字符串字面量....51
4.7.3 浮点字面量....51
4.8 汇编器符号....51
4.8.1 标识符....52
4.8.2 标签....52
4.8.3 局部标签....52
4.8.4 符号常数....53
4.8.5 定义符号常量（--asm_define 选项）....54
4.8.6 预定义符号常量....54
4.8.7 寄存器....55
4.8.8 替代符号....56
4.9 表达式....57
4.9.1 数学和逻辑运算符....57
4.9.2 关系运算符和条件表达式....59
4.9.3 明确定义的表达式....59
4.9.4 可重定位的符号和合法表达式....59
4.9.5 表达式示例....60
4.10 内置函数和运算符....61
4.10.1 内置数学和三角函数....61
4.11 统一汇编语言语法支持....62
4.12 源程序列表....62
4.13 调试汇编源文件....65
4.14 交叉引用列表....66
5 汇编器指令....67
5.1 指令摘要....68
5.2 用于定义段的指令....72
5.3 用于更改指令类型的指令....75
5.4 用于初始化值的指令....75
5.5 执行对齐和保留空间的指令....77
5.6 用于设置输出列表格式的指令....79
5.7 用于引用其他文件的指令....80
5.8 用于启用条件汇编的指令....80
5.9 用于定义联合体或结构体类型的指令....81
5.10 用于定义枚举类型的指令....81
5.11 在汇编时用于定义符号的指令....81
5.12 其他命令....82
5.13 指令参考....83
6 宏语言说明....153
6.1 使用宏....154
6.2 定义宏....154
6.3 宏参数/替代符号....156
6.3.1 用于定义替代符号的指令....157
6.3.2 内置替代符号函数....158

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

6.3.3 递归替代符号 159
6.3.4 强制替代 159
6.3.5 访问带下标的替代符号的各个字符 159
6.3.6 替代符号作为宏中的局部变量 160
6.4 宏库 160
6.5 在宏中使用条件汇编 161
6.6 在宏中使用标签 163
6.7 在宏中生成消息 164
6.8 使用指令设置输出列表的格式 165
6.9 使用递归和嵌套宏 166
6.10 宏指令摘要 167
7 归档器说明 169
7.1 归档器概述 170
7.2 归档器在软件开发流程中的作用 170
7.3 调用归档器 171
7.4 归档器示例 171
7.5 库信息归档器说明 173
7.5.1 调用库信息归档器 173
7.5.2 库信息归档器示例 173
7.5.3 列出索引库的内容 174
7.5.4 要求 174
8 链接器说明 175
8.1 链接器概述 176
8.2 链接器在软件开发流程中的作用 176
8.3 调用链接器 177
8.4 链接器选项 178
8.4.1 文件、段和符号模式中的通配符 180
8.4.2 通过链接器选项指定 C/C++ 符号 180
8.4.3 重定位功能（--absolute_exe 和 --relocatable 选项） 180
8.4.4 分配存储器供加载器使用以传递参数（--arg_size 选项） 181
8.4.5 更改大端字节序指令的编码 181
8.4.6 压缩（--cinit_compression 和 --copy_compression 选项） 181
8.4.7 压缩 DWARF 信息（--compress_dwarf 选项） 182
8.4.8 控制链接器诊断 182
8.4.9 自动选择库（--disable_auto_rts 选项） 183
8.4.10 不要删除未使用的段（--unused_section_elimination 选项） 183
8.4.11 链接器命令文件预处理（--disable_pp、--define 和 --undefine 选项） 183
8.4.12 纠错码测试（--ecc 选项） 185
8.4.13 定义入口点（--entry_point 选项） 185
8.4.14 设置默认填充值（--fill_value 选项） 186
8.4.15 生成死函数列表（--generate_dead_funcs_list 选项） 186
8.4.16 定义堆大小（--heap_size 选项） 186
8.4.17 隐藏符号 186
8.4.18 改变库搜索算法（--library、--search_path 和 TI_ARM_C_DIR） 187
8.4.19 更改符号局部化 189
8.4.20 创建映射文件（--map_file 选项） 190
8.4.21 管理映射文件内容（--mapfile_contents 选项） 191
8.4.22 禁用名称还原 (--no_demangle) 192
8.4.23 禁止合并符号调试信息（--no_sym_merge 选项） 192
8.4.24 去除符号信息（--no_symtable 选项） 193
8.4.25 指定输出模块（--output_file 选项） 193
8.4.26 确定函数放置优先级（--preferred_order 选项） 193
8.4.27 C 语言选项（--ram_model 和 --rom_model 选项） 193
8.4.28 保留丢弃的段（--retain 选项） 194
8.4.29 创建绝对列表文件（--run_abs 选项） 194
8.4.30 扫描所有库中的重复符号定义（--scan_libraries） 194

Copyright © 2023 Texas Instruments Incorporated

8.4.31 定义栈大小（--stack_size 选项）......194
8.4.32 符号映射（--symbol_map 选项）......195
8.4.33 生成 Far 调用 Trampoline（--trampolines 选项）......195
8.4.34 引入未解析的符号（--undef_sym 选项）......197
8.4.35 创建未定义的输出段时显示一条消息 (--warn_sections)......197
8.4.36 生成 XML 链接信息文件（--xml_link_info 选项）......197
8.4.37 零初始化（--zero_init 选项）......197
8.5 链接器命令文件......199
8.5.1 链接器命令文件中的保留名称......200
8.5.2 链接器命令文件中的常量......200
8.5.3 从链接器命令文件访问文件和库......200
8.5.4 MEMORY 指令......202
8.5.5 SECTIONS 指令......205
8.5.6 在不同的加载和运行地址放置段......216
8.5.7 使用 GROUP 和 UNION 语句......219
8.5.8 特殊段类型（DSECT、COPY、NOLOAD 和 NOINIT）......222
8.5.9 使用链接器配置纠错码 (ECC)......223
8.5.10 在链接时分配符号......225
8.5.11 创建和填充孔洞......231
8.6 链接器符号......233
8.6.1 链接器定义的函数和数组......234
8.6.2 链接器定义的整数值......234
8.6.3 链接器定义的地址......234
8.6.4 有关 _symval 运算符的更多信息......234
8.6.5 弱符号......235
8.6.6 利用对象库解析符号......236
8.7 默认放置算法......238
8.7.1 分配算法如何创建输出段......238
8.7.2 减少存储器碎片......239
8.8 使用由链接器生成的复制表......239
8.8.1 使用复制表进行引导加载......239
8.8.2 在复制表中使用内置链接运算符......240
8.8.3 重叠管理示例......240
8.8.4 使用 table() 运算符生成复制表......241
8.8.5 压缩......244
8.8.6 复制表内容......248
8.8.7 通用复制例程......249
8.9 由链接器生成的 CRC 表......250
8.9.1 在 SECTIONS 指令中使用 crc_table() 运算符......250
8.9.2 关于 TMS570_CRC64_ISO 算法的注意事项......254
8.10 部分（增量）链接......254
8.11 链接 C/C++ 代码......255
8.11.1 运行时初始化......255
8.11.2 对象库和运行时支持......256
8.11.3 设置堆栈段的大小......256
8.11.4 在运行时初始化和自动初始化变量......256
8.11.5 Cinit 的初始化和看门狗计时器保持......256
8.12 链接器示例......256
9 绝对列表器说明......261
9.1 生成绝对列表......262
9.2 调用绝对列表器......263
9.3 绝对列表器示例......264
10 交叉参考列表器说明......267
10.1 生成交叉参考列表......268
10.2 调用交叉参考列表器......269
10.3 交叉参考列表示例......270

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

11 目标文件实用程序......271
11.1 调用目标文件显示实用程序......272
11.2 调用反汇编器......272
示例 11-1. 目标文件 memcpy32.asm......274
示例 11-2. 从 memcpy32.asm 进行反汇编......274
示例 11-3. 具有不同加载和运行地址的部分复制记录输出......274
11.3 调用名称实用程序......274
11.4 调用符号去除实用程序......275
12 十六进制转换实用程序说明......277
12.1 软件开发流程中十六进制转换实用程序的作用......278
12.2 调用十六进制转换实用程序......279
12.2.1 从命令行调用十六进制转换实用程序......280
12.2.2 使用命令文件调用十六进制转换实用程序......282
12.3 理解存储器宽度......282
12.3.1 目标宽度......283
12.3.2 指定存储器宽度......283
12.3.3 将数据分入输出文件......285
12.4 ROMS 指令......287
12.4.1 何时使用 ROMS 指令......287
12.4.2 ROMS 指令的示例......288
12.5 SECTIONS 指令......290
12.6 加载映像格式（--load_image 选项）......290
12.6.1 加载映像段形成......291
12.6.2 加载映像特性......291
12.7 排除指定段......291
12.8 分配输出文件名......292
12.9 映像模式和 --fill 选项......293
12.9.1 生成存储器映像......293
12.9.2 指定填充值......293
12.9.3 使用映像模式的步骤......294
12.10 数组输出格式......294
12.11 为片上引导加载程序编译引导表......295
12.11.1 引导表说明......295
12.11.2 引导表格式......295
12.11.3 如何构建引导表......295
12.11.4 从器件外设进行引导......296
12.11.5 设置引导表的进入点......296
12.11.6 使用 ARM 引导加载程序......296
12.12 在 TMS320F2838x 器件上使用安全闪存引导功能......300
12.13 控制 ROM 器件地址......301
12.14 控制十六进制转换实用程序诊断......302
12.15 目标格式说明......303
12.15.1 ASCII 十六进制对象格式（--ascii 选项）......303
12.15.2 Intel MCS-86 目标格式（--intel 选项）......304
12.15.3 Motorola Exorciser 对象格式（--motorola 选项）......305
12.15.4 扩展的 Tektronix 目标格式（--tektronix 选项）......306
12.15.5 德州仪器 (TI) SDSMAC (TI-Tagged) 目标格式（--ti_tagged 选项）......307
12.15.6 TI-TXT 十六进制格式（--ti_txt 选项）......308
13 与汇编源代码共享 C/C++ 头文件......311
13.1 .cdecls 指令概述......312
13.2 C/C++ 转换注意事项......313
13.2.1 说明......313
13.2.2 条件编译（#if/#else/#ifdef/等）......313
13.2.3 Pragma......313
13.2.4 #error 和 #warning 指令......313
13.2.5 预定义符号 __ASM_HEADER__......313

Copyright © 2023 Texas Instruments Incorporated

13.2.6 在 C/C++ asm( ) 语句中的用法......313
13.2.7 #include 指令......313
13.2.8 转换 #define 宏......314
13.2.9 #undef 指令......314
13.2.10 枚举......315
13.2.11 C 字符串......315
13.2.12 C/C++ 内置函数......315
13.2.13 结构体和联合体......315
13.2.14 函数/变量原型......316
13.2.15 C 常量后缀......316
13.2.16 基本 C/C++ 类型......316
13.3 C++ 专有转换注意事项......316
13.3.1 名称改编......316
13.3.2 衍生类......317
13.3.3 模板......317
13.3.4 虚拟函数......317
13.4 汇编器特殊支持......318
13.4.1 枚举 (.enum/.emember/.endenum)......318
13.4.2 .define 指令......318
13.4.3 .undefine/.unasg 指令......318
13.4.4 $$defined( ) 内置函数......319
13.4.5 $$sizeof 内置函数......319
13.4.6 结构体/联合体对齐和 $$alignof( )......319
13.4.7 .cstring 指令......319
A 符号调试指令......321
A.1 DWARF 调试格式......322
A.2 调试指令语法......322
B XML 链接信息文件说明......323
B.1 XML 信息文件元素类型......324
B.2 文档元素......324
B.2.1 标头元素......324
B.2.2 输入文件列表......325
B.2.3 对象组件列表......326
B.2.4 逻辑组列表......327
B.2.5 放置映射......329
B.2.6 Far Call Trampoline 列表......330
B.2.7 符号表......331
C 十六进制转换实用程序示例......333
C.1 场景 1 -- 为单个 8 位 EPROM 构建十六进制转换命令文件......334
示例 C-1. 场景 1 的链接器命令文件和链接映射......335
示例 C-2. 场景 1 的十六进制转换命令文件......336
示例 C-3. 十六进制映射文件 example1.mxp 的内容......337
C.2 场景 2 -- 为 16-BIS 代码构建十六进制转换命令文件......338
示例 C-4. 场景 2 的链接器命令文件......339
示例 C-5. 场景 2 的十六进制转换命令文件......340
示例 C-6. 十六进制映射文件 example2.mxp 的内容......340
C.3 场景 3 -- 为两个 8 位 EPROM 构建十六进制转换命令文件......341
示例 C-7. 场景 3 的链接器命令文件......341
示例 C-8. 场景 3 的十六进制转换命令文件......342
示例 C-9. 十六进制映射文件 example3.mxp 的内容......343
D 术语表......345
D.1 术语......345
E 修订历史记录......351

插图清单

图 1-1. ARM 器件 软件开发流程......16

Copyright © 2023 Texas Instruments Incorporated

图 2-1. 存储器信息逻辑块分区......21
图 2-2. 使用 Sections 指令示例......25
图 2-3. 图 2-2 中由文件生成的目标代码......26
图 2-4. 组合输入段以构成可执行对象模块......27
图 3-1. 引导加载序列（简化）......33
图 3-2. 使用次级引导加载程序的引导加载序列......34
图 3-3. 运行时自动初始化......38
图 3-4. 加载时初始化......38
图 4-1. ARM 软件开发流程中的汇编器......42
图 4-2. 汇编列表示例......63
图 4-3. 汇编列表示例（续）......64
图 5-1. .field 指令......76
图 5-2. 初始化指令......77
图 5-3. .align 指令......78
图 5-4. .space 和 .bes 指令......79
图 5-5. 双精度浮点格式......99
图 5-6. .field 指令......107
图 5-7. 单精度浮点格式......108
图 5-8. .usect 指令......149
图 7-1. ARM 软件开发流程中的归档器......170
图 8-1. ARM 软件开发流程中的链接器......176
图 8-2. 由 SECTIONS 指令定义的段放置示例......206
图 8-3. 在运行时将一个函数从慢速存储器移动到快速存储器 的运行时执行......219
图 8-4. 存储器分配如 UNION 语句和 UNION 段的单独加载地址所示......220
图 8-5. 压缩的复制表......244
图 8-6. 处理程序表......245
图 8-7. CRC_TABLE 概念模型......252
图 9-1. 绝对列表器开发流程......262
图 10-1. 交叉参考列表器开发流程......268
图 12-1. ARM 软件开发流程中的十六进制转换实用程序......278
图 12-2. 十六进制转换实用程序处理流程......283
图 12-3. 目标文件数据与存储器宽度......284
图 12-4. 数据、存储器和 ROM 宽度......286
图 12-5. infile.out 文件分区为四个输出文件......288
图 12-6. 从 8 位 SPI Boot 引导的示例十六进制转换器输出文件......298
图 12-7. ARM 16 位并行引导 GP I/O 的示例十六进制转换输出文件......299
图 12-8. ASCII 十六进制对象格式......303
图 12-9. Intel 十六进制目标格式......304
图 12-10. Motorola-S 格式......305
图 12-11. 扩展的 Tektronix 目标格式......306
图 12-12. TI-Tagged 目标格式......307
图 12-13. TI-TXT 目标格式......308
图 C-1. 场景 1 的 EPROM 存储器系统......334
图 C-2. 十六进制输出文件 example1.hex 的内容......337
图 C-3. 场景 2 的 EPROM 存储器系统......338
图 C-4. 十六进制输出文件 example2.hex 的内容......340
图 C-5. 场景 3 的 EPROM 存储器系统......341
图 C-6. 十六进制输出文件 lower16.bit 的内容......343
图 C-7. 十六进制输出文件 upper16.bit 的内容......343

表格清单

表 4-1. ARM 汇编器选项......43
表 4-2. ARM 处理器符号常量......55
表 4-3. ARM 寄存器符号与别名......55
表 4-4. ARM 状态寄存器和别名......56
表 4-5. 表达式中使用的运算符（优先级）......58

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

表 4-6. 具有绝对符号和可重定位符号的表达式......59
表 4-7. 内置数学函数......61
表 4-8. 符号属性......66
表 5-1. 供控制段使用的指令......68
表 5-2. 用于将段收集到通用组中的指令......68
表 5-3. 会影响未使用段消除的指令......68
表 5-4. 用于初始化值（数据和存储器）的指令......68
表 5-5. 用于执行对齐和保留空间的指令......69
表 5-6. 用于更改指令类型的指令......69
表 5-7. 用于设置输出列表格式的指令......69
表 5-8. 用于引用其他文件的指令......69
表 5-9. 会影响符号链接和可见性的指令......70
表 5-10. 定义符号的指令......70
表 5-11. 用于启用条件汇编的指令......70
表 5-12. 用于定义联合体或结构体类型的指令......70
表 5-13. 用于创建或影响宏的指令......71
表 5-14. 用于控制诊断的指令......71
表 5-15. 用于执行汇编源代码调试的指令......71
表 5-16. 供绝对列表器使用的指令......71
表 5-17. 用于执行其他函数的指令......71
表 6-1. 替代符号函数和返回值......158
表 6-2. 创建宏......167
表 6-3. 操作替代符号......167
表 6-4. 条件汇编......167
表 6-5. 生成汇编时消息......167
表 6-6. 格式化列表......167
表 8-1. 基本选项汇总......178
表 8-2. 文件搜索路径选项汇总......178
表 8-3. 命令文件预处理选项汇总......178
表 8-4. 诊断选项汇总......178
表 8-5. 链接器输出选项汇总......178
表 8-6. 符号管理选项汇总......179
表 8-7. 运行时环境选项汇总......179
表 8-8. 链接时优化选项汇总......179
表 8-9. 其他选项汇总......180
表 8-10. 预定义的 ARM 宏名称......184
表 8-11. 表达式中使用的运算符组（优先级）......227
表 10-1. 交叉参考列表中的符号属性......270
表 12-1. 基本十六进制转换实用程序选项......280
表 12-2. 引导加载程序选项......295
表 12-3. 引导表源格式......296
表 12-4. 引导表格式......297
表 12-5. 用于指定十六进制转换格式的选项......303
表 A-1. 符号调试指令......322

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

Preface
请先阅读

关于本手册

ARM 汇编语言工具用户指南 介绍了如何使用以下德州仪器 (TI) 代码生成目标文件工具：

- 汇编器
- 归档器
- 链接器
- 库信息归档器
- 绝对列表器
- 交叉引用列表器
- 反汇编器
- 目标文件显示实用程序
- 名称实用程序
- 符号去除实用程序
- 十六进制转换实用程序

如何使用本手册

本手册将帮助您了解如何使用德州仪器 (TI) 目标文件和专为 ARM® 32 位 器件设计的汇编语言工具。本手册包含四个部分：

- **介绍性信息**，包括章节 1 至章节 3，概述了目标文件和汇编语言开发工具。尤其是章节 2，介绍了目标模块以及如何管理它们以为 ARM 应用加载和运行提供帮助。强烈建议开发人员在使用汇编器和链接器之前，先熟悉什么是目标模块以及如何使用它们。
- **汇编器说明**，包括章节 4 至章节 6，其中包含有关使用汇编器的详细信息。章节 4 和章节 5 介绍了如何调用汇编器，并讨论了源语句格式、有效常量和表达式、汇编器输出和汇编器指令。章节 6 重点介绍了宏语言。
- **链接器和其他目标文件工具说明**，包括章节 7 至章节 12，详细描述了与汇编器一同提供的可为创建可执行目标文件提供帮助的每个工具。章节 7 提供了有关使用归档器创建对象库的详细信息。章节 8 介绍了链接器的调用方法、链接器的运行方式以及链接器指令的使用方法。章节 11 对一些目标文件实用程序进行了简要概述，这些实用程序在检查目标文件内容方面，以及删除符号和调试信息以减小给定目标文件的大小方面均非常有用。章节 12 介绍了如何使用十六进制转换实用程序。
- **其他参考资料**，包括附录 A 至附录 D，提供包含 ARM C/C++ 编译器使用的符号调试指令在内的补充信息。它还提供了十六进制实用程序示例。此外还提供了 XML 链接信息文件的说明以及一个术语表。

DRAFT ONLY
TI Confidential—NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

命名规则

本文档使用以下惯例：

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

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

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

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

  --library= *libraryname*

- 在汇编器语法语句中，最左侧字符位置（列 1）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、指示或参数都不能从列 1 开始。

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

- 有些指令的参数数量可变。例如，.byte 指令可以具有多个参数。此语法显示为 [, ..., *parameter*]。

.byte $parameter_1$[, ..., $parameter_n$]

- TMS470 和 TMS570 器件统称为 ARM。
- ARM 16 位指令集被称为 16-BIS。
- ARM 32 位指令集被称为 32-BIS。
- 本文档中使用的其他符号和缩写词包括：

符号	定义
B、b	后缀 — 二进制整数
H、h	后缀 — 十六进制整数
LSB	最低有效位
MSB	最高有效位
0x	前缀 — 十六进制整数
Q、q	后缀 — 八进制整数

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

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 社区：软件工具论坛

以下书籍可以作为本用户指南的补充：

SPNU151 **ARM 优化 C/C++ 编译器用户指南。**描述了 ARM C/C++ 编译器。此 C/C++ 编译器支持 ANSI 标准 C/C++ 源代码，并可为 ARM 平台器件生成汇编语言源代码。

SPNU134 **TMS470R1x 用户指南。**介绍了 TMS470R1x RISC 微控制器、其架构（包括寄存器）、ICEBreaker 模块、接口（存储器、协处理器和调试器）、16 位和 32 位指令集以及电气规范。

商标

ARM® is a registered trademark of ARM Limited.
所有商标均为其各自所有者的财产。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

This page intentionally left blank.

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

章节 1

软件开发工具简介

ARM® 具有一系列配套的软件开发工具，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。本章将概述这些工具。

以下汇编语言开发工具均支持 ARM device：

- 汇编器
- 归档器
- 链接器
- 库信息归档器
- 绝对列表器
- 交叉参考列表器
- 目标文件显示实用程序
- 反汇编器
- 命名实用程序
- 符号去除实用程序
- 十六进制转换实用程序

本章介绍了这些工具如何融入一般软件工具开发流程，并简要说明了每个工具。为了方便起见，本章还汇总了 C/C++ 编译器和调试工具。有关编译器和调试器的详细信息，以及有关 ARM 器件的完整说明，请参阅*德州仪器 (TI) 相关文档*中列出的书籍。

1.1 软件开发工具概述..16
1.2 工具说明..17

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

1.1 软件开发工具概述

图 1-1 展示了 ARM 器件 的软件开发流程。阴影部分突出显示了最常用的开发路径；其他部分是可选的。其他部分是用于增强开发流程的外围功能。

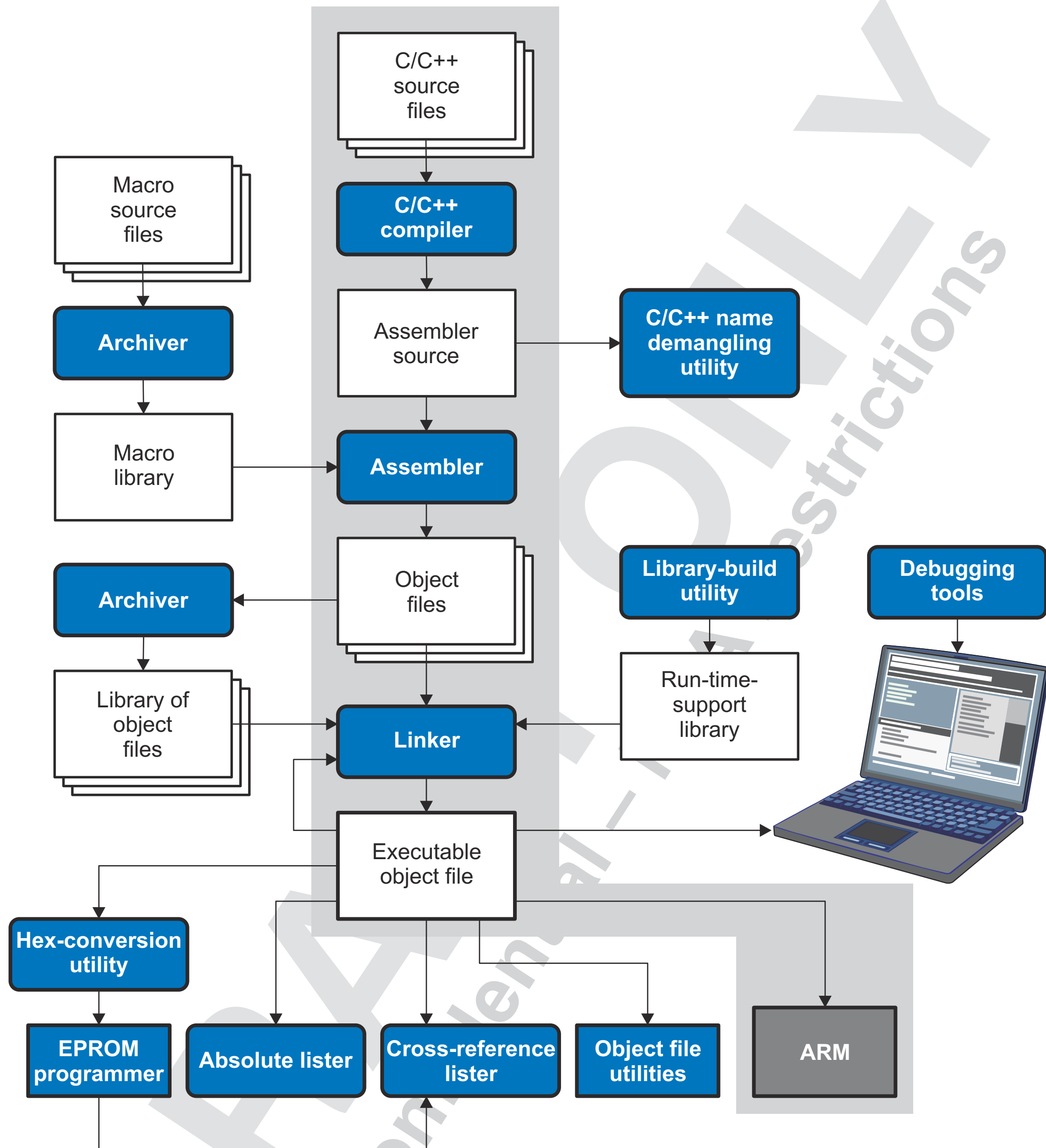


图 1-1. ARM 器件 软件开发流程

Copyright © 2023 Texas Instruments Incorporated

1.2 工具说明

以下列表描述了图 1-1 中显示的工具：

- **C/C++ 编译器**接受 C/C++ 源代码并生成 ARM 机器码目标模块。如需更多信息，请参阅 *《ARM 优化 C/C++ 编译器用户指南》*。安装时会包含 **shell** 程序、优化器和 **interlist** 实用程序：
  - shell 程序只需一步即可编译、汇编和链接源代码模块。
  - 优化器可修改代码，以提升 C/C++ 程序的效率。
  - interlist 实用程序可将 C/C++ 源语句插入到汇编语言输出中，以将由编译器生成的代码与源代码相关联。
- **汇编器**将汇编语言源文件转换为机器语言目标模块。源文件可以包含指令、汇编器指令和宏指令。您可以使用汇编器指令控制汇编过程，包括源代码列表格式、数据对齐和段内容。请参阅章节 4 至章节 6。有关汇编语言指令集的详细信息，请参阅 *《TMS470R1x 用户指南》*。
- **链接器**将目标文件组合为单个可执行目标模块。它会执行符号重定位并解析外部引用。链接器接受可重定位的目标模块（由汇编器创建）作为输入。它还接受通过之前运行链接器创建的归档器库成员和输出模块。链接器指令可组合目标文件段、将段或符号与地址绑定或绑定到存储器范围中，以及定义全局符号。请参阅章节 8。
- **归档器**可将一组文件收集到被称为库的单个存档文件中。归档器最常见的用途是将一组目标文件收集到一个目标库中。链接器在链接时会提取目标库成员，以解析外部引用。您也可以使用归档器将多个宏收集到一个宏库中。汇编器会搜索库并使用被源文件作为宏调用的成员。归档器允许通过删除、替换、提取或添加成员来修改库。请参阅节 7.1。
- **库信息归档器**可用于创建多个目标文件库变体的索引库，如果库的多个变体有不同的可用选项，此功能很有用。您可以不必引用具体库，而是链接索引库，链接器将从索引库中选择最佳匹配项。如需有关使用归档器来管理库内容的更多信息，请参阅节 7.5。
- 可以使用**库编译实用程序**编译您自有的自定义运行时支持库。如需更多信息，请参阅 *《ARM 优化 C/C++ 编译器用户指南》*。
- **十六进制转换实用程序**将目标文件转换为 TI-Tagged、ASCII-Hex、Intel、Motorola-S 或 Tektronix 目标格式。转换后的文件可下载到 EPROM 编程器。请参阅章节 12。
- **绝对列表器**使用链接的目标文件来创建 .abs 文件。这些文件经汇编后可生成目标代码绝对地址的列表。请参阅章节 9。
- **交叉引用列表器**使用目标文件生成交叉引用列表，展示符号、符号定义以及符号在链接的源文件中的引用。请参阅章节 10。
- 此开发流程的主要产品是一个可在 **ARM** 器件上执行的可执行目标文件。可使用多种调试工具之一来改进和更正代码。 可用产品包括：
  - 指令精确和时钟精确的软件仿真器
  - XDS 仿真器

此外，还提供了以下实用程序来帮助检查或管理给定目标文件的内容：

- **目标文件显示实用程序**以可读的或 XML 格式打印目标文件和目标库的内容。请参阅节 11.1。
- **反汇编器**解码目标模块中的机器代码以显示其所表示的汇编指令。请参阅节 11.2。
- **名称实用程序**打印目标文件或目标存档中定义或引用的目标和函数的符号名称列表。请参阅节 11.3。
- **符号去除实用程序**从目标文件和目标库中删除符号表和调试信息。请参阅节 11.4。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

DRAFT ONLY
TI Confidential – NDA Restrictions

This page intentionally left blank.

Copyright © 2023 Texas Instruments Incorporated

章节 2

目标模块简介

汇编器根据汇编代码创建目标模块，而链接器根据目标模块创建可执行目标文件。这些可执行目标文件可以通过 ARM 器件执行。

建议您在编写汇编语言程序时从代码*块* 和数据块的角度去考虑目标模块，这样便可以更轻松地进行模块化编程。这些块被称为段。汇编器和链接器提供了可用于创建和操作段的指令。

本章着重介绍汇编语言程序中段的概念和使用。

2.1 目标文件格式规范 20
2.2 可执行目标文件 20
2.3 段简介 20
2.4 汇编器如何处理段 21
2.5 链接器如何处理段 26
2.6 符号 27
2.7 符号重定位 29
2.8 加载程序 30

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

2.1 目标文件格式规范

由汇编器和链接器创建的目标文件符合 ELF（可执行连接格式）二进制格式规范，该格式由嵌入式应用二进制接口 (EABI) 使用。有关 EABI ABI 的信息，请参阅 *ARM 优化 C/C++ 编译器用户指南* (SPNU151)。在 ARM 信息中心可找到完整的 ARM ABI 规范。

v15.6.0.STS 和更高版本的 TI 代码生成工具不支持 COFF 目标文件以及传统的 TIABI 和 TI ARM9 ABI 模式。如果希望生成 COFF 输出文件，请使用 5.2 版本的 ARM 代码生成工具，并参考 SPNU151J 文档。

由汇编器和链接器生成的 ELF 目标文件与 2003 年 12 月 17 日生成的 System V generic ABI（或 gABI）快照相符。 此规范目前由 SCO 负责维护。

2.2 可执行目标文件

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

若要运行程序，必须将可执行目标模块中的数据传输或加载到目标系统存储器中。有关加载和运行程序的详细信息，请参阅章节 3。

2.3 段简介

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

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

目标文件通常包含三个默认段：

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

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

有两种基本类型的段：

已初始化的段	包含数据或代码。.text 和 .data 段已被初始化；使用 .sect 汇编器指令创建的用户命名段也已初始化。
未初始化的段	在存储器映射中为未初始化的数据预留空间。.bss 段未被初始化；使用 .usect 汇编器指令创建的用户命名段也已被初始化。

您可以使用几个汇编器指令将代码和数据的各个部分与相应的段关联。汇编器在汇编过程中构建这些段，创建按图 2-1 所示整理的目标文件。

链接器的其中一项功能是将段重新定位到目标系统的存储器映射中；此功能被称为*放置*。大多数系统都包含几种类型的存储器，因此使用段可帮助您更高效地使用目标存储器。所有段均可单独重定位；您可以将任何段放入目标存储器的任何已分配块中。例如，您可以定义一个包含初始化例程的段，然后将该例程分配到包含 ROM 的存储器映射的一个部分。有关段放置的更多信息，请参阅 *ARM 优化 C/C++ 编译器用户指南* 中的“指定在存储器中分配段的位置”一节。

图 2-1 展示了目标文件中的段与虚构目标存储器之间的关系。ROM 可以是 EEPROM、FLASH 或实际系统中的一些其他类型的物理存储器。

[1] 某些目标允许在 .text 段中使用除文本以外的内容，例如变量。

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential – NDA Restrictions

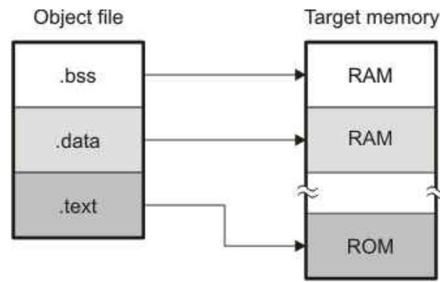


图 2-1. 存储器信息逻辑块分区

2.3.1 特殊段名

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

一些常见的特殊段有：

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

有关段的更多信息，请参阅 *ARM 优化 C/C++ 编译器用户指南* 中的“指定在存储器中分配段的位置”一节。

2.4 汇编器如何处理段

汇编器会识别属于给定段的汇编语言程序部分。汇编器具有以下支持该功能的指令：

- .bss
- .data
- .sect
- .text
- .usect

.bss 和 .usect 指令创建*未初始化的段*；.text、.data 和 .sect 指令创建*已初始化的段*。

为了更严格控制存储器映射，可创建任何段的子段。子段是使用 .sect 和 .usect 指令创建的。通过用冒号分隔的基础段名和子段名可以识别子段；请参阅节 2.4.6。

备注

如果不使用段指令，汇编器会将所有内容都汇编到 .text 段。

2.4.1 未初始化的段

未初始化的段在 ARM 存储器中保留空间；它们通常位于 RAM 中。这些段在目标文件中并无实际内容；只是保留存储器。程序可以在运行时使用此空间来创建和存储变量。

未初始化的数据区域是使用以下汇编器指令编译的。

- .bss 指令用于在 .bss 段中保留空间。
- .usect 指令用于在特定未初始化的用户命名段中保留空间。

每次调用 .bss 或 .usect 指令时，汇编器都会在 .bss 或用户命名段中保留额外空间。语法为：

.bss *symbol* **,** *size in bytes*[**,***alignment* [**,***bank offset*]]

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY Confidential – NDA Restrictions

```
symbol    .usect " section name ", size in bytes[,alignment[,bank offset] ]
```

symbol（符号）	指向此次通过调用 .bss 或 .usect 指令保留的第一个字节。*符号* 对应于要保留空间的变量的名称。它可由任何其他段引用，也可声明为全局符号（使用 .global 指令）。
size in bytes（以字节表示的大小）	是一个绝对表达式（请参阅节 4.9）。.bss 指令用于在 .bss 段中保留 *以字节表示的大小* 个字节。.usect 指令用于在 *段名* 中保留 *以字节表示的大小* 个字节。对于这两条指令，用户必须指定大小；无默认值。
alignment（对齐）	是一个可选参数。它指定分配的空间所需的最小对齐量（以字节为单位）。默认值是字节对齐；此选项由值 1 表示。该值必须是 2 的幂。
bank offset（模块偏移量）	是一个可选参数。可确保分配给符号的空间出现在特定存储器组的边界上。*模块偏移量* 测量在将符号分配给该位置之前从指定的对齐偏移的字节数。
section name（段名）	指定要保留空间的用户命名段。请参阅节 2.4.3。

初始化段的指令（.text、.data 和 .sect）可更改将哪个段视为*当前* 段（请参阅节 2.4.4）。但 .bss 和 .usect 指令*不会* 更改当前段；它们只是临时离开当前段。在 .bss 或 .usect 指令之后，汇编器会立即继续汇编到这些指令之前的当前段。.bss 和 .usect 指令可出现在初始化段中的任何位置，不会影响其内容。请参阅节 2.4.7 的示例。

.usect 指令也可用于创建未初始化的子段。有关创建子段的更多信息，请参阅节 2.4.6。

.common 指令与创建未初始化数据段的指令类似，只是通用符号是由链接器创建的。

2.4.2 已初始化段

已初始化段中包含可执行代码或已初始化的数据。这些段的内容存储在目标文件中，并在加载程序时置于 ARM 存储器中。每个已初始化的段可以单独重定位，并可能引用已在其他段中定义的符号。链接器会自动解析这些引用。以下指令指示汇编器将代码或数据置于段中。这些指令的语法为：

```
.text
.data
.sect " section name "
```

.sect 指令也可用于创建已初始化的子段。有关创建子段的更多信息，请参阅节 2.4.6。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

2.4.3 用户命名的段

用户命名的段是由*用户* 创建的段。其使用方式与默认 .text、.data 和 .bss 段相同，但具有独特名称的每个段在汇编期间都会区分开。

例如，重复使用 .text 指令可在目标文件中生成单一 .text 段。此 .text 段在存储器中是作为一个单元分配的。假设您希望链接器将一部分可执行代码（例如初始化例程）放在与其余 .text 不同的位置，如果您将此段代码汇编到用户命名的段，它会与 .text 分别汇编，您可以使用链接器将其分配到存储器中的其他位置。您也可以汇编与 .data 段分开的初始化数据，并为与 .bss 段分开的未初始化变量保留空间。

以下指令可用于创建用户命名的段：

- **.usect** 指令可创建未初始化的段，它们的使用方式与 .bss 段相同。这些段在 RAM 中为变量保留空间。
- **.sect** 指令可创建初始化段，其中可包含代码或数据，与默认 .text 和 .data 段相似。.sect 指令创建的用户命名段具有可重定位地址。

这些指令的语法为：

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

段的数量上限为 2^{32}-1 (4294967295)。

section name 参数是段的名称。对于 .usect 和 .sect 指令，段名可引用子段；请参阅节 2.4.6 了解详情。

每次您用新名称调用这些指令之一，就创建了一个新的用户命名段。每次您用已有名称调用这些指令之一，汇编器会在具有该名称的段中继续汇编代码或数据（或保留空间）。*不能在不同指令中使用相同名称*。也就是说，您无法使用 .usect 指令创建段，更不用说通过 .sect 使用该段。

2.4.4 当前段

汇编器一次将代码或数据添加到一个段。汇编器当前填充的段是*当前段*。.text、.data、 和 .sect 指令用于更改将哪个段视为当前段。当汇编器遇到其中某个指令时，它会停止汇编到当前段（作为当前段命令的隐式结束）。汇编器将指定段设置为当前段并将后续代码汇编到指定段中，直到遇到另一个 .text、.data、 或 .sect 指令。

如果其中某个指令将当前段设置为已包含之前在文件中添加的代码或数据的段，汇编器将继续添加到该段的末尾。汇编器只为每个给定段名生成一个连续段。该段是通过连接放置在该段中的所有代码或数据而形成的。

2.4.5 段程序计数器

汇编器会为每个段分别保留程序计数器。这些程序计数器被称为*段程序计数器*，或 *SPC*。

SPC 表示代码段或数据段中的当前地址。汇编器最初将每个 SPC 设为 0。汇编器在段中填充代码或数据时，相应 SPC 会递增。如果继续汇编到某个段，汇编器会记得相应 SPC 之前的值，并从该值继续递增 SPC。

汇编器处理每个段时都会当作从地址 0 开始；链接器会根据定义符号的段的最终地址在每个段中重定位该符号。请参阅节 2.7，了解重定位的相关信息。

2.4.6 子段

创建段时在名称中包含冒号可创建子段。子段是较大的段中的逻辑细分。子段本身是段，可由汇编器和链接器处理。

汇编器没有子段的概念；对于汇编器而言，名称中的冒号没有特殊含义。子段 .text:rts 被认为与其父段 .text 完全无关，汇编器不会将子段与其父段合并。

子段用于区分段的各个部分，以便分别处理。例如，将每个函数和对象置于具有唯一名称的子段，链接器可以更加精细地将段放置于存储器中，消除未使用的函数。

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

默认情况下，链接器如果在链接器命令文件（如 ".text"）中发现一条 SECTION 指令，就会收集 .text 以及 .text 的所有子段，并放入一个大的输出段，名为 ".text"。您也可以使用 SECTION 指令分别控制子段。请参阅节 8.5.5.1 中的示例。

创建子段的方式与创建其他用户命名段的方式相同：即使用 .sect 或 .usect 指令。

子段名称的语法为：

symbol	**.usect "** *section_name* **:** *subsection_name* **",** *size in bytes*[,*alignment*[,*bank offset*]]
	.sect " *section_name* **:** *subsection_name* **"**

基础段名后跟冒号和子段名即为子段。子段名不包含任何空格。

子段可分别分配，或使用同一基础名称与其他段组合。例如，您在 .text 段中创建了一个名为 _func 的子段：

```
      .sect ".text:_func"
```

使用链接器的 SECTIONS 指令，您可以单独分配 .text:_func，或与所有 .text 段共同分配。

您可以创建两类子段：

- 初始化的子段使用 .sect 指令创建。请参阅节 2.4.2。
- 未初始化的子段使用 .usect 指令创建。请参阅节 2.4.1。

子段与段的放置方法相同。有关 SECTIONS 指令的信息，请参阅节 8.5.5。

2.4.7 使用 Sections 指令

图 2-2 显示了如何使用 Sections 指令在不同段之间来回切换，从而增量式构建段。用户可以使用 Sections 指令首次开始汇编到某段，或继续汇编到已包含代码的段。在第二种情况下，汇编器会将新代码添加到该段中已有的代码之后。

图 2-2 中的格式是列表文件。图 2-2 显示了汇编期间如何修改 SPC。列表文件中的一行包含 4 个字段：

字段 1	包含源代码行计数器。
字段 2	包含段程序计数器。
字段 3	包含目标代码。
字段 4	包含原始源语句。

请参阅节 4.12，了解源列表中各字段的解释信息。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

```
 1                            *************************************************
 2                            ** Assemble an initialized table into .data.  **
 3                            *************************************************
 4  00000000                          .data
 5  00000000 00000011  coeff   .word          011h, 022h, 033h
    00000004 00000022
    00000008 00000033
 6
 7                            *************************************************
 8                            **  Reserve space in .bss for a variable.     **
 9                            *************************************************
10  00000000                          .bss           buffer,10
11                            *************************************************
12                            **          Still in .data.                   **
13                            *************************************************
14  0000000c 00000123  ptr     .word          0123h
15                            *************************************************
16                            **  Assemble code into the .text section.     **
17                            *************************************************
18  00000000                       .text
19  00000000 E59F14D2  add:    LDR            R1, #1234
20  00000004 E2511001  aloop:  SUBS           R1, R1, #1
21  00000008 1AFFFFFD     BNE                 aloop
22                            *************************************************
23                            **  Another initialized table into .data.     **
24                            *************************************************
25  00000010                          .data
26  00000010 000000AA  ivals   .word          0AAh, 0BBh, 0CCh
    00000014 000000BB
    00000018 000000CC
27                            *************************************************
28                            ** Define another section for more variables.**
29                            *************************************************
30  00000000           var2    .usect         "newvars", 1
31  00000001           inbuf   .usect         "newvars", 7
32                            *************************************************
33                            **      Assemble more code into .text.        **
34                            *************************************************
35  0000000c                          .text
36  0000000c E59F3D80  mpy:    LDR            R3, #3456
37  00000010 E0120293  mloop:  MULS           R2, R3, R2
38  00000014 1AFFFFFD          BNE            mloop
39                            *************************************************
40                            ** Define a named section for int. vectors.   **
41                            *************************************************
42  00000000                          .sect          "vectors"
43  00000000 00000011                 .word          011h,033h
    00000004 00000033
 Field 1   Field 2    Field 3                     Field 4
```

图 2-2. 使用 Sections 指令示例

如图 2-3 所示，图 2-2 中的文件创建了 5 个段：

.text	包含 6 个 32 位目标代码字。
.data	包含 7 个 32 位初始化数据字。
vectors	是使用 .sect 指令创建的用户命名段；包含 2 个 32 位初始化数据字。
.bss	保留存储器中的 10 个字节。
newvars	是使用 .usect 指令创建的用户命名段；保留存储器中的 8 个字节。

第二列显示了汇编到这些段的目标代码；第一列显示了生成这些目标代码的源语句。

Copyright © 2023 Texas Instruments Incorporated

Line numbers	Object code	Section
19 20 21 36 37 38	E59F14D2 E2511001 1AFFFFFD E59F3D80 E0120293 1AFFFFFD	.text
5 5 5 14 26 26 26	00000011 00000022 00000033 00000123 000000AA 000000BB 000000CC	.data
43 43	00000011 00000033	vectors
10	No data - ten bytes reserved	.bss
30 31	No data - eight bytes reserved	newvars

图 2-3. 图 2-2 中由文件生成的目标代码

2.5 链接器如何处理段

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

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

使用子段可更精确地控制段的放置。您可以使用链接器的 SECTIONS 指令指定每个子段的位置。如果不指定子段，则该子段将与具有相同基础段名的其他段组合在一起。请参阅节 8.5.5.1。

并不总是需要使用链接器指令。如果不使用此类指令，链接器将使用节 8.7 中所述的目标处理器的默认放置算法。当您*确实* 使用链接器指令时，必须在链接器命令文件中指定它们。

有关链接器命令文件和链接器指令的更多信息，请参阅以下部分：

- 节 8.5，*链接器命令文件*
- 节 8.5.4，*MEMORY 指令*
- 节 8.5.5，*SECTIONS 指令*
- 节 8.7，*默认放置算法*

2.5.1 合并输入段

图 2-4 通过一个简化示例介绍了将两个文件链接在一起的过程。

请注意，这是一个简化示例，因此它不会显示待创建的所有段或这些段的实际顺序。请参阅节 8.7，了解 ARM 的实际默认存储器放置映射。

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

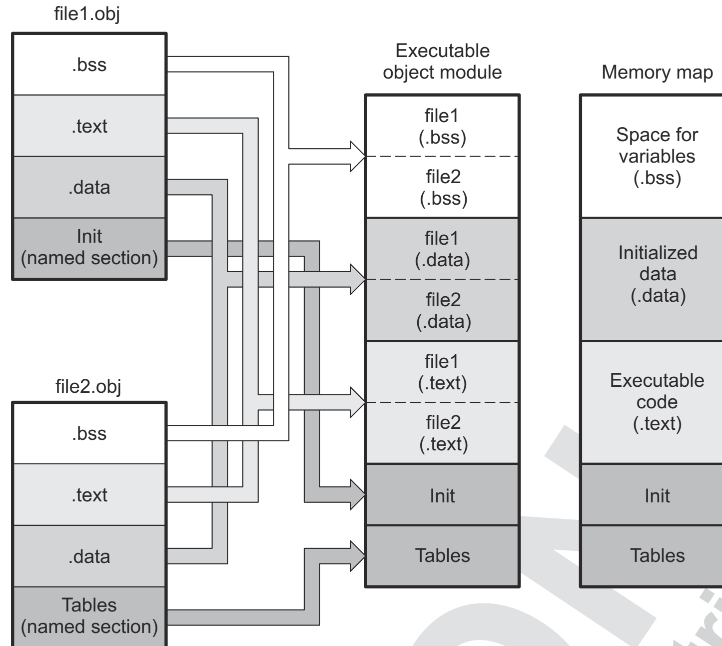


图 **2-4.** 组合输入段以构成可执行对象模块

在图 2-4 中，已将 file1.obj 和 file2.obj 组合在一起用作链接器输入。每个都包含 .text、.data 和 .bss 默认段；此外，每个都包含一个用户命名的段。可执行对象模块显示了各组合段。链接器将 file1.obj 的 .text 段和 file2.obj 的.text 段组合成一个 .text 段，然后将两个 .data 段和两个 .bss 段组合在一起，最后将用户命名的段放在末尾。存储器映射显示了待置于存储器中的组合段。

2.5.2 放置段

图 2-4 展示了链接器合并段的默认方法。有时，您可能不想使用默认设置。例如，您可能不想将所有 .text 段合并到一个 .text 段中。或者，您可能想要将用户命名的段放置在 .data 段通常所在的位置。大多数存储器映射都包含多种不同类型的存储器（RAM、ROM、EEPROM、FLASH 等），对应的空间量各不相同；您可能想要将某个段放在特定类型的存储器中。

如需存储器映射中段放置的进一步说明，请参阅 节 8.5.4 和 节 8.5.5 中的讨论。请参阅节 8.7，了解 ARM 的实际默认存储器分配映射。

2.6 符号

目标文件包含一个符号表，存储着目标文件中相关*符号* 的信息。链接器在执行重定位时会使用此表。请参阅节 2.7。

目标文件符号是一个指定的 32 位整数值，通常表示一个地址。符号可以表示诸如函数、变量、段的起始地址，或绝对整数（如栈大小）之类的东西。

在汇编中定义符号时，需添加标签或 .set .equ .bss、.usect 等指令。

符号具有*绑定* 属性，类似于 C 语言的*链接* 概念。ELF 文件可能包含绑定为*局部符号*、*全局符号* 和 *弱符号* 的符号。

- **全局符号**在整个程序中可见。链接器不允许特定符号具有超过一个全局定义；如果全局符号定义超出一次，会生成多重定义错误。（汇编器可以为局部符号提供类似的“多重定义”错误。）从目标文件引用全局符号，会引用该符号唯一允许的全局定义。汇编代码必须添加 .def、.ref 或 .global 指令，显式定义全局符号。（请参阅节 2.6.1。）

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI confidential – NDA restrictions

- **局部符号**只在一个目标文件中可见；每个使用某符号的目标文件均需要自己的局部定义。引用一个目标文件中的局部符号与另一个目标文件中的同名局部符号完全无关。默认情况下符号是局部符号。（请参阅节 2.6.2。）
- **弱符号**是在当前模块中使用但未定义的符号。它们可能在另一模块中定义，也可能未在另一模块中定义。弱符号会被另一个目标文件中同名的强（非弱）全局符号定义覆盖。如果具有可用的强定义，弱符号会被强符号替代。如果没有可用定义（即弱符号未解析），则不会生成错误，但弱变量的地址将为 null (0)。因此，访问弱变量的应用代码*在尝试访问该变量之前必须确保其地址非零*。（请参阅节 2.6.3。）

绝对符号 是具有数字值的符号。它们可能是常量。对于链接器而言，这样的符号是无符号的，但整数会根据其使用方式被视为有符号或无符号类型。若处理为无符号类型，则绝对整数合法值的范围是 0 到 2^32-1，若处理为有符号类型，则为 -2^31 到 2^31-1。

一般而言，*通用符号*（请参阅 .common 指令）优先于弱符号。

有关*汇编器符号* 的信息，请参阅节 4.8。

2.6.1 全局（外部）符号

全局符号是在当前模块中访问但在另一模块中定义的符号（外部符号），或是在当前模块中定义并在另一模块中访问的符号。此类符号在各目标模块中均可见。您必须使用 .def、.ref 或 .global 指令将符号识别为外部符号：

.def	该符号在当前文件中定义，并可在另一文件中使用。
.ref	该符号在当前文件中引用，但在另一文件中定义。
.global	该符号可以是上述任一种。汇编器会根据每个符号的情况选择 .def 或 .ref。

以下代码片段说明了 .global 指令的用法。

```
x:  ADD      R0, #56h     ; Define x
    .global x ; acts as .def x
```

因为 x 是在此模块中定义的，所以汇编器将“.global x”视为“.def x”。现在，其他模块可以引用 x。

```
    B         y              ; Reference y
    .global y ; .ref of y
```

因为 y 不是在此模块中定义的，所以汇编器将“.global y”视为“.ref y”。必须在另一模块中定义符号 y。

符号 x 和 y 都是外部符号，并放置在目标文件的符号表中；x 作为已定义的符号，y 作为未定义的符号。当该目标文件与其他目标文件链接时，x 的条目将用于解析其他文件对 x 的引用。y 的条目使链接器在其他文件的符号表中查找 y 的定义。

链接器尝试将所有引用与相应的定义匹配。如果链接器找不到符号的定义，它将打印有关未解析引用的错误消息。此类错误会阻止链接器创建可执行目标模块。

如果多次定义同一个符号，也会出现错误。

2.6.2 局部符号

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

默认情况下符号是局部符号。

2.6.3 弱符号

弱符号是可能定义，也可能不定义的符号。

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

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential – NDA Restrictions

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

用户可以使用 .weak 汇编指令或链接器命令文件中的弱运算符来定义弱符号。

- **使用汇编指令**：要在输入目标文件中定义一个弱符号，源文件可以用汇编指令编写。如以下示例所示，组合使用 .weak 和 .set 指令，定义弱符号“ext_addr_sym”：

```
                .weak   ext_addr_sym
ext_addr_sym    .set    0x12345678
```

  汇编定义了弱符号的源文件，并在链接中包含生成的目标文件。此示例中的“ext_addr_sym”在最终链接中作为弱符号提供。如果应用中的其他位置未引用该符号，它会成为候选的删除对象。请参阅 .weak 指令。
- **使用链接器命令文件**：若要在链接器命令文件中定义弱符号，请在赋值表达式中使用“弱”运算符来指定可以从输出文件的符号表中删除该符号（如果它未被引用）。在链接器命令文件中，可使用 MEMORY 或 SECTIONS 指令之外的赋值表达式来定义由链接器定义的弱符号。例如，可将“ext_addr_sym”定义如下：

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

  如果链接器命令文件执行最终链接，“ext_addr_sym”会作为弱符号提供给链接器；如果未引用该符号，它将不会被包含在生成的输出文件中。请参阅节 8.6.5。
- **使用 C/C++ 代码**：请参阅 *ARM 优化 C/C++ 编译器用户指南* 中的 WEAK pragma 和 weak GCC 样式变量属性信息。

如果同一符号有多个定义，链接器会根据特定规则确定优先使用哪个定义。一些定义可能有弱绑定，另一些定义可能有强绑定。“强”在此上下文中意为该符号*并未*通过上述任一种方法进行弱绑定。一些定义可能来自输入目标文件（即使用汇编指令），另一些可能来自链接器命令文件中的赋值语句。

链接器根据以下指导原则确定将引用解析为符号时使用哪个定义：

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

2.6.4 符号表

对于以下每一项，汇编器都会在符号表中生成全局（外部）绑定条目：

- 每条 .ref、.def 或 .global 指令（请参阅节 2.6.1）
- 每个段的起始

对于每个局部可用的函数，汇编器都会生成局部绑定条目。

符号表中还提供程序中每个符号的条目，仅供参考。

2.7 符号重定位

汇编器在处理每一个段时均假设它始于地址 0。当然，在存储器中不可能所有段都始于地址 0，因此链接器必须将段重定位。重定位与符号相关，而不是与段相关。

链接器将段*重定位*的方式是：

- 将它们分配到存储器映射，使它们从由链接器的 MEMORY 指令定义的适当地址开始。
- 调整符号值，以对应到新的段地址
- 调整对重定位符号的引用，以反映调整后的符号值

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

链接器使用*重定位条目*调整对符号值的引用。汇编器在可重定位的符号被引用时会创建一个重定位条目。然后链接器会使用这些条目在符号重新定位后修补引用。以下示例包含 ARM 器件的代码片断，汇编器会针对它们生成重定位条目。

```
 1                      **********************************************
 2                      **       生成重定位条目        **
 3                      **********************************************
 4                              .ref X
 5                              .def Y
 6 00000000                     .text
 7 00000000 E0921003            ADDS    R1, R2, R3
 8 00000004 0A000001            BEQ     Y
 9 00000008 E1C410BE            STRH    R1, [R4, #14]
10 0000000c EAFFFFFB!           B       X    ; generates a relocation entry
11 00000010 E0821003    Y:      ADD     R1, R2, R3
```

在以上示例中，符号 X 和 Y 均可重定位。Y 在本模块的 .text 段中定义；X 在另一模块中定义。汇编代码时，X 的值为 0（汇编器假定所有未定义的外部符号的值均为 0），Y 的值为 16（相对于 .text 段中的地址 0）。汇编器生成两个重定位条目：一个用于 X，一个用于 Y。对 X 的引用是外部引用（由列表中的 **!** 字符表示）。 对 Y 的引用是对内部定义的可重定位符号的引用（由列表中的 **'** 字符表示）。

链接代码后，假设将 X 重定位到地址 0x10014。再假设将 .text 段重定位为从地址 0x10000 开始；Y 目前的重定位值为 0x10010。链接器使用引用 X 的重定位条目来修补目标代码中的分支指令：

EAFFFFFB! B X	becomes	EA000000

2.8 加载程序

链接器会创建一个可执行的目标文件，根据您的执行环境，可以通过多种方式加载该文件。这些方法包括使用 Code Composer Studio 或十六进制转换实用程序。相关详细信息，请参阅节 3.1。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

章节 3
程序加载和运行

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

上述一些任务需要构建到程序本身中。*自举*是程序执行一些自身初始化的过程。很多必要任务都由编译器和链接器替用户完成，但如果需要更多地控制这些任务，那么了解各段代码如何配合工作会有所帮助。

本章将介绍程序加载、初始化和启动中涉及的各个概念。

本章当前提供了适用于 C6000 器件系列的示例。请参阅器件文档，以了解自举的各种器件特定方面。

3.1 负载..32
3.2 入口点..36
3.3 运行时初始化..37
3.4 main 的参数..39
3.5 运行时重定位..39
3.6 其他信息..39

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

3.1 负载

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

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

可以通过以下方式加载程序：

- **在已连接的主机工作站上运行的调试器。**在典型的嵌入式开发设置中，器件隶属于运行 Code Composer Studio (CCS) 等调试器的主机。器件与 JTAG 接口等通信通道连接。CCS 读取程序，并通过通信接口将加载映像直接写入到目标存储器。
- **将加载映像"烧录"到 EPROM 模块。** 十六进制转换器 (armhex) 可以将可执行目标文件转换为适合输入到 EPROM 编程器的格式，从而帮助完成此操作。将 EPROM 放入器件本身，并成为器件存储器的一部分。相关详细信息，请参阅章节 12。
- **从专用外设（例如，I²C 外设）加载引导程序。**器件可能需要一个被称为引导加载程序的小程序来执行从外设加载。十六进制转换器可帮助创建引导加载程序。
- **器件上运行的另一个程序。**运行的程序可以创建加载映像并将控制权移交给加载的程序。如果有操作系统，则它可能具有加载和运行程序的能力。

3.1.1 加载和运行地址

假设有一个嵌入式器件，程序的加载映像烧录到 EPROM/ROM。程序中的可变数据必须是可写入的，因此必须位于可写存储器中，通常是 RAM。但是，RAM 具有 *易失性*，这意味着断电时其中的内容会丢失。如果此数据必须有初始值，该初始值必须存储在加载映像中的其他位置，否则在断电并重新上电时该值会丢失。在使用之前，初始值必须从非易失性 ROM 复制到它在 RAM 中的运行时位置。有关完成此操作的方法，请参阅节 8.8。

加载地址 是对象在加载映像中的位置。

运行地址 是对象在程序执行期间所处的位置。

*对象*是一块存储器。它代表一个段、区段、函数或数据。

*对象的加载地址和运行地址可能相同。*对于程序代码和只读数据（例如 .const 段），这两个地址通常是相同的。在这种情况下，程序会直接从加载地址读取数据。没有初始值的段（例如 .bss 段）没有加载数据，因此被视为具有相同的加载地址和运行地址。如果为未初始化的段指定不同的加载地址和运行地址，链接器会发出警告并忽略加载地址。

*对象的加载地址和运行地址可能不同。*对于可写入数据（例如 .data 段），通常是这种情况。将 .data 段的起始内容放入 ROM 并复制到 RAM。通常在程序启动时执行此操作，但根据对象的需求，该操作可能会延迟执行，如节 3.5 所述。

汇编代码和目标文件中的符号通常是指运行地址。查看程序中的地址时，通常查看的是运行地址。加载地址很少用于初始化以外的任何其他事务。

段的加载地址和运行地址由链接器命令文件控制并记录在目标文件元数据中。

加载地址决定了加载程序将段的原始数据放在何处。对该段的任何引用（例如对其中所含标签的引用）都会引用其运行地址。应用程序在运行期间遇到符号的第一次引用时，必须将段从其加载地址复制到其运行地址；即使指定了单独的运行地址，此操作也 *不会* 自动执行。有关指定加载地址和运行地址的示例，请参阅节 8.5.6.1。

有关说明如何在运行时移动代码块的示例，请参阅在运行时将一个函数从慢速存储器移动到快速存储器。若要创建使您可以引用加载时地址而不是运行时地址的符号，请参阅 .label 指令。若要在启动时使用复制表将对象从加载空间复制到运行空间，请参阅节 8.8。

ELF 格式可执行目标文件包含*区段*。有关段和区段的信息，请参阅节 2.3。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

3.1.2 引导加载

不同器件的引导加载细节有很大差异。并非每个器件都支持每种引导加载模式，并且对引导加载程序的使用是可选的。本节讨论了各种引导加载方案，以帮助您了解它们的工作原理。请参阅您器件的数据表，了解哪些引导加载方案可用以及如何使用它们。

典型的嵌入式系统使用引导加载功能来初始化器件。程序代码和数据可以存储在 ROM 或 FLASH 存储器中。上电时，内置于器件硬件中的片上引导加载程序（*主引导加载程序*）会自动启动。

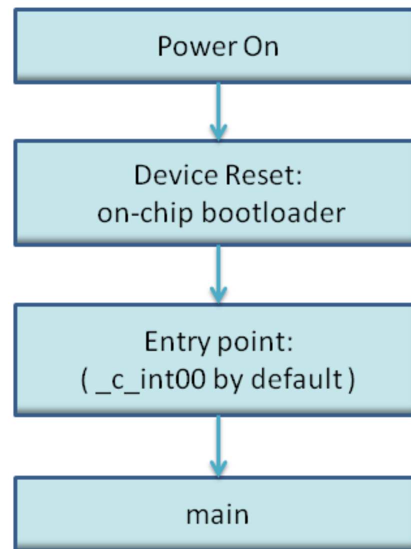


图 3-1. 引导加载序列（简化）

主引导加载程序通常非常小，可将有限数量的存储器从 ROM 中的专用位置复制到 RAM 中的专用位置。（一些引导加载程序支持从 I/O 外设复制程序。）复制完成后，它会将控制权转移给程序。

对于许多程序，主引导加载程序无法加载整个程序，因此这些程序提供了一个功能更强大的次级引导加载程序。主引导加载程序会加载次级引导加载程序并将控制权转移给它。然后，次级引导加载程序会加载程序的其余部分并将控制权转移给它。可以有任意数量的引导加载程序层，每层加载一个功能更强大的引导加载程序，它将控制权转移到该引导加载程序。

DRAFT ONLY TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

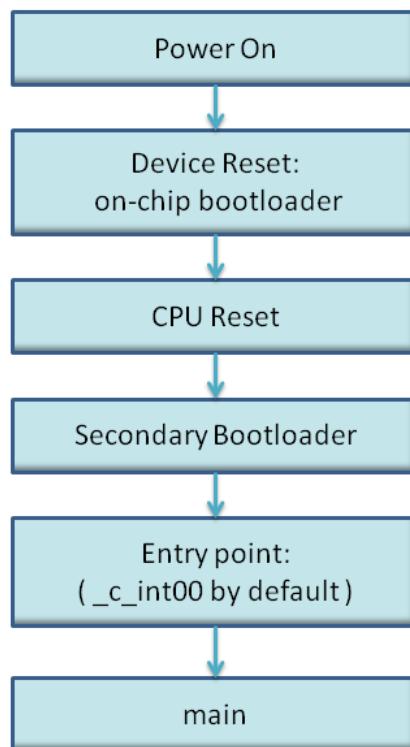


图 3-2. 使用次级引导加载程序的引导加载序列

3.1.2.1 引导、加载和运行地址

被引导加载对象的 *引导地址* 是通电前其原始数据在 ROM 中的位置。

一个对象的引导、加载和运行地址可能都相同；.const 数据通常就是这种情况。如果它们不同，则必须先将对象的内容复制到正确的位置，然后才能使用该对象。

引导地址可能与加载地址不同。引导加载程序负责将原始数据复制到加载地址。

引导地址不受链接器命令文件的控制或也未被记录在目标文件中；严格来说，它是引导加载程序和程序共享的一项规则。

3.1.2.2 主引导加载程序

主引导加载程序的详细操作情况取决于器件。一些器件具有复杂功能，例如从 I/O 外设引导或配置存储器控制器参数。

3.1.2.3 次级引导加载程序

十六进制转换器假定次级引导加载程序为特定格式。十六进制转换器的模型引导加载程序使用 *引导表*。您可以使用需要的任何格式，但如果使用此模型，十六进制转换器可以自动创建引导表。

3.1.2.4 引导表

模型次级引导加载程序的输入是 *引导表*。引导表包含的记录会指示次级引导加载程序将表中包含的数据块复制到指定目标地址。十六进制转换实用程序会自动编译次级引导加载程序的引导表。使用该实用程序，您可以指定要初始化的段、引导表位置以及包含次级引导加载程序例程及其所在位置的段的名称。十六进制转换实用程序会编译表的完整图像并将其添加到程序中。

引导表是特定于目标的。对于 C6000，引导表的格式很简单。标头记录包含一个 4 字节字段，用于指示引导加载程序在完成复制数据后应该分支的位置。在标头之后，要包含在引导表中的每个段都包含以下内容：

- 包含段大小的 4 字节字段

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential – NDA Restrictions

- 包含复制目标地址的 4 字节字段
- 原始数据
- 0 到 3 个字节的尾部填充，使下一个字段对齐到 4 个字节

可以输入多个段；包含全零 4 字节字段的终止块跟在最后一段之后。

有关引导表格式的详细信息，请参阅节 12.11.2。

3.1.2.5 引导加载程序例程

引导加载程序例程是一个普通函数，只不过它是在设置 C 环境之前执行的。为此，它不能使用 C 栈，也不能调用任何尚未加载的函数！

以下示例代码适用于 C6000，取自*使用 Code Composer Studio 在 TMS320C6000 平台上创建用于闪存引导加载的次级引导加载程序* (SPRA999)。

示例 3-1. 次级引导加载程序例程示例

```
;  ======== boot_c671x.s62 ========
; global EMIF symbols defined for the c671x family
            .include        boot_c671x.h62
            .sect ".boot_load"
            .global _boot
_boot:
;*************************************************************************
;* DEBUG LOOP - COMMENT OUT B FOR NORMAL OPERATION
;*************************************************************************
zero B1
_myloop: ;   [!B1] B _myloop
            nop  5
_myloopend: nop
;*************************************************************************
;* CONFIGURE EMIF
;*************************************************************************
        ;*****************************************************************
        ; *EMIF_GCTL = EMIF_GCTL_V;
        ;*****************************************************************
            mvkl  EMIF_GCTL,A4
      ||    mvkl  EMIF_GCTL_V,B4
            mvkh  EMIF_GCTL,A4
      ||    mvkh  EMIF_GCTL_V,B4
            stw   B4,*A4
        ;*****************************************************************
        ; *EMIF_CE0 = EMIF_CE0_V
        ;*****************************************************************
            mvkl  EMIF_CE0,A4
      ||    mvkl  EMIF_CE0_V,B4
            mvkh  EMIF_CE0,A4
      ||    mvkh  EMIF_CE0_V,B4
            stw   B4,*A4
        ;*****************************************************************
        ; *EMIF_CE1 = EMIF_CE1_V (setup for 8-bit async)
        ;*****************************************************************
            mvkl  EMIF_CE1,A4
      ||    mvkl  EMIF_CE1_V,B4
            mvkh  EMIF_CE1,A4
      ||    mvkh  EMIF_CE1_V,B4
            stw   B4,*A4
        ;*****************************************************************
        ; *EMIF_CE2 = EMIF_CE2_V (setup for 32-bit async)
        ;*****************************************************************
            mvkl  EMIF_CE2,A4
      ||    mvkl  EMIF_CE2_V,B4
            mvkh  EMIF_CE2,A4
      ||    mvkh  EMIF_CE2_V,B4
            stw   B4,*A4
        ;*****************************************************************
        ; *EMIF_CE3 = EMIF_CE3_V (setup for 32-bit async)
        ;*****************************************************************
      ||    mvkl  EMIF_CE3,A4
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

```
    ||    mvkl  EMIF_CE3_V,B4      ;
          mvkh  EMIF_CE3,A4
    ||    mvkh  EMIF_CE3_V,B4
          stw   B4,*A4
      ;*****************************************************************
      ; *EMIF_SDRAMCTL = EMIF_SDRAMCTL_V
      ;*****************************************************************
    ||    mvkl  EMIF_SDRAMCTL,A4
    ||    mvkl  EMIF_SDRAMCTL_V,B4      ;
          mvkh  EMIF_SDRAMCTL,A4
    ||    mvkh  EMIF_SDRAMCTL_V,B4
          stw   B4,*A4
      ;*****************************************************************
      ; *EMIF_SDRAMTIM = EMIF_SDRAMTIM_V
      ;*****************************************************************
    ||    mvkl  EMIF_SDRAMTIM,A4
    ||    mvkl  EMIF_SDRAMTIM_V,B4      ;
          mvkh  EMIF_SDRAMTIM,A4
    ||    mvkh  EMIF_SDRAMTIM_V,B4
          stw   B4,*A4
      ;*****************************************************************
      ; *EMIF_SDRAMEXT = EMIF_SDRAMEXT_V
      ;*****************************************************************
    ||    mvkl  EMIF_SDRAMEXT,A4
    ||    mvkl  EMIF_SDRAMEXT_V,B4      ;
          mvkh  EMIF_SDRAMEXT,A4
    ||    mvkh  EMIF_SDRAMEXT_V,B4
          stw   B4,*A4
;****************************************************************************
; copy sections
;****************************************************************************
            mvkl  COPY_TABLE, a3 ; load table pointer
            mvkh  COPY_TABLE, a3
            ldw   *a3++, b1      ; Load entry point
copy_section_top:
            ldw   *a3++, b0      ; byte count
            ldw   *a3++, a4      ; ram start address
            nop   3
[!b0]       b copy_done          ; have we copied all sections?
            nop   5
copy_loop:
            ldb   *a3++,b5
            sub   b0,1,b0        ; decrement counter
[ b0]       b     copy_loop      ; setup branch if not done
[!b0]       b     copy_section_top
            zero  a1
[!b0]       and   3,a3,a1
            stb   b5,*a4++
[!b0]       and   -4,a3,a5       ; round address up to next multiple of 4
[ a1]       add   4,a5,a3        ; round address up to next multiple of 4
;****************************************************************************
; jump to entry point
;****************************************************************************
copy_done:
            b    .S2 b1
            nop  5
```

3.2 入口点

入口点是开始执行程序的地址。这是启动例程的地址。启动例程负责初始化和调用程序的其余部分。对于 C/C++ 程序，启动例程通常命名为 _c_int00（请参阅节 3.3.1）。加载程序后，入口点的值将放入 PC 寄存器中，并允许 CPU 运行。

目标文件有一个入口点字段。对于 C/C++ 程序，链接器默认会填入 _c_int00。您可以选择自定义入口点；请参阅节 8.4.13。器件本身无法从目标文件中读取入口点字段，因此必须在程序中的某处对其进行编码。

- 如果您使用引导加载程序，则引导表包含一个入口点字段。当它完成运行时，引导加载程序将跳转到入口点。
- 如果您使用中断矢量，入口点将安装为 RESET 中断处理程序。应用 RESET 时，将调用启动例程。
- 如果您使用托管的调试器（如 CCS），该调试器可以将程序计数器 (PC) 显式设置为入口点的值。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

3.3 运行时初始化

在加载映像就位后，程序便可以运行。后续的小节描述了 C/C++ 程序的自举初始化。仅汇编程序可能不需要完成所有这些步骤。

3.3.1 _c_int00 函数

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

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

启动例程负责执行以下操作：

1. 切换到用户模式并设置用户模式栈
2. 设置状态寄存器和配置寄存器
3. 设置栈
4. 处理特殊二进制符号复制表（若存在）。
5. 处理运行时初始化表以自动初始化全局变量（使用 --rom_model 选项时）
6. 调用所有全局构造函数
7. 调用 main 函数
8. 当 main 返回时调用 exit

3.3.2 RAM 模型与 ROM 模型

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

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

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

在 EABI RAM 模型中，启动时不会生成 .cinit 记录。

3.3.2.1 在运行时自动初始化变量 (--rom_model)

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

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

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

图 3-3 演示了使用 ROM 模型时的运行时自动初始化（ ）。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

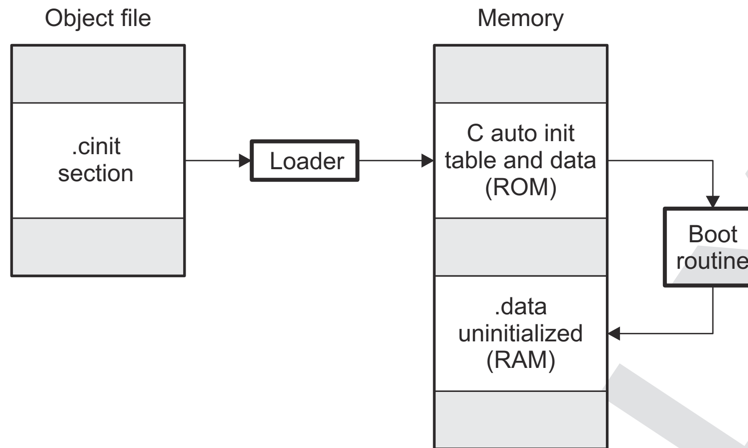


图 **3-3.** 运行时自动初始化

3.3.2.2 在加载时初始化变量 (--ram_model)

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

此模型可以减少启动时间并节省初始化表使用的存储器空间。

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

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

加载器会将值直接从 .data 段复制到存储器中。

图 3-4 演示了加载时变量的初始化。

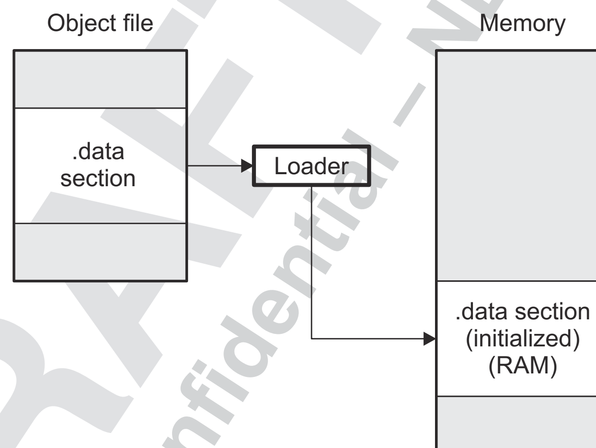


图 **3-4.** 加载时初始化

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

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

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

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

3.3.3 关于链接器生成的复制表

RTS 函数 copy_in 可在运行时用来移动代码和数据，通常是从加载地址移动到运行地址。此函数从复制表中读取大小和位置信息。链接器会自动生成几种复制表。请参阅节 8.8。

使用复制表可以创建和控制代码叠加。请参阅节 8.8.4，了解详细信息和示例。

链接器可以使用复制表来实现运行时重定位（如节 3.5 所述），但是复制表需要特定的表格式。

3.3.3.1 BINIT

BINIT（启动时初始化）复制表的特殊之处在于目标将在自动初始化时自动执行复制。更多有关 BINIT 复制表名称的信息，请参阅节 8.8.4.2。在复制 BINIT 复制表之后需要进行 .cinit 处理。

3.3.3.2 CINIT

EABI .cinit 表是特殊类型的复制表。 有关将 .cinit 段用于 ROM 模型的更多信息，请参阅节 3.3.2.1；有关将该段用于 RAM 模型的更多信息，请参阅节 3.3.2.2。

3.4 main 的参数

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

有关分配存储器以进行参数传递的信息，请参阅节 8.4.4。

3.5 运行时重定位

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

链接器提供了处理此任务的一种方式。使用 SECTIONS 指令，您可以选择使链接器分配一个段两次：首先设置其加载地址，接着设置其运行地址。加载地址使用 *load* 关键字，运行地址使用 *run* 关键字。请参阅节 3.1.1，以进一步了解加载地址和运行地址。如果段在链接时被分配了两个地址，该段中定义的所有标签会重新定位为引用运行时地址，以便在代码运行时正确引用该段（例如，分支）。

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

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

有关运行时重定位的完整说明，请参阅节 8.5.6。

3.6 其他信息

请参阅以下章节和文档了解详情：

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential – NDA Restrictions

节 8.4.4“分配存储器供加载器使用以传递参数（--arg_size 选项）”

节 8.4.13“定义入口点（--entry_point 选项）”

节 8.5.6.1“指定加载和运行地址”

节 8.8“由链接器生成的复制表”

节 8.11.1“运行时初始化”

.label 指令

章节 12“十六进制转换实用程序说明”

《*ARM 优化 C/C++ 编译器用户指南*》中的“运行时初始化”和“系统初始化”章节

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

章节 4
汇编器说明

ARM 汇编器将汇编语言源文件转换成机器语言的目标文件。这些文件是目标模块（请参阅章节 2 中的相关讨论）。源文件可以包含以下汇编语言元素：

汇编器指令	如章节 5 所述
宏命令	如章节 6 所述
汇编语言指令	如《*TMS470R1x 用户指南*》所述

4.1 汇编器概述....42
4.2 汇编器在软件开发流程中的作用....42
4.3 调用汇编器....43
4.4 控制应用二进制接口....43
4.5 指定用于汇编器输入的备用目录....44
4.6 源语句格式....46
4.7 字面常量....49
4.8 汇编器符号....51
4.9 表达式....57
4.10 内置函数和运算符....61
4.11 统一汇编语言语法支持....62
4.12 源程序列表....62
4.13 调试汇编源文件....65
4.14 交叉引用列表....66

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

4.1 汇编器概述

双程汇编器将执行以下操作：

- 处理文本文件中的源语句以生成可重定位的目标文件
- 生成源程序列表（如果需要）并能够控制此列表
- 能够将代码划分为多个段，并为每个目标代码段维护一个段程序计数器 (SPC)
- 定义和引用全局符号，并将交叉参考列表附加到源程序列表（如果需要）
- 允许条件汇编
- 支持宏，从而能够以内联方式或在库中定义宏

4.2 汇编器在软件开发流程中的作用

图 4-1 展示了汇编器在软件开发流程中的作用。阴影部分突出显示了最常用的汇编器开发路径。汇编器接受汇编语言源文件作为输入，这些文件由您创建或由 ARM C/C++ 编译器创建。

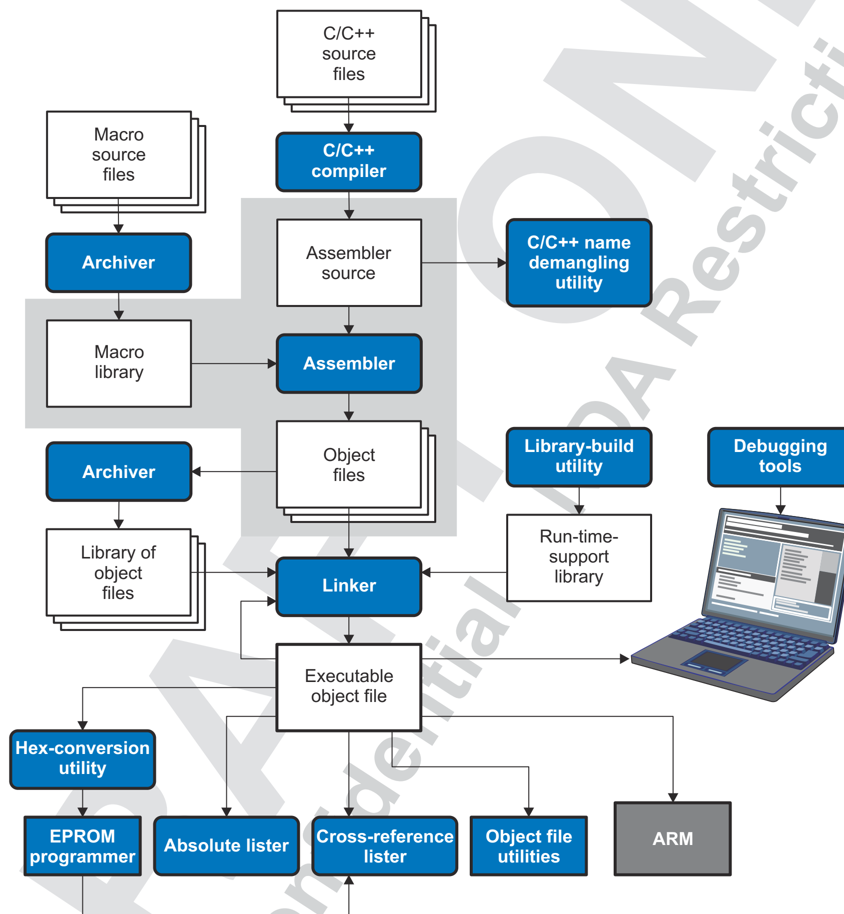


图 4-1. ARM 软件开发流程中的汇编器

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

4.3 调用汇编器

若要调用汇编器，请输入以下命令：

armcl *input file* [*options*]

armcl	是通过编译器调用汇编器的命令。编译器将任何带有 .asm 扩展名的文件视为汇编文件并调用汇编器。
input file	指定汇编语言源文件的名称。
options	标识要使用的汇编器选项。选项区分大小写，可出现在命令行中的命令之后的任意位置。如示例所示，在每个选项前面加一个或两个连字符。

表 4-1 中列出了有效的汇编器选项。

表 4-1. ARM 汇编器选项

选项	别名	说明
--absolute_listing	**-aa**	创建绝对列表。使用 --absolute_listing 时，汇编器不会生成目标文件。--absolute_listing 选项与绝对列表器搭配使用。
--asm_define=*name*[=*def*]	**-ad**	设置 *名称* 符号。这等效于在数值情况下使用 .set 指令定义 *名称*，或在其他情况下使用 .asg 指令进行定义。如果省略*值*，则该符号将设置为 1。请参阅节 4.8.5。
--asm_dependency	**-apd**	对执行汇编文件进行预处理，但不是写入预处理输出，而是将适合于输入的依赖行列表写入标准 make 实用程序。该列表将写入与源文件同名但扩展名为 .ppa 的文件中。
--asm_includes	**-api**	对汇编文件进行预处理，但不是写入预处理后的输出，而是写入 .include 指令包含的文件列表。该列表将写入与源文件同名但扩展名为 .ppa 的文件中。
--asm_listing	**-al**	生成一个与输入文件同名但扩展名为 .lst 的列表文件。
--asm_cross_reference_listing	**-ax**	生成交叉参考表并将它附加到列表文件的末尾；还将交叉参考信息添加到目标文件中以供交叉参考实用程序使用。如果您不请求列表文件，但使用 --asm_cross_reference_listing 选项，则汇编器会自动创建列表文件，使用与输入文件相同的名称为它命名，但使用 .lst 扩展名。请参阅节 4.14。
--asm_undefine=*name*	**-au**	取消定义预定义的常数 *名称*，这会覆盖指定常数的任何 --asm_define 选项。
--cmd_file=*filename*	**-@**	将文件的内容附加到命令行。您可以使用此选项来避免主机操作系统对命令行长度的限制。在命令文件的行首使用星号或分号（* 或 ; ）以包括注释。在任何其他列开始的注释必须以分号开头。在命令文件内，包含嵌入空格或连字符的文件名或选项参数必须用引号引起来。例如："this-file.asm"
--code_state={16|32}	**-mt**	--code_state=16（或 -mt）指示汇编器开始将指令汇编为 16 位指令；UAL 语法 (.thumb) 适用于 ARMv7，在其他场合使用非 UAL 语法 (.state16)。默认情况下，汇编器开始汇编 32 位指令。您可以通过指定 --code_state=32 来重置默认行为。有关 16 位与 32 位代码中的间接调用的信息，请参阅 *ARM 优化 C/C++ 编译器用户指南*。
--endian	**-me**	以小端字节序格式生成目标代码。如需更多信息，请参阅 *ARM 优化 C/C++ 编译器用户指南*。
--include_file=*filename*	**-ahi**	包含汇编模块的指定文件。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。
--include_path=*pathname*	**-I**	指定一个目录，让汇编器可以在其中找到由 .copy、.include 或 .mlib 指令命名的文件。以这种方式指定的目录数量没有限制；每个 pathname 前面必须有 --include_path 选项。请参阅节 4.5.1。
--quiet	**-q**	不显示横幅和进度信息（汇编器以静默模式运行）。
--symdebug:dwarf or **--symdebug:none**	**-g**	（默认设置为 DWARF）在 C 源调试器中启用汇编器源代码调试。每个汇编源代码行的行信息都输出到目标模块。不能对包含 .line 指令的汇编代码使用此选项。请参阅节 4.13。

4.4 控制应用二进制接口

应用二进制接口 (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 的存在是为了能让符合 ABI 的目标代码链接在一起，而不管其来源如何，并允许生成的可执行文件在支持该 ABI 的任何系统上运行。有关 EABI ABI 的信息，请参阅《*ARM 优化 C/C++ 编译器用户指南*》(SPNU151)。在 ARM 信息中心可找到完整的 ARM ABI 规范。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

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

EABI 应用程序中的所有目标文件都必须是针对 EABI 构建的。链接器会检测目标模块符合不同 ABI 的情况并生成错误。

请注意，将汇编文件从 COFF API 转换为 EABI 需要对汇编代码进行一些更改。

4.5 指定用于汇编器输入的备用目录

.copy、.include 和 .mlib 指令告诉汇编器需使用外部文件中的代码。.copy 和 .include 指令告诉汇编器需从另一个文件中读取源语句，并告诉 .mlib 指令需指定一个包含宏函数的库。章节 5 提供了 .copy、.include 和 .mlib 指令的示例。这些指令的语法为：

> **.copy** ["]*filename*["]
> **.include** ["]*filename*["]
> **.mlib** ["]*filename*["]

filename 指定一个供汇编器读取语句的复制文件/头文件，或一个包含宏定义的宏库。如果 *filename* 以数字开头，则需要双引号。之所以建议使用引号，是为了在处理包含在文件名规格中的路径信息或包含空格的路径名时不会出现问题。文件名可以是完整路径名、部分路径名或没有路径信息的文件名。

汇编器按照给定的顺序在以下位置中搜索文件：

1. 包含当前源文件的目录。当前源文件是遇到 .copy、.include 或 .mlib 指令时所汇编的文件。
2. 使用 --include_path 选项指定的任何目录
3. 使用 TI_ARM_C_DIR 环境变量指定的任何目录
4. 使用 TI_ARM_C_DIR 环境变量指定的任何目录

由于这种搜索层次结构，您可以使用 --include_path 选项（在节 4.5.1 中介绍）或 TI_ARM_A_DIR 环境变量（在节 4.5.2 中介绍）来增强汇编器的目录搜索算法。有关 TI_ARM_C_DIR 环境变量的讨论，请参阅《*ARM 优化 C/C++ 编译器用户指南*》。

备注

如果 TI_ARM_C_DIR 环境变量与较旧的 TMS470_C_DIR 环境变量都已定义，则前者优先于后者。如果只设置了 TMS470_C_DIR，则将继续使用它。同样，如果 TI_ARM_A_DIR 环境变量与较旧的 TMS470_A_DIR 环境变量都已定义，则前者优先于后者。。如果只设置了 TMS470_A_DIR，则将继续使用它。

4.5.1 使用 --include_path 汇编器选项

--include_path 汇编器选项指定了包含复制/头文件或宏库的备用目录。--include_path 选项的格式如下：

> **armcl --include_path=** *pathname source filename* [*other options*]

每次调用时，--include_path 选项的数量没有限制；每个 --include_path 选项指定一个路径名。在汇编源中，可以使用 .copy、.include 或 .mlib 指令，无需指定路径信息。如果汇编器在包含当前源文件的目录中找不到文件，会搜索由 --include_path 选项指定的路径。

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

```
    .copy "copy.asm"
```

假设 copy.asm 文件位于以下路径：

UNIX:	/tools/files/copy.asm
Windows：	c:\tools\files\copy.asm

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential – NDA Restrictions

您可以使用如下所示的命令来设置搜索路径：

操作系统	输入
UNIX (Bourne shell)	`armcl --include_path=/tools/files source.asm`
Windows	`armcl --include_path=c:\tools\files source.asm`

汇编器首先在当前目录中搜索 source.asm，因为 source.asm 位于当前目录中。然后汇编器会搜索 --include_path 选项指定的目录。

4.5.2 使用 TI_ARM_A_DIR 环境变量

环境变量是由用户定义并向其分配字符串的系统符号。汇编器使用 TI_ARM_C_DIR 环境变量来指定包含复制文件/头文件或宏库的备用目录。

汇编器会寻找 TI_ARM_A_DIR 环境变量，然后对其进行读取和处理。如果汇编器未找到 TI_ARM_A_DIR 变量，接下来会搜索 TI_ARM_C_DIR。如同时将德州仪器 (TI) 工具用于不同的处理器，那么特定于处理器的变量就很有用。

请参阅 *ARM 优化 C/C++ 编译器用户指南*，了解有关 TI_ARM_C_DIR 的详细信息。

备注

TI_ARM_C_DIR 环境变量优先于较旧的 TMS470_C_DIR 环境变量（如果两者均已定义）。如果只设置了 TMS470_C_DIR，则将继续使用它。同样，TI_ARM_A_DIR 环境变量优先于较旧的 TMS470_A_DIR 环境变量（如果两者均已定义）。如果只设置了 TMS470_A_DIR，则将继续使用它。

分配环境变量的命令语法如下：

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

pathnames 是包含复制/头文件或宏库的目录。pathnames 必须遵循以下约束：

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

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

- 路径中允许使用空格和制表符来适应包含空格的 Windows 目录。例如，以下命令中的路径名是有效的：

在汇编源中，可以使用 .copy、.include 或 .mlib 指令，无需指定路径信息。如果汇编器在包含当前源文件或由 --include_path 选项指定的目录中找不到文件，会搜索由环境变量指定的路径。

例如，假设名为 source.asm 的文件中包含以下语句：

```
      .copy "copy1.asm"
      .copy "copy2.asm"
```

假设文件位于以下路径：

UNIX：	/tools/files/copy1.asm 和 /dsys/copy2.asm
Windows：	c:\tools\files\copy1.asm 和 c:\dsys\copy2.asm

您可以使用如下所示的命令设置搜索路径：

Copyright © 2023 Texas Instruments Incorporated

操作系统	输入
UNIX (Bourne shell)	`TI_ARM_A_DIR="/dsys"; export TI_ARM_A_DIR` `armcl --include_path=/tools/files source.asm`
Windows	`TI_ARM_A_DIR=c:\dsys` `armcl --include_path=c:\tools\files source.asm`

汇编器首先在当前目录中搜索 copy1.asm 和 copy2.asm，因为 source.asm 位于当前目录中。然后汇编器会搜索 --include_path 选项指定的目录并寻找 copy1.asm。最后，汇编器会搜索 TI_ARM_A_DIR 指定的目录并寻找 copy2.asm。

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之一来重置变量：

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

4.6 源语句格式

ARM 汇编输入文件中的每一行都可以是空白、注释、汇编器指令、宏调用或汇编指令。

汇编语言源语句可以包含四个有序字段（标号、助记符、操作数列表和注释）。源语句的一般语法如下：

[*label*[:]]*mnemonic* [*operand list*][;*comment*]

以下是源语句示例：

```
SYM1     .set    2              ; Symbol SYM1 = 2
Begin:  MOV     R0, #SYM1      ; Load R0 with 2
         .word  016h            ; Initialize word (016h)
```

ARM 汇编器每行可读取的字符数量不受限制。长度超过 400 字符的源语句（含注释）会在列表文件中被截断。

请遵循以下指南：

- 所有语句都必须以标号、空白、星号或分号开头。
- 标号对于大多数语句来说是可选的；如果使用，则必须从第 1 列开始。
- 每个字段之间必须用一个或多个空格或制表符分隔。
- 注释为可选项。从第 1 列开始的注释可以用星号或分号（* 或 ; ）开头，但在任何其他列中开始的注释都*必须*以分号开头。

备注

助记符不能在第 1 列中开始，否则它会被解析为标号。助记符操作码和不带 . 前缀的汇编器指令名称都是有效的标号名称。务必要在助记符前面加上空格字符，否则汇编器会将该标识符视为新的标号定义。

以下各节将分别介绍各个字段。

4.6.1 标签字段

标签必须是放置在第 1 列中的合法标识符（请参阅节 4.8.1）。每条指令都可以选择性地设置一个标签。许多指令允许使用标签，有些指令必须使用标签。

标签后面可以跟一个冒号 (:)。冒号不属于标签名称的一部分。如果不使用标签，则第一个字符位置必须包含空格、分号或星号。

Copyright © 2023 Texas Instruments Incorporated

对汇编指令或数据指令使用标签时，会创建一个同名的*汇编器符号*（节 4.8）。此符号的值是*段程序计数器*（SPC，请参阅节 2.4.5）的当前值。此符号表示该指令的地址。在以下示例中，.word 指令用于创建一个包含 3 个字的数组。由于使用了标签，汇编符号 Start 指的是第一个字，且该符号的值将为 40h。

```
.    .    .    .
      9                   * Assume some code was assembled
     10 00000040 0000000A  Start:  .word 0Ah,3,7
        00000044 00000003
        00000048 00000007
```

一行中的标签本身就是一条有效的语句。当一个标签单独出现在一行中时，它指向下一行上的指令（SPC 不递增）：

```
      1 00000000          Here:
      2 00000000 00000003          .word 3
```

一行中的标签本身相当于以下写法：

```
Here: .equ $  ; $ provides the current value of the SPC
```

如果不使用标签，则第 1 列中的字符必须是空白、星号或分号。

4.6.2 助记符字段

助记符字段位于标签字段之后。助记符字段不能在第 1 列中开始，否则会被解析为标签。但有一个例外：助记符字段的双竖线 (||) 可以在第 1 列中开始。助记符字段包含以下项之一：

- 机器指令助记符（如 ADD、MUL、STR）
- 汇编器指令（如 .data、.list、.equ）
- 宏指令（如 .macro、.var、.mexit）
- 宏启用

4.6.3 操作数字段

操作数字段位于助记符字段之后，包含零个或更多由逗号分隔的操作数。操作数包括：

- 立即操作数（通常是一个常数或符号）（请参阅节 4.7 和节 4.8）
- 寄存器操作数
- 存储器引用操作数
- 评估以上操作数之一的表达式（请参阅节 4.9）

立即操作数 直接在指令中进行编码。立即操作数的值必须为一个*常数表达式*。具有立即操作数的大多数指令都需要一个*绝对常数表达式*，例如 1234。一些指令（例如调用指令）支持*可重定位的常数表达式*，例如在另一个文件中定义的符号。（请参阅节 4.9，了解表达式类型的详细信息。）

寄存器操作数 是一个特殊的预定义符号，表示一个 CPU 寄存器。

存储器引用操作数 使用若干存储器寻址模式之一来引用存储器中的一个位置。存储器引用操作数使用一种特殊的特定目标语法，该语法在相应的 *CPU 和指令集参考指南* 中定义。

您必须使用逗号分隔操作数。并非所有操作数均支持所有操作数类型。请参阅适用于您的器件系列的 *CPU 和指令集参考指南*，了解具体指令。

4.6.3.1 指令中的操作数语法

汇编器支持将操作数用作地址、直接值、间接地址、寄存器、移位寄存器或寄存器列表。以下规则适用于指令的操作数。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

- **# 前缀 — 操作数是直接值**。如果使用 # 符号作为前缀，汇编器会将操作数视为直接值。即使操作数为寄存器也适用；汇编器会将寄存器视为值，而不是使用寄存器的内容。例如：

```
Label:  ADD R1, R1, #123
        ; Add 123 (decimal) to the value of R1 and place the result in R1.
```

- **方括号 — 操作数是间接地址**。如果将操作数放在方括号内，汇编器会将操作数视为间接地址；即将操作数的内容作为地址使用。间接地址由基址和偏移组成。基址由寄存器指定并由寄存器中的值构成。偏移可以由寄存器、直接值或移位寄存器指定。此外，偏移可以指定为以下之一：
  - 前索引，其中基址和偏移合并构成地址。若要指定前索引偏移，请将偏移放在右侧结束括号内。
  - 后索引，其中从基址构成地址，然后将基址与偏移合并。若要指定后索引偏移，请将偏移放在右侧括号之外。

  偏移可以加到基址或从基址减去。以下是使用间接地址作为操作数的指令示例：

```
A: LDR R1, [R1]
     ; Load from address in R1 into R1.
   LDR R7, [R1, #5]
     ; Form address by adding the value in R1 to 5.Load from address into R7.
   STR R3, [R1, -R2]
     ; Form address by subtracting the value in R2 from the value in R1.Store from R3
     ; to memory at address.
   STR R14, [R1, +R3, LSL #2]
     ; Form address by adding the value in R3 shifted left by 2 to the value in R1.
     ; Store from R14 to memory at address.
   LDR R1, [R1], #5
     ; Load from address in R1 into R1, then add 5 to the address.
   STR R2, [R1], R5
     ; Store value in R2 in the address in R1, then add the value in R5 to the address.
```

- **! 后缀 — 写回寄存器**。如果使用 ! 符号作为后缀，汇编器会将计算出的地址写回基址寄存器。写回寄存器功能仅与间接寻址模式语法搭配使用。

  以下示例展示的指令使用了写回寄存器后缀：

```
LDR R1, [R4, #4]!
  ; Form address by adding the value in R4 to 4.Load from this address into R1,
  ; then replace the value in R4 with the address.
```

- **^ 后缀 — 设置 S 位**。如果使用 ^ 符号作为后缀，汇编器会设置 S 位。所产生的操作取决于所执行指令的类型以及 R15 是否在传送列表中。更多信息，请参阅 *TMS470R1x 用户指南* 中的 LDM 和 STM 指令。

```
LDMIA SP, {R4-R11, R15}^
  ; Load registers R4 through R11 and R15 from memory at SP.Load CPSR with SPSR.
```

- **移位寄存器**。如果寄存器符号后跟移位类型，计算的值为寄存器中根据以下所定义类型移位的值：

LSL	逻辑左移
LSR	逻辑右移
ASL	算术左移
ASR	算术右移
ROR	向右旋转
RRX	向右旋转扩展

  移位类型可以后跟寄存器或直接值，其值定义了移位量。以下是使用移位寄存器作为操作数的指令示例：

```
B: ADD R1, R4, R5, LSR R2
     ; Logical shift right the value in R5 by the value in R2.Add the value in R5 to R4.
     ; Place result in R1.
   LDR R1, [R5, R4, LSL #4]
     ; Form address by adding the value in R4 shifted left by 4 to the value in R5.
     ; Load from address into R1.
   CMP R3, R4, RRX
     ; Compare the value in R3 with the value in R4 rotate right extend.
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

- **花括号 - 操作数为寄存器列表**。如果在寄存器两边加上花括号，汇编器会将操作数视为寄存器列表。您可以用逗号分隔各个寄存器，也可以使用连字符指示寄存器范围。以下是使用寄存器列表的指令示例：

```
      LDMEA R2, {R1, R3, R6}
        ; Pre-decrement stack load.Load registers R1, R3 and R6 from memory at the address in R2.
      STMFD R12, {R1, R3-R5}
        ; Pre-increment stack store.Store from registers R1 and R3 through R5 to memory at the
        ; address in R12.
```

4.6.3.2 立即值作为指令的操作数

主要将立即值用作指令的操作数。在某些情况下，可将立即值与指令的操作数一同使用。例如，可将立即值与 .byte 指令一同用于将值加载到当前段中。

通常不需要为指令使用 # 前缀。比较以下语句：

```
   ADD R1, #10
   .byte 10
```

在第一条语句中，# 前缀是必要的，用于告知汇编器将值 10 与 R1 相加。但是，第二条语句中没有使用 # 前缀；汇编器期望操作数是一个值并通过值 10 初始化一个字节。

请参阅章节 5，了解更多有关指令的语法和用途的信息。

4.6.4 注释字段

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

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

4.7 字面常量

字面常量（也被称为*字面量*，在某些其他文档中被称为*立即值*）是表示自身的值，例如 12、3.14 或“hello”。

汇编器支持几种类型的字面量：

- 二进制整数字面量
- 八进制整数字面量
- 十进制整数字面量
- 十六进制整数字面量
- 字符字面量
- 字符串字面量
- 浮点字面量

汇编器会进行错误检查以查找无效或不完整的字面量。

4.7.1 整数字面量

汇编器在内部将每个整数字面量作为 32 位无符号量进行处理。字面量被认为是无符号值，并且不进行符号扩展。例如，字面量 00FFh 等于 00FF（十六进制）或 255（十进制）；它*不等于* -1，即 0FFFFFFFFh（十六进制）。请注意，如果将 0FFh 存储在 .byte 位置，则相应的位将与存储 -1 时完全相同。位的符号性由该位置的阅读器来解释。

4.7.1.1 二进制整数字面量

二进制整数字面量是一个字符串，最多包含 32 个二进制数字（0 和 1），后跟后缀 B（或 b）。还支持“0[bB][10]+”形式的二进制字面量。如果指定的位数少于 32 位，则汇编器会右对齐该值并用零填充未指定的位。以下是有效的二进制字面量的示例：

00000000B	等于 0_{10} 或 0_{16} 的字面量
0100000b	等于 32_{10} 或 20_{16} 的字面量

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

01b	等于 1_{10} 或 1_{16} 的字面量
11111000B	等于 248_{10} 或 $0F8_{16}$ 的字面量
0b00101010	等于 42_{10} 或 $2A_{16}$ 的字面量
0B101010	等于 42_{10} 或 $2A_{16}$ 的字面量

4.7.1.2 八进制整数字面量

八进制整数字面量是一个字符串，最多包含 11 个八进制数字（0 到 7），后跟后缀 Q（或 q）。八进制字面量也可能以 0 开头，不包含数字 8 或 9，也不以后缀结尾。以下是有效的八进制字面量的示例：

10Q	等于 8_{10} 或 8_{16} 的字面量
054321	等于 22737_{10} 或 $58D1_{16}$ 的字面量
100000Q	等于 32768_{10} 或 8000_{16} 的字面量
226q	等于 150_{10} 或 96_{16} 的字面量

4.7.1.3 十进制整数字面量

十进制整数字面量是包含十进制数字（范围从 -2147 483 648 到 4 294 967 295）的字符串。以下是有效的十进制整数字面量的示例：

1000	等于 1000_{10} 或 $3E8_{16}$ 的字面量
-32768	等于 $-32\,768_{10}$ 或 -8000_{16} 的字面量
25	等于 25_{10} 或 19_{16} 的字面量
4815162342	等于 4815162342_{10} 或 $11F018BE6_{16}$ 的字面量

4.7.1.4 十六进制整数字面量

十六进制整数字面量是一个字符串，最多包含 8 个十六进制数字，后跟后缀 H（或 h）或跟在 0x 之后。*如果十六进制字面量由 H 或 h 后缀表示，则它必须以十进制值 (0-9) 开头。*

十六进制数字包括十进制值 0-9 和字母 A-F 或 a-f。如果指定的十六进制数字少于八个，则汇编器将右对齐这些位。

以下是有效的十六进制字面量的示例：

78h	等于 120_{10} 或 0078_{16} 的字面量
0x78	等于 120_{10} 或 0078_{16} 的字面量
0Fh	等于 15_{10} 或 $000F_{16}$ 的字面量
37ACh	等于 14252_{10} 或 $37AC_{16}$ 的字面量

4.7.1.5 字符字面量

字符字面量是用*单*引号引起来的单个字符。字符在内部表示为 8 位 ASCII 字符。对于字符字面量中的每个单引号，需要使用两个连续的单引号进行表示。仅由两个单引号组成的字符字面量是有效的，并被赋值 0。以下是有效的字符字面量的示例：

'a'	定义字符字面量 *a* 并在内部表示为 61_{16}
'C'	定义字符字面量 *C* 并在内部表示为 43_{16}
''''	定义字符字面量 *'* 并在内部表示为 27_{16}
''	定义一个 null 字符并在内部表示为 00_{16}

注意字符*字面量*和*字符串字面量*的区别（节 4.7.2 讨论了字符串）。字符字面量表示单个整数值；字符串是一个字符序列。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

4.7.2 字符串字面量

字符串是用*双*引号引起来的字符序列。字符串中的双引号由两个连续的双引号表示。字符串的最大长度各不相同，对于每个需要字符串的指令，都要定义此长度。字符在内部表示为 8 位 ASCII 字符。

以下是有效字符串的示例：

"sample program"	定义包含 14 个字符的字符串 *sample program*。
"PLAN ""C"""	定义包含 8 个字符的字符串 *PLAN "C"*。

字符串用于以下用途：

- 文件名，如 .copy "filename"
- 段名，如 .sect "section name"
- 数据初始化指令，如 .byte "charstring"
- .string 指令的操作数

4.7.3 浮点字面量

浮点字面量包含一个十进制数字字符串，后跟一个必需的小数点、一个可选的小数部分和一个可选的指数部分。浮点数的语法为：

[+|-] *nnn* .[*nnn*] [**E**|**e** [+|-] *nnn*]

将 *nnn* 替换为一个十进制数字字符串。在 *nnn* 前面可以加上一个 + 或 -。必须指定一个小数点。例如，3.e5 有效，但 3e5 无效。指数表示 10 的幂。以下是有效的浮点字面量的示例：

```
3.0
3.14
3.
-0.314e13
+314.59e-2
```

汇编器语法并不支持所有 C89 样式的浮点字面量，也不支持 C99 样式的十六进制常量，但 $$strtod 内置数学函数支持这两者。如果您想使用其中一种格式来指定浮点字面量，请使用 $$strtod。例如：

```
$$strtod(".3")
$$strtod("0x1.234p-5")
```

不能直接使用 NaN、Inf 或 -Inf 作为浮点字面量，而是应该使用 $$strtod 来表示这些值。“NaN”和“Inf”字符串的处理不区分大小写。有关内置函数，请参阅节 4.10.1。

```
$$strtod("NaN")
$$strtod("Inf")
```

4.8 汇编器符号

汇编器符号是一个指定的 32 位无符号整数值，通常表示地址或绝对整数。符号可以表示诸如函数、变量或段的起始地址之类的内容。符号的名称必须是合法标识符。标识符成为符号值的符号表示，并可在后续指令中用于引用符号的位置或值。

一些汇编器符号成为外部符号，并放置在目标文件的符号表中。一个符号仅在定义这个符号的模块内有效，除非使用 .global 指令或 .def 指令将其声明为*外部符号*（请参阅 .global 指令）。

有关符号以及目标文件中的符号表的更多信息，请参阅节 2.6。

DRAFT ONLY
FI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

4.8.1 标识符

标识符是用作标签、寄存器、符号和替代符号的名称。标识符是由字母数字字符、美元符号和下划线（A-Z、a-z、0-9、$ 和 _）组成的字符串。标识符中的第一个字符不能是数字，标识符不能包含嵌入式空串。您定义的标识符区分大小写；例如，汇编器会将 ABC、Abc 和 abc 识别为三个不同的标识符。

4.8.2 标签

用作标签的标识符变成了汇编器符号，表示程序中的地址。文件内的标签必须是唯一的。

备注

助记符不能在第 1 列中开始，否则它会被解析为标签。助记符操作码和不带 . 前缀的汇编器指令名称都是有效的标签名称。请务必记得要在助记符前面加上空格字符，否则汇编器会将该标识符视为新的标签定义。

从标签派生而来的符号也可以用作 .bss、.global、.ref 或 .def 指令的操作数。

```
        .global_f
        LDR     A1, CON1
        STR     A1, [sp, #0]
        BL      _f
CON1:   .field  -269488145,32
```

4.8.3 局部标签

局部标签是特殊的标签，其范围和作用是暂时的。可通过两种方式定义局部标签：

- $n，其中 n 是 0-9 范围内的十进制数字。例如，$4 和 $1 是有效的局部标签。请参阅示例 4-1。
- *name*?，其中 *name* 是上述任何合法标识符。汇编器会将问号替换为一个句点，后跟一个唯一的数字。扩展源代码时，*您不会在列表文件中看到这个唯一的数字*。您的标签显示时带有问号，与源代码定义中的相同。

不能将这些类型的标签声明为全局标签。

普通标签必须具有唯一性（它们只能被声明一次），并且它们可以用作操作数字段中的常量。但是，局部标签可以取消定义并重新定义。局部标签不能由指令定义。

可通过以下方式之一取消定义或重置局部标签：

- 使用 .newblock 指令
- 更改段（使用 .sect、.text 或 .data 指令）
- 更改生成的代码的状态（使用 .state16 或 .state32 指令）
- 进入头文件（由 .include 或 .copy 指令指定）
- 离开头文件（由 .include 或 .copy 指令指定）

示例 4-1. $n 格式的局部标签

以下示例中的代码合法声明并使用了局部标签：

```
Label1: CMP     r1, #0          ; Compare r1 to zero.
        BCS     $1              ; If carry is set, branch to $1;
        ADDS    r0, r0, #1      ; else increment to r0
        MOVCS   pc, lr          ; and return.
$1:     LDR     r2, [r5], #4    ; Load indirect of r5 into r2
                                ; with write back.
        .newblock               ; Undefine $1 so it can be used
                                ; again.
        ADDS    r1, r1, r2      ; Add r2 to r1.
        BPL     $1              ; If the negative bit isn't set,
                                ; branch to $1;
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

```
        MVNS    r1, r1         ; else negate r1.
$1:     MOV     pc, lr         ; Return.
```

以下代码非法使用了局部标签：

```
        BCS     $1             ; If carry is set, branch to $1;
        ADDS    r0, r0, #1     ; else increment to r0
        MOVCS   pc, lr         ; and return.
$1:     LDR     r2, [r5], #4   ; Load indirect of r5 into r2
                               ; with write-back.
        ADDS    r1, r1, r2     ; Add r2 to r1.
        BPL     $1             ; If the negative bit isn't set,
                               ; branch to $1;
        MVNS    r1, r1         ; else negate r1.
$1:     MOV     pc, lr         ; Return.
```

$1 标签在由第二条分支指令重用之前已定义。因此，重新定义 $1 是非法的。

局部标签在宏中特别有用。如果宏包含普通标签并且被多次调用，则汇编器会发出“多重定义”错误。但是，如果在某个宏中使用局部标签和 .newblock，则每次展开这个宏时都会使用并重置局部标签。

一次最多启用 10 个 $n 形式的局部标签。name? 形式的局部标签不受限制。取消定义某个局部标签后，可以再次定义并使用这个标签。局部标签不会出现在目标代码符号表中。

有关在宏中使用标签的更多信息，请参阅节 6.6。

4.8.4 符号常数

符号常数是指值为绝对常数表达式的符号（请参阅节 4.9）。通过使用符号常数，您可以为常数表达式分配有意义的名称。.set 和 .struct/.tag/.endstruct 指令让您可以设置符号常数（请参阅“Define 汇编时常数”）。定义后，符号常数便*无法*重新定义。

如果您使用 .set 指令来为符号分配值，该符号便会成为符号常数，并可以在需要使用常数表达式的地方使用。例如：

```
shift3      .set    3
            MOV     R0, #shift3
```

您还可以使用 .set 指令来为寄存器名称等其他符号分配符号常数。在这种情况下，符号常数会成为对应寄存器的同义词：

```
AuxR1       .set  R1
            LDR  AuxR1, [SP]
```

下面的例子显示了 .set 指令可以如何与 .struct、.tag 和 .endstruct 指令搭配使用。它会创建符号常数 K、maxbuf、item、value、delta 和 i_len。

```
K      .set     1024                ;constant definitions
maxbuf .set     2*K
item   .struct                      ;item structure definition
       .int     value               ;constant offsets value = 0
       .int     delta               ;constant offsets delta = 1
i_len  .endstruct
array  .tag     item                ;array declaration
       .bss     array, i_len*K
```

汇编器还包含很多预定义的符号常数；节 4.8.6 中讨论了这些符号常数。

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential – NDA Restrictions

4.8.5 定义符号常量（--asm_define 选项）

--asm_define 选项使常量值或字符串与符号等同。随后可以使用该符号代替汇编源代码中的值。--asm_define 选项的格式如下：

armcl--asm_define= *name*[**=** *value*]

name 是用户要定义的符号的名称。*value* 是用户要分配给符号的常量或字符串值。如果省略 *value*，则该符号将设为 1。如果用户要定义带引号的字符串并保留引号，请执行以下操作之一：

- 对于 Windows，请使用 **--asm_define=** *name* **="\"** *value* **\""**。例如，--asm_define=car="\"sedan\""
- 对于 UNIX，请使用 **--asm_define=** *name* **='"** *value* **"'**。例如，--asm_define=car='"sedan"'
- 对于 Code Composer，请在文件中输入定义并使用 --cmd_file（或 -@）选项包含该文件。

一旦用户使用 --asm_define 选项定义了名称，该符号就可与汇编指令和其他指令一同使用，就像该符号是使用 .set 指令定义的一样。例如，在命令行中输入：

```
armcl --asm_define=SYM1=1 --asm_define=SYM2=2 --asm_define=SYM3=3 --asm_define=SYM4=4 value.asm
```

用户已经为 SYM1、SYM2、SYM3 和 SYM4 赋值，因此可以在源代码中使用它们。本节末尾的示例显示了 value.asm 文件如何在不显式定义这些符号的情况下使用它们。

在汇编器源代码中，可通过以下指令测试使用 --asm_define 选项定义的符号：

测试类型	指令用法
存在	**.if $$isdefed("** *name* **")**
不存在	**.if $$isdefed("** *name* **") = 0**
等于值	**.if** *name* **=** *value*
不等于值	**.if** *name* **!=** *value*

$$isdefed 内置函数的参数必须用引号引起来。引号使参数按字面解释而非作为替代符号。

该示例使用在本节前面的命令行中定义的符号常量：

```
IF_4:  .if      SYM4 = SYM2 * SYM2
       .byte    SYM4          ; Equal values
       .else
       .byte    SYM2 * SYM2   ; Unequal values
       .endif
IF_5:  .if      SYM1 <= 10
       .byte    10            ; Less than / equal
       .else
       .byte    SYM1          ; Greater than
       .endif
IF_6:  .if      SYM3 * SYM2 != SYM4 + SYM2
       .byte    SYM3 * SYM2   ; Unequal value
       .else
       .byte    SYM4 + SYM4   ; Equal values
       .endif
IF_7:  .if      SYM1 = SYM2
       .byte    SYM1
       .elseif  SYM2 + SYM3 = 5
       .byte    SYM2 + SYM3
       .endif
```

4.8.6 预定义符号常量

汇编器具有几类预定义符号。

$，美元符号，表示段程序计数器（SPC）的当前值。

此外，还提供以下预定义处理器符号常量：

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

表 4-2. ARM 处理器符号常量

宏名称	说明
.TI_ARM	始终设为 1
.TI_ARM_16BIS	如果默认状态为 16 位 Thumb 模式（--code_state=16 选项用于 ARMv6 或更早架构），则设为 1；否则设为 0。
.TI_ARM_32BIS	如果默认状态为 32 位（未使用 --code_state=16 选项，或使用 --code_state=32 选项），则设为 1；否则设为 0。
.TI_ARM_T2IS	如果默认状态为 Thumb-2 模式（--code_state=16 选项用于 ARMv7 或更高架构），则设为 1；否则设为 0。
.TI_ARM_LITTLE	如果选择小端字节序模式（使用 --endian 汇编器选项），则设为 1；否则设为 0。
.TI_ARM_BIG	如果选择大端字节序模式（未使用 --endian 汇编器选项），则设为 1；否则设为 0。
__TI_ARM7ABI_ASSEMBLER	如果启用 TI ARM7 ABI（使用 --abi=tiabi 选项），则设为 1；否则设为 0。（已弃用此选项。）
__TI_ARM9ABI_ASSEMBLER	如果启用 TI ARM9 ABI（使用 --abi=ti_arm9_abi 选项），则设为 1；否则设为 0。（已弃用此选项。）
__TI_EABI_ASSEMBLER	如果启用 EABI ABI，则设为 1。EABI 目前是唯一受支持的 ABI；请参阅节 4.4。
__TI_NEON_SUPPORT__	如果目标是 NEON SIMD 扩展（使用 --neon 选项），则设为 1；否则设为 0。
__TI_ARM_V4__	如果目标是 v4 架构（ARM7）（使用 -mv4 选项），则设为 1；否则设为 0。
__TI_ARM_V5E__	如果目标是 v5E 架构（ARM9E）（使用 -mv5e 选项），则设为 1；否则设为 0。
__TI_ARM_V6__	如果目标是 v6 架构（ARM11）（使用 -mv6 选项），则设为 1；否则设为 0。
__TI_ARM_V6M0__	如果目标是 v6M0 架构（Cortex-M0）（使用 -mv6M0 选项），则设为 1；否则设为 0。
__TI_ARM_V7__	如果目标是任何 v7 架构（Cortex），则设为 1；否则设为 0。
__TI_ARM_V7A8__	如果目标是 v7A8 架构（Cortex-A8）（使用 -mv7A8 选项），则设为 1；否则设为 0。
__TI_ARM_V7M3__	如果目标是 v7M3 架构（Cortex-M3）（使用 -mv7M3 选项），则设为 1；否则设为 0。
__TI_ARM_V7M4__	如果目标是 v7M4 架构（Cortex-M4）（使用 -mv7M4 选项），则设为 1；否则设为 0。
__TI_ARM_V7R4__	如果目标是 v7R4 架构（Cortex-R4）（使用 -mv7R4 选项），则设为 1；否则设为 0。
__TI_VFP_SUPPORT__	如果启用 VFP 协处理器（使用任何 --float_support 选项），则设为 1；否则设为 0。
__TI_VFPV3_SUPPORT__	如果启用 VFP 协处理器（使用 --float_support=vfpv3 选项），则设为 1；否则设为 0。
__TI_VFPV3D16_SUPPORT__	如果启用 VFP 协处理器（使用 --float_support=vfpv3d16 选项），则设为 1；否则设为 0。
__TI_FPV4SPD16_SUPPORT__	如果启用 FP 协处理器（使用 --float_support=fpv4spd16 选项），则设为 1；否则设为 0。

4.8.7 寄存器

此外，控制寄存器的名称为预定义符号。

ARM 寄存器的名称及其别名为寄存器符号，包括：

- 协处理器寄存器，包括 C0-C15。
- 协处理器 ID，包括 P0-P15。
- VFP 寄存器，包括 D0-D31、S0-S31。
- NEON 寄存器，包括 D0-D31、Q0-Q15。

表 4-3. ARM 寄存器符号与别名

寄存器名称	别名	寄存器名称	别名
R0	A1	R8	V5
R1	A2	R9	V6
R2	A3	R10	V7
R3	A4	R11	V8
R4	V1	R12	V9，IP
R5	V2	R13	SP
R6	V3	R14	LR
R7	V4，AP	R15	PC

Copyright © 2023 Texas Instruments Incorporated

寄存器符号和别名可输入全大写或全小写字符。 例如，R13 也可输入为 r13、SP 或 sp。

控制寄存器符号可输入全大写或全小写字符。

请参阅《*ARM 优化 C/C++ 编译器用户指南*》中的“寄存器惯例”一节，了解寄存器及其使用的详细信息。

状态寄存器可输入全大写或全小写字符；即 CPSR 可输入为 cpsr、CPSR_ALL 或 cpsr_all。

表 4-4. ARM 状态寄存器和别名

注册	别名	说明
CPSR	CPSR_ALL	当前处理器状态寄存器
CPSR_FLG		仅限当前处理器状态寄存器的标志位
SPSR	SPSR_ALL	已保存的处理器状态寄存器
SPSR_FLG		仅限已保存的处理器状态寄存器的标志位

4.8.8 替代符号

可以为符号分配字符串值。如此，用户便可以通过使字符串等于符号名称来为字符串创建别名。表示字符串的符号被称为替代符号。汇编器在遇到替代符号时，会将符号名称替换为其字符串值。与符号常数不同，替代符号可以重新定义。

用户可以在程序中的任意位置为替代符号分配字符串；例如：

```
    .asg   "SP", stack-pointer
       ; Assigns the string SP to the substitution symbol stack-pointer.
    .asg   "#0x20", block2
       ; Assigns the string #0x20 to the substitution symbol block2.
    ADD    stack-pointer, stack-pointer, block2
       ; Adds the value in SP to #0x20 and stores the result in SP.
```

使用宏时，替代符号非常重要，因为宏参数实际上就是已分配宏参数的替代符号。以下代码显示了如何在宏中使用替代符号：

```
addl  .macro dest, src
         ; addl macro definition
      ADDS   dest, dest, src
         ; Add the value in register dest to the value in register src,
         ; and store the result in src.
      BLCS   reset_ctr
         ; Handle overflow.
      .endm
*addl invocation
      addl   R4, R5
         ; Calls the macro addl and substitutes R4 for dest and R5 for src.
         ; The macro adds the value of R4 and the value of R5, stores the
         ; result in R4, and handles overflow.
```

有关宏的更多信息，请参阅章节 6。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

4.9 表达式

汇编语言中几乎所有的值和操作数都是*表达式*，可能是以下任何一种：

- 字面常量
- 寄存器
- 寄存器对
- 存储器引用
- 符号
- 内置函数调用
- 对一个或多个表达式进行的数学或逻辑运算

本节定义了本文档中引用的几种表达式。一些指令操作数接受的表达式类型有限。例如，.if 指令要求其操作数是具有整数值的绝对常量表达式。在汇编代码的上下文中，“绝对”意味着必须在汇编时知道表达式的值。

*常量表达式*是指任何不以任何方式涉及寄存器或存储器引用的表达式。*立即操作数*通常不接受寄存器或内存引用。必须为其指定一个常量表达式。常量表达式可能是以下任何一种：

- 字面常量
- 地址常量表达式
- 值为常量表达式的符号
- 对常量表达式进行的内置函数调用
- 对一个或多个常量表达式进行的数学或逻辑运算

*地址常量表达式*是常量表达式的一个特例。一些需要地址值的立即操作数可以接受一个符号加一个加数；例如，一些分支指令。该符号必须具有地址值，并且可以是外部符号。该加数必须是具有整数值的绝对常量表达式。例如，一个有效的地址常量表达式为“array+4”。

常量表达式可以是绝对表达式或可重定位表达式。*绝对*意味着在汇编时已知。*可重定位*意味着会是常量的，但直到链接时才已知。外部符号即使引用同一模块中定义的符号，也是可重定位的符号。

*绝对常量表达式*不能引用表达式中任何位置的任何外部符号。换句话说，绝对常量表达式可以是以下任何一种：

- 字面常量
- 绝对地址常量表达式
- 值为绝对常量表达式的符号
- 参数全部是绝对常量表达式的内置函数调用
- 对一个或多个绝对常量表达式进行的数学或逻辑运算

*可重定位的常量表达式*引用至少一个外部符号。对于 ELF，此类表达式最多可包含一个外部符号。可重定位的常量表达式可以是以下任何一种：

- 外部符号
- 可重定位的地址常量表达式
- 值为可重定位的常量表达式的符号
- 任何参数是可重定位的常量表达式的内置函数调用
- 对一个或多个表达式（其中至少有一个是可重定位的常量表达式）进行的数学或逻辑运算

在某些情况下，可重定位的地址表达式的值在汇编时可能是已知的。例如，相对位移分支可能会分支到在同一个段中进行定义的标签。

4.9.1 数学和逻辑运算符

数学或逻辑运算符的操作数必须是正确定义的表达式。也就是说，必须使用正确数量的操作数，而且运算必须有意义。例如，不能对浮点值进行 XOR 运算。此外，正确定义的表达式仅包含符号或汇编时常量，它们已提前定义，然后才能出现在指令的表达式中。

三个主要因素会影响表达式求值顺序：

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

圆括号

总是首先计算圆括号中的表达式。
8 / (4 / 2) = 4，但 8 / 4 / 2 = 1
不能 用大括号 ({ }) 或方括号 ([ ]) 代替圆括号。

优先级组

表 4-5 中列出的运算符分为九个优先级组。当圆括号不能确定表达式求值顺序时，首先计算优先级最高的运算。
8 + 4 / 2 = 10（首先计算 4 / 2）

从左到右求值

当圆括号和优先级组不能确定表达式求值顺序时，表达式从左到右求值，但第 1 组是从右到左求值。
8 / 4*2 = 4，但 8 / (4*2) = 1

表 4-5 根据优先级组列出了可在表达式中使用的运算符。

表 4-5. 表达式中使用的运算符（优先级）

组[(1)]	运算符	描述[(2)]
1	+	一元加
	-	一元减
	~	一的补码
	!	逻辑非
2	*	乘法
	/	除法
	%	模数
3	+	加法
	-	减法
4	<<	左移
	>>	右移
5	<	小于
	<=	小于或等于
	>	大于
	>=	大于或等于
6	=[=]	等于
	!=	不等于
7	且	按位与
8	^	按位异或 (XOR)
9	|	按位或

(1) 从右到左计算第 1 组运算符。从左到右计算所有其他运算符。
(2) 一元 + 和 - 比二元形式具有更高的优先级。

在汇编期间执行算术运算时，汇编器会检查是否有上溢和下溢情况。每当发生上溢或下溢时，汇编器都会发出警告（“value truncated”消息）。汇编器不检查乘法中的上溢或下溢。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

4.9.2 关系运算符和条件表达式

汇编器支持关系运算符，可用于任何表达式；这对于条件汇编特别有用。有如下关系运算符：

=	等于	**! =**	不等于
<	小于	**<=**	小于或等于
>	大于	**> =**	大于或等于

条件表达式如果为真则判断为 1，如果为假则判断为 0，只能用于相同类型的操作数；例如绝对值可与绝对值比较，但绝对值不能与可重定位的值比较。

4.9.3 明确定义的表达式

一些汇编器指令（例如 .if）需要使用明确定义的绝对常数表达式作为操作数。明确定义的表达式仅包含符号或汇编时常数，它们已提前定义，然后才能出现在指定的表达式中。此外，它们必须使用正确数量的操作数，而且操作必须有意义。对明确定义的表达式进行的评估必须无歧义。

这是明确定义的表达式示例：

```
1000h+X
```

其中 X 已在之前定义为绝对符号。

4.9.4 可重定位的符号和合法表达式

所有合法表达式均可简化为以下两种形式之一：

可重定位的符号 ± *绝对符号*

或

绝对值

一元运算符只能应用于绝对值；无法应用于可重定位的符号。表达式如果无法简化为仅包含一个可重定位的符号，则是非法的。

表 4-6 总结了针对绝对、可重定位和外部符号的有效操作。表达式无法包含可重定位的符号或外部符号的乘法或除法。表达式无法包含可重定位到其他段的未解析符号。

利用 .global 指令定义为全局符号的符号也可用于表达式中；在表 4-6 中，这些符号被称为*外部*符号。

表 4-6. 具有绝对符号和可重定位符号的表达式

如果 A 是... 并且	如果 B 是...，那么	A + B 是... 并且	A - B 是...
绝对	绝对	绝对	绝对
绝对	可重定位	可重定位	非法
绝对	外部	外部	非法
可重定位	绝对	可重定位	可重定位
可重定位	可重定位	非法	绝对(1)
可重定位	外部	非法	非法
外部	绝对	外部	外部
外部	可重定位	非法	非法
外部	外部	非法	非法

(1) A 和 B 必须在同一段中；否则将可重定位的符号添加到可重定位的符号是非法的。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

4.9.5 表达式示例

以下是使用可重定位符号和绝对符号的表达式示例。这些示例使用了在同一个段中定义的四个符号：

```
          .global extern_1  ; Defined in an external module
intern_1: .word '"D'        ; Relocatable, defined in current
                            ;   module
LAB1:     .set 2            ; LAB1 = 2
intern_2                    ; Relocatable, defined in current
                            ;   module
intern_3                    ; Relocatable, defined in  current
                            ;   module
```

- **示例 1**

  此示例中的语句使用已定义且值为 2 的绝对符号 LAB1。第一条语句将值 51 加载到 R0 中。第二条语句将值 27 加载到 R0 中。

```
      MOV  R0, #LAB1  +  ((4+3)  *  7) ;  R0 = 51
                                       ; 2 + ((7) * 7)
                                       ; 2 + (49) = 51

      MOV  R0, #LAB1  +  4  +  (3*7)   ;  R0 = 27
                                       ; 2 + 4 + (21) = 27
```

- **示例 2**

  以下示例中的第一条语句是有效的，但其后的语句是无效的。

```
      LDR R1, intern_1 - 10       ; Legal
      LDR R1, 10-intern_1         ; Can't negate reloc. symbol
      LDR R1, -(intern_1)         ; Can't negate reloc. symbol
      LDR R1, intern_1/10         ; / isn't additive operator
      LDR R1, intern_1 + intern_2 ; Multiple relocatables
```

- **示例 3**

  下面的第一条语句是合法的；尽管 intern_1 和 intern_2 是可重定位的符号，但它们的差值是绝对值，因为它们位于同一个段中。从一个可重定位的符号减去另一个可重定位的符号会将表达式简化为*可重定位的符号* + *绝对值*。第二条语句是非法的，因为两个可重定位符号的总和并非绝对值。

```
      LDR R1, intern_1 - intern_2 + intern_3   ; Legal
      LDR R1, intern_1 + intern_2 + intern_3   ; Illegal
```

- **示例 4**

  可重定位符号在表达式中的放置位置对于表达式求值很重要。尽管下面的语句与前面示例中的第一条语句类似，但采用的却是从左到右的运算符优先级，因此这条语句是非法的；汇编器会尝试将 intern_1 与 extern_3 相加。

```
      LDR R1, intern_1 + intern_3 - intern_2   ; Illegal
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

4.10 内置函数和运算符

汇编器支持内置数学函数和内置寻址运算符。

节 6.3.2 中讨论了内置替代符号函数。

4.10.1 内置数学和三角函数

汇编器支持用于转换和各种数学计算的内置函数。表 4-7 描述了这些内置函数。*expr* 必须是一个常数值。

表 4-7. 内置数学函数

函数	说明
$$acos(*expr*)	以浮点值形式返回 *expr* 的反余弦值
$$asin(*expr*)	以浮点值形式返回 *expr* 的反正弦值
$$atan(*expr*)	以浮点值形式返回 *expr* 的反正切值
$$atan2(*expr*, *y*)	以 [-π，π] 范围中的浮点值形式返回 *expr* 的反正切值
$$ceil(*expr*)	返回不小于 *expr* 的最小整数
$$cos(*expr*)	以浮点值形式返回 *expr* 的余弦值
$$cosh(*expr*)	以浮点值形式返回 *expr* 的双曲余弦值
$$cvf(*expr*)	将 *expr* 转换为浮点值
$$cvi(*expr*)	将 *expr* 转换为整数值
$$exp(*expr*)	返回指数函数 e^{expr}
$$fabs(*expr*)	以浮点值形式返回 *expr* 的绝对值
$$floor(*expr*)	返回不大于 *expr* 的最大整数
$$fmod(*expr*, *y*)	返回 *expr1* ÷ *expr2* 的余数
$$int(*expr*)	如果 *expr* 具有整数值，则返回 1；否则返回 0。返回一个整数。
$$ldexp(*expr*, *expr2*)	将 *expr* 乘以 2 的整数幂。即 $expr1 \times 2^{expr2}$
$$log(*expr*)	返回 *expr* 的自然对数，其中 *expr*>0
$$log10(*expr*)	返回 *expr* 的以 10 为底的对数，其中 *expr*>0
$$max(*expr1*, *expr2*)	返回两个值中的最大值
$$min(*expr1*, *expr2*)	返回两个值中的最小值
$$pow(*expr1*, *expr2*)	返回 *expr1* 的 *expr2* 次幂
$$round(*expr*)	返回 *expr* 并舍入到最接近的整数
$$sgn(*expr*)	返回 *expr* 的符号。
$$sin(*expr*)	返回 *expr* 的正弦值
$$sinh(*expr*)	以浮点值形式返回 *expr* 的双曲正弦值
$$sqrt(*expr*)	以浮点值形式返回 expr 的平方根 (expr≥0)
$$strtod(*str*)	将字符串转换为双精度浮点值。该字符串包含格式正确的 C99 样式浮点字面量。
$$tan(*expr*)	以浮点值形式返回 *expr* 的正切值
$$tanh(*expr*)	以浮点值形式返回 *expr* 的双曲正切值
$$trunc(*expr*)	返回 *expr* 并向 0 舍入

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

4.11 统一汇编语言语法支持

统一汇编语言 (UAL) 是一种新的汇编语法，由 ARM Ltd. 推出，可控制原始 Thumb-2 汇编语法带来的歧义，并为 ARM、Thumb 和 Thumb-2 提供类似的语法。UAL 向后兼容旧版 ARM 汇编，但与之前的 Thumb 汇编语法不兼容。

从 ARMv7 架构开始，UAL 语法是默认的汇编语法。编写汇编代码时，.arm 和 .thumb 指令分别用于指定 ARM 和 Thumb UAL 语法。.state32 和 .state16 指令仍可指定非 UAL ARM 和 Thumb 语法。.arm 和 .state32 指令是等效的，因为在 ARM 模式下 UAL 语法是向后兼容的。Thumb-2 指令不支持非 UAL 语法，因此在 .state16 段中不能使用 Thumb-2 指令。但具有 .state16 段的汇编代码如果仅包含非 UAL Thumb 代码，可针对 ARMv7 架构进行汇编，从而能够方便地移植更早的代码。

请参阅节 5.3，了解有关 .state16、.state32、.arm 和 .thumb 指令的更多信息。

ARM Ltd. 文档中可找到有关 UAL 语法的全面介绍，但有一些与 Thumb-2 语法相关的关键差异：

- .W 扩展用于指定，应以 32 位形式编码指令。.N 扩展用于指定，应以 16 位形式编码指令；如果不可能，汇编器将报错。如果未使用扩展，汇编器将尽可能使用 16 位编码。
- 16 位 Thumb ALU 指令利用具有“S”修饰符的语法指示状态设置。这与 ARM ALU 指令一直以来设置状态的方式相同。

4.12 源程序列表

源程序列表显示了源语句及其生成的目标代码。若要获得源程序列表文件，请在调用汇编器时使用 --asm_listing 选项（请参阅节 4.3）。

每个源程序列表页面的顶部都有两个横幅行、一个空白行和一个标题行。由 .title 指令提供的任何标题都会列印在标题行中。页码会列印在标题的右侧。如果不使用 .title 指令，则会列印源文件的名称。汇编器会在标题行下插入一个空白行。

源文件中的每一行都会在源程序列表文件中生成至少一行。此行会显示源语句编号、SPC 值、汇编而来的目标代码以及源语句。图 4-2 显示了这些项目在实际源程序列表文件中的情况。

字段 1：源语句编号

行编号

源语句编号是十进制数。汇编器会按照源文件中出现的顺序对源代码行进行编号；一些语句会使行数计数器递增，但不会列出。（例如，.title 语句和跟在 .nolist 之后的语句不会列出。）两个连续源代码行编号之差表示源文件中这两个语句之间并未列出的语句数量。

包含文件字母

行编号前面的字母表示该行是从该字母所指定的包含文件汇编而来。

嵌套级别编号

行编号前面的数字表示宏扩展或循环块的嵌套级别。

字段 2：段程序计数器

此字段包含十六进制 SPC 值。所有段（.text、.data、.bss 和命名段）都会维护单独的 SPC。一些指令不影响 SPC 并会将此字段留空。

字段 3：目标代码

此字段包含目标代码的十六进制表示形式。所有机器指令都使用此字段来列出目标代码。此字段还会指示与该行源代码的操作数相关联的重定位类型。如果多个操作数可以重定位，此列会指示第一个操作数的重定位类型。下面列出了会出现在此列的字符及关联的重定位类型：

!	未定义的外部引用
'	可重定位的 .text

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

+	可重定位的 .sect
”	可重定位的 .data
-	可重定位的 .bss、.usect
%	重定位表达式

字段 4：源语句字段

此字段包含源语句中由汇编器扫描而来的字符。汇编器每行最多可以接受 200 个字符。此字段中的空格方式由源语句中的空格方式决定。

图 4-2 显示了一个汇编列表，其中分别标出了这四个字段。

```
Include file  Line number
letter
        1 00000000                    .state32
        2                             .copy "mac1.inc"
A       1                    to16     .macro
A       2                             ADD     r0, pc, #1
A       3                             BX      r0
A       4                             .state16
A       5
A       6                             .endm
        3
        4                             .global __stack
        5                    ;**********************************************
        6                    ;* DEFINE THE USER MODE STACK                 **
        7                    ;**********************************************
        8          00000200  STACKSIZE .set 512
        9 00000000           __stack:  .usect ".stack", STACKSIZE, 4
       10                    ;**********************************************
       11                    ;* INTERRUPT VECTORS                          **
       12                    ;**********************************************
       13                             .global reset
       14 00000000                    .sect ".intvecs"
       15
       16 00000000 EAFFFFFE'          B       reset
       17 00000004 00000000           .word 0
       18 00000008 00000000           .word 0
       19 0000000c 00000000           .word 0
       20 00000010 00000000           .word 0
       21 00000014 00000000           .word 0
       22 00000018 00000000           .word 0
       23 0000001c 00000000           .word 0
       24
       25 00000000                    .text
       26                             .global dispatch
       27                             .global reset
       28                    ;**********************************************
       29                    ;* RESET ROUTINE                              **
       30                    ;**********************************************
       31 00000000           reset:
       32                             ;*-----------------------------------
       33                             ;* SET TO USER MODE
       34                             ;*-----------------------------------
       35 00000000 E10F0000           MRS     r0, cpsr
       36 00000004 E3C0001F           BIC     r0, r0, #0x1F ; Clear modes
       37 00000008 E3800010           ORR     r0, r0, #0x10 ; Set user mode
       38 0000000c E129F000           MSR     cpsr, r0
       39
  Field 1   Field 2   Field 3                 Field 4
```

图 4-2. 汇编列表示例

DRAFT ONLY
TI Confidential – NDA Restrictions

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

Nesting level number

```
       40                                  ;*-------------------------------------
       41                                  ;* CHANGE TO 16 BIT STATE
       42                                  ;*-------------------------------------
       43 00000010                         to16
1         00000010 E28F0001                ADD    r0, pc, #1
1         00000014 E12FFF10                BX     r0
1         00000018                         .state16
1
       44
       45                                  ;*-------------------------------------
       46                                  ;* INITIALIZE THE USER MODE STACK
       47                                  ;*-------------------------------------
       48 00000018 4802                    LDR    r0, stack
       49 0000001a 4685                    MOV    sp, r0
       50 0000001c 4802                    LDR    r0, stacksz
       51 0000001e 4485                    ADD    sp, r0
       52
       53                                  ;*-------------------------------------
       54                                  ;* DISPATCH TASKS
       55                                  ;*-------------------------------------
       56 00000020 F7FF!                   BL     dispatch
          00000022 FFEE
       57 00000024 00000000- stack         .long  __stack
       58 00000028 00000200  stacksz       .long  STACKSIZE
       59
       60
       61
```

Field 1 Field 2 Field 3 Field 4

图 **4-3.** 汇编列表示例（续）

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

4.13 调试汇编源文件

默认情况下，当您编译汇编文件时，汇编器会提供符号调试信息，允许您在调试器中逐步调试汇编代码，而不是使用 Code Composer Studio 中的“Disassembly”窗口。这样，您可以在调试时查看源代码注释和其他源代码注释。默认与使用 --symdebug:dwarf 选项操作相同。您可以使用 --symdebug:none 选项禁止生成调试信息。

.asmfunc 和 .endasmfunc（请参阅 .asmfunc 指令）指令使您能够在汇编代码中使用 C 特性，这使得调试汇编文件的过程更类似于调试 C/C++ 源文件。

使用 .asmfunc 和 .endasmfunc 指令，您可以命名代码的某些区域，并使这些区域在调试器中显示为 C 函数。未包含在 .asmfunc 和 .endasmfunc 指令中的连续汇编代码段将自动放置在使用以下语法命名的汇编器定义函数中：

$ ***filename*** **:** ***starting source line*** **:** ***ending source line*** **$**

如果您想在 C 代码中将变量视为用户定义的类型，则必须声明类型且必须在 C 文件中定义变量。然后，可以使用 .ref 指令（请参阅 .ref 指令）在汇编代码中引用此 C 文件。下面的 C 示例显示了 cvar.c 程序，该程序将变量 svar 定义为结构类型 X。然后在后面的 addfive.asm 汇编程序中引用 svar 变量，并将 5 添加到 svar 的第二个数据成员。

使用 --symdebug:dwarf 选项 (-g) 编译两个源文件，并按如下方式链接它们：

```
armcl --symdebug:dwarf cvars.c addfive.asm --run_linker --library=lnk.cmd
      --library=rtsv4_A_be_eabi.lib --output_file=addfive.out
```

当您将此程序加载到符号调试器中时，addfive 将显示为 C 函数。您可以在逐步执行 main 的同时，像监视任何常规 C 变量一样监视 svar 中的值。

将汇编变量视为 C 类型 C 程序

```
typedef struct {
   int m1;
   int m2;
} X;
X svar = { 1, 2 };
```

addfive.asm 汇编语言程序

```
; Tell the assembler we're referencing variable "_svar", which is defined in
; another file (cvars.c).
;------------------------------------------------------------------------------
          .ref _svar
;------------------------------------------------------------------------------
; addfive() - Add five to the second data member of _svar
;------------------------------------------------------------------------------
          .text
          .global addfive
addfive:  .asmfunc
           LDW      .D2T2    *+B14(_svar+4),B4 ; load svar.m2 into B4
           RET      .S2      B3                ; return from function
           NOP               3                 ; delay slots 1-3
           ADD      .D2      5,B4,B4           ; add 5 to B4 (delay slot 4)
           STW      .D2T2    B4,*+B14(_svar+4) ; store B4 back into svar.m2
                                               ; (delay slot 5)
          .endasmfunc
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

4.14 交叉引用列表

交叉引用列表会显示符号及其定义。要获得交叉引用列表，请使用 --asm_cross_reference_listing 选项来调用汇编器（请参阅节 4.3），或将 .option 指令与 X 操作数一起使用（请参阅选择列表选项）。汇编器会将交叉引用附加到源代码列表的末尾。以下示例展示了交叉引用列表中包含的四个字段。

```
LABEL                              VALUE       -DEFN    REF
.TI_ARM                            00000001        0
.TI_ARM_16BIS                      00000000        0
.TI_ARM_32BIS                      00000001        0
.TI_ARM_BIG                        00000001        0
.TI_ARM_LITTLE                     00000000        0
.ti_arm                            00000001        0
.ti_arm_16bis                      00000000        0
.ti_arm_32bis                      00000001        0
.ti_arm_big                        00000001        0
.ti_arm_little                     00000000        0
STACKSIZE                          00000200        9      10      63
__stack                            00000000-      10       5      62
dispatch                           REF            29      60
reset                              00000000'      34      16      19      30
stack                              00000024'      62      52
stacksz                            00000028'      63      54
```

Label（标签） 列包含在汇编过程中定义或引用的每个符号。

Value（值） 列包含一个 8 位十六进制数（这是分配给符号的值）*或*一个描述符号属性的名称。值的前面也可以是一个描述符号属性的字符。下表列出了这些字符和名称。

Definition (DEFN)（定义）列包含定义该符号的语句编号。对于未定义的符号，此列为空白。

Reference (REF)（引用）列会列出引用该符号的语句的行号。此列空白表示从未使用过该符号。

表 4-8. 符号属性

字符或名称	含义
REF	外部引用（全局符号）
UNDF	未定义
'	在 .text 段中定义的符号
"	在 .data 段中定义的符号
+	在 .sect 段中定义的符号
-	在 .bss 或 .usect 段中定义的符号

编译器也提供了类似的 --gen_cross_reference_listing 选项，该选项会生成一个列表文件，其中包含 C/C++ 源文件中标识符的参考信息。 请参阅 *ARM 优化 C/C++ 编译器用户指南* 中的“生成交叉参考列表信息”部分。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

汇编器指令

汇编器指令用于为程序提供数据并控制汇编过程。汇编器指令使您能够执行以下操作：

- 将代码和数据汇编到指定段
- 在存储器中为未初始化的变量预留空间
- 控制列表的外观
- 初始化存储器
- 汇编条件代码块
- 定义全局变量
- 指定汇编器可以从中获取宏的库
- 检查符号调试信息

本章分为两部分：第一部分（节 5.1 至节 5.12）根据功能描述了相关指令，第二部分（节 5.13）是按字母顺序给出的参考。

5.1 指令摘要 68
5.2 用于定义段的指令 72
5.3 用于更改指令类型的指令 75
5.4 用于初始化值的指令 75
5.5 执行对齐和保留空间的指令 77
5.6 用于设置输出列表格式的指令 79
5.7 用于引用其他文件的指令 80
5.8 用于启用条件汇编的指令 80
5.9 用于定义联合体或结构体类型的指令 81
5.10 用于定义枚举类型的指令 81
5.11 在汇编时用于定义符号的指令 81
5.12 其他命令 82
5.13 指令参考 83

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

5.1 指令摘要

表 5-1 至表 5-17 汇总了汇编器指令。除文中所述的汇编器指令之外，ARM 器件软件工具还支持以下指令：

- 宏指令在章节 6 中讨论；本章不予讨论。
- C 编译器使用指令进行符号调试。与其他指令不同，大多数汇编语言程序中不使用符号调试指令。附录 A 讨论了这些指令；本章不予讨论。

备注

标签和注释未显示在语法中：大多数包含指令的源语句还可以包含标签和注释。标签从第一列开始（只有标签和注释可以出现在第一列中），注释前面必须有分号，如果注释是该行中的唯一元素，则必须在前面加上星号。为提高可读性，此处未将标签和注释显示为指令语法的一部分。请参阅每个指令的详细说明以在指令中使用标签。

表 5-1. 供控制段使用的指令

助记符和语法	说明	请参阅
.bss *symbol, size in bytes*[,*alignment* [,*bank offset*]]	在 .bss（未初始化数据）段中保留 *size* 个字节	.bss 主题
.data	汇编到 .data（已初始化的数据）段	.data 主题
.sect " *section name* **"**	汇编到命名（已初始化）段	.sect 主题
.text	汇编到 .text（可执行代码）段	.text 主题
symbol **.usect "** *section name* **",** *size in bytes* [,*alignment*[,*bank offset*]]	在命名（未初始化）段中保留 *size* 个字节	.usect 主题

表 5-2. 用于将段收集到通用组中的指令

助记符和语法	说明	请参阅
.endgroup	结束组声明。	.endgroup 主题
.gmember *section name*	将 *段名* 指定为该组成员。	.gmember 主题
.group *group section name group type* **:**	开始组声明。	.group 主题

表 5-3. 会影响未使用段消除的指令

助记符和语法	说明	请参阅
.retain " *section name* **"**	指示链接器在链接的输出文件中包含当前段或指定段，无论该段是否被引用	.retain 主题
.retainrefs " *section name* **"**	指示链接器包含引用当前段或指定段的任何数据对象。	.retain 主题

表 5-4. 用于初始化值（数据和存储器）的指令

助记符和语法	说明	请参阅
.bits *value*$_1$[, ... , *value*$_n$]	在当前段中初始化一个或多个连续位	.bits 主题
.byte *value*$_1$[, ... , *value*$_n$]	在当前段中初始化一个或多个连续字节	.byte 主题
.char *value*$_1$[, ... , *value*$_n$]	在当前段中初始化一个或多个连续字节	.char 主题
.cstring {*expr*$_1$|" *string*$_1$ "}[,..., {*expr*$_n$|" *string*$_n$ "}]	初始化一个或多个文本字符串	.string 主题
.double *value*$_1$[, ... , *value*$_n$]	初始化一个或多个 64 位、IEEE 双精度、浮点常量	.double 主题
.field *value*[, *size*]	使用*值* 初始化 *size* 位 (1-32) 字段	.field 主题
.float *value*$_1$[, ... , *value*$_n$]	初始化一个或多个 32 位、IEEE 单精度、浮点常量	.float 主题
.half *value*$_1$[, ..., *value*$_n$]	初始化一个或多个 16 位整数（半字）	.half 主题
.int *value*$_1$[, ..., *value*$_n$]	初始化一个或多个 32 位整数	.int 主题
.long *value*$_1$[, ..., *value*$_n$]	初始化一个或多个 32 位整数	.long 主题
.short *value*$_1$[, ..., *value*$_n$]	初始化一个或多个 16 位整数（半字）	.short 主题
.string {*expr*$_1$|" *string*$_1$ "}[,..., {*expr*$_n$|" *string*$_n$ "}]	初始化一个或多个文本字符串	.string 主题
.ubyte *value*$_1$[, ..., *value*$_n$]	在当前段中初始化一个或多个连续无符号字节	.ubyte 主题

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential - NDA Restrictions

表 **5-4.** 用于初始化值（数据和存储器）的指令 **(continued)**

助记符和语法	说明	请参阅
.uchar *value*₁[, ... , *value*ₙ]	在当前段中初始化一个或多个连续无符号字节	.uchar 主题
.uhalf *value*₁[, ..., *value*ₙ]	初始化一个或多个无符号 16 位整数（半字）	.uhalf 主题
.uint *value*₁[, ..., *value*ₙ]	初始化一个或多个无符号 32 位整数	.uint 主题
.ulong *value*₁[, ..., *value*ₙ]	初始化一个或多个无符号 32 位整数	.long 主题
.ushort *value*₁[, ..., *value*ₙ]	初始化一个或多个无符号 16 位整数（半字）	.short 主题
.uword *value*₁[, ..., *value*ₙ]	初始化一个或多个无符号 32 位整数	.uword 主题
.word *value*₁[, ..., *value*ₙ]	初始化一个或多个 32 位整数	.word 主题

表 **5-5.** 用于执行对齐和保留空间的指令

助记符和语法	说明	请参阅
.align [*size in bytes*]	在 *size in bytes*（必须是 2 的幂）指定的边界上对齐 SPC；默认为字节边界	.align 主题
.bes *size*	在当前段中保留 *size* 个字节；一个标签指向保留空间的末尾	.bes 主题
.space *size*	在当前段中保留 *size* 个字节；一个标签指向保留空间的开头	.space 主题

表 **5-6.** 用于更改指令类型的指令

助记符和语法	说明	请参阅
.arm	开始汇编 ARM UAL 指令。等同于 .state32。	.arm 主题
.state16	开始汇编非 UAL 16 位指令	.state16 主题
.state32	开始汇编 32 位指令（默认）	.state32 主题
.thumb	开始汇编 Thumb 或 Thumb-2 UAL 指令	.thumb 主题

表 **5-7.** 用于设置输出列表格式的指令

助记符和语法	说明	请参阅
.drlist	允许列出所有指令行（默认）	.drlist 主题
.drnolist	禁止列出特定指令行	.drnolist 主题
.fclist	允许列出错误条件代码块（默认）	.fclist 主题
.fcnolist	禁止列出错误条件代码块	.fcnolist 主题
.length [*page length*]	设置源代码列表的页长度	.length 主题
.list	重新启动源代码列表	.list 主题
.mlist	允许列出宏和循环块（默认）	.mlist 主题
.mnolist	禁止列出宏和循环块	.mnolist 主题
.nolist	停止源代码列表	.nolist 主题
.option *option*₁[, *option*₂ , ...]	选择输出列表选项；可用的选项有 A、B、H、M、N、O、R、T、W 和 X	.option 主题
.page	在源列表中弹出一页	.page 主题
.sslist	允许列出展开的替代符号	.sslist 主题
.ssnolist	禁止列出展开的替代符号（默认）	.ssnolist 主题
.tab *size*	将制表符设为 *size* 个字符	.tab 主题
.title " *string* **"**	在列表页标题中列印一个标题	.title 主题
.width [*page width*]	设定源代码列表的页宽度	.width 主题

表 **5-8.** 用于引用其他文件的指令

助记符和语法	说明	请参阅
.copy ["]*filename*["]	包括来自另一个文件的源语句	.copy 主题
.include ["]*filename*["]	包括来自另一个文件的源语句	.include 主题

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

表 5-8. 用于引用其他文件的指令 (continued)

助记符和语法	说明	请参阅
.mlib ["]*filename*["]	指定从中检索宏定义的宏库	.mlib 主题

表 5-9. 会影响符号链接和可见性的指令

助记符和语法	说明	请参阅
.common *symbol, size in bytes* [, *alignment*] **.common** *symbol, structure tag* [, *alignment*]	为变量定义通用符号。	.common 主题
.def *symbol*$_1$[, ..., *symbol*$_n$]	标识在当前模块中定义并且可以在其他模块中使用的一个或多个符号。	.def 主题
.global *symbol*$_1$[, ..., *symbol*$_n$]	标识一个或多个全局（外部）符号。	.global 主题
.ref *symbol*$_1$[, ..., *symbol*$_n$]	标识用在当前模块中并在其他模块中定义的一个或多个符号。	.ref 主题
.symdepend *dst symbol name*[, *src symbol name*]	创建从段到符号的人工引用。	.symdepend 主题
.weak *symbol name*	标识用在当前模块中并在其他模块中定义的符号。	.weak 主题

表 5-10. 定义符号的指令

助记符和语法	说明	请参阅
.asg ["]*character string*["], *substitution symbol*	为*替代符号* 分配字符串。由 .asg 创建的替代符号可以重新定义。	.asg 主题
.define ["]*character string*["], *substitution symbol*	为*替代符号* 分配字符串。由 .define 创建的替代符号无法重新定义。	.asg 主题
symbol **.equ** *value*	使*值* 等同于*符号*	.equ 主题
.elfsym *name*, SYM_SIZE(*size*)	提供 ELF 符号信息	.elfsym 主题
.eval *expression* , *substitution symbol*	对数字*替代符号* 执行算术运算	.eval 主题
.label *symbol*	在段中定义加载时可重定位标签	.label 主题
.newblock	取消定义局部标签	.newblock 主题
symbol **.set** *value*	使*值* 等同于*符号*	.set 主题
.unasg *symbol*	取消将*符号* 分配为替代符号	.unasg 主题
.undefine *symbol*	取消将*符号* 分配为替代符号	.unasg 主题

表 5-11. 用于启用条件汇编的指令

助记符和语法	说明	请参阅
.if *condition*	如果*条件* 成立，则汇编代码块	.if 主题
.else	如果 .if *条件* 为不成立，则汇编代码块。使用 .if 构造时，.else 构造是可选的。	.else 主题
.elseif *condition*	如果 .if *条件* 不成立，且 .elseif 条件成立，则汇编代码块。使用 .if 构造时，.elseif 构造是可选的。	.elseif 主题
.endif	结束 .if 代码块	.endif 主题
.loop [*count*]	开始反复汇编代码块；循环计数由 *count* 确定。	.loop 主题
.break [*end condition*]	如果*结束条件* 成立，则结束 .loop 汇编。使用 .loop 构造时，.break 构造是可选的。	.break 主题
.endloop	结束 .loop 代码块	.endloop 主题

表 5-12. 用于定义联合体或结构体类型的指令

助记符和语法	说明	请参阅
.cstruct	行为类似于 .struct，但添加了填充和对齐，就像对 C 结构体所做的那样	.cstruct 主题
.cunion	行为类似于 .union，但添加了填充和对齐，就像对 C 联合体所做的那样	.cunion 主题
.emember	在汇编代码中设置类似 C 语言的枚举类型	节 5.10
.endenum	在汇编代码中设置类似 C 语言的枚举类型	节 5.10

DRAFT ONLY
Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

表 **5-12.** 用于定义联合体或结构体类型的指令 **(continued)**

助记符和语法	说明	请参阅
.endstruct	结束结构体定义	.cstruct 主题， .struct 主题
.endunion	结束联合体定义	.cunion 主题，.union 主题
.enum	在汇编代码中设置类似 C 语言的枚举类型	节 5.10
.union	开始联合体定义	.union 主题
.struct	开始结构体定义	.struct 主题
.tag	为标签分配结构体属性	.cstruct 主题， .struct 主题 .union 主题

表 **5-13.** 用于创建或影响宏的指令

助记符和语法	说明	请参阅
macname **.macro** [*parameter*$_1$][,..., *parameter*$_n$]	开始定义名为 *macname* 的宏	.macro 主题
.endm	终止宏定义	.endm 主题
.mexit	转至 .endm	节 6.2
.mlib *filename*	识别包含宏定义的库	.mlib 主题
.var	将局部替代符号添加到宏的参数列表	.var 主题

表 **5-14.** 用于控制诊断的指令

助记符和语法	说明	请参阅
.emsg *string*	将用户定义的错误信息发送至输出器件；不产生 .obj 文件	.emsg 主题
.mmsg *string*	将用户定义的消息发送至输出器件	.mmsg 主题
.wmsg *string*	将用户定义的警告消息发送至输出器件	.wmsg 主题

表 **5-15.** 用于执行汇编源代码调试的指令

助记符和语法	说明	请参阅
.asmfunc	标识包含函数的代码块的开头	.asmfunc 主题
.endasmfunc	标识包含函数的代码块的结尾	.endasmfunc 主题

表 **5-16.** 供绝对列表器使用的指令

助记符和语法	说明	请参阅
.setsect	由绝对列表器生成；设置段	章节 9
.setsym	由绝对列表器生成；设置符号	章节 9

表 **5-17.** 用于执行其他函数的指令

助记符和语法	说明	请参阅
.cdecls [*options* ,]" *filename* "[, " *filename2* "[, ...]	在 C 和汇编代码之间共享 C 头文件	.cdecls 主题
.end	终止程序	.end 主题

DRAFT ONLY TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

除了可以在代码中使用的汇编指令之外，C/C++ 编译器在创建汇编代码时还会生成若干指令。这些指令只能由编译器使用；请勿尝试使用这些指令：

- 节 A.1 中列出的 DWARF 指令
- **.battr** 指令用于编码目标文件的编译属性。
- **.bound** 指令仅在内部使用。
- **.comdat** 指令仅在内部使用。
- **.compiler_opts** 指令指示汇编代码是由编译器生成的，以及此文件使用的编译模型选项。

5.2 用于定义段的指令

这些指令可将汇编语言程序的各个部分与相应的段相关联：

- **.bss** 指令用于在 .bss 段中为未初始化的变量保留空间。
- **.data** 指令用于标识 .data 段中的代码部分。.data 段通常包含已初始化的数据。
- **.retain** 指令可用于指示当前段或指定段必须包含在已链接的输出中。因此，即使链接中没有其他段引用当前段或指定段，该段仍会包含在链接中。
- **.retainrefs** 指令可用于强制相关段引用指定段。这在中断矢量中会很有用。
- **.sect** 指令用于定义已初始化的命名段，并将后续代码或数据与该段相关联。用 .sect 定义的段可以包含代码或数据。
- **.text** 指令用于标识 .text 段中的代码部分。.text 段通常包含可执行代码。
- **.usect** 指令用于保留未初始化命名段中的空间。.usect 指令类似于 .bss 指令，但它允许用户将空间与 .bss 段分开保留。

章节 2 详细讨论了这些段。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

下面的示例显示了如何使用段指令将代码和数据与正确的段相关联。这是一个输出列表；第 1 列显示行号，第 2 列显示 SPC 值。（每个段都自带程序计数器，即 SPC。）当代码第一次被放入某个段中时，它的 SPC 值等于 0。当用户在汇编其他代码后继续汇编到某个段时，该段的 SPC 会恢复计数，就好像之间没有代码一样。

该示例中的指令执行以下任务：

.text	用值 1、2、3、4、5、6、7 和 8 初始化字。
.data	用值 9、10、11、12、13、14、15 和 16 初始化字。
var_defs	用值 17 和 18 初始化字。
.bss	保留 19 个字节。
xy	保留 20 个字节。

.bss 和 .usect 指令不会结束当前段或开始新段；它们会保留指定的空间量，然后汇编器继续将代码或数据汇编到当前段中。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

```
 1                      **************************************************
 2                      *     Start assembling into the .text section    *
 3                      **************************************************
 4 00000000                      .text
 5 00000000 00000001             .word 1,2
   00000004 00000002
 6 00000008 00000003             .word 3,4
   0000000c 00000004
 7
 8                      **************************************************
 9                      *     Start assembling into the .data section    *
10                      **************************************************
11 00000000 .data
12 00000000 00000009             .word 9, 10
   00000004 0000000A
13 00000008 0000000B             .word 11, 12
   0000000c 0000000C
14
15                      *************************************************
16                      *         Start assembling into a named,        *
17                      *         initialized section, var_defs         *
18                      *************************************************
19 00000000                      .sect "var_defs"
20 00000000 00000011             .word 17, 18
   00000004 00000012
21
22                      *************************************************
23                      *    Resume assembling into the .data section   *
24                      *************************************************
25 00000010 .data
26 00000010 0000000D             .word 13, 14
   00000014 0000000E
27 00000000                      .bss sym, 19    ; Reserve space in .bss
28 00000018 0000000F             .word 15, 16    ; Still in .data
   0000001c 00000010
29
30                      *************************************************
31                      *    Resume assembling into the .text section   *
32                      *************************************************
33 00000010                      .text
34 00000010 00000005             .word 5, 6
   00000014 00000006
35 00000000             usym     .usect "xy", 20 ; Reserve space in xy
36 00000018 00000007             .word 7, 8      ; Still in .text
   0000001c 00000008
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

5.3 用于更改指令类型的指令

默认情况下，汇编器开始将文件中的所有指令汇编为 32 位指令。可使用 --code_state=16 汇编器（请参阅节 4.3）选项更改默认行为；该选项会使汇编器开始将文件中的所有指令汇编为 16 位指令。还可以使用四个指令来更改汇编器从指令出现的位置开始汇编指令的方式：

- **.arm** 指令告诉汇编器从指令位置开始汇编 ARM UAL 语法 32 位指令。.arm 指令用于在将任何指令写入相应的段之前执行隐式字对齐，以确保所有 32 位指令都是字对齐的。.arm 指令还会重置已定义的所有局部标签。.arm 指令等效于 .state32 指令。
- **.state16** 指令使汇编器从指令位置开始汇编非 UAL 16 位指令。.state16 指令用于在将任何指令写入相应的段之前执行隐式半字对齐，以确保所有 16 位指令都是半字对齐的。.state16 指令还会重置已定义的所有局部标签。
- **.state32** 指令告诉汇编器从指令位置开始汇编 32 位指令。.state32 指令用于在将任何指令写入相应的段之前执行隐式字对齐，以确保所有 32 位指令都是字对齐的。.state32 指令还会重置已定义的所有局部标签。
- **.thumb** 指令告诉汇编器从指令位置开始汇编 Thumb 或 Thumb-2 UAL 语法指令。.thumb 指令用于在将任何指令写入相应的段之前执行隐式字对齐，以确保所有指令都是字对齐的。.thumb 指令还会重置已定义的所有局部标签。

5.4 用于初始化值的指令

若干指令用于汇编当前段的值。例如：

- **.byte** 和 **.char** 指令用于将一个或多个 8 位值置于当前段的连续字节中。这些指令类似于 .word、.int 和 .long，不同之处在于每个值的宽度限制为 8 位。
- **.double** 指令用于计算一个或多个浮点值的双精度（64 位）IEEE 浮点表示，并将其存储在当前段的两个连续字中。.double 指令用于自动与双字边界对齐。
- **.field** 和 **.bits** 指令用于将单个值置于当前字的指定位数中。借助 .field，用户可以将多个字段打包成一个字；汇编器在填充完一个字之前不会递增 SPC。如果某个字段无法保留在当前字中剩余的空间内，.field 将插入 0 来填充当前字，然后将该字段置于下一个字中。.bits 指令具有类似情况，但不强制对齐到字段边界。请参阅 .field 主题和 .bits 主题。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

图 5-1 显示了如何将字段打包成一个字。使用以下汇编代码时，请注意：前三个字段的 SPC 没有更改（这些字段被打包到同一个字中）：

```
     1 00000000 60000000          .field 3, 3
     2 00000000 64000000          .field 8, 6
     3 00000000 64400000          .field 16, 5
     4 00000004 01234000          .field 01234h, 20
     5 00000008 00001234          .field 01234h, 32
```

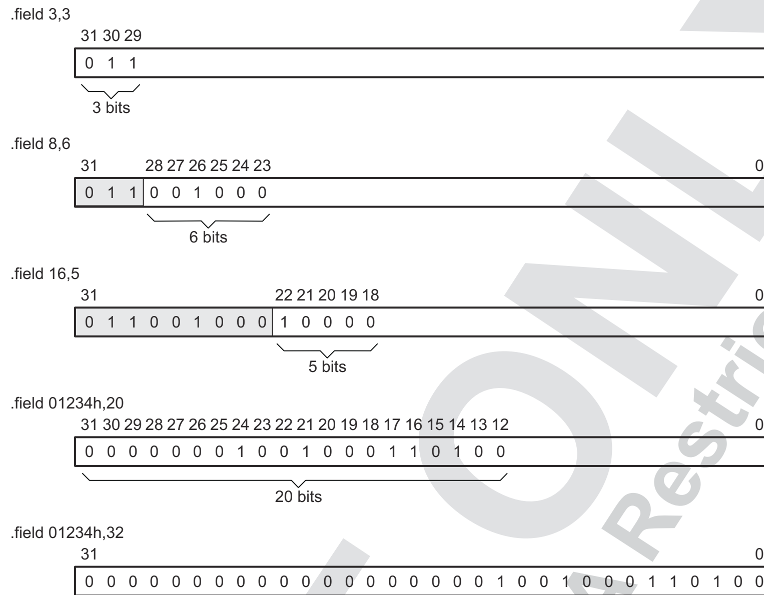


图 5-1. .field 指令

- **.float** 指令用于计算一个浮点值的单精度（32 位）IEEE 浮点表示，并将其存储在当前段中与字边界对齐的字中。
- **.half** 和 **.short** 指令用于将一个或多个 16 位值置于当前段的连续 16 位字段（半字）中。.half 和 .short 指令用于自动与短（2 字节）边界对齐。
- **.int**、**.long** 和 **.word** 指令用于将一个或多个 32 位值置于当前段的连续 32 位字段（字）中。.int、.long 和 .word 指令用于自动与字边界对齐。
- **.string** 和 **.cstring** 指令用于将一个或多个字符串中的 8 位字符置于当前段中。.string 和 .cstring 指令类似于 .byte，将 8 位字符置于当前段的每个连续字节中。.cstring 指令用于添加 C 所需的 NUL 字符；.string 指令不会添加 NUL 字符。
- **.ubyte**、**.uchar**、**.uhalf**、 **.uint**、**.ulong**、**.ushort** 和 **.uword** 指令作为其各自有符号指令的无符号版本提供。这些指令主要由 C/C++ 编译器用于支持 C/C++ 中的无符号类型。

备注

用于初始化常量的指令（用在 .struct/.endstruct 序列中）：

当 .bits、.byte、.char、.int、.long、.word、.double、.half、.short、.ubyte、.uchar、.uhalf、.uint、.ulong、.ushort、.uword、.string、.float 和 .field 指令是 .struct/.endstruct 序列的一部分时，它们不会初始化存储器，而是会定义某个成员的大小。如需更多信息，请参阅 .struct/.endstruct 指令。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

图 5-2 使用以下已汇编代码比较了 .byte、.char、.short、.int、.long、.float、.double、.word 和 .string 指令：

```
1 00000000 AA                .byte      0AAh, 0BBh
  00000001 BB
2 00000002 CC                .char      0CCh
3 00000004 ABCD              .short     0ABCDh
4 00000006 0000DDDD          .word      0DDDDh
5 0000000a EEEEFFFF          .long      0EEEEFFFFh
6 0000000e 0000DDDD          .int       0DDDDh
7 00000012 3FFFFCB9          .float     1.9999
8 00000016 3FFFFFF5          .double    1.99999
  0000001a 83A53B8E
9 0000001e 48                .string    "Help"
  0000001f 65
  00000020 6C
  00000021 70
```

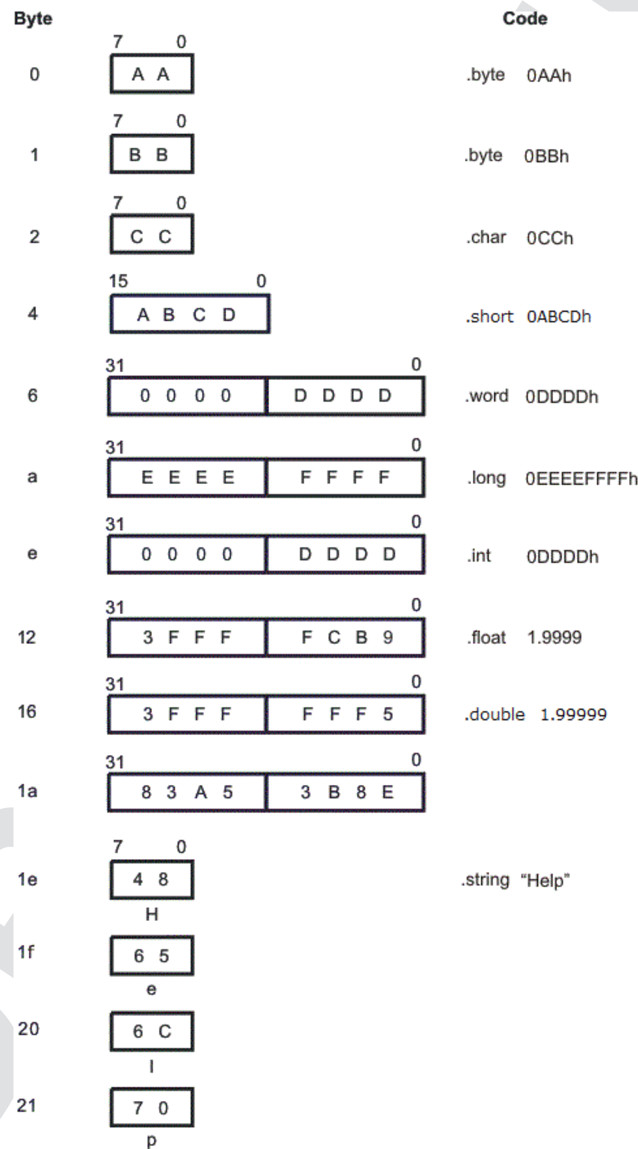


图 5-2. 初始化指令

5.5 执行对齐和保留空间的指令

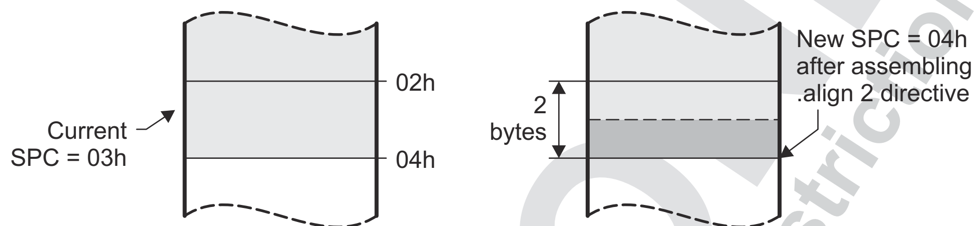
这些指令用于对齐段程序计数器（SPC）或在段中保留空间：

Copyright © 2023 Texas Instruments Incorporated

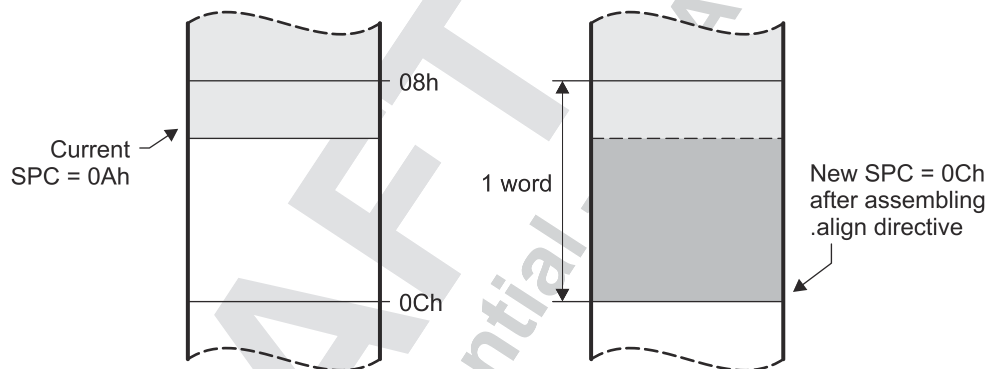
- **.align** 指令用于使 SPC 在 1 字节到 32K 字节边界对齐。这可确保指令后面的代码从用户指定的字节值开始。如果 SPC 已在所选边界对齐，则不会递增。.align 指令的操作数必须等于 2 的幂，介于 2^0 和 2^{15} 之间（包含端点值）。

  图 5-3 演示了 .align 指令。使用以下汇编代码：

```
      1 00000000 40000000          .field 2,3
      2 00000000 4000000B          .field 11, 21
      3                            .align 2
      4 00000004 45                .string "Errcnt"
        00000005 72
        00000006 72
        00000007 63
        00000008 6E
        00000009 74
      5                            .align
      6 0000000c 04                .byte 4
```



(a) Result of .align 2



(b) Result of .align without an argument

图 5-3. .align 指令

Copyright © 2023 Texas Instruments Incorporated

- **.bes** 和 **.space** 指令用于在当前段中保留指定字节数。汇编器用 0 填充这些保留字节。用户可以通过将字节数乘以 4 来保留指定数量的字。
  - 如果用户使用带 .space 的标签，它会指向包含保留位的*第一个*字节。
  - 如果用户使用带 .bes 的标签，它会指向包含保留位的*最后*一个字节。
- 图 5-4 显示了 .space 和 .bes 指令如何用于以下汇编代码：

```
     1
     2 00000000 00000100          .word 100h, 200h
       00000004 00000200
     3 00000008          Res_1:   .space 17
     4 0000001c 0000000F          .word 15
     5 00000033          Res_2:   .bes 20
     6 00000034 BA                .byte 0BAh
```

Res_1 指向 .space 保留的空间中的第一个字节。Res_2 指向 .bes 保留的空间中的最后一个字节。

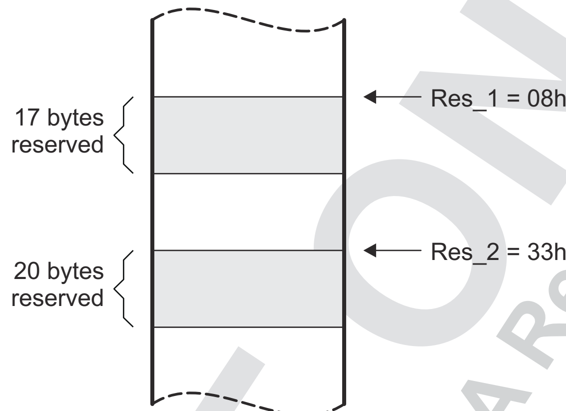


图 **5-4. .space** 和 **.bes** 指令

5.6 用于设置输出列表格式的指令

这些指令用于设置列表文件的格式：

- **.drlist** 指令用于将指令行列印在列表中；**.drnolist** 指令用于因某些指令而将其关闭。用户可以使用 .drnolist 指令来禁止列印以下指令。用户可以使用 .drlist 指令重新开启列表。

.asg	.eval	.length	.mnolist	.var
.break	.fclist	.mlist	.sslist	.width
.emsg	.fcnolist	.mmsg	.ssnolist	.wmsg

- 源代码列表包含不生成代码的错误条件块。**.fclist** 和 **.fcnolist** 指令用于开启和关闭此列表。用户可以使用 .fclist 指令准确列出源代码中显示的错误条件块。用户可以使用 .fcnolist 指令仅列出实际汇编的条件块。
- **.length** 指令用于控制列表文件的页长度。用户可以使用此指令调整各种输出器件的列表。
- **.list** 和 **.nolist** 指令用于开启和关闭输出列表。用户可以使用 .nolist 指令来防止汇编器列印列表文件中选定的源语句。可使用 .list 指令重新开启列表。
- 源代码列表包含宏扩展和循环块。**.mlist** 和 **.mnolist** 指令用于开启和关闭此列表。用户可以使用 .mlist 指令将所有宏扩展和循环块列印在列表中，并使用 .mnolist 指令关闭此列表。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

- **.option** 指令可控制列表文件中的某些功能。该指令具有以下操作数：

A	开启所有指令和数据的列表，以及随后的扩展、宏和块。
B	将 .byte 和 .char 指令的列表限制为一行。
H	将 .half 和 .short 指令的列表限制为一行。
M	关闭列表中的宏扩展。
N	关闭列表（执行 .nolist）。
O	开启列表（执行 .list）。
R	复位 B、H、M、T 和 W 指令（关闭 B、H、M、T 和 W 的限制）。
T	将 .string 指令的列表限制为一行。
W	将 .word 和 .int 指令的列表限制为一行。
X	生成符号的交叉参考列表。您还可以通过使用 --asm_cross_reference_listing 选项调用汇编器来获取交叉参考列表（请参阅节 4.14）。

- **.page** 指令用于在输出列表中弹出页。
- 源代码列表包含替代符号扩展。**.sslist** 和 **.ssnolist** 指令用于开启和关闭此列表。用户可以使用 .sslist 将所有替代符号扩展列印在列表中，并使用 .ssnolistto 关闭此列表。这些指令对于调试替代符号扩展非常有用。
- **.tab** 指令用于定义制表符大小。
- **.title** 指令用于提供汇编器在每页顶部列印的标题。
- **.width** 指令用于控制列表文件的页面宽度，用户可能需要针对各种输出器件对其进行调整。

5.7 用于引用其他文件的指令

这些指令为可在当前文件的程序集中使用的其他文件提供信息或关于这些文件的信息：

- **.copy** 和 **.include** 指令用于告知汇编器开始从另一个文件读取源语句。当汇编器读取完复制/头文件中的源语句时，它会继续从当前文件读取源语句。从复制文件中读取的语句打印在列表文件中；从头文件中读取的语句*不会*打印在列表文件中。
- **.def** 指令用于识别在当前模块中定义但可用在另一模块中的符号。汇编器将此符号包含在符号表中。
- **.global** 指令用于在外部声明一个符号，以便在链接时其他模块可以使用它。（有关全局符号的更多信息，请参阅节 2.6.1。）.global 指令有双重作用，作为已定义符号的 .def 和未定义符号的 .ref。只有在程序中使用符号时，链接器才会解析未定义的全局符号引用。.global 指令用于声明一个 16 位符号。
- **.mlib** 指令用于为汇编器提供包含宏定义的存档库的名称。当汇编器遇到当前模块中未定义的宏时，它会在用 .mlib 指定的宏库中搜索它。
- **.ref** 指令用于识别在当前模块中使用但在另一模块中定义的符号。汇编器将该符号标记为未定义的外部符号并将其输入至目标符号表，以便链接器可以解析其定义。.ref 指令用于强制链接器解析符号引用。
- **.symdepend** 指令用于创建一个从定义源符号名称的段到目标符号的人工引用。如果源符号段包含在输出模块中，.symdepend 指令可防止链接器删除包含目标符号的段。
- **.weak** 指令用于识别在当前模块中使用但在另一模块中定义的符号。它等同于 .ref 指令，只是引用具有弱链接性。

5.8 用于启用条件汇编的指令

条件汇编指令使您能够指示汇编器根据表达式的评估结果（true 或 false）来汇编某些代码段。两组指令允许您汇编条件代码块：

- **.if/.elseif/.else/.endif** 指令会告知汇编器根据表达式的计算结果有条件地汇编代码块。

.if *condition*	如果 .if *condition* 为 true，则标记条件代码块的开头并汇编代码。
[**.elseif** *condition*]	如果 .if *condition* 为 false，且 .elseif 条件为 true，则标记要汇编的代码块。
.else	如果 .if *condition* 为 false，且任何 .elseif 条件为 false，则标记要汇编的代码块。
.endif	标记条件代码块的结束并终止代码块。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

- **.loop/.break/.endloop** 指令会告知汇编器根据表达式的计算结果反复汇编代码块。

.loop [*count*]	标记可重复代码块的开始。可选表达式会根据循环计数求值。
.break [*end condition*]	告知汇编器在 .break *end condition* 为 false 时重复汇编，并在表达式为 true 或省略时转到紧跟 .endloop 的代码。
.endloop	标记可重复代码块的结束。

汇编器支持多种可用于条件表达式的关系运算符。有关关系运算符的更多信息，请参阅节 4.9.2。

5.9 用于定义联合体或结构体类型的指令

这些指令设置了专门的类型供之后用于 .tag 指令，允许您使用符号名称来引用复合对象的各部分。创建的类型类似于 C 语言的结构体和联合体类型。

.struct、.union、.cstruct 和 .cunion 指令将相关数据分组到更容易访问的聚合结构中。这些指令不会为任何对象分配空间。对象必须单独分配，并且必须使用 .tag 指令为对象分配类型。

```
type    .struct             ; structure tag definition
X       .int
Y       .int
T_LEN   .endstruct
COORD   .tag type           ; declare COORD (coordinate)
COORD   .space T_LEN        ; actual memory allocation
        LDR R0, COORD.Y     ; load member Y of structure
                            ; COORD into register R0.
```

.cstruct 和 .cunion 指令可保证数据结构将具有相同的对齐和填充，就好像该结构是在类似的 C 代码中定义的一样。这允许在 C 和汇编代码之间共享结构。请参阅章节 13。对于 .struct 和 .union，元素偏移量的计算由汇编器决定，因此布局可能与 .cstruct 和 .cunion 不同。

5.10 用于定义枚举类型的指令

这些指令设置了供以后在表达式中使用的专门类型，使用户能够使用符号名称来引用编译时常量。创建的类型类似于 C 语言的枚举类型。这实现了在 C 语言代码和汇编代码之间共享枚举类型。请参阅章节 13。

有关使用 .enum 的示例，请参阅节 13.2.10。

5.11 在汇编时用于定义符号的指令

汇编时符号指令将有意义的符号名称等同于常量值或字符串。

- **.asg** 指令将字符串分配给替代符号。值存储在替代符号表中。汇编器在遇到替代符号时，会将该名称替换为其字符串值。由 .asg 创建的替代符号可以重新定义。

```
      .asg "10, 20, 30, 40", coefficients
              ; Assign string to substitution symbol.
      .byte coefficients
              ; Place the symbol values 10, 20, 30, and 40
              ; into consecutive bytes in current section.
```

- **.define** 指令将字符串分配给替代符号。值存储在替代符号表中。汇编器在遇到替代符号时，会将该名称替换为其字符串值。由 .define 创建的替代符号无法重新定义。
- **.eval** 指令用于计算定义明确的表达式，将结果转换为字符串，并将字符串分配给替代符号。该指令对于操作计数器非常有用：

```
      .asg      1 , x    ; x = 1
      .loop              ; Begin conditional loop.
      .byte     x*10h    ; Store value into current section.
      .break    x = 4    ; Break loop if x = 4.
      .eval     x+1, x   ; Increment x by 1.
      .endloop           ; End conditional loop.
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

- **.label** 指令用于定义一个特殊符号，以引用当前段中的加载时地址。当某个段在一个地址加载但在不同的地址运行时，这会很有用。例如，用户可能需要将性能关键代码块加载到速度较慢的片外存储器中以节省空间，并将代码移动到高速片上存储器中运行。有关使用加载时地址标签的示例，请参阅 .label 主题。
- **.set** 和 **.equ** 指令用于将常量值设置为符号。该符号存储在符号表中，不能重新定义；例如：
- **.unasg** 指令用于关闭通过 .asg 进行的替代符号分配。
- **.undefine** 指令用于关闭通过 .define 进行的替代符号分配。
- **.var** 指令使用户能够使用替代符号作为宏中的局部变量。

5.12 其他命令

这些指令用于实现其他函数或功能：

- **.asmfunc** 和 **.endasmfunc** 指令用于标记函数边界。这些指令与编译器 --symdebug:dwarf (-g) 选项一同使用，可生成汇编函数的调试信息。
- **.cdecls** 指令允许使用混合汇编和 C/C++ 环境的编程器共享 C 头文件，此头文件包含 C 和汇编代码间的声明和原型。
- **.end** 指令用于终止汇编。如果使用 .end 指令，则它应该是程序的最后一条源语句。该指令与文件结尾字符具有相同的效果。
- **.group**、**.gmember** 和 **.endgroup** 指令用于定义由多个段共享的 ELF 组段。
- **.newblock** 指令用于重置局部标签。局部标签是表单 $n 中的符号，其中 n 是一个十进制数字。当局部标签显示在标签字段中时，才对其进行定义。局部标签是可用作跳转指令操作数的临时标签。.newblock 指令用于通过在使用局部标签后对它们进行重置来限制局部标签的范围。有关局部标签的信息，请参阅节 4.8.3。

用户可以通过以下三个指令来定义自有的错误和警告消息：

- **.emsg** 指令用于向标准输出器件发送错误消息。.emsg 指令生成错误的方式与汇编器相同：递增错误计数，阻止汇编器生成目标文件。
- **.mmsg** 指令用于向标准输出器件发送汇编时消息。.mmsg 指令的运行方式与 .emsg 和 .wmsg 指令相同，但不设置错误计数或警告计数。它不影响目标文件的创建。
- **.wmsg** 指令用于向标准输出器件发送警告消息。.wmsg 指令的运行方式与 .emsg 指令相同，但会使警告计数而非错误计数递增。它不影响目标文件的创建。

有关在宏中使用错误和警告指令的更多信息，请参阅节 6.7。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

5.13 指令参考

本章的其余部分是参考。通常，指令按字母顺序排列，每个主题一个指令。但是，相关指令（例如 .if/.else/.endif）在一个主题中一起呈现。

.align

在下一边界上对齐 SPC

语法

.align [*size in bytes*]

说明

.align 指令会在下一边界上对齐段程序计数器 (SPC)，具体取决于 *size in bytes* 参数。*size* 可以是 2 的任意次幂，但只有某些值可用于对齐。操作数 1 会在下一字节边界上对齐 SPC，如果未提供 *以字节表示的大小*，则此为默认值。*以字节表示的大小* 必须等于 2 的幂；该值必须介于 1 和 32,768 之间（包含端点值）。汇编器会将包含空值 (0) 的字汇编到下一以字节表示的大小边界：

1	将 SPC 与字节边界对齐
2	将 SPC 与半字边界对齐
4	将 SPC 与字边界对齐
8	将 SPC 与双字边界对齐
128	将 SPC 与页边界对齐

使用 .align 指令具有两种效果：

- 汇编器会在当前段*内* 的 x-byte 边界对齐 SPC。
- 汇编器会设置一个标志，强制链接器对齐该段，以便在将段加载到存储器中时，各个对齐方式保持不变。

示例

此示例显示了多种对齐方式，包括 .align 2、.align 8 和默认的 .align。

```
 1 00000000 04                 .byte 4
 2                             .align 2
 3 00000002 45                 .string "Errorcnt"
   00000003 72
   00000004 72
   00000005 6F
   00000006 72
   00000007 63
   00000008 6E
   00000009 74
 4                             .align
 5 0000000c 60000000           .field 3,3
 6 0000000c 6A000000           .field 5,4
 7                             .align 2
 8 0000000c 6A006000           .field 3,3
 9                             .align 8
10 00000010 50000000           .field 5,4
11                             .align
12 00000014 04                 .byte 4
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.asg/.define/.eval

分配替代符号

语法

.asg " *character string* **",** *substitution symbol*

.define " *character string* **",** *substitution symbol*

.eval *expression* **,** *substitution symbol*

说明

.asg 和 **.define** 指令用于将字符串分配给替代符号。替代符号存储在替代符号表中。.asg 指令的使用方法与 .set 指令有诸多相同之处，区别在于 .set 会为符号分配一个常量值（不能重新定义），而 .asg 会为替代符号分配一个字符串（可重新定义）。

- 汇编器会为替代符号分配 *字符串*。
- *替代符号* 必须是有效的符号名称。替代符号最多具有 128 个字符，并且必须以字母开头。该符号的其余字符可以是字母数字字符、下划线（_）和美元符号（$）的组合。

.define 指令的运行方式与 .asg 指令相同，但 .define 不允许创建与寄存器符号或助记符同名的替代符号。它不会在汇编器中创建新的符号命名空间，而是使用现有的替代符号命名空间。.define 指令用于在转换 C/C++ 头文件时防止汇编环境遭损坏。有关在汇编源代码中使用 C/C++ 头文件的更多信息，请参见章节 13。

.eval 指令对存储在替代符号表中的替代符号执行算术运算。该指令会计算 *表达式* 并将结果的字符串值分配给替代符号。.eval 指令作为 .loop/.endloop 块中的计数器特别有用。

- 该*表达式* 是一个定义明确的字母数字表达式，其中所有符号之前都已在当前源代码模块中定义，因此结果是一个绝对表达式。
- *替代符号* 必须是有效的符号名称。替代符号最多具有 128 个字符，并且必须以字母开头。该符号的其余字符可以是字母数字字符、下划线（_）和美元符号（$）的组合。

有关如何关闭替代符号的信息，请参阅 .unasg/.undefine 主题。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.asg/.define/.eval (continued)

分配替代符号

示例

此示例演示了如何使用 .asg 和 .eval。

```
      1                                .sslist ; show expanded sub. symbols
      2                                        ; using .asg and .eval
      3
      4                                .asg R13, STACKPTR
      5                                .asg &, AND
      6
      7 00000000 E28DD018              ADD STACKPTR, STACKPTR, #280 AND 255
#                                      ADD R13, R13, #280 & 255
      8 00000004 E28DD018              ADD STACKPTR, STACKPTR, #280 & 255
#                                      ADD R13, R13, #280 & 255
      9
     10                                .asg 0, x
     11                                .loop 5
     12                                .eval x+1, x
     13                                .word x
     14                                .endloop
1                                      .eval x+1, x
#                                      .eval 0+1, x
1         00000008 00000001            .word x
#                                      .word 1
1                                      .eval x+1, x
#                                      .eval 1+1, x
1         0000000c 00000002            .word x
#                                      .word 2
1                                      .eval x+1, x
#                                      .eval 2+1, x
1         00000010 00000003            .word x
#                                      .word 3
1                                      .eval x+1, x
#                                      .eval 3+1, x
1         00000014 00000004            .word x
#                                      .word 4
1                                      .eval x+1, x
#                                      .eval 4+1, x
1         00000018 00000005            .word x
#                                      .word 5
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.asmfunc/.endasmfunc

标记函数边界

语法

symbol **.asmfunc [stack_usage(** *num* **)]**

.endasmfunc

说明

.asmfunc 和 **.endasmfunc** 指令用于标记函数边界。这些指令与编译器 -g 选项 (--symdebug:dwarf) 一同使用，以允许以与 C/C++ 函数相同的方式调试汇编代码段。

您不应使用由编译器生成的相同指令（参见附录 A）来完成汇编调试；这些指令只能由编译器用于为 C/C++ 源文件生成符号调试信息。

符号 是必须显示在标签字段中的标签。

.asmfunc 指令有一个可选参数 stack_usage，它表示该函数最多可以使用 *num* 个字节。

未包含在 .asmfunc 和 .endasmfunc 指令对中的连续范围汇编代码被赋予以下格式的默认名称：

$ *filename* **:** *beginning source line* **:** *ending source line* **$**

示例

在此示例中，汇编源代码会为 user_func 段生成调试信息。

```
 1 00000000                      .sect   ".text"
 2                               .global user_func
 3                               .global printf
 4
 5                               .align   4
 6
 7 00000000                      .state32
 8
 9                    user_func: .asmfunc
10 00000000 E92D4008             STMFD     SP!, {A4, LR}
11 00000004 E28F000C             ADR       A1, SL1
12 00000008 EBFFFFFC!            BL        printf
13 0000000c E3A00000             MOV       A1, #0
14 00000010 E8BD4008             LDMFD     SP!, {A4, LR}
15 00000014 E12FFF1E             BX        LR
16                               .endasmfunc
17
18                               .align  4
19 00000018 48        SL1:       .string "Hello World!",10,0
   00000019 65
   0000001a 6C
   0000001b 6C
   0000001c 6F
   0000001d 20
   0000001e 57
   0000001f 6F
   00000020 72
   00000021 6C
   00000022 64
   00000023 21
   00000024 0A
   00000025 00
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.bits

初始化位

语法

.bits *value*[, *size in bits*]

说明

.bits 指令用于将值放入当前段的连续位中。

.bits 指令类似于 .field 指令（请参阅 .field 主题）。但是，.bits 指令不会强制将值与字段边界对齐。如果 .bits 指令后跟不同的空间创建指令，则 SPC 将与后跟指令的适当值对齐。

此指令有两个操作数：

- *值* 是必需参数；它是一个表达式，计算得出后放置在当前位置的当前段中。值必须是绝对值。
- *以位表示的大小* 是一个可选参数；它用于指定从 1 到 32 的数字，表示值的位数。默认大小为 32 位。如果指定的值无法容纳在 *以位表示的大小* 中，汇编器会截断该值并发出警告消息。例如，.bits 3,1 会使汇编器将值 3 截断为 1；汇编器还会列印以下消息：

```
*** WARNING! line 21: W0001: Field value truncated to 1
            .bits  3, 1
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.bss

在 .bss 段中保留空间

语法

.bss *symbol* **,** *size in bytes*[**,** *alignment*]

说明

.bss 指令在 .bss 段中为变量保留空间。该指令通常用于在 RAM 中分配空间。

该指令类似于 .usect 指令（参阅 .usect 主题）；两者都只是为数据保留空间，而该空间没有内容。但是，.usect 定义了可以放置在存储器中任何位置的其他段，这些段独立于 .bss 段。

- *symbol* 是一个必需参数。它定义了一个符号，用于指向指令保留的第一个位置。符号名称必须与要保留空间的变量相对应。
- *size in bytes* 是一个必需参数，而且必须是一个绝对常量表达式。汇编器在 .bss 段中分配 size 个字节。没有默认大小。
- *alignment* 是一个可选参数，可确保分配给符号的空间出现在指定的边界上。边界指示必须设置为 2 的幂，介于 2^0 和 2^{15} 之间（包含端点值）。如果 SPC 已在指定的边界对齐，则不会递增。

有关段的更多信息，请参阅章节 2。

示例

在本例中，.bss 指令为两个变量 TEMP 和 ARRAY 分配空间。符号 TEMP 指向四个字节的未初始化空间（在 .bss SPC = 0 处）。符号 ARRAY 指向 100 个字节的未初始化空间（在 .bss SPC = 04h 处）。使用 .bss 指令声明的符号可按与其他符号相同的方式引用，也可以在外部声明。

```
 1                           ***********************************************
 2                           **   开始汇编到 .text 段。**
 3                           ***********************************************
 4 00000000                           .text
 5 00000000 E3A00000                  MOV     R0, #0
 6
 7                           ***********************************************
 8                           **      在 .bss 段中为 TEMP 分配 4 个字节。    **
 9                           ***********************************************
10 00000000                  Var_1:  .bss    TEMP, 4
11
12                           ***********************************************
13                           **               仍处于 .text 段中。              **
14                           ***********************************************
15 00000004 E2801056                 ADD     R1, R0, #56h
16 00000008 E0020091                 MUL     R2, R1, R0
17
18                           ***********************************************
19                           ** 在 .bss 段中为名为 ARRAY 的符号分配 100 个字节。 **
20                           **                                        **
21                           ***********************************************
22 00000004                          .bss    ARRAY, 100, 4
23
24                           ***********************************************
25                           **       将更多代码汇编到 .text 段。      **
26                           ***********************************************
27 0000000c E1A0F00E                 MOV     PC, LR
28
29                           ***********************************************
30                           **      声明外部 .bss 符号。       **
31                           ***********************************************
32                                   .global ARRAY, TEMP
33                                   .end
```

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.byte/.ubyte/.char/.uchar

初始化字节

语法

.byte *value*$_1$[, ..., *value*$_n$]

.ubyte *value*$_1$[, ..., *value*$_n$]

.char *value*$_1$[, ..., *value*$_n$]

.uchar *value*$_1$[, ..., *value*$_n$]

说明

.byte、**.ubyte**、**.char** 和 **.uchar** 指令将一个或多个值放入当前段的连续字节中。*value* 可以是以下任一项：

- 汇编器计算并视为 8 位有符号数的表达式
- 用双引号引起来的字符串。字符串中的每个字符代表一个单独的值，并且值存储在连续字节中。整个字符串*必须*用双引号引起来。

第一个字节占用完整 32 位字的 8 个最低有效位。第二个字节占用第 8 到 15 位，而第三个字节占用第 16 到 23 位。汇编器会截断大于八位的值。

如果使用标签，则它指向已初始化的第一个字节的位置。

当您在 .struct/.endstruct 序列中使用这些指令时，它们会定义成员的大小，但不会初始化存储器。如需更多信息，请参阅 .struct/.endstruct/.tag 主题。

示例

在本例中，8 位值（10、－1、abc 和 a）被放入中。具有 .byte 的存储器中的字节。此外，8 位值（8、－3、def 和 b）被放入具有 .char 的存储器中的连续字节中。标签 STRX 的值为 0h，这是第一个已初始化字节的位置。标签 STRY 的值为 6h，这是由 .char 指令初始化的第一个字节。

```
1 00000000              .space 100h
2 00000100 0A   STRX    .byte 10, -1, "abc", 'a'
  00000101 FF
  00000102 61
  00000103 62
  00000104 63
  00000105 61
3 00000106 08   STRY    .char 8, -3, "def", 'b'
  00000107 FD
  00000108 64
  00000109 65
  0000010a 66
  0000010b 62
```

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.cdecls

*在 **C** 和汇编代码之间共享 **C** 头文件*

语法 单行：

.cdecls [*options* **,**] **"** *filename* **"**[**,** **"** *filename2* **"**[**,**...]]

语法 多行：

.cdecls [*options*]

%{

/*--*/

/ C/C++ code - Typically a list of #includes and a few defines */*

/*--*/

%}

说明 **.cdecls** 指令允许使用混合汇编和 C/C++ 环境的编程器共享 C 头文件，此头文件包含使用 C 和 汇编代码的声明和原型。任何合法的 C/C++ 都可以在 .cdecls 块中使用，并且 C/C++ 声明会使系统自动生成合适的汇编语言，使用户可以在汇编代码中引用 C/C++ 结构，比如调用函数、分配空间、访问结构成员和使用等效的汇编机制。虽然函数和变量定义会被忽略，但常见的 C/C++ 元素被转换为汇编语言，例如：枚举、(非函数类)宏、函数和变量原型、结构体和联合体。

.cdecls 选项会控制将代码视为 C 代码还是 C++ 代码；以及如何呈现 .cdecls 块和转换后的代码。选项必须用英文逗号分隔；它们可以按任何顺序出现：

C	将 .cdecls 块中的代码视为 C 源代码(默认)。
CPP	将 .cdecls 块中的代码视为 C++ 源代码。这和 C 选项相反。
NOLIST	不要在为包含汇编文件生成的任何列表文件中包含转换后的汇编代码(默认)。
LIST	在为包含汇编文件生成的任何列表文件中包含转换后的汇编代码。这和 NOLIST 选项相反。
NOWARN	不要在 STDERR 上发出有关在解析 .cdecls 源块时无法转换 C/C++ 构造的警告(默认)。
WARN	在 STDERR 上发出有关在解析 .cdecls 源块时无法转换 C/C++ 构造的警告(默认)。这和 NOWARN 选项相反。

在单行格式中，选项后跟一个或多个要包含的文件名。文件名和选项用英文逗号分隔。列出的每个文件犹如 #include "filename" 是以多行格式指定。

在多行格式中，.cdecls 后面的行必须包含开始的 .cdecls 块指示符 %{。%{ 之后的所有内容，直到结束块指示符 %}，都被视为 C/C++ 源代码并进行处理。普通汇编器进行处理，然后在结束 %} 后面的行上恢复。

%{ 和 %} 中的文本被传递给 C/C++ 编译器以转换为汇编语言。许多 C 语言语法(包括函数和变量定义以及类函数宏)不受支持，在转换过程中会被忽略。但是，传统上出现在 C 头文件中的所有内容都受支持，包括函数和变量原型、结构体和联合体声明、非类函数宏、枚举和 #defines。

生成的汇编语言包含在汇编文件中的 .cdecls 指令处。如果使用 LIST 选项，则转换后的汇编语句将列印在列表文件中。

由 .cdecls 指令生成的汇编语言的处理方式与 .include 文件类似。因此， .cdecls 指令可以嵌套在被复制或包含的文件中。汇编器将嵌套限制为十级；主机操作系统可能会设置额外

Copyright © 2023 Texas Instruments Incorporated

DRAFT ONLY
Confidential – NDA Restrictions

.cdecls (continued)

*在 **C** 和汇编代码之间共享 **C** 头文件*

的限制。汇编器在复制文件的行号之前加上一个字母代码来标识复制级别。A 表示第一个复制的文件，B 表示第二个复制的文件，以此类推。

.cdecls 指令可以出现在汇编源文件中的任何位置，并且可以在一个文件中多次出现。但是，一个 .cdecls 创建的 C/C++ 环境不会被后面的 .cdecls 继承；每个 .cdecls 的 C/C++ 环境在开始时都是全新的。

有关在 C 头文件中设置和使用 .cdecls 指令的更多信息，请参阅章节 13。

示例

在此示例中，使用 .cdecls 指令调用 C header.h 文件。

C 头文件：

```
#define WANT_ID 10
#define NAME "John\n"
extern int a_variable;
extern float cvt_integer(int src);
struct myCstruct { int member_a; float member_b; };
enum status_enum { OK = 1, FAILED = 256, RUNNING = 0 };
```

源文件：

```
         .cdecls C,LIST,"myheader.h"
size:    .int $$sizeof(myCstruct)
aoffset: .int myCstruct.member_a
boffset: .int myCstruct.member_b
okvalue: .int status_enum.OK
failval: .int status_enum.FAILED
         .if $$defined(WANT_ID)
id       .cstring NAME
         .endif
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.cdecls (continued)

在 C 和汇编代码之间共享 C 头文件

列表文件：

```
        1                             .cdecls C,LIST,"myheader.h"
A       1                    ; ------------------------------------------
A       2                    ; Assembly Generated from C/C++ Source Code
A       3                    ; ------------------------------------------
A       4
A       5                    ; =========== MACRO DEFINITIONS ===========
A       6                               .define "10",WANT_ID
A       7                               .define """John\n""",NAME
A       8
A       9                    ; =========== TYPE DEFINITIONS ===========
A      10                    status_enum      .enum
A      11           00000001 OK               .emember 1
A      12           00000100 FAILED           .emember 256
A      13           00000000 RUNNING          .emember 0
A      14                                     .endenum
A      15
A      16                    myCstruct        .struct 0,4
       17                    ; struct size=(8 bytes|64 bits), alignment=4
A      18           00000000 member_a         .field 32
       19                    ; int member_a - offset 0 bytes, size (4 bytes|32
bits)
A      20           00000004 member_b         .field 32
       21                    ; float member_b - offset 4 bytes, size (4 bytes|32
bits)
A      22           00000008                  .endstruct
       23                    ; final size=(8 bytes|64 bits)
A      24
A      25                    ; =========== EXTERNAL FUNCTIONS ===========
A      26                                     .global _cvt_integer
A      27
A      28                    ; =========== EXTERNAL VARIABLES ===========
A      29                                     .global _a_variable
        2 00000000 00000008 size:    .int $$sizeof(myCstruct)
        3 00000004 00000000 aoffset: .int myCstruct.member_a
        4 00000008 00000004 boffset: .int myCstruct.member_b
        5 0000000c 00000001 okvalue: .int status_enum.OK
        6 00000010 00000100 failval: .int status_enum.FAILED
        7                            .if $$defined(WANT_ID)
        8 00000014 0000004A id       .cstring NAME
          00000015 0000006F
          00000016 00000068
          00000017 0000006E
          00000018 0000000A
          00000019 00000000
        9                            .endif
```

DRAFT ONLY TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.common

创建通用符号

语法

.common *symbol* **,** *size in bytes*[**,** *alignment*]

.common *symbol* **,** *structure tag*[**,** *alignment*]

说明

.common 指令用于在通用块中创建通用符号，而不是在一个存储器段中放置变量。

使用通用符号的好处是生成的代码可以删除未使用的变量（如果不删除，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。）

当启用 --common 选项（默认）时，编译器将使用该指令，这会导致未初始化的文件作用域变量作为通用符号发出。除非使用 --common=off 编译器选项，否则默认情况下会对 C/C++ 代码进行这种优化。

- *symbol* 是一个必需参数。它为由该指令创建的符号定义了一个名称。符号名称必须与要保留空间的变量相对应。
- *size in bytes* 是一个必需参数；它必须是一个绝对表达式。汇编器在用于通用符号的段中分配字节大小。没有默认大小。
- *structure tag* 可用于代替大小来指定由 .struct 指令创建的结构。该参数需要一个大小或一个结构标签。
- *alignment* 是一个可选参数，可确保分配给符号的空间出现在指定的边界上。该边界必须设置为 2 的幂，介于 2^0 和 2^{15} 之间（包含端点值）。如果 SPC 已在指定的边界对齐，则不会递增。

通用符号是放置在 ELF 目标文件的符号表中的符号。它们表示未初始化的变量。通用符号不会引用段。（相反，初始化变量需要引用包含初始化数据的段。）通用符号的值是其所需的对齐方式；它没有地址，也没有存储地址。虽然未初始化的通用块的符号可以出现在可执行目标文件中，但通用符号只能出现在可重定位的目标文件中。通用符号优于弱符号。有关通用符号的更多信息，请参阅 System V ABI 规范中有关“符号表”的部分。

链接包含通用符号的目标文件之时，会在未初始化段 (.common) 中为每个通用符号保留空间。创建一个符号来代替通用符号以引用其保留位置。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.copy/.include

复制源文件

语法

.copy " *filename* **"**

.include " *filename* **"**

说明

.copy 和 **.include** 指令用于告诉汇编器从另一个文件读取源语句。从副本文件汇编的语句会列印在汇编列表中。汇编的 .list/.nolist 指令无论数量多少，从头文件中汇编的语句均*不会*列印在汇编列表中。

在汇编 .copy 或 .include 指令后，汇编器会：

1. 停止汇编当前源文件中的语句
2. 汇编复制/头文件中的语句
3. 从 .copy 或 .include 指令后面的语句开始，继续汇编主源文件中的语句

文件名 是指定源文件的必需参数。该参数用双引号引起来，并且必须遵循操作系统规则。

您可以指定完整路径名（例如，/320tools/file1.asm）。如果未指定完整路径名，汇编器将在以下位置搜索文件：

1. 包含当前源文件的目录
2. 使用 --include_path 汇编器选项指定的任何目录
3. 由 TI_ARM_A_DIR 环境变量指定的任何目录
4. 由 TI_ARM_C_DIR 环境变量指定的任何目录

有关 --include_path 选项和 TI_ARM_A_DIR 的更多信息，请参阅节 4.5。有关 TI_ARM_C_DIR 的更多信息，请参阅 *ARM 优化 C/C++ 编译器用户指南*。

.copy 和 .include 指令可以嵌套在所复制或包含的文件中。汇编器将嵌套限制为 32 级；主机操作系统可能会设置额外的限制。汇编器在复制文件的行号之前加上一个字母代码来标识复制级别。A 表示第一个复制的文件，B 表示第二个复制的文件，以此类推。

示例 1

在此示例中，.copy 指令用于从其他文件读取和汇编源语句；然后，汇编器继续汇编到当前文件中。

原始文件 copy.asm 包含一条用于复制文件 byte.asm 的 .copy 语句。当 copy.asm 进行汇编时，汇编器会将 byte.asm 复制到它在列表中的位置（注意下面的列表）。副本文件 byte.asm 包含用于第二个文件 word.asm 的 .copy 语句。

当遇到 word.asm 的 .copy 语句时，汇编器会切换到 word.asm 以继续复制和汇编。然后，汇编器返回到它在 byte.asm 中的位置以继续进行复制和汇编。完成 byte.asm 的汇编后，汇编器返回到 copy.asm 以汇编其剩余的语句。

copy.asm （源文件）	byte.asm （第一个副本文件）	word.asm （第二个副本文件）

`        .space 29`
`        .copy "byte.asm"`
`** Back in original file`
`        .string "done"`
	`** In byte.asm`
	`        .byte 32,1+ 'A'`
	`        .copy "word.asm"`
	`** Back in byte.asm`
	`        .byte 67h + 3q`
		`** In word.asm`
		`        .word 0ABCDh, 56q`

列表文件：

```
       1 00000000                 .space 29
       2                          .copy "byte.asm"
A      1                      ** In byte.asm
A      2 0000001d 20              .byte 32,1+ 'A'
```


Copyright © 2023 Texas Instruments Incorporated

.copy/.include (continued)

复制源文件

```
                0000001e 42
A       3                          .copy "word.asm"
B       1                      ** In word.asm
B       2 00000020 0000ABCD        .word 0ABCDh, 56q
          00000024 0000002E
A       4                      ** Back in byte.asm
A       5 00000028 6A              .byte 67h + 3q
        3
        4                      ** Back in original file
        5 00000029 64              .string "done"
          0000002a 6F
          0000002b 6E
          0000002c 65
```

示例 2 在此示例中，.include 指令用于从其他文件读取和汇编源语句；然后，汇编器继续汇编到当前文件中。该机制类似于 .copy 指令，只是语句不会列印在列表文件中。

include.asm （源文件）	byte2.asm （第一个副本文件）	word2.asm （第二个副本文件）

`.space 29`
`.include "byte2.asm"`
`** Back in original file`
`.string "done"`
	`** In byte2.asm`
	`.byte 32,1+ 'A'`
	`.include`
	`"word2.asm"`
	`** Back in byte2.asm`
	`.byte 67h + 3q`
		`** In word2.asm`
		`.word 0ABCDh, 56q`

列表文件：

```
        1 00000000                 .space 29
        2                          .include "byte2.asm"
        3
        4                      ** Back in original file
        5 00000029 64              .string "done"
          0000002a 6F
          0000002b 6E
          0000002c 65
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.cstruct/.cunion/.endstruct/.endunion/.tag

*声明 **C** 结构类型*

语法

[*stag*] **.cstruct|.cunion** [*expr*]

[mem_0] *element* [$expr_0$]
[mem_1] *element* [$expr_1$]
...
...

[mem_n] **.tag** *stag* [$expr_n$]

[mem_N] *element* [$expr_N$]

[*size*] **.endstruct|.endunion**

label **.tag** *stag*

说明

添加了 **.cstruct** 和 **.cunion** 指令，以支持在汇编代码和 C 代码之间轻松共享通用数据结构。.cstruct 和 .cunion 指令的使用方法与现有 .struct 和 .union 指令相同，还可确保执行与 C 编译器用于 C 结构体和联合体数据类型的布局匹配的数据布局。

特别是，当这些类型嵌套在复合数据结构中时，.cstruct 和 .cunion 指令会强制执行与 C 编译器所用相同的对齐和填充方式。

.endstruct 指令用于终止结构定义。**.endunion** 指令用于终止联合体定义。

.tag 指令为 *label* 提供结构特征，简化符号表示，还可定义包含其他结构的结构。.tag 指令不会分配存储器空间，必须先定义 .tag 指令的结构标签 (*stag*)。

以下是要与 .struct、.endstruct 和 .tag 指令一同使用的参数的说明：

- *stag* 是结构体的标签，其值与结构体的开头相关联。如果不存在 stag，汇编器会将结构成员放在全局符号表中，其值是它们距结构体顶部的绝对偏移量。stag 对于 .struct 而言是可选项，但对于 .tag 而言是必备项。
- *元素* 是以下描述符之一：.byte、.char、.int、.long、.word、.double、.half、.short、.string、.float 和 .field。除了 .tag 之外的所有上述指令都是用于初始化存储器的典型指令。在 .struct 指令之后，这些指令描述结构元素的大小，不分配存储器空间。.tag 指令是一种特例，因为必须使用 stag（如在 stag 的定义中）。
- *expr* 是一个可选表达式，指示结构体的开始偏移量。结构体的默认起始点为 0。
- $expr_{n/N}$ 是表示所描述元素数量的可选表达式，该值默认为 1。.string 元素大小是一个字节，.field 元素大小是一位。
- $mem_{n/N}$ 是结构成员的可选标签。该标签是绝对值，相当于距结构体开始处的当前偏移量。结构成员的标签不能声明为全局标签。
- *size* 是表示结构体总大小的可选标签。

示例

此示例演示了以 C 表示、将使用汇编代码访问的结构体。

```
typedef struct STRUCT1
; {     int i0;       /* offset 0 */
;       short s0;     /* offset 4 */
; } struct1;          /* size 8, alignment 4 */
;
; typedef struct STRUCT2
; {     struct1 st1;  /* offset 0 */
;       short s1;     /* offset 8 */
; } struct2;          /* size 12, alignment 4 */
;
; The structure will get the following offsets once the C compiler lays out the
```


English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

.cstruct/.cunion/.endstruct/.endunion/.tag (continued)

声明 C 结构类型

```
structure
; elements according to the C standard rules:
;
; offsetof(struct1, i0) = 0
; offsetof(struct1, s0) = 4
; sizeof(struct1)       = 8
; offsetof(struct2, s1) = 0
; offsetof(struct2, i1) = 8
; sizeof(struct2)       = 12
;
; Attempts to replicate this structure in assembly using the .struct/.union
directives will not
; create the correct offsets because the assembler tries to use the most
compact arrangement:
struct1     .struct
i0          .int                    ; bytes 0-3
s0          .short; bytes 4-5
struct1len  .endstruct              ; size 6, alignment 4
struct2     .struct
st1         .tag struct1            ; bytes 0-5
s1          .short                  ; bytes 6-7
endstruct2  .endstruct              ; size 8, alignment 4
        .sect "data1"
        .word struct1.i0            ; 0
        .word struct1.s0            ; 4
        .word struct1len            ; 6
        .sect "data2"
        .word struct2.st1           ; 0
        .word struct2.s1            ; 6
        .word endstruct2            ; 8
;
; The .cstruct/.cunion directives calculate offsets in the same manner as the C
compiler.The resulting
; assembly structure can be used to access the elements of the C
structure.Compare the difference
; in the offsets of those structures defined via .struct above and the offsets
for the C code.
cstruct1    .cstruct
i0          .int                    ; bytes 0-3
s0          .short; bytes 4-5
cstruct1len .endstruct              ; size 8, alignment 4
cstruct2    .cstruct
st1         .tag cstruct1           ; bytes 0-7
s1          .short; bytes 8-9
cendstruct2 .endstruct              ; size 12, alignment 4
        .sect "data3"
        .word cstruct1.i0, struct1.i0   ; 0
        .word cstruct1.s0, struct1.s0   ; 4
        .word cstruct1len, struct1len   ; 8
        .sect "data4"
        .word cstruct2.st1, struct2.st1 ; 0
        .word cstruct2.s1, struct2.s1   ; 8
        .word cendstruct2, endstruct2   ; 12
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.data

汇编到 .data 段

语法

.data

说明

.data 指令用于将 .data 设置为当前段；其后面的行将汇编到 .data 段。.data 段通常用于包含数据表或预初始化的变量。

有关各段的更多信息，请参阅章节 2。

示例

在此示例中，代码汇编到 .data 和 .text 段。

```
 1                              ***********************************************
 2                              **           保留 .data 段中的空间。          **
 3                              ***********************************************
 4 00000000                             .data
 5 00000000                             .space  0CCh
 6
 7                              ***********************************************
 8                              **           汇编到 .text 段。               **
 9                              ***********************************************
10 00000000                             .text           ; Constant into .data
11          00000000            INDEX   .set    0
12 00000000 E3A00000                    MOV     R0, #INDEX
13
14                              ***********************************************
15                              **           汇编到 .data 段。               **
16                              ***********************************************
17 000000cc                     Table:  .data
18 000000cc FFFFFFFF                    .word   -1      ; Assemble 32-bit
19                                                      ; constant into .data.
20
21 000000d0 FF                          .byte   0FFh    ; Assemble 8-bit
22                                                      ; constant into .data.
23
24                              ***********************************************
25                              **           汇编到 .text 段。               **
26                              ***********************************************
27 00000004                             .text
28 00000004 000000CC"           con:    .field  Table, 32
29 00000008 E51F100C                    LDR     R1, con
30 0000000c E5912000                    LDR     R2, [R1]
31 00000010 E0802002                    ADD     R2, R0, R2
32                              ***********************************************
33                              **   恢复汇编到地址 0Fh 处 **
34                              **    的 .data 段。                           **
35                              ***********************************************
36 000000d1                             .data
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

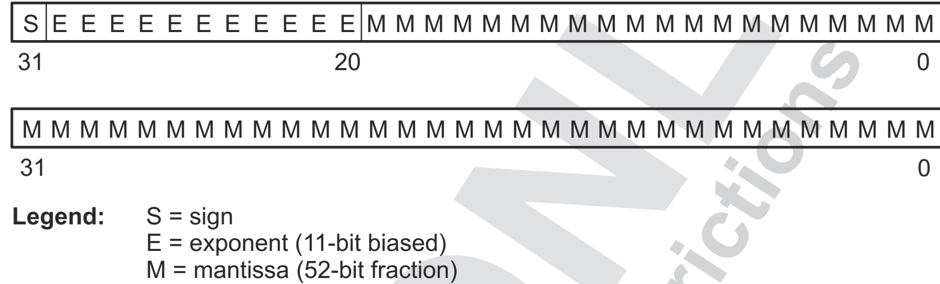
.double

初始化双精度浮点值

语法 **.double** *value*$_1$ [, ..., *value*$_n$]

说明 **.double** 指令用于将一个或多个浮点值的 IEEE 双精度浮点表示置于当前段中。每个*值*必须是具有算术类型的绝对常量表达式，或等同于具有算术类型的绝对常量表达式的符号。每个常量都转换为 IEEE 双精度 64 位格式的浮点值。双精度浮点常量与双字边界对齐。

64 位值以图 5-5 中所示的格式存储。



Legend: S = sign
E = exponent (11-bit biased)
M = mantissa (52-bit fraction)

图 5-5. 双精度浮点格式

当用户在 .struct/.endstruct 序列中使用 .double 时，.double 会定义成员的大小，但不会初始化存储器。如需更多信息，请参阅 .struct/.endstruct/.tag 主题。

示例 此示例显示了 .double 指令。

```
      1 00000000 C5308B2A          .double -2.0e25
        00000004 2C280291
      2 00000008 40180000          .double 6
        0000000c 00000000
      3 00000010 407C8000          .double 456
        00000014 00000000
```

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.drlist/.drnolist

控制指令列表

语法

.drlist

.drnolist

说明

两个指令可使用户控制将汇编器指令列印到列表文件中：

.drlist 指令用于将所有指令列印到列表文件中。

.drnolist 指令用于禁止将以下指令列印到列表文件中。.drnolist 指令在宏中不起任何作用。

- .asg
- .break
- .emsg
- .eval
- .fclist
- .fcnolist
- .mlist
- .mmsg
- .mnolist
- .sslist
- .ssnolist
- .var
- .wmsg

默认情况下，汇编器的行为就像已指定了 .drlist 指令一样。

示例

此示例演示了 .drnolist 如何禁止列出指定的指令。

源文件：

```
      .asg     0, x
      .loop    2
      .eval    x+1, x
      .endloop
      .drnolist
      .asg     1, x
      .loop    3
      .eval    x+1, x
      .endloop
```

列表文件：

```
        3                            .asg     0, x
        4                            .loop    2
        5                            .eval    x+1, x
        6                            .endloop
1                                    .eval    0+1, x
1                                    .eval    1+1, x
        7
        8                            .drnolist
       12                            .loop    3
       13                            .eval    x+1, x
       14                            .endloop
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.elfsym

ELF 符号信息

语法 **.elfsym** *name* **, SYM_SIZE(** *size* **)**

说明 .elfsym 指令为 ELF 格式的符号提供了附加信息。该指令旨在传达不同类型的信息，因此每种类型都使用 *type*(*value*) 语法。目前，该指令仅支持 SYM_SIZE 类型。

备注

SYM_SIZE 指示由 *名称* 表示的符号的分配大小（以字节为单位）。

示例 该示例展示了 ELF 符号信息指令的用法。

```
            .sect      ".examp"
            .align 4
            .elfsym    ex_sym, SYM_SIZE(4)
    ex_sym:
             .word     0
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.emsg/.mmsg/.wmsg

定义消息

语法

.emsg *string*

.mmsg *string*

.wmsg *string*

说明

这些指令可使用户定义自有的错误和警告消息。当用户使用这些指令时，汇编器会跟踪遇到的错误和警告的数量，并将这些数字列印在列表文件的最后一行。

.emsg 指令以与汇编器相同的方式向标准输出器件发送错误消息。它会增加错误计数并阻止汇编器生成目标文件。

.mmsg 指令以与 .emsg 和 .wmsg 指令相同的方式向标准输出器件发送汇编时消息。但是，它不会设置错误或警告计数，也不会阻止汇编器生成目标文件。

.wmsg 指令以与 .emsg 指令相同的方式向标准输出器件发送警告消息。然而，它会增加警告计数而非错误计数，也不会阻止汇编器生成目标文件。

示例

该示例向标准输出器件发送“ERROR -- MISSING PARAMETER”消息。

源文件：

```
MSG_EX   .macro parm1
         .if    $$symlen(parm1) = 0
         .emsg  "ERROR -- MISSING PARAMETER"
         .else
          ADD   parm1, r7, r8
         .endif
         .endm
         MSG_EX R0
         MSG_EX
```

列表文件：

```
       1                     MSG_EX  .macro   parm1
       2                             .if      $$symlen(parm1) = 0
       3                             .emsg    "ERROR -- MISSING PARAMETER"
       4                             .else
       5                             ADD      parm1, r7, r8
       6                             .endif
       7                             .endm
       8
       9 00000000                    MSG_EX   R0
1                                    .if      $$symlen(parm1) = 0
1                                    .emsg    "ERROR -- MISSING PARAMETER"
1                                    .else
1        00000000 E0870008           ADD      R0, r7, r8
1                                    .endif
      10
      11 00000004                    MSG_EX
1                                    .if      $$symlen(parm1) = 0
1                                    .emsg    "ERROR -- MISSING PARAMETER"
  ***** USER ERROR ***** - : ERROR -- MISSING PARAMETER
1                                    .else
1                                    ADD      parm1, r7, r8
1                                    .endif
 1 Error, No Warnings
```

此外，汇编器将以下消息发送到标准输出：

```
*** ERROR!   line 11:  ***** USER ERROR ***** - : ERROR -- MISSING PARAMETER
                .emsg   "ERROR -- MISSING PARAMETER"    ]]
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.emsg/.mmsg/.wmsg (continued)

定义消息

```
1 Error, No Warnings
Errors in source - Assembler Aborted
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.end

结束汇编

语法 **.end**

说明 **.end** 指令是可选的，用于终止汇编。汇编器会忽略 .end 指令后面的所有源语句。如果使用 .end 指令，则其必须是程序的最后一条源语句。

该指令与文件结束符具有相同的效果。当用户正在调试并希望在代码中的特定点停止汇编时，可以使用 .end。

备注

结束宏

请勿使用 .end 指令来终止宏，而要使用 .endm 宏指令。

示例 该示例展示了 .end 指令如何终止汇编。汇编器会忽略 .end 指令后面的所有源语句。

源文件：

```
START:  .space   300
TEMP    .set     15
        .bss     LOC1, 48h
LOCL_n  .word    LOC1
        MVN      R0, R0
        ADD      R0, R0, #TEMP
        LDR      R4, LOCL_n
        STR      R0, [R4]
        .end
        .byte    4
        .word    CCCh
```

列表文件：

```
       1 00000000          START:  .space  300
       2          0000000F TEMP    .set    15
       3 00000000                  .bss    LOC1, 48h
       4 0000012c 00000000- LOCL_n .word   LOC1
       5 00000130 E1E00000         MVN     R0, R0
       6 00000134 E280000F         ADD     R0, R0, #TEMP
       7 00000138 E51F4014         LDR     R4, LOCL_n
       8 0000013c E5840000         STR     R0, [R4]
       9                           .end
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.fclist/.fcnolist

控制错误条件代码块的列表

语法

.fclist

.fcnolist

说明

可使用两组指令来控制错误条件代码块的列表：

.fclist 指令用于列出错误条件代码块（不生成代码的条件代码块）。

.fcnolist 指令会抑制错误条件代码块的列出，直到遇到 .fclist 指令。使用 .fcnolist，列表中仅显示实际汇编的条件代码块中的代码。不会显示 .if、.elseif、.else, 和 .endif 指令。

默认情况下列出所有条件代码块；汇编器的行为就像使用了 .fclist 指令一样。

示例

该示例显示了带和不带所列条件代码块的代码的汇编语言和列表文件。

源文件：

```
AAA    .set  1
BBB    .set  0
       .fclist
       .if   AAA
       ADD   R0, R0, #1024
       .else
       ADD   R0, R0, #1024*10
       .endif
       .fcnolist
       .if   AAA
       ADD   R0, R0, #1024
       .else
       ADD   R0, R0, #1024*10
       .endif
```

列表文件：

```
***ARM***
       1          00000001  AAA    .set  1
       2          00000000  BBB    .set  0
       3                           .fclist
       4
       5                           .if   AAA
       6 00000000 E2800B01         ADD   R0, R0, #1024
       7                           .else
       8                           ADD   R0, R0, #1024*10
       9                           .endif
      10
      11                           .fcnolist
      12
      14 00000004 E2800B01         ADD   R0, R0, #1024
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.field

初始化字段

语法 **.field** *value*[**,** *size in bits*]

说明 **.field** 指令用于初始化存储器单个字内的多位字段（32 位）。此指令有两个操作数：

- *value* 是必需参数；它是一个被求值并且置于字段中的表达式。值必须是绝对值。
- *以位表示的大小* 是一个可选值（取值范围为 1 至 32），表示字段中的位数。默认大小为 。 32 位开始。如果指定的值无法容纳在 *以位表示的大小* 中，汇编器会截断该值并发出警告消息。例如，.field 3,1 会使汇编器将值 3 截断为 1 并列印以下消息：

```
*** WARNING! line 21: W0001: Field value truncated to 1
                 .field  3, 1
```

连续的 .field 指令从当前字开始将值打包成指定位数。字段从字的最高有效位开始打包，随着更多字段的添加，移向最低有效位。如果汇编器遇到不适合当前字的字段大小，则会写出该字，并开始将字段打包成下一个字。

.field 指令类似于 .bits 指令（请参阅 .bits 主题）。但是，.bits 指令不会强制对齐到字段边界，并且不会在到达字边界时自动递增 SPC。

使用 .align 指令来强制下一个 .field 指令开始打包一个新字。

如果使用标签，则其指向包含指定字段的字节。

当用户在 .struct/.endstruct 序列中使用 .field 时，.field 会定义成员的大小；但不会初始化存储器。如需更多信息，请参阅 .struct/.endstruct/.tag 主题。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.field (continued)

初始化字段

示例

该示例展示了如何将字段打包成一个字。在填满一个字并开始下一个字之前，SPC 不会改变。

```
 1                        ************************************
 2                        **    Initialize a 14-bit field.  **
 3                        ************************************
 4 00000000 2AF00000              .field  0ABCh, 14
 5
 6                        ************************************
 7                        **    Initialize a 5-bit field    **
 8                        **        in the same word.       **
 9                        ************************************
10 00000000 2AF14000      L_F:    .field  0Ah, 5
11
12                        ************************************
13                        **       Write out the word.      **
14                        ************************************
15                                .align  4
16
17                        ************************************
18                        **    Initialize a 4-bit field.   **
19                        ** This fields starts a new word.**
20                        ************************************
21 00000004 C0000000      x:      .field  0Ch, 4
22
23                        ************************************
24                        **    32-bit relocatable field    **
25                        **        in the next word.       **
26                        ************************************
27 00000008 00000004'             .field  x
28
29                        ************************************
30                        **   Initialize a 32-bit field.   **
31                        ************************************
32 0000000c 00004321              .field  04321h, 32
```

图 5-6 展示了该示例中的指令如何影响存储器。

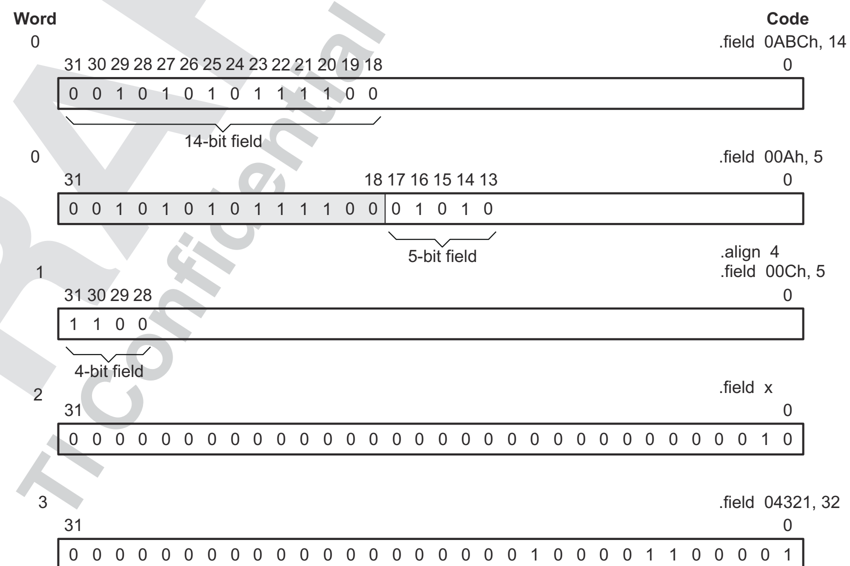


图 5-6. .field 指令

Copyright © 2023 Texas Instruments Incorporated

.float

初始化单精度浮点值

语法 **.float** *value*[, ..., $value_n$]

说明 **.float** 指令将单浮点常量的 IEEE 单精度浮点表示应用于当前段中的一个字中。*value* 必须是具有算术类型的绝对常量表达式，或等同于具有算术类型的绝对常量表达式的符号。每个常量都转换为 IEEE 单精度 32 位格式的浮点值。

存储 32 位值时首先存储指数字节，然后存储小数的最高有效字节，最后存储小数的最低有效字节，其格式如图 5-7 所示。

S	E E E E E E E E	M M M M M M M M M M M M M M M M M M M M M M M
31	23	0

value = $(-1)^S \times (1.0 + \text{mantissa}) \times (2)^{\text{exponent}-127}$

Legend: S = sign (1 bit)
E = exponent (8-bit biased)
M = mantissa (23-bit fraction)

图 5-7. 单精度浮点格式

当您在 .struct/.endstruct 序列中使用 .float 时，.float 会定义成员的大小；但不会初始化存储器。如需更多信息，请参阅 .struct/.endstruct/.tag 主题。

示例 以下是 .float 指令的示例：

```
1 00000000 E9045951        .float  -1.0e25
2 00000004 40400000        .float  3
3 00000008 42F60000        .float  123
```

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.global/.def/.ref

识别全局符号

语法

.global *symbol*$_1$[, ..., *symbol*$_n$]

.def *symbol*$_1$[, ..., *symbol*$_n$]

.ref *symbol*$_1$[, ..., *symbol*$_n$]

说明

三条指令用于识别在外部定义或可在外部引用的全局符号：

.def 指令用于识别在当前模块中定义并可由其他文件访问的符号。汇编器将此符号置于符号表中。

.ref 指令用于识别在当前模块中使用但在另一模块中定义的符号。链接器在链接时解析此符号的定义。

.global 指令可根据需要充当 .ref 或 .def。

全局符号 的定义方式与任何其他符号相同；即，它显示为标签或由 .set、.equ、.bss 或 .usect 指令定义。如果多次定义了某个全局符号，链接器会发出“多重定义”错误。（汇编器可以为局部符号提供类似的“多重定义”错误。）.ref 指令始终为一个符号创建符号表条目，无论模块是否使用该符号；而 .global 仅在模块实际使用该符号时才创建条目。

一个符号可以声明为全局符号的原因有两种：

- 如果该符号*不是在当前模块中定义*（当前模块包括宏文件、副本文件和包含文件），则 .global 或 .ref 指令告知汇编器该符号是在外部模块中定义的。这可防止汇编器发出未解析的引用错误。链接时，链接器会在其他模块中查找该符号的定义。
- 如果该符号*是在当前模块中定义*，则 .global 或 .def 指令声明该符号及其定义可由其他模块在外部使用。这些类型的引用在链接时解析。

示例

本示例显示四个文件。对于使用的所有符号，file1.lst 和 file2.lst 相互引用；file3.lst 和 file4.lst 也有类似的关系。

file1.lst 和 **file3.lst** 文件是等效的。这两个文件都定义了符号 INIT 并使其可用于其他模块；这两个文件都使用了外部符号 X、Y 和 Z。此外，file1.lst 使用 .global 指令来识别这些全局符号；file3.lst 使用 .ref 和 .def 来识别符号。

file2.lst 和 **file4.lst** 文件是等效的。这两个文件都定义了符号 X、Y 和 Z，并使其可用于其他模块；这两个文件都使用了外部符号 INIT。此外，file2.lst 使用 .global 指令来识别这些全局符号；file4.lst 使用 .ref 和 .def 来识别符号。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.global/.def/.ref (continued)

识别全局符号

file1.lst

```
      1                    ; Global symbol defined in this file
      2                             .global INIT
      3                    ; Global symbols defined in file2.lst
      4                             .global X, Y, Z
      5 00000000           INIT:
      6 00000000 E2800056           ADD     R0, R0, #56h
      7 00000004 00000000!          .word   X
      8                    ;        .
      9                    ;        .
     10                    ;        .
     11                             .end
```

file2.lst

```
      1                    ; Global symbols defined in this file
      2                             .global X, Y, Z
      3                    ; Global symbol defined in file1.lst
      4                             .global INIT
      5          00000001  X:       .set    1
      6          00000002  Y:       .set    2
      7          00000003  Z:       .set    3
      8 00000000 00000000!          .word   INIT
      9                    ;        .
     10                    ;        .
     11                    ;        .
     12                             .end
```

file3.lst

```
      1                    ; Global symbols defined in this file
      2                             .def    INIT
      3                    ; Global symbol defined in file4.lst
      4                             .ref    X, Y, Z
      5 00000000           INIT:
      6 00000000 E2800056           ADD     R0, R0, #56
      7 00000004 00000000!          .word   X
      8                    ;        .
      9                    ;        .
     10                    ;        .
     11                             .end
```

file4.lst

```
      1                    ; Global symbols defined in this file
      2                             .def    X, Y, Z
      3                    ; Global symbol defined in file3.lst
      4                             .ref    INIT
      5          00000001  X:       .set    1
      6          00000002  Y:       .set    2
      7          00000003  Z:       .set    3
      8 00000000 00000000!          .word   INIT
      9                    ;        .
     10                    ;        .
     11                    ;        .
     12                             .end
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.group/.gmember/.endgroup

定义通用数据段

语法

.group *group_section_name group_type*

.gmember *section_name*

.endgroup

说明

三条指令用于指示汇编器使某些段成为 ELF 组段的成员（有关组段的更多信息，请参阅 ELF 规范）。

.group 指令用于开始组声明。*group_section_name* 用于指定组段的名称。*group_type* 指定组类型。支持以下类型：

0x0	常规 ELF 组
0x1	COMDAT ELF 组

多个模块中允许存在重复的 COMDAT（通用数据）组；链接器只保留一个组。创建此类重复组对于 C++ 模板的延迟实例化和提供调试信息非常有用。

.gmember 指令用于将 *section_name* 指定为该组成员。

.endgroup 指令用于结束组声明。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.half/.short/.uhalf/.ushort

初始化 16 位整数

语法

.half *value*$_1$[, ..., *value*$_n$]

.short *value*$_1$[, ..., *value*$_n$]

.uhalf *value*$_1$[, ..., *value*$_n$]

.ushort *value*$_1$[, ..., *value*$_n$]

说明

.half 和 **.short** 指令将一个或多个值放入当前段的连续半字中。*value* 可以是以下任一项：

- 汇编器计算并视为 16 位有符号或无符号数的表达式
- 用双引号引起来的字符串。字符串中的每个字符表示一个单独的值，并单独存储在 16 位字段的八个最低有效位中，用 0 填充。

汇编器会截断大于 16 位的值。

如果在 .half 或 .short 中使用标签，则它指向汇编器放置第一个字节的位置。

这些指令在数据写入段之前执行半字（16 位）对齐。这可保证数据驻留在 16 位边界上。

当您在 .struct/.endstruct 序列中使用 .half 或 .short 指令时，它们会定义成员的大小，但不会初始化存储器。如需更多信息，请参阅 .struct/.endstruct/.tag 主题。

示例

在此示例中，.half 用于将 16 位值（10、-1、abc 和 a）放入存储器的连续半字中；.short 用于将 16 位值（8、-3、def 和 b）放入存储器的连续半字中。标签 STRN 的值为 100ch，这是 .short 的第一个已初始化半字的位置。

```
1 00000000                 .space  100h * 16
2 00001000 000A            .half   10, -1, "abc", 'a'
  00001002 FFFF
  00001004 0061
  00001006 0062
  00001008 0063
  0000100a 0061
3 0000100c 0008     STRN   .short   8, -3, "def", 'b'
  0000100e FFFD
  00001010 0064
  00001012 0065
  00001014 0066
  00001016 0062
```

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.if/.elseif/.else/.endif

汇编条件代码块

语法

.if *condition*

[**.elseif** *condition*]

[**.else**]

.endif

说明

.if 指令用于标记条件代码块的开始。*条件* 是必备项。

- 如果表达式计算结果为 true（非零），则汇编器会汇编表达式后面的代码（一直到 .elseif、.else 或 .endif）。
- 如果表达式计算结果为 false (0)，则汇编器会汇编 .elseif（如果存在）、.else（如果存在）或 .endif（如果 .elseif 或 .else 不存在）后面的代码。

.elseif 指令用于标识当 .if 表达式为 false (0) 和 .elseif 表达式为 true（非零）时要汇编的代码块。当 .elseif 表达式计算结果为 false 时，汇编器继续执行到下一个 .elseif（如果存在）、.else（如果存在）或 .endif（如果 .elseif 或 .else 不存在）。在条件代码块中，.elseif 是可选项，可以使用多个 .elseif。如果表达式计算结果为 false，且没有 .elseif，则汇编器会继续执行 .else（如果存在）或 .endif 后面的代码。

.else 指令用于标识当 .if 表达式和所有 .elseif 表达式的计算结果均为 false (0) 时汇编器汇编的代码。.else 指令是可选的；如果表达式计算结果为 false，且没有 .else 语句，则汇编器会继续执行 .endif 后面的代码。可以在同一个条件代码块中使用 .elseif 和 .else 指令。

.endif 指令用于终止条件代码块。

请参阅节 4.9.2，了解有关关系运算符的信息。

示例

此示例展示了条件汇编：

```
 1          00000001  SYM1    .set    1
 2          00000002  SYM2    .set    2
 3          00000003  SYM3    .set    3
 4          00000004  SYM4    .set    4
 5
 6                    If_4:   .if     SYM4 = SYM2 * SYM2
 7 00000000 04                .byte   SYM4            ; Equal values
 8                            .else
 9                            .byte   SYM2 * SYM2     ; Unequal values
10                            .endif
11
12                    If_5:   .if     SYM1 <= 10
13 00000001 0A                .byte   10              ; Less than / equal
14                            .else
15                            .byte   SYM1            ; Greater than
16                            .endif
17
18                    If_6:   .if     SYM3 * SYM2 != SYM4 + SYM2
19                            .byte   SYM3 * SYM2     ; Unequal value
20                            .else
21 00000002 08                .byte   SYM4 + SYM4     ; Equal values
22                            .endif
23
24                    If_7:   .if     SYM1 = SYM2
25                            .byte   SYM1
26                            .elseif SYM2 + SYM3 = 5
27 00000003 05                .byte   SYM2 + SYM3
28                            .endif
```

DRAFT ONLY TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.int/.unint/.long/.ulong/.word/.uword

初始化 32 位整数

语法

.int *value*$_1$[, ..., *value*$_n$]

.uint *value*$_1$[, ..., *value*$_n$]

.long *value*$_1$[, ..., *value*$_n$]

.ulong *value*$_1$[, ..., *value*$_n$]

.word *value*$_1$[, ..., *value*$_n$]

.uword *value*$_1$[, ..., *value*$_n$]

说明

.int、**.unint**、**.long**、**.ulong**、 **.word** 和 **.uword** 指令将一个或多个值置于当前段的连续字中。每个值单独置于一个 32 位字中，并在字边界上对齐。*值* 可以是以下任一项：

- 汇编器计算并视为 32 位有符号或无符号数的表达式
- 用双引号引起来的字符串。字符串中的每个字符表示一个单独的值，并单独存储在 32 位字段的八个最低有效位中，用 0 s 填充。

值可以是绝对表达式或可重定位表达式。如果某个表达式可重定位，则汇编器会生成引用适当符号的重定位条目；然后链接器可以正确修补（重定位）引用。这可使用户使用指向变量或标签的指针来初始化存储器。

如果在这些指令中使用标签，则它会指向初始化的第一个字。

当用户在 .struct/.endstruct 序列中使用这些指令时，它们会定义成员的大小，但不会初始化存储器。请参阅 .struct/.endstruct/.tag 主题。

示例 1

此示例使用 .int 指令来初始化字。

```
1 00000000                   .space 73h
2 00000000                   .bss   PAGE, 128
3 00000080                   .bss   SYMPTR, 4
4 00000074 E3A00056  INST:   MOV    R0, #056h
5 00000078 0000000A          .int   10, SYMPTR, -1, 35 + 'a', INST, "abc"
  0000007c 00000080-
  00000080 FFFFFFFF
  00000084 00000084
  00000088 00000074'
  0000008c 00000061
  00000090 00000062
  00000094 00000063
```

示例 2

此示例展示了 .long 指令如何初始化字。符号 DAT1 指向保留的第一个字。

```
1 00000000 0000ABCD  DAT1:   .long   0ABCDh, 'A' + 100h, 'g', 'o'
  00000004 00000141
  00000008 00000067
  0000000c 0000006F
2 00000010 00000000'         .long   DAT1, 0AABBCCDDh
  00000014 AABBCCDD
3 00000018           DAT2:
```

示例 3

在这个示例中，使用 .word 指令来初始化字。符号 WORDX 指向保留的第一个字。

```
1 00000000 00000C80  WORDX:   .word   3200, 1 + 'AB', -0AFh, 'X'
  00000004 00004242
  00000008 FFFFFF51
  0000000c 00000058
```


Copyright © 2023 Texas Instruments Incorporated

.label

创建加载时地址标签

语法 **.label** *symbol*

说明 **.label** 指令用于定义一个特殊*符号*，引用当前段内的加载时地址，而不是运行时地址。由汇编器创建的大部分段都有可重定位地址。汇编器汇编每个段，就好像它是从 0 开始的一样，而链接器会将其重新定位到其加载和运行时的地址。

对于某些应用，需要在一个地址加载段，然后在*另*一个 地址运行。例如，您可能需要将对性能至关重要的代码块加载到较慢的存储器以节省空间，然后将该代码移至高速存储器中运行。这样的段在链接时分配两个地址：一个加载地址和一个运行地址。该段中定义的所有标签会重新定位为引用运行时地址，以便在代码运行时正确引用该段（例如，分支）。有关运行时重新定位的更多信息，请参阅节 3.5。

.label 指令用于创建一个特殊标签来引用*加载时* 地址。此功能主要用于为重新定位该段的代码指定段的加载位置。

示例 此示例展示了加载时地址标签的用法。

```
sect  ".examp"
       .label examp_load ; load address of section
start:                   ; run address of section
       <code>
finish:                  ; run address of section end
       .label examp_end  ; load address of section end
```

有关在链接器中分配运行时和加载时地址的更多信息，请参阅节 8.5.6。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.length/.width

设置列表页大小

语法

.length [*page length*]

.width [*page width*]

说明

两条指令可用于控制输出列表文件的大小。

.length 指令用于设置输出列表文件的页长度。它会影响当前页和后续页。您可以使用另一条 .length 指令来重置页长度。

- 默认长度：60 行。如果未使用 .length 指令或使用 .length 指令时未指定 *页长度*，则输出列表长度默认为 60 行。
- 最小长度：1 行
- 最大长度：32 767 行

.width 指令用于设置输出列表文件的页宽度。它会影响汇编的下一行及后续行。您可以使用另一条 .width 指令来重置页宽度。

- 默认宽度：132 个字符。如果未使用 .width 指令或使用 .width 指令时未指定 *页宽度*，则输出列表宽度默认为 132 个字符。
- 最小宽度：80 个字符
- 最大宽度：200 个字符

宽度指的是列表文件中一整行的宽度；行计数器值、SPC 值和目标代码算作行宽度的一部分。源语句中超出页宽度的注释和其他部分会在列表中被截断。

汇编器不会列出 .width 和 .length 指令。

示例

以下示例展示了如何更改页长度和宽度。

```
********************************************
**        页长度 = 65 行            **
**        页宽度 = 85 个字符        **
********************************************
          .length    65
          .width     85
********************************************
**        页长度 = 55 行            **
**        页宽度 = 100 个字符       **
********************************************
          .length    55
          .width     100
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.list/.nolist

启动/停止源列表

语法

.list

.nolist

说明

两条指令可用于控制源列表的列印：

.list 指令用于列印源列表。

.nolist 指令用于抑制源列表的输出，直至遇到 .list 指令。.nolist 指令可用于减少汇编时间和源列表大小。它可以在宏定义中使用，用于抑制宏扩展的列出。

汇编器不会输出在 .nolist 指令后面出现的 .list 或 .nolist 指令或源语句。但是，它会继续增加行计数器。您可以嵌套 .list/.nolist 指令；每条 .nolist 指令都需要匹配的 .list 以恢复列表。

默认情况下，源列表会列印到列表文件；汇编器的行为就像使用了 .list 指令一样。但是，如果在调用汇编器时未通过在命令行上包括 --asm_listing 选项来请求列表文件（请参阅节 4.3），则汇编器会忽略 .list 指令。

示例

此示例演示了 .copy 指令如何从另一个文件插入源语句。第一次遇到此指令时，汇编器会在列表文件中列出已复制的源行。第二次遇到此指令时，因为汇编了 .nolist 指令，汇编器不会列出已复制的源行。.nolist、第二个 .copy 和 .list 指令不会出现在列表文件中。此外，即使没有列出源语句，行计数器也会递增。

源文件：

```
.copy"copy2.asm"
* Back in original file
NOP
.nolist
.copy"copy2.asm"
.list
* Back in original file
.string"Done"
```

列表文件：

```
      1                            .copy   "copy2.asm"
 A    1                    * In copy2.asm (copy file)
 A    2 00000000 00000020          .word 32, 1 + 'A'
        00000004 00000042
      2                    * Back in original file
      3 00000008 E1A00000          NOP
      7                    * Back in original file
      8 00000014 44                .string "Done"
        00000015 6F
        00000016 6E
        00000017 65
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.loop/.endloop/.break

反复汇编代码块

语法

.loop [*count*]

.break [*end-condition*]

.endloop

说明

您可以使用三条指令反复汇编代码块：

.loop 指令用于开始一个可重复的代码块。如果使用了可选的 *count* 操作数，则它必须是定义明确的整数表达式。*计数* 指示要执行的循环数（循环计数）。如果省略了 *计数*，则默认为 1024。循环将反复为次数计数，除非由 .break 指令提前终止。

可选的 **.break** 指令用于提早终止 .loop。您可以在不使用 .break 的情况下使用 .loop。仅当 *end-condition* 表达式为 true（计算结果为非零）时，.break 指令才会终止 .loop。如果省略了可选的 *end-condition* 操作数，则默认为 true。如果 *end-condition* 为 true，则汇编器会立即停止重复 .loop 主体；不会汇编 .break 之后和 .endloop 之前的任何剩余语句。汇编器恢复汇编 .endloop 指令之后的语句。如果 *end-condition* 为 false（计算结果为 0），则循环继续。

.endloop 指令用于标记可重复代码块的结束。当循环终止时，不管是通过 .break 指令的 true *end-condition* 还是通过对迭代次数执行循环计数，汇编器都会停止重复循环体，并恢复汇编 .endloop 指令之后的语句。

示例

此示例说明这些指令如何与 .eval 指令一同使用。前六行中的代码可扩展到紧跟在这六行之后的代码。

```
       1                        .eval      0,x
       2                  COEF  .loop
       3                        .word      x*100
       4                        .eval      x+1, x
       5                        .break     x = 6
       6                        .endloop
1        00000000 00000000      .word      0*100
1                               .eval      0+1, x
1                               .break     1 = 6
1        00000004 00000064      .word      1*100
1                               .eval      1+1, x
1                               .break     2 = 6
1        00000008 000000C8      .word      2*100
1                               .eval      2+1, x
1                               .break     3 = 6
1        0000000c 0000012C      .word      3*100
1                               .eval      3+1, x
1                               .break     4 = 6
1        00000010 00000190      .word      4*100
1                               .eval      4+1, x
1                               .break     5 = 6
1        00000014 000001F4      .word      5*100
1                               .eval      5+1, x
1                               .break     6 = 6
```


Copyright © 2023 Texas Instruments Incorporated

DRAFT ONLY TI Confidential - NDA Restrictions

.macro/.endm

定义宏

语法

macname **.macro** [*parameter*$_1$[, ..., *parameter*$_n$]]

model statements or macro directives

.endm

说明

.macro 和 **.endm** 指令用于定义宏。

可在程序中的任何位置定义宏，但必须先定义宏，然后才能使用宏。可以在源文件的开头、.include/.copy 文件中或宏库中对宏进行定义。

macname	指定宏的名称。必须将名称放在源语句的标签字段中。
.macro	将源语句标识为宏定义的第一行。必须将 .macro 放在操作码字段中。
[*parameters*]	是作为 .macro 指令操作数出现的可选替代符号。
model statements	是每次调用宏时执行的指令或汇编器指令。
宏指令	用于控制宏扩展。
.endm	标记宏定义的结束。

如需进一步详细了解宏，请参阅章节 6。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.mlib

定义宏库

语法

.mlib " *filename* **"**

说明

.mlib 指令用于为汇编器提供宏库的 *文件名*。宏库是包含宏定义的文件集合。多个宏定义文件通过归档器合并至单个文件（被称为库或归档）。

宏库中的每个文件都包含一个与文件名对应的宏定义。宏库成员的 *文件名* 必须与宏名称相同，且其扩展名必须是 .asm。文件名必须遵循主机操作系统规则；它可以用双引号引起来。您可以指定完整路径名（例如，c:\320tools\macs.lib）。如果未指定完整路径名，汇编器将按照指定的顺序在以下位置搜索文件：

1. 包含当前源文件的目录
2. 使用 --include_path 汇编器选项指定的任何目录
3. 由 TI_ARM_A_DIR 环境变量指定的任何目录
4. 由 TI_ARM_C_DIR 环境变量指定的任何目录

有关 -include_path 选项的更多信息，请参阅节 4.5。

.mlib 指令用于指示汇编器打开指定的库并创建一个库内容表。汇编器会将各个库成员的名称作为库条目存储到操作码表中。这会重新定义任何具有相同名称的现有操作码或宏。如果这些宏之一被调用，汇编器会提取库条目并将其加载到宏表中。汇编器采用与扩展其他宏相同的方式扩展库条目，但它不会将相应源代码置于列表中。只会从库中提取宏，并且只提取一次。有关详细信息，请参阅章节 6。

示例

该代码会创建一个宏库，用于定义以下两个宏：inc4.asm 和 dec4.asm。inc4.asm 文件包含 inc4 的定义，而 dec4.asm 文件包含 dec4 的定义。

递增宏：inc4.asm

```
inc4 .macro reg1, reg2, reg3, reg4
    Add reg1, reg1, #1
    ADD reg2, reg2, #1
    ADD reg3, reg3, #1
    ADD reg4, reg4, #1
    .endm
```

递减宏：dec4.asm

```
dec4 .macro reg1, reg2, reg3, reg4
     SUB reg1, reg1, #1
     SUB reg2, reg2, #1
     SUB reg3, reg3, #1
     SUB reg4, reg4, #1
     .endm
```

可以通过如下所示的命令行使用归档器来创建宏库：

```
armar -a mac inc4.asm dec4.asm
ar32 -a mac inc4.asm dec4.asm
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.mlib (continued)

定义宏库

使用 .mlib 来引用宏库。 定义 inc4.asm 和 dec4.asm 宏：

```
       1                         .mlib    "mac.lib"
       2                    ; Macro call
       3 00000000                inc4  R7, R6, R5, R4
1        00000000 E2877001          ADD     R7, R7, #1
1        00000004 E2866001          ADD     R6, R6, #1
1        00000008 E2855001          ADD     R5, R5, #1
1        0000000c E2844001          ADD     R4, R4, #1
       4
       5                    ; Macro call
       6 00000010                dec4  R0, R1, R2, R3
1        00000010 E2400001        SUB    R0, R0, #1
1        00000014 E2411001        SUB    R1, R1, #1
1        00000018 E2422001        SUB    R2, R2, #1
1        0000001c E2433001        SUB    R3, R3, #1
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.mlist/.mnolist

启动/停止宏扩展列表

语法

.mlist

.mnolist

说明

两个指令可用于控制在列表中列出宏和可重复块扩展：

.mlist 指令允许在列表文件中列出宏和 .loop/.endloop 块扩展。

.mnolist 指令抑制在列表文件中列出宏和 .loop/.endloop 块扩展。

默认情况下，汇编器的行为就像指定了 .mlist 指令一样。

有关宏和宏库的更多信息，请参阅章节 6。请参阅 .loop/.break/.endloop 主题，了解有关条件块的信息。

示例

以下示例将定义一个名为 STR_3 的宏。首次调用该宏时，会列出对应的宏扩展（默认）。第二次调用该宏时，不会列出对应的宏扩展，因为这时已汇编了 .mnolist 指令。第三次调用该宏时，会再次列出对应的宏扩展，因为这时已汇编了 .mlist 指令。

```
        1                      STR_3  .macro   P1, P2, P3
        2                             .string ":p1:", ":p2:", ":p3:"
        3                             .endm
        4
        5 00000000                    STR_3 "as", "I", "am"    ; Invoke STR_3 macro.
1         00000000 3A                 .string ":p1:", ":p2:", ":p3:"
          00000001 70
          00000002 31
          00000003 3A
          00000004 3A
          00000005 70
          00000006 32
          00000007 3A
          00000008 3A
          00000009 70
          0000000a 33
          0000000b 3A
        6                             .mnolist                 ; Suppress expansion.
        7 0000000c                    STR_3 "as", "I", "am"    ; Invoke STR_3 macro.
        8                             .mlist                   ; Show macro
expansion.
        9 00000018                    STR_3 "as", "I", "am"    ; Invoke STR_3 macro.
1         00000018 3A                 .string ":p1:", ":p2:", ":p3:"
          00000019 70
          0000001a 31
          0000001b 3A
          0000001c 3A
          0000001d 70
          0000001e 32
          0000001f 3A
          00000020 3A
          00000021 70
          00000022 33
          00000023 3A
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.newblock

终止局部符号块

语法 **.newblock**

说明 **.newblock** 指令用于取消定义任何当前定义的局部标签。本质上，局部标签是临时标签；.newblock 指令用于对它们进行重置并终止其范围。

局部标签可以是 $*n* 形式，其中 *n* 可以是一个十进制数字，也可以是 *name*? 形式，而 *name* 是合法符号名称。与其他标签不同，局部标签供局部使用，不能用在表达式中。它们只能用作 8 位跳转指令中的操作数。局部标签不会出现在符号表中。

在定义和（或许）使用局部标签后，您应该使用 .newblock 指令来重置局部标签。.text、.data 和 .sect 指令也会重置局部标签。包含文件中定义的局部标签在包含文件之外无效。

有关局部标签用法的更多信息，请参阅节 4.8.3。

示例 下面的示例展示了如何声明、重置和再重新声明局部标签 $1。

```
 1 00000000 E3510000  LABEL1: CMP    r1, #0
 2 00000004 2A000001          BCS    $1
 3 00000008 E2900001          ADDS   r0, r0, #1
 4 0000000c 21A0F00E          MOVCS  pc, lr
 5 00000010 E4952004  $1:     LDR    r2, [r5], #4
 6                            .newblock    ; Undefine $1 to use again.
 7 00000014 E0911002          ADDS   r1, r1, r2
 8 00000018 5A000000          BPL    $1
 9 0000001c E1F01001          MVNS   r1, r1
10 00000020 E1A0F00E  $1:     MOV    pc, lr
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.option

选择列表选项

语法

.option *option*$_1$[, *option*$_2$,...]

说明

.option 指令用于为汇编器输出列表选择选项。*选项*必须用逗号分隔；每个选项选择一个列表特性。选项*不*区分大小写。有效选项包括：

A	开启所有指令、数据、后续扩展、宏和代码块的列表。
B	将 .byte 和 .char 指令的列表限制为一行。
H	将 .half 和 .short 指令的列表限制为一行。
L	将 .long 指令的列表限制为一行。
M	关闭列表中的宏扩展。
N	关闭列表（执行 .nolist）。
O	开启列表（执行 .list）。
R	重置任何 B、H、M、T 和 W（关闭 B、H、M、T 和 W 的限制）。
T	将 .string 指令的列表限制为一行。
W	将 .word 和 .int 指令的列表限制为一行。
X	生成交叉引用符号列表。您还可以通过 --asm_cross_reference_listing 选项调用汇编器来获取此列表（请参阅节 4.14）。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.option (continued)

选择列表选项

示例

该示例将 .byte、.char、.int、.long、.word 和 .string 列表限制为每个一行。

```
       1
*****************************************************
       2                    ** Limit the listing of .byte, .char, .int, .long,
**
       3                    **  .word, and .string directives to 1 line each.
**
       4
*****************************************************
       5                            .option B, W, T
       6 00000000 BD                .byte   -'C', 0B0h, 5
       7 00000003 BC                .char   -'D', 0C0h, 6
       8 00000008 0000000A          .int    10, 35 + 'a', "abc"
       9 0000001c AABBCCDD          .long   0AABBCCDDh, 536 + 'A'
      10 00000024 000015AA          .word   5546, 78h
      11 0000002c 45                .string "Extended Registers"
      12
      13                    ****************************************************
      14                    **            Reset the listing options.          **
      15                    ****************************************************
      16                            .option R
      17 0000003e BD                .byte   -'C', 0B0h, 5
         0000003f B0
         00000040 05
      18 00000041 BC                .char   -'D', 0C0h, 6
         00000042 C0
         00000043 06
      19 00000044 0000000A          .int    10, 35 + 'a', "abc"
         00000048 00000084
         0000004c 00000061
         00000050 00000062
         00000054 00000063
      20 00000058 AABBCCDD          .long   0AABBCCDDh, 536 + 'A'
         0000005c 00000259
      21 00000060 000015AA          .word   5546, 78h
         00000064 00000078
      22 00000068 45                .string "Extended Registers"
         00000069 78
         0000006a 74
         0000006b 65
         0000006c 6E
         0000006d 64
         0000006e 65
         0000006f 64
         00000070 20
         00000071 52
         00000072 65
         00000073 67
         00000074 69
         00000075 73
         00000076 74
         00000077 65
         00000078 72
         00000079 73
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.page

在列表中弹出页

语法 **.page**

说明 **.page** 指令用于在列表文件中弹出页。.page 指令不会列印在源列表中，但汇编器在遇到 .page 指令时会使行数计数器递增。使用 .page 指令将源列表拆分为逻辑切分可以提高程序可读性。

示例 此示例展示了 .page 指令如何使汇编器开始源列表的新一页。

源文件：

```
Source file (generic)
        .title    "**** Page Directive Example ****"
;         .
;         .
;         .
        .page
```

列表文件：

```
TMS470R1x Assembler    Version x.xx        Day    Time    Year
Copyright (c) 1996-2011 Texas Instruments Incorporated
**** Page Directive Example ****                                    PAGE    1
       2                    ;        .
       3                    ;        .
       4                    ;        .
TMS470R1x Assembler    Version x.xx        Day    Time    Year
Copyright (c) 1996-2011 Texas Instruments Incorporated
**** Page Directive Example ****                                    PAGE    2
 No Errors, No Warnings
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.retain/.retainrefs

有条件地将段保留在目标模块输出内

语法

.retain[" *section name* "]

.retainrefs[" *section name* "]

说明

.retain 指令用于指示当前段或指定的段不符合通过条件链接进行移除的条件。用户也可以使用 --retain 链接器选项为指定的段覆盖条件链接。用户也可以使用 --unused_section_elimination=off 链接器选项完全禁用条件链接。

.retainrefs 指令指示引用当前段或指定段的任何段均不符合通过条件链接进行移除的条件。例如，应用程序可能使用 .intvecs 段来设置中断矢量。默认情况下，.intvecs 段符合在条件链接期间进行移除的条件。用户可以通过对 .intvecs 段应用 .retain 和 .retainrefs 指令，强制保留 .intvecs 段以及任何引用它的段。

*段名*用于标识段。如果使用该指令时未搭配段名，则它适用于当前已初始化的段。如果指令应用于未初始化的段，则需要段名。段名必须用双引号引起来。段名可包含 *段名*：*子段名* 形式的子段名。

链接器假定默认情况下所有段都不符合通过条件链接进行移除的条件。（不过，链接器会自动保留 .reset 段。）如果用户希望将某段保留在链接中（即使链接中的任何其他段均未引用该段），.retain 指令对于覆盖段的这种默认条件链接行为非常有用。例如，用户可以对以汇编语言编写但在应用程序中任何正常入口点均未引用的中断函数应用 .retain 指令。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.sect

汇编到命名段

语法

.sect " *section name* **"**

.sect " *section name* **"** [,{RO|RW}] [,{ALLOC|NOALLOC}]

说明

.sect 指令用于定义命名段，它可作为默认的 .text 段和 .data 段使用。.sect 指令用于将 *段名* 设置为当前段；后面的行将汇编到 *段名* 段。

段名 用于标识段，段名必须用双引号引起来。段名可以包含 *段名*:*子段名* 形式的子段名。有关段的更多信息，请参阅章节 2。

该段可标记为只读 (RO) 或读写 (RW)。另外，这些段可标记为分配 (ALLOC) 或未分配 (NOALLOC)。这些属性可按任意顺序指定，但每组中只能选择一个属性。RO 与 RW 冲突，ALLOC 与 NOALLOC 冲突。如果指定了冲突属性，汇编器会生成一个错误，例如：

```
"t.asm", ERROR!  at line 1:[E0000] Attribute RO cannot be combined with attr RW
          .sect "illegal_sect",RO,RW
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.sect (continued)

汇编到命名段

示例

本示例定义了两个具有特殊作用的段，Sym_Defs 和 Vars，并将代码汇编到其中。

```
       1
******************************************************
       2                    **       Begin assembling into .text section.
**
       3
******************************************************
       4 00000000                  .text
       5 00000000 E3A00078         MOV     R0, #78h
       6 00000004 E2801078         ADD     R1, R0, #78h
       7
******************************************************
       8                    **      Begin assembling into Sym_Defs section.
**
       9
******************************************************
      10 00000000                  .sect   "Sym_Defs"
      11 00000000 3D4CCCCD         .float  0.05          ; Assembled into
Sym_Defs
      12 00000004 000000AA  X:     .word   0AAh          ; Assembled into
Sym_Defs
      13 00000008 E2833028         ADD     R3, R3, #28h  ; Assembled into
Sym_Defs
      14
******************************************************
      15                    **      Begin assembling into Vars section.
**
      16
******************************************************
      17 00000000                  .sect   "Vars"
      18          00000010  WORD_LEN       .set    16
      19          00000020  DWORD_LEN      .set    WORD_LEN * 2
      20          00000008  BYTE_LEN       .set    WORD_LEN / 2
      21
******************************************************
      22                    **       Resume assembling into .text section.
**
      23
******************************************************
      24 00000008                  .text
      25 00000008 E2802042         ADD     R2, R0, #42h ; Assembled into .text
      26 0000000c 03               .byte   3, 4         ; Assembled into .text
         0000000d 04
      27
******************************************************
      28                    **       Resume assembling into Vars section.
**
      29
******************************************************
      30 00000000                  .sect   "Vars"
      31 00000000 000D0000         .field  13, WORD_LEN
      32 00000000 000D0A00         .field  0Ah, BYTE_LEN
      33 00000004 00000008         .field  10q, DWORD_LEN
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.set/.equ

定义汇编时常量

语法

symbol **.set** *value*

symbol **.equ** *value*

说明

.set 和 **.equ** 指令将一个常量值等同于 .set/.equ 符号。之后，可以使用该符号代替汇编源代码中的值。这样可将常量和其他值等同于有意义的名称。.set 和 .equ 指令是相同的，可以互换使用。

- *符号* 是必须显示在标签字段中的标签。
- *值* 必须是一个定义明确的表达式，即表达式中的所有符号都必须先在当前源代码模块中定义。

表达式中不能使用未定义的外部符号和之后会在模块中定义的符号。如果表达式可重定位，所指定的符号也是可重定位的。

表达式的值出现在列表的对象字段中。此值不属于实际目标代码，不会写入输出文件。

利用由 .set 或 .equ 定义的符号可通过 .def 或 .global 指令使之变得外部可见（请参阅 .global/.def/.ref 主题）。您可以通过这种方式定义全局绝对常量。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.set/.equ (continued)

定义汇编时常量

示例

此示例展示了如何使用 .set 和 .equ 指定符号。

```
       1
*****************************************************
       2                    **   Equate symbol AUX_R1 to register AR1 and use
**
       3                    **          it instead of the register.
**
       4                    **********************************************
       5          00000001  AUX_R1  .set    R1
       6 00000000 E3A01056          MOV     AUX_R1, #56h
       7
       8
*****************************************************
       9                    **    Set symbol index to an integer expression.
**
      10                    **        and use it as an immediate operand.
**
      11
*****************************************************
      12          00000035  INDEX   .equ    100/2 +3
      13 00000004 E2810035          ADD     R0, AUX_R1, #INDEX
      14
      15
*****************************************************
      16                    **   Set symbol SYMTAB to a relocatable expression.**
      17                    **       and use it as a relocatable operand.
**
      18
*****************************************************
      19 00000008 0000000A  LABEL   .word   10
      20          00000009' SYMTAB  .set    LABEL + 1
      21
      22
*****************************************************
      23                    **    Set symbol NSYMS equal to the symbol INDEX
**
      24                    **       INDEX and use it as you would INDEX.
**
      25
*****************************************************
      26          00000035  NSYMS   .set    INDEX
      27 0000000c 00000035          .word   NSYMS
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.space/.bes

保留空间

语法

[*label*] **.space** *size in bytes*

[*label*] **.bes** *size in bytes*

说明

.space 和 **.bes** 指令在当前段中保留的字节数源于 *size in bytes*，指令会将其填充为 0。段程序计数器会递增，指向保留空间后的字。

如果将 .space 指令与标签一同使用，它会指向保留的*第一个*字节。如果将 .bes 指令与标签一同使用，它会指向保留的*最后一个*字节。

示例

本示例展示了如何使用 .space 和 .bes 指令来保留存储器。

```
         1                    ***************************************************
         2                    **   开始汇编到 .text 段中。   **
         3                    ***************************************************
         4 00000000                   .text
         5
         6                    ***************************************************
         7                    **    在 .text 段保留 0F0 字节。    **
         8                    ***************************************************
         9 00000000                   .space  0F0h
        10 000000f0 00000100          .word   100h, 200h
           000000f4 00000200
        11                    ***************************************************
        12                    **   开始汇编到 .data 段中。   **
        13                    ***************************************************
        14 00000000                   .data
        15 00000000 49                .string "In .data"
           00000001 6E
           00000002 20
           00000003 2E
           00000004 64
           00000005 61
           00000006 74
           00000007 61
        16                    ****************************************************
        17                    ** 在 .data 段保留 100 个字节; RES_1 **
        18                    **    指向      **
        19                    **             包含保留字节的第一个字
节。                **
        20                    ****************************************************
        21 00000008           RES_1:  .space  100
        22 0000006c 0000000F          .word   15
        23 00000070 00000008"         .word   RES_1
        24
        25                    ****************************************************
        26                    **   在 .data 段保留 20 个位;  RES_2   **
        27                    **      指向      **
        28                    **             包含保留字节的最后一个字
节。                **
        29                    ***************************************************
        30 00000087           RES_2:  .bes    20
        31 00000088 00000036          .word   36h
        32 0000008c 00000087"         .word   RES_2
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.sslist/.ssnolist

替代符号控制列表

语法

.sslist

.ssnolist

说明

两条指令用于控制列表文件中的替代符号扩展：

.sslist 指令用于实现列表文件中的替代符号扩展。扩展代码行显示在实际源代码行之下。

.ssnolist 指令用于抑制列表文件中的替代符号扩展。

默认情况下会抑制列表文件中的所有替代符号扩展；汇编器的行为与使用 .ssnolist 指令的情况相同。

具有井号 (#) 字符的行表示扩展的替代符号。

示例

此示例展示了默认情况下抑制替代符号扩展列表的代码，它显示已汇编的 .sslist 指令，指示汇编器列出替代符号代码扩展。

```
       1                   ADDL     .macro  dest, src
       2                            .global reset_ctr
       3                            ADDS    dest, dest, src
       4                            BLCS    reset_ctr
       5                            .endm
       6
       7 00000000                   ADDL    R4, R5
1                                   .global reset_ctr
1          00000000 E0944005        ADDS    R4, R4, R5
1          00000004 2BFFFFFD!       BLCS    reset_ctr
       8 00000008 E5954000          LDR     R4, [R5]
       9 0000000c                   ADDL    R0, R4
1                                   .global reset_ctr
1          0000000c E0900004        ADDS    R0, R0, R4
1          00000010 2BFFFFFA!       BLCS    reset_ctr
      10
      11                            .sslist
      12
      13 00000014 E5B53004           LDR    R3, [R5, #4]!
      14 00000018 E5954000           LDR    R4, [R5]
      15 0000001c                    ADDL   R4, R3
1                                   .global reset_ctr
1          0000001c E0944003        ADDS    dest, dest, src
#                                   ADDS    R4, R4, R3
1          00000020 2BFFFFF6!       BLCS    reset_ctr
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.state16

汇编 16 位指令（非 UAL 语法）

语法 **.state16**

说明 默认情况下，汇编器开始将文件中的所有指令汇编为 32 位指令。使用 **.state16** 指令指示汇编器从 16 位指令的位置开始汇编所有指令。该指令和 .state32 指令可实现在非 UAL 语法的两种汇编模式下进行切换。如果需要将整个文件汇编为 16 位指令以用于 V6 和更早的架构，可使用 −mt 汇编器选项，指示汇编器开始汇编过程，将所有指令汇编为 16 位指令。

.state16 指令会在将任何指令写入相应的段之前执行隐式半字对齐，以确保所有 16 位指令都是半字对齐的。.state16 指令还会重置已定义的所有局部标签。

示例 在本示例中，汇编器汇编 16 位指令，开始汇编 32 位指令，并返回以汇编 16 位指令。

```
 1                                .global glob1, glob2
 2                        ****************************************************
 3                        **      开始汇编 16 位指令。     **
 4                        ****************************************************
 5 00000000                       .state16
 6
 7 00000000 4808                  LDR     r0, glob1_a
 8 00000002 4909                  LDR     r1, glob2_a
 9 00000004 6800                  LDR     r0, [r0]
10 00000006 6809                  LDR     r1, [r1]
11 00000008 0080                  LSL     r0, r0, #2
12 0000000a 3156                  ADD     r1, #56h
13 0000000c 4778                  BX      pc
14 0000000e 46C0                  NOP
15                        ****************************************************
16                        **    切换到 32 位指令以使用     **
17                        **      32 位状态长乘法指令。     **
18                        ****************************************************
19 00000010                       .state32
20
21 00000010 E0845190              UMULL   r5, r4, r0, r1
22 00000014 E28FE001              ADD     lr, pc, #1
23 00000018 E12FFF1E              BX      lr
24                        ****************************************************
25                        **    继续汇编 16 位指令。     **
26                        ****************************************************
27 0000001c                       .state16
28
29 0000001c 1A2D                   SUB    r5, r5, r0
30 0000001e D200                   BCS    $1
31 00000020 3C01                   SUB    r4, #1
32 00000022               $1
33 00000024 00000000! glob1_a .word   glob1
34 00000028 00000000! glob2_a .word   glob2
```

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.state32/.arm

汇编 32 位指令

语法

.state32

.arm

说明

默认情况下，汇编器开始将文件中的所有指令汇编为 32 位指令。在使用 -mt 汇编器选项或 .state16 指令汇编 16 位指令时，可以使用 **.state32** 或 **.arm** 指令指示汇编器开始将 .state32/.arm 指令之后的所有指令汇编为 32 位指令。

在编写汇编代码时，.arg 指令用于指定 ARM UAL 语法。.state32 指令和 .arm 指令是等效的，因为 UAL 语法是向后兼容的。

这些指令会在将任何指令写入相应的段之前执行隐式字对齐，以确保所有 32 位指令都是字对齐的。这些指令还会重置已定义的所有局部标签。

示例

在本示例中，汇编器汇编 32 位指令，开始汇编 16 位指令，并返回以汇编 32 位指令。

```
 1                              .global globs, filter
 2                      ****************************************************
 3                      **      开始汇编 32 位指令。     **
 4                      ****************************************************
 5 00000000                     .state32
 6 00000000 E28F4001            ADD     r4, pc, #1
 7 00000004 E12FFF14            BX      r4
 8                      ****************************************************
 9                      **      切换到 16 位指令以占用      **
10                      **                更少的代码空间。               **
11                      ****************************************************
12 00000008                     .state16
13 00000008 2200                MOV     r2, #0
14 0000000a 2300                MOV     r3, #0
15 0000000c 4C0B                LDR     r4, globs_a
16 0000000e 2500                MOV     r5, #0
17 00000010 2600                MOV     r6, #0
18 00000012 2700                MOV     r7, #0
19 00000014 4690                MOV     r8, r2
20 00000016 4691                MOV     r9, r2
21 00000018 4692                MOV     r10, r2
22 0000001a 4693                MOV     r11, r2
23 0000001c 4694                MOV     r12, r2
24 0000001e 4695                MOV     r13, r2
25 00000020 4778                BX      pc
26 00000022 46C0                NOP
27                      ****************************************************
28                      **    继续汇编 32 位指令。    **
29                      ****************************************************
30 00000024                     .state32
31 00000024 E4940004            LDR     r0, [r4], #4
32 00000028 E5941000            LDR     r1, [r4]
33 0000002c EBFFFFF3!           BL      filter
34 00000030 E1500001            CMP     r0, r1
35 00000034 30804005            ADDCC   r4, r0, r5
36 00000038 20464001            SUBCS   r4, r6, r1
37 0000003c 00000000!           globs_a .word globs
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.string/.cstring

初始化文本

语法

.string {$expr_1$ **|** **"** $string_1$ **"**} [**,** ..., {$expr_n$ **|** **"** $string_n$ **"**}]

.cstring {$expr_1$ **|** **"** $string_1$ **"**} [**,** ..., {$expr_n$ **|** **"** $string_n$ **"**}]

说明

.string 和 **.cstring** 指令用于将一个字符串中的 8 位字符放入当前段中。*expr* 或 *string* 可以是：

- 汇编器计算并视为 8 位有符号数的表达式。
- 用双引号引起来的字符串。字符串中的每个字符代表一个单独的值，并且值存储在连续字节中。整个字符串*必须*用双引号引起来。

.cstring 指令用于添加 C 所需的 NUL 字符；.string 指令不会添加 NUL 字符。此外，.cstring 会解释 C 转义符 (\\ \a \b \f \n \r \t \v \<octal>)。

汇编器会截断大于八位的所有值。操作数必须纳入单一源语句行中。

如果使用标签，它会指向已初始化的第一个字节的位置。

如果在 .struct/.endstruct 序列中使用 .string 和 .cstring，该指令仅定义成员的大小；但不会初始化存储器。如需更多信息，请参阅 .struct/.endstruct/.tag 主题。

示例

在本例中，8 位值置于当前段的连续字节中。

```
       1 00000000 41        Str_Ptr:   .string  "ABCD"
         00000001 42
         00000002 43
         00000003 44
       2 00000004 41                   .string  41h, 42h, 43h, 44h
         00000005 42
         00000006 43
         00000007 44
       3 00000008 41                   .string "Austin", "Houston", "Dallas"
         00000009 75
         0000000a 73
         0000000b 74
         0000000c 69
         0000000d 6E
         0000000e 48
         0000000f 6F
         00000010 75
         00000011 73
         00000012 74
         00000013 6F
         00000014 6E
         00000015 44
         00000016 61
         00000017 6C
         00000018 6C
         00000019 61
         0000001a 73
       4 0000001b 30                   .string  36 + 12
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.struct/.endstruct/.tag

声明结构体类型

语法

[*stag*] **.struct** [*expr*]

[mem_0] *element* [$expr_0$]
[mem_1] *element* [$expr_1$]
...

[mem_n] **.tag** *stag* [$expr_n$]
...

[mem_N] *element* [$expr_N$]

[*size*] **.endstruct**

label **.tag** *stag*

说明

.struct 指令用于为数据结构定义的元素分配符号偏移量。这样便可将类似的数据元素分为一组，并由汇编器计算元素偏移量。这与 C 结构或 Pascal 记录相似。.struct 指令不会分配存储器空间；仅创建一个可重复使用的符号模板。

.endstruct 指令用于终止结构定义。

.tag 指令用于为 *label* 提供结构特征，简化符号表示，还可定义包含其他结构体的结构。.tag 指令不会分配存储器空间，必须事先定义 .tag 指令的结构标签（*stag*）。

与 .struct、.endstruct 和 .tag 指令一起使用的参数包括：

- *stag* 是结构体的标签，其值与结构体的开头相关联。如果不存在 stag，汇编器会将结构体成员置于全局符号表中，其值是它们距结构体顶部的绝对偏移量。stag 对于 .struct 而言是可选项，但对于 .tag 而言是必备项。
- *expr* 是一个可选表达式，指示结构体的开始偏移量。结构体的默认起始点为 0。
- $mem_{n/N}$ 是结构体成员的可选标签。该标签是绝对值，相当于距结构体开始处的当前偏移量。结构体成员的标签不能声明为全局标签。
- *element*（元素）是以下描述符之一：.byte、.char、.int、.long、.word、.double、.half、.short、.string、.float、.field 和 .tag。除了 .tag 之外的所有上述描述符都是用于初始化存储器的典型指令。在 .struct 指令之后，这些指令描述结构体元素的大小，不分配存储器空间。.tag 指令是一个特例，因为必须使用 stag（如在 stag 的定义中）。
- $expr_{n/N}$ 是表示所描述元素数量的可选表达式，该值默认为 1。.string 元素大小是一个字节，.field 元素大小是一位。
- *size* 是表示结构体总大小的可选标签。

备注

可以出现在 .struct/.endstruct 序列中的指令：在 .struct/.endstruct 序列中只能出现元素描述符、条件汇编指令以及 .align 指令（在字边界对齐成员偏移量）这些指令。空结构体是非法的。

以下示例展示了 .struct、.tag 和 .endstruct 指令的各种用途。

示例 1

```
1                    REAL_REC    .struct              ; stag
2          00000000  NOM         .int                 ; member1 = 0
3          00000004  DEN         .int                 ; member2 = 1
4          00000008  REAL_LEN    .endstruct           ; real_len = 4
5
6 00000000 E59F0004              LDR R0, REAL_A
7 00000004 E5904004              LDR R4, [R0, #REAL_REC.DEN]
8 00000008 E0811004              ADD R1, R1, R4
```


English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY
TI Confidential – DP Restrictions

.struct/.endstruct/.tag (continued)

声明结构体类型

```
       9 00000000                      .bss REAL, REAL_LEN      ; allocate mem
rec
      10 0000000c 00000000- REAL_A     .word REAL
      11
```

示例 2

```
      12                    CPLX_REC   .struct
      13           00000000 REALI      .tag REAL_REC            ; stag
      14           00000008 IMAGI      .tag REAL_REC            ; member1 = 0
      15           00000010 CPLX_LEN   .endstruct               ; cplx_len = 8
      16
      17                    COMPLEX    .tag CPLX_REC            ; assign
structure
      18                                                        ; attribute
      19 00000010           COMPLEX    .space CPLX_LEN          ; allocate space
      20 00000020 E51F4018             LDR R4, COMPLEX.REALI    ; access
structure
      21 00000024 E0811004             ADD R1, R1, R4
```

示例 3

```
       1                               .struct            ; no stag puts mems
into
       2                                                  ; global symbol table
       3           00000000 X          .int               ; create 3 dim
templates
       4           00000004 Y          .int
       5           00000008 Z          .int
       6           0000000C            .endstruct
```

示例 4

```
       1                    BIT_REC    .struct                  ; stag
       2           00000000 STREAM     .string 64
       3           00000040 BIT7       .field  7                ; bit7 = 64
       4           00000040 BIT8       .field  9                ; bit9 = 64
       5           00000042 BIT10      .field  10               ; bit10 = 64
       6           00000044 X_INT      .int                     ; x_int = 68
       7           00000048 BIT_LEN    .endstruct               ; length = 72
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.symdepend

创建从段到符号的人工引用

语法

.symdepend *dst symbol name*[**,** *src symbol name*]

说明

.symdepend 指令会创建一个从定义 *src symbol name* 的段到 *dst symbol name* 符号的人工引用。如果定义 *src symbol name* 的段包含在输出模块中，这样可防止链接器删除包含 *dst symbol name* 的段。如果未指定 *src symbol name*，将创建从当前段的引用。

全局*符号* 的定义方式与任何其他符号相同；即，它显示为标签或由 .set、.equ、.bss 或 .usect 指令定义。如果多次定义了某个全局符号，链接器会发出“多重定义”错误。（汇编器可以为局部符号提供类似的“多重定义”错误。）

.symdepend 指令只在模块实际使用某符号时才创建符号表条目，而 .weak 指令始终为一个符号创建符号表条目，无论模块是否使用该符号（请参阅 .weak 主题）。

如果该符号*在当前模块中定义*，则使用 .symdepend 指令声明该符号及其定义可由其他模块在外部使用。这些类型的引用在链接时解析。

DRAFT ONLY
TI Confidential – NDA Restrictions

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

.tab

定义制表符大小

语法

.tab *size*

说明

.tab 指令用于定义制表符大小。源代码输入中的制表符会转换为列表中的 *size* 字符空间。默认制表符大小为八个空间。

示例

在本例中，.tab 语句后的每一行代码均包含单一制表符字符，后跟一条 NOP 指令。

源文件：

```
; default tab size
    NOP
    NOP
    NOP
      .tab 4
    NOP
    NOP
    NOP
      .tab 16
    NOP
    NOP
    NOP
```

列表文件：

```
       1                   ; default tab size
       2 00000000 E1A00000         NOP
       3 00000004 E1A00000         NOP
       4 00000008 E1A00000         NOP
       5
       7 0000000c E1A00000     NOP
       8 00000010 E1A00000     NOP
       9 00000014 E1A00000     NOP
      10
      12 00000018 E1A00000                 NOP
      13 0000001c E1A00000                 NOP
      14 00000020 E1A00000                 NOP
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.text

汇编到 .text 段

语法

.text

说明

.text 将 .text 设为当前段。将此指令之后的代码行汇编至 .text 段，其中通常包含可执行代码。如果尚未将任何内容汇编到 .text 段，段程序计数器将设为 0。如果已将代码汇编到 .text 段，段程序计数器将恢复为段中之前的值。

.text 段是默认段。因此，在汇编开始时，汇编器会将代码汇编到 .text 段，除非您使用 .data 或 .sect 指令来指定其他段。

有关段的更多信息，请参阅章节 2。

示例

本示例将代码汇编到 .text 和 .data 段。

```
      1                        ******************************************
      2                        ** 开始汇编到 .data 段。**
      3                        ******************************************
      4 00000000                       .data
      5 00000000 0A                    .byte    0Ah, 0Bh
        00000001 0B
      6                        ******************************************
      7                        ** 开始汇编到 .text 段。**
      8                        ******************************************
      9 00000000                       .text
     10 00000000 41            START:  .string "A","B","C"
        00000001 42
        00000002 43
     11 00000003 58            END:    .string "X","Y","Z"
        00000004 59
        00000005 5A
     12 00000008 E3A01003              MOV     R1, #END-START
     13 0000000c E1A01181              MOV     R1, R1, LSL #3
     14
     15                        ******************************************
     16                        ** 继续汇编到 .data 段。**
     17                        ******************************************
     18 00000002                       .data
     19 00000002 0C                    .byte    0Ch, 0Dh
        00000003 0D
     20                        ******************************************
     21                        ** 继续汇编到 .text 段。**
     22                        ******************************************
     23 00000010                       .text
     24 00000010 51                    .string "QUIT"
        00000011 55
        00000012 49
        00000013 54
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.thumb

汇编 Thumb 或 Thumb-2 指令（UAL 语法）

语法 .thumb

说明 可以使用 **.thumb** 指令指示汇编器使用 Thumb（32 位）或 Thumb-2（16 位或 32 位）UAL 语法开始汇编 .thumb 指令之后的所有指令。汇编器会根据代码的语法结构确定指令是 16 位还是 32 位。

.thumb 指令会在将任何指令写入段之前执行隐式半字对齐，以确保所有 Thumb/Thumb-2 指令都是半字对齐的。这些指令还会重置已定义的所有局部标签。

示例 在本示例中，汇编器汇编 16 位指令，开始汇编 32 位指令，并返回以汇编 16 位指令。

```
       1                            .global glob1, glob2
       2                    ****************************************************
       3                    **      开始汇编 Thumb 指令。      **
       4                    ****************************************************
       5 00000000                   .thumb
       6
       7 00000000 4808              LDR     r0, glob1_a
       8 00000002 4909              LDR     r1, glob2_a
       9 00000004 6800              LDR     r0, [r0]
      10 00000006 6809              LDR     r1, [r1]
      11 00000008 0080              LSLS    r0, r0, #2
      12 0000000a 3156              ADDS    r1, #56h
      13 0000000c 4778              BX      pc
      14 0000000e 46C0              NOP
      15                    ****************************************************
      16                    **      切换到 ARM 模式以使用        **
      17                    **            长乘法指令。              **
      18                    ****************************************************
      19 00000010                   .arm
      20
      21 00000010 E0845190          UMULL   r5, r4, r0, r1
      22 00000014 E28FE001          ADD     lr, pc, #1
      23 00000018 E12FFF1E          BX      lr
      24                    ****************************************************
      25                    **     继续汇编 Thumb 指令。     **
      26                    ****************************************************
      27 0000001c                   .thumb
      28
      29 0000001c 1A2D              SUBS    r5, r5, r0
      30 0000001e D201              BCS     $1
      31 00000020 3C01              SUBS    r4, #1
      32 00000024           $1
      33 00000024 00000000! glob1_a .word glob1
      34 00000028 00000000! glob2_a .word glob2
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.title

定义页标题

语法 **.title " *string* "**

说明 **.title** 指令提供了打印在每个列表页顶部的标题。源语句本身不会打印，但行计数器会递增。

string 是置于双引号中的标题，最多为 64 个字符。如果字符超过 64 个，汇编器会截断该字符串并发出警告：

```
*** WARNING! line x: W0001: String is too long - will be truncated
```

汇编器会将标题打印在该指令之后的页上，直到处理另一个 .title 指令。如果希望标题出现在第一页上，第一个源语句必须包含 .title 指令。

示例 在本例中，一个标题打印在第一页上，另一个标题打印在后续页上。

源文件：

```
       .title  "**** Fast Fourier Transforms ****"
;        .
;        .
;        .
       .title  "**** Floating-Point Routines ****"
       .page
```

列表文件：

```
TMS470R1x Assembler    Version x.xx        Day    Time    Year
Copyright (c) 1996-2011 Texas Instruments Incorporated
**** Fast Fourier Transforms ****                              PAGE    1
       2                  ;        .
       3                  ;        .
       4                  ;        .
TMS470R1x Assembler    Version x.xx        Day    Time    Year
Copyright (c) 1996-2011 Texas Instruments Incorporated
**** Floating-Point Routines ****                              PAGE    2
 No Errors, No Warnings
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.unasg/.undefine

关闭替代符号

语法

.unasg *symbol*

.undefine *symbol*

说明

.unasg 和 **.undefine** 指令用于删除通过 .asg 或 .define 创建的替代符号定义。替代符号表中从 .undefine 或 .unasg 位置到汇编文件结尾之间的指定*符号*将被删除。有关替代符号的更多信息，请参阅节 4.8.8。

这些指令可用于从汇编环境中删除可导致问题的所有 C/C++ 宏。有关在汇编源代码中使用 C/C++ 头文件的更多信息，请参阅章节 13。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.union/.endunion/.tag

声明联合体类型

语法

[*utag*] **.union** [*expr*]

[mem_0] *element* [$expr_0$]
[mem_1] *element* [$expr_1$]
...

[mem_n] **.tag** *utag* [$expr_n$]
...

[mem_N] *element* [$expr_N$]

[*size*] **.endunion**

label **.tag** *utag*

说明

.union 指令用于为（将在同一存储器空间中分配的）备用数据结构体定义的元素分配符号偏移量。这样可使用户定义多个备用结构体，然后由汇编器计算元素偏移量。它与 C 联合体类似。.union 指令不会分配任何存储器空间；仅创建一个可重复使用的符号模板。

.struct 定义可包含一个 .union 定义，且.structs 和 .unions 可嵌套。

.endunion 指令用于终止联合体定义。

.tag 指令为 *label* 提供结构或联合体特征，以简化符号表示和定义包含其他结构或联合体的结构或联合体。.tag 指令不会分配存储器空间，必须事先定义 .tag 指令的结构体或联合体标签。

与 .struct、.endstruct 和 .tag 指令一起使用的参数包括：

- *utag* 是联合体的标签。其值与联合体的开头相关联。如果不存在 utag，汇编器会将联合体成员放在全局符号表中，其值是它们距联合体顶部的绝对偏移量。在这种情况下，每个成员必须有唯一的名称。
- *expr* 是一个可选表达式，指示联合体的开始偏移量。联合体默认始于 0。此参数只能与顶层联合体一同使用，不能用于定义嵌套联合体。
- $mem_{n/N}$ 是联合体成员的可选标签。该标签是绝对值，相当于距联合体开始处的当前偏移量。联合体成员的标签不能声明为全局标签。
- *element*（元素）是以下描述符之一：.byte、.char、.int、.long、.word、.double、.half、.short、.string、.float 和 .field。元素也可以是嵌套结构体或联合体，或由其标签声明的结构体或联合体的完整声明。在 .union 指令之后，这些指令描述元素的大小，不分配存储器空间。
- $expr_{n/N}$ 是表示所描述元素数量的可选表达式，该值默认为 1。.string 元素大小是一个字节，.field 元素大小是一位。
- *size* 是表示联合体总大小的可选标签。

备注

可以出现在 .union/.endunion 序列中的指令：.union/.endunion 序列中只能出现元素描述符、结构体和联合体标签以及条件汇编指令。空结构体是非法的。

这些示例展示了有标签和无标签的联合体。

示例 1

```
1                         .global employid
2           xample        .union                    ; utag
3      0000 ival          .word                     ; member1 = int
4      0000 fval          .float                    ; member2 = float
5      0000 sval          .string                   ; member3 = string
6      0002 real_len      .endunion                 ; real_len = 2
```

DRAFT ONLY TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.union/.endunion/.tag (continued)

声明联合体类型

```
      7
      8 000000                  .bss   employid, real_len  ;allocate memory
      9
     10              employid   .tag   xample              ; name an instance
     11 000000 0000-            ADD    employid.fval, A    ; access union element
```

示例 2

```
      1
      2                                               ; utag
      3        0000   x          .long                       ; member1 = long
      4        0000   y          .float                      ; member2 = float
      5        0000   z          .word                       ; member3 = word
      6        0002   size_u     .endunion                   ; real_len = 2
      7
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.usect

保留未初始化的空间

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

说明 **.usect** 指令用于在未初始化的命名段中为变量保留空间。该指令类似于 .bss 指令（参阅 .bss 主题）；两者都只是为数据保留空间，而该空间没有内容。但是，.usect 定义了可以放置在存储器中任何位置的其他段，这些段独立于 .bss 段。

- 该*符号* 指向此次调用 .usect 指令保留的第一个位置。符号对应于要保留空间的变量的名称。
- *段名* 必须用双引号引起来。此参数为未初始化的段命名。段名可以包含 *段名*:*子段名* 形式的子段名。
- *size in bytes* 是一个表达式，定义在 *section name* 中保留的字节数。
- *alignment* 是一个可选参数，可确保分配给符号的空间出现在指定的边界上。该边界可设为 2 的任意次幂。
- *bank offset* 是一个可选参数，可确保分配给符号的空间出现在特定存储器组边界上。该值表示在将符号分配给该位置之前从指定的对齐偏移的字节数。

初始化段的指令（.text、.data 和 .sect）指示汇编器暂停汇编到当前段，并开始汇编到另一个段中。在当前段中遇到的 .usect 或 .bss 指令会直接汇编，然后在当前段中继续汇编。

可在存储器中连续放置的变量可在同一指定段中定义；方法是用同一指令名和后续符号（变量名）重复 .usect 指令。

有关各段的更多信息，请参阅章节 2。

示例 此示例使用 .usect 指令定义两个未初始化的命名段 var1 和 var2。符号 ptr 指向 var1 段中保留的第一个字节。符号 array 指向 var1 中保留的 100 个字节的块中的第一个字节，dflag 指向 var1 中 50 个字节的块中的第一个字节。符号 vec 指向 var2 段中保留的第一个字节。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.usect (continued)

保留未初始化的空间

图 5-8 展示了此示例如何在两个未初始化的段 var1 和 var2 中保留空间。

```
        1
******************************************************
        2                     **         汇编到 .text 段。         **
        3
******************************************************
        4 00000000                    .text
        5 00000000 E3A01003           MOV     R1, #03h
        6
        7
******************************************************
        8                     **       保留 var1 段中的 1 个字节。       **
        9
******************************************************
       10 00000000            ptr     .usect  "var1", 1
       11
       12
******************************************************
       13                     **      保留 var1 段中的 100 个字节。      **
       14
******************************************************
       15 00000001            array   .usect  "var1", 100
       16
       17 00000004 E281001F           ADD     R0, R1, #037 ; Still in .text
       18
       19
******************************************************
       20                     **      保留 var1 段中的 50 个字节。       **
       21
******************************************************
       22 00000065            dflag   .usect  "var1", 50
       23
       24 00000008 E2812064           ADD     R2, R1, #dflag - array ; Still
in .text
       25
       26
******************************************************
       27                     **      保留 var2 段中的 100 个字节。      **
       28
******************************************************
       29 00000000            vec     .usect  "var2", 100
       30
       31 0000000c E0824000           ADD     R4, R2, R0   ; Still in .text
       32
******************************************************
       33                     **      将 .usect 符号声明为一个外部符号。      **
       34
******************************************************
       35                             .global array
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.usect (continued)

保留未初始化的空间

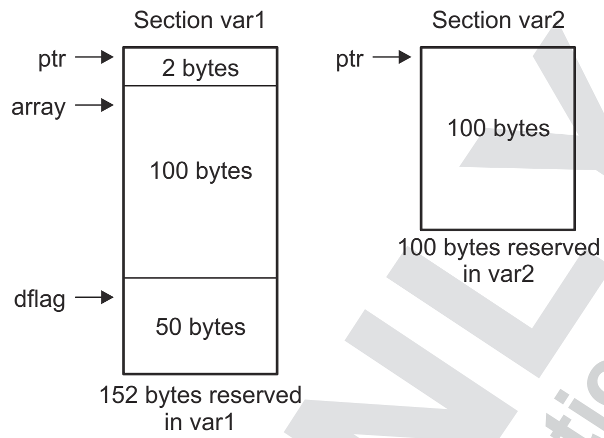


图 **5-8. .usect 指令**

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.var

使用替代符号作为局部变量

语法 **.var** *sym*$_1$ [, *sym*$_2$, ..., *sym*$_n$]

说明 **.var** 指令允许您使用替代符号作为宏中的局部变量。使用该指令，最多可为每个宏定义 32 个局部宏替代符号（包括参数）。

.var 指令用于创建临时的替代符号并将初始值设置为空字符串。这些符号不会作为参数传递，并会在扩展后丢失。

请参阅节 4.8.8 了解更多有关替代符号的信息。请参阅章节 6 了解更多有关宏的信息。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.weak

标识一个符号，将其作为弱符号处理

语法

.weak *symbol name*

说明

.weak 指令用于标识在当前模块中使用但在另一模块中定义的符号。链接器在链接时解析此符号的定义。如果某弱符号需要解析某以其他方式未解析的引用，链接器不会默认在输出文件的符号表中包含该弱符号（就像全局符号那样），而只会在“最终”链接的输出中包含该弱符号。有关链接器如何处理弱符号的详细信息，请参阅节 2.6.3。

.weak 指令等同于 .ref 指令，只是引用具有弱链接。

.weak 指令始终为一个符号创建符号表条目，无论模块是否使用该符号。而 .symdepend 指令只在模块实际使用某符号时才为其创建符号表条目（请参阅 .symdepend 主题）。

如果某符号*并未在当前模块中定义*（当前模块包括宏文件、副本文件和包含文件），则使用 .weak 指令指示汇编器：该符号在外部模块中定义。这可防止汇编器发出未解析的引用错误。链接时，链接器会在其他模块中查找该符号的定义。

例如，如以下示例所示组合使用 .weak 和 .set 指令来定义弱绝对符号“ext_addr_sym”：

```
                .weak   ext_addr_sym
ext_addr_sym    .set    0x12345678
```

如果汇编此汇编源代码，并在链接中包含生成的目标文件，本例中的“ext_addr_sym”可用作最终链接中的弱绝对符号。如果应用中的其他位置未引用该符号，它会成为候选的删除对象。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

This page intentionally left blank.

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

章节 6

宏语言说明

ARM device 汇编程序支持一种使您能够创建自有指令的宏语言。这在程序多次执行特定任务时特别有用。宏语言使您能够：

- 定义您自有的宏以及重新定义现有的宏
- 简化冗长或复杂的汇编代码
- 访问使用归档器创建的宏库
- 在宏中定义条件块和可重复块
- 在宏中操作字符串
- 控制扩展列表

6.1 使用宏.....154
6.2 定义宏.....154
6.3 宏参数/替代符号.....156
6.4 宏库.....160
6.5 在宏中使用条件汇编.....161
6.6 在宏中使用标签.....163
6.7 在宏中生成消息.....164
6.8 使用指令设置输出列表的格式.....165
6.9 使用递归和嵌套宏.....166
6.10 宏指令摘要.....167

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

6.1 使用宏

程序经常包含可执行多次的例程。您可以将一个例程定义为宏，然后在需要重复例程的位置调用该宏，而不必重复例程的源语句。这样可简化并缩短源程序。

如果您希望多次调用一个宏，但每次使用不同的数据，则可以在宏中分配参数。这样每次调用宏时都能够传递不同的信息。宏语言支持一种特殊的符号，被称为*替代符号*，用于宏参数。请参阅节 6.3，了解详情。

使用宏是一个包含 3 个步骤的过程。

1. **定义宏**。在程序中使用宏之前必须进行定义。可以通过两种方法来定义宏：
   a. 宏可以在*源文件* 的开头定义，或在复制/包含文件中定义。相关详细信息，请参阅节 6.2，*定义宏*。
   b. 宏也可以在*宏库* 中定义。宏库是一组归档格式的文件，由归档器创建。归档文件（宏库）中的每个成员都包含一个与成员名称对应的宏定义。您可以使用 .mlib 指令来访问宏库。如需更多信息，请参阅 节 6.4。
2. **调用宏**。定义宏之后，可在源程序中将宏名称作为助记符进行调用。这被称为*宏调用*。
3. **扩展宏**。源程序调用各个宏时，汇编器会将其扩展。在扩展期间，汇编器会通过变量将参数传递至宏参数，用宏定义替换宏调用语句，然后对源代码进行汇编。默认情况下，宏表达式在列表文件中列印。您可以使用 .mnolist 指令来关闭扩展列表。如需更多信息，请参阅 节 6.8。

当汇编器遇到宏定义时，会替换操作码表中的宏名称。这样会重新定义原先定义的所有与该宏同名的宏、库条目、指令或指令助记符。这样您可扩展指令的功能，并添加新指令。

6.2 定义宏

可在程序中的任何位置定义宏，但必须先定义宏，然后才能使用宏。宏可以在源文件的开头定义，或在 .copy/.include 文件（请参阅复制源文件）中定义；也可以在宏库中定义。有关宏库的更多信息，请参阅节 6.4。

宏定义可以被嵌套，也可以调用其他宏，但宏的所有元素必须在同一个文件中定义。节 6.9 中讨论了嵌套宏。

宏定义是一系列采用以下格式的源语句：

macname	**.macro** [*parameter*$_1$] [, ..., *parameter*$_n$]
	model statements or macro directives
	[**.mexit**]
	.endm

macname	指定宏的名称。必须将名称放在源语句的标签字段中。只有宏名称的前 128 个字符有意义。汇编器会将宏名称放在内部操作码表中，并会替换同名的任何指令或先前的宏定义。
.macro	是将源语句标识为宏定义第一行的指令。必须将 .macro 放在操作码字段中。
parameter $_1$, *parameter* $_n$	是作为 .macro 指令操作数出现的可选替代符号。节 6.3 中讨论了参数。
model statements	是每次调用宏时执行的指令或汇编器指令。
macro directives	用于控制宏扩展。
.mexit	是一个用作 *goto .endm* 的指令。当错误测试确认宏扩展失败并且不需要完成宏的其余部分时，.mexit 指令会很有用。
.endm	是终止宏定义的指令。

如果要在宏定义中添加注释但*不希望* 这些注释显示在宏扩展中，请在注释前使用感叹号。如果*希望* 注释显示在宏扩展中，请使用星号或分号。有关宏注释的更多信息，请参阅节 6.7。

以下示例展示了宏的定义、调用和扩展。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

宏定义、调用和扩展

宏定义：以下代码使用四个参数定义了一个宏 add3：

```
    1                   *
    2
    3                   *      add3
    4                   *
    5                   *      ADDRP = P1 + P2 + P3
    6
    7                   add3   .macro P1, P2, P3, ADDRP
    8
    9                          ADD    ADDRP, P1, P2
   10                          ADD    ADDRP, ADDRP, P3
   11                          .endm
```

宏调用：以下代码使用四个参数调用宏 add3：

```
   12
   13 00000000                 add3 R1, R2, R3, R0
```

宏扩展：以下代码展示了替换针对宏调用的宏定义。汇编器将 add3 的 P1、P2、P3 和 ADDRP 参数替换为 R1、R2、R3 和 R0。

```
1
1        00000000 E0810002         ADD    R0, R1, R2
1        00000004 E0800003         ADD    R0, R0, R3
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

6.3 宏参数/替代符号

如果您希望多次调用一个宏，但每次使用不同的数据，则可以在该宏中指定参数。宏语言支持一种特殊的符号，被称为*替代符号*，用于宏参数。

宏参数是表示字符串的替代符号。这些符号也可以在宏以外使用，使字符串等于符号名称（请参阅节 4.8.8）。

有效的替代符号最长可以包含 128 个字符，并且*必须以字母开头*。该符号的其余字符可以是字母数字字符、下划线和美元符号的组合。

用作宏参数的替代符号是定义宏中的局部替代符号。您可以为每个宏定义最多 32 个局部宏替代符号（包括使用 .var 指令定义的替代符号）。有关 .var 指令的更多信息，请参阅节 6.3.6。

在宏扩展期间，汇编器会通过变量将实参传递给宏参数。每个实参的字符串等效值会被分配给对应的参数。没有对应实参的参数会被设置为空字符串。如果实参数量超过参数数量，则会将剩余所有实参的字符串等效值分配给最后一个参数。

如果要将一个实参列表分配一个参数，或者如果要将逗号或分号分配给一个参数，则必须在两边加上引号。

在汇编时，汇编器会将宏参数/替代符号替换为相应的字符串，然后将源代码转换为对象代码。

以下示例展示了具有不同数量实参的宏的扩展。

调用具有不同数量实参的宏

宏定义：

```
Parms.macro      a,b,c
;                 a = :a:
;                 b = :b:
;                 c = :c:
      .endm
```

调用宏：

```
      Parms   100,label             Parms   100,label,x,y
;         a = 100                   ;    a = 100
;         b = label                 ;    b = label
;         c = ""                    ;    c = x,y
      Parms   100, , x              Parms   "100,200,300",x,y
;         a = 100                   ;    a = 100,200,300
;         b = ""                    ;    b = x
;         c = x                     ;    c = y
      Parms   """string""",x,y
;         a = "string"
;         b = x
;         c = y
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

6.3.1 用于定义替代符号的指令

可使用 **.asg** 和 **.eval** 指令操作替代符号。

- **.asg** 指令将字符串分配给替代符号。

  对于 .asg 指令，引号是可选项。如果没有引号，汇编器会读取第一个逗号之前的所有字符并删除前导空格和尾随空格。无论是哪一种情况，都会读取字符串并将其分配给*替代符号*。.asg 指令的语法为：

  .asg[**"**]*character string*[**"**], *substitution symbol*

.asg 指令展示了分配给替代符号的字符串。

.asg 指令

```
    .asg   R13, stack_ptr     ; stack pointer
```

- **.eval** 指令对数字替代符号执行算术运算。

  .eval 指令会计算 *表达式* 并将结果字符串值分配给 *替代符号*。如果表达式 (expression) 定义不明确，汇编器会生成错误并将 null 字符串分配给符号。.eval 指令的语法为：

  .eval *well-defined expression* **,** *substitution symbol*

.eval 指令展示了对替代符号执行的算术运算。

.eval 指令

```
    .asg   1,counter
    .loop  100
    .word  counter
    .eval  counter + 1,counter
    .endloop
```

在 .eval 指令中，.asg 指令可以替换为 .eval 指令 (.eval 1, counter) 而不改变输出。在像这样的简单情况下，可以将 .eval 和 .asg 互用。但是，如果要从表达式计算 *值*，则必须使用 .eval。.asg 仅将字符串分配给替代符号，而 .eval 会计算表达式，然后将等效的字符串分配给替代符号。

有关 .asg 和 .eval 汇编器指令的更多信息，请参阅分配替代符号。

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY
TI Confidential – NDA Restrictions

6.3.2 内置替代符号函数

以下内置替代符号函数使您能够根据替代符号的字符串值做出决定。这些函数总是会返回值，并可以在表达式中使用。内置替代符号函数在条件汇编表达式中特别有用。这些函数的参数是替代符号或字符串常量。

在表 6-1 所示的函数定义中，*a* 和 *b* 是表示替代符号或字符串常量的参数。术语 *字符串* 指的是参数的字符串值。符号 *ch* 表示字符常量。

表 6-1. 替代符号函数和返回值

函数	返回值
$$symlen *(a)*	字符串 *a* 的长度
$$symcmp *(a,b)*	< 0 if *a* < *b*; 0 if *a* = *b*; > 0 if *a* > *b*
$$firstch *(a,ch)*	字符常量 *ch* 在字符串 *a* 中第一次出现的索引
$$lastch *(a,ch)*	字符常量 *ch* 在字符串 *a* 中最后一次出现的索引
$$isdefed *(a)*	如果在符号表中定义了字符串 *a*，则为 1 如果在符号表中未定义字符串 *a*，则为 0
$$ismember *(a,b)*	列表 *b* 的顶部成员被分配给字符串 *a* 如果 *b* 是空字符串，则为 0
$$iscons *(a)*	如果字符串 *a* 是二进制常量，则为 1 如果字符串 *a* 是八进制常量，则为 2 如果字符串 *a* 是十六进制常量，则为 3 如果字符串 *a* 是字符常量，则为 4 如果字符串 *a* 是十进制常量，则为 5
$$isname *(a)*	如果字符串 *a* 是有效的符号名称，则为 1 如果字符串 *a* 不是有效的符号名称，则为 0
$$isreg *(a)* [1]	如果字符串 *a* 是有效的预定义寄存器名称，则为 1 如果字符串 *a* 不是有效的预定义寄存器名称，则为 0

(1) 更多有关预定义寄存器名称的信息，请参阅节 4.8.6。

以下示例显示了内置替代符号函数。

使用内置替代符号函数

```
    .asg    label, ADDR                  ; ADDR = label
    .if     ($$symcmp(ADDR, "label") = 0) ; evaluates to true
    LDR     R4, ADDR
    .endif
    .asg    "x,y,z" , list               ; list = x,y,z
    .if     ($$ismember(ADDR,list))      ; ADDR = x, list = y,z
    SUB     R4, R4, #4                   ; sub x
    .endif
```


Copyright © 2023 Texas Instruments Incorporated

DRAFT ONLY
TI Confidential – NDA Restrictions

6.3.3 递归替代符号

汇编器在遇到替代符号时，会尝试替换为对应的字符串。如果该字符串也是替代符号，汇编器会再次执行替换。汇编器会继续执行此操作，直到遇到不是替代符号的令牌，或者直到遇到此评估过程中已遇到过的替代符号。

在以下示例中，用 x 替代 z；用 z 替代 y，并用 y 替代 x。汇编器识别出这是一个无限递归并停止替换。

递归替代

```
   .asg  "x",z  ; declare z and assign z = "x"
   .asg  "z",y  ; declare y and assign y = "z"
   .asg  "y",x  ; declare x and assign x = "y"
   LDR   R0, x
 * LDR   R0, x  ; recursive expansion
```

6.3.4 强制替代

在某些情况下，汇编器无法识别替代符号。强制替代运算符是符号两边的一组冒号，使您能够强制替代符号的字符串。只需在符号两边使用冒号，即可强制进行替代。冒号和符号之间不要有任何空格。强制替代运算符的语法为：

```
:symbol:
```

汇编器在展开其他替代符号之前，会展开两边有冒号的替代符号。

只能在宏内部使用强制替代运算符，而不能在另一个强制替代运算符中嵌套强制替代运算符。

使用强制替代运算符展示了如何使用强制替代运算符。

使用强制替代运算符

```
       1                    force   .macro
       2                            .asg    0,x
       3                            .loop   8
       4                    AUX:x:  .set    x
       5                            .eval   x+1,x
       6                            .endloop
       7                            .endm
       8
       9 00000000                   force
1                                   .asg    0,x
1                                   .loop   8
1                           AUX:x:  .set    x
1                                   .eval   x+1,x
1                                   .endloop
2                00000000   AUX0  .set    0
2                                   .eval   0+1,x
2                00000001   AUX1  .set    1
2                                   .eval   1+1,x
2                00000002   AUX2  .set    2
2                                   .eval   2+1,x
2                00000003   AUX3  .set    3
2                                   .eval   3+1,x
2                00000004   AUX4  .set    4
2                                   .eval   4+1,x
2                00000005   AUX5  .set    5
2                                   .eval   5+1,x
2                00000006   AUX6  .set    6
2                                   .eval   6+1,x
2                00000007   AUX7  .set    7
2                                   .eval   7+1,x
```

6.3.5 访问带下标的替代符号的各个字符

在宏中，可通过使用下标和强制的替代运算符（符号和下标周围的冒号）来访问替代符号的单个字符（子字符串）。可通过两种方式访问子字符串：

- :symbol (*well-defined expression*): 采用这种下标方法进行计算的结果是，得到了一个包含一个字符的字符串。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

- :symbol (*well-defined expression* $_1$, *well-defined expression* $_2$): 在这种方法中，expression$_1$ 表示子字符串的起始位置，而 expression$_2$ 表示子字符串的长度。您可以准确指定从哪里开始添加下标以及计算得出的字符串的确切长度。*子字符串字符的索引从 1 开始，而不是从 0 开始。*

以下示例显示了带下标的替代符号函数。在第一个示例中，带下标的替代符号重新定义了 ADD 指令，以便它能够处理短立即值。在第二个示例中，带下标的替代符号用于查找从字符串 strg2 中的 start 位置开始的子字符串 strg1。子字符串 strg1 的位置被分配给替代符号 pos。

使用带下标的替代符号重新定义指令

```
ADDX    .macro    dst, imm
        .var      TMP
        .asg      :imm(1):, TMP
        .if       $$symcmp(TMP,"#") = 0
        ADD       dst, dst, imm
        .else
        .emsg     "Bad Macro Parameter"
        .endif
        .endm
        ADDX      R9, #100              ; macro call
        ADDX      R9, R8                ; macro call
```

使用带下标的替代符号查找子字符串

```
substr  .macro    start,strg1,strg2,pos
        .var      LEN1,LEN2,I,TMP
        .if       $$symlen(start) = 0
        .eval     1,start
        .endif
        .eval     0,pos
        .eval     1,i
        .eval     $$symlen(strg1),LEN1
        .eval     $$symlen(strg2),LEN2
        .loop
        .break    I = (LEN2 - LEN1 + 1)
        .asg      ":strg2(I,LEN1):",TMP
        .eval     i,pos
        .break
        .else
        .eval     I + 1,i
        .endif
        .endloop
        .endm
        .asg      0,pos
        .asg      "ar1 ar2 ar3 ar4",regs
        substr    1,"ar2",regs,pos
        .word     pos
```

6.3.6 替代符号作为宏中的局部变量

如果您想要将替代符号用作宏中的局部变量，您可以使用 **.var** 指令来为每个宏定义最多 32 个局部宏替代符号（包括参数）。.var 指令会创建临时的替代符号并将初始值设置为空字符串。这些符号不会作为参数传递，并会在扩展后丢失。

.var *sym*$_1$ [,*sym*$_2$, ...,*sym*$_n$]

节 6.3.5 的示例中使用了 .var 指令。

6.4 宏库

定义宏的一种方法是创建宏库。宏库是包含宏定义的文件集合。必须使用归档器将这些文件或成员收集到单个文件（称为存档）中。宏库的每个成员都包含一个宏定义。宏库中的文件必须是未汇编的源文件。宏名和成员名必须相同，且宏文件名的扩展名必须为 .asm。例如：

宏名	宏库中的文件名
simple	simple.asm

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY
TI Confidential – NDA Restrictions

宏名	宏库中的文件名
add3	add3.asm

您可以使用 .mlib 汇编器指令来访问宏库（如定义宏库中所述）。语法为：

```
.mlib filename
```

汇编器遇到 .mlib 指令时，会打开由文件名 (filename) 指定的库并创建一个库内容表。汇编器会将库中各个成员的名称作为库条目输入到操作码表中；这将重新定义同名的任何现有操作码或宏。如果这些宏之一被调用，汇编器会从库中提取条目并将其加载到宏表中。

汇编器采用与扩展其他宏相同的方式扩展库条目。请参阅节 6.1，了解汇编器如何扩展宏。您可以使用 .mlist 指令控制库条目扩展的列表。有关 .mlist 指令的信息，请参阅节 6.8 和启动/停止宏扩展列表。只会提取实际从库中调用的宏，并且只提取一次。

您可以使用归档器通过在存档中包含所需文件来创建宏库。宏库与任何其他存档文件没有什么不同，只是汇编器希望宏库包含宏定义。汇编器*只*期望宏库中包含宏定义；将目标代码或其他源文件放入宏库中可能会产生不良结果。有关创建宏库存档的信息，请参阅节 7.1。

6.5 在宏中使用条件汇编

条件汇编指令包括 **.if/.elseif/.else/.endif** 和 **.loop/ .break/.endloop**。它们可以互相嵌套，深度最多可达 32 级。条件代码块的格式为：

```
.if 明确定义的表达式
[.elseif 明确定义的表达式]
[.else]
.endif
```

.elseif 和 **.else** 指令在条件汇编中是可选的。.elseif 指令可在条件汇编代码块中多次使用。省略 .elseif 和 .else 时，如果 .if 表达式为 false (0)，汇编器会继续汇编位于 .endif 指令之后的代码。请参阅 汇编条件代码块 了解有关 .if/ .elseif/.else/.endif 指令的更多信息。

.loop/.break/.endloop 指令可反复汇编一个代码块。可重复代码块的格式为：

```
.loop [明确定义的表达式]
[.break [明确定义的表达式]]
.endloop
```

.loop 指令可选的*明确定义的表达式*会评估循环记数（要执行的循环数）。如果忽略此表达式，循环记数默认为 1024 次，除非汇编器遇到 .break 指令，其表达式为 true（非零）。请参阅 重复汇编条件代码块 了解有关 .loop/.break/.endloop 指令的更多信息。

在重复汇编中 **.break** 指令及其表达式是可选的。如果表达式评估为 false，循环会继续。如果 .break 表达式评估为 true，或省略 .break 表达式，汇编器会中断循环。循环中断后，汇编器会继续处理位于 .endloop 指令之后的代码。如需更多信息，请参阅 节 5.8。

.loop/.break/.endloop 指令、嵌套条件汇编指令和条件汇编代码块中的内置替代符号函数展示了 .loop/.break/ .endloop 指令，正确嵌套的条件汇编指令，以及条件汇编代码块中使用的内置替代符号函数。

.loop/.break/.endloop 指令

```
        .asg    1,x
        .loop
        .break  (x == 10)  ; if x == 10, quit loop/break with expression
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

```
        .eval   x+1,x
        .endloop
```

嵌套条件汇编指令

```
        .asg    1,x
        .loop
        .if     (x == 10)  ; if x == 10, quit loop
        .break  (x == 10)  ; force break
        .endif
        .eval   x+1,x
        .endloop
```

条件汇编代码块中的内置替代符号函数

```
        .fcnolist
*
*Double Add or Subtract
*
DBL     .macro ABC, dsth, dstl, srch, srcl  ; add or subtract double
        .if     $$symcmp(ABC,"+")
        ADDS    dstl, dstl, srcl            ; add double
        ADC     dsth, dsth, srch
        .elseif $$symcmp(ABC,"-")
        SUBS    dstl, dstl, srcl            ; subtract double
        SUBS    dsth, dsth, srch
        .else
        .emsg   "Incorrect Operator Parameter"
        .endif
        .endm
*Macro Call
        DBL  -, R4, R5, R6, R7
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

6.6 在宏中使用标签

汇编语言程序中的所有标签都必须是唯一的。其中也包括宏中的标签。如果宏会扩展多次，其标签也要多次定义。*多次定义一个标签是非法的*。宏语言提供一种在宏中定义标签的方法，可使标签唯一化。只需在每个标签后跟一个问号，汇编器会将问号替换为一个句点，后跟一个唯一的数字。扩展宏时，*您不会在列表文件中看到这些数字*。您的标签显示时带有问号，与宏定义中的相同。您无法将些标签声明为全局标签。请参阅节 4.8.3 进一步了解标签。

唯一标签的语法为：

label **?**

宏中的独特标签展示了了如何在宏中生成唯一标签。标签长度上限缩短，为唯一后缀留出空间。例如，如果宏的扩展次数少于 10 次，标签长度上限为 126 个字符。如果宏的扩展次数在 10 次到 99 次之间，标签长度上限为 125 个字符。具有唯一后缀的标签显示在交叉列表文件中。要获得交叉列表文件，请在调用汇编器时使用 --asm_cross_reference_listing 选项（请参阅节 4.14）。

宏中的独特标签

```
       1                    ; define macro to find minimum
       2                    MIN    .macro dst, src1, src2
       3                           CMP    src1, src2
       4                           BCC    m1?
       5                           MOV    dst, src1
       6                           B      m2?
       7
       8                    m1?    MOV    dst, src2
       9                    m2?
      10                           .endm
      11
      12                    ; call macro
      13 00000000                  .state16
      14 00000000                  MIN    r4, r1, r2
1        00000000 4291             CMP    r1, r2
1        00000002 D301             BCC    m1?
1        00000004 1C0C             MOV    r4, r1
1        00000006 E000             B      m2?
1
1        00000008 1C14      m1?    MOV    r4, r2
1        0000000a           m2?
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

6.7 在宏中生成消息

宏语言支持三条指令，可供用户定义自有的汇编时错误和警告消息。如果要根据需求创建特定消息，这些指令特别有用。列表文件的最后一行显示错误和警告计数。这些计数提醒用户代码中存在问题，在调试期间会特别有用。

.emsg 将错误消息发送到列表文件。.emsg 指令生成错误的方式与汇编器相同：递增错误计数并阻止汇编器生成目标文件。

.mmsg 将汇编时消息发送到列表文件。.mmsg 指令的运行方式与 .emsg 指令相同，但不设置错误计数，也不会阻止生成目标文件。

.wmsg 将警告消息发送到列表文件。.wmsg 指令的运行方式与 .emsg 指令相同，但会递增警告计数，不会阻止生成目标文件。

宏注释是出现在宏定义中的注释，*但不出现在宏扩展中*。第 1 列的感叹号标识一条宏注释。如果希望注释出现在宏扩展中，请在注释前添加星号或分号。

在宏中生成消息 展示了宏中的用户消息，以及不会出现在宏扩展中的宏注释。有关 .emsg、.mmsg 和 .wmsg 汇编器指令的更多信息，请参阅定义消息。

在宏中生成消息

```
       1                    MUL_I   .macro x,y
       2                            .if ($$symlen(x) ==0)
       3                                .emsg "ERROR -- Missing Parameter"
       4                                .mexit
       5                            .elseif ($$symlen(y) == 0)
       6                                .emsg "ERROR -- Missing Parameter"
       7                                .mexit
       8                            .else
       9                                MOV  R1, x
      10                                MOV  R2, y
      11                                MUL  R0, R1, R2
      12                            .endif
      13                            .endm
      14
      15 00000000                   MUL_I #50, #51
1                                   .if ($$symlen(x) ==0)
1                                       .emsg "ERROR -- Missing Parameter"
1                                       .mexit
1                                   .elseif ($$symlen(y) == 0)
1                                       .emsg "ERROR -- Missing Parameter"
1                                       .mexit
1                                   .else
1         00000000 E3A01032             MOV  R1, #50
1         00000004 E3A02033             MOV  R2, #51
1         00000008 E0000291             MUL  R0, R1, R2
1                                   .endif
      16
      17 0000000c                   MUL_I
1                                   .if ($$symlen(x) ==0)
1                                       .emsg "ERROR -- Missing Parameter"
  ***** USER ERROR ***** - : ERROR -- Missing Parameter
1                                       .mexit
 1 Error, No Warnings
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

6.8 使用指令设置输出列表的格式

宏、替代符号和条件汇编指令可能会隐藏信息。用户可能需要查看这些隐藏信息，因此宏语言支持扩展的列表功能。

默认情况下，汇编器会在列表输出文件中显示宏扩展和错误条件块。用户可能希望在列表文件中打开或关闭此列表。四组指令可用于控制此信息的列表：

- **宏和循环扩展列表**

  .mlist 扩展宏和 .loop/.endloop 块。.mlist 指令用于输出这些块中的所有代码。

  .mnolist 抑制宏扩展和 .loop/ .endloop 块列表。

  对于宏和循环扩展列表，.mlist 是默认指令。

- **错误条件块列表**

  .fclist 使汇编器在列表文件中包含所有不会生成代码的条件块（错误条件块）。条件块在列表中的形式与在源代码中完全相同。

  .fcnolist 抑制错误条件块列表。列表中仅显示条件块中实际汇编的代码。.if、.elseif、.else, 和 .endif 指令不会出现在列表中。

  对于错误条件块列表，.fclist 是默认指令。

- **替代符号扩展列表**

  .sslist 扩展列表中的替代符号。这对于调试替代符号的扩展非常有用。扩展代码行显示在实际源代码行下方。

  .ssnolist 关闭列表中的替代符号扩展。

  对于替代符号扩展列表，.ssnolist 是默认指令。

- **指令列表**

  .drlist 使汇编器将所有指令行输出到列表文件中。

  .drnolist 抑制在列表文件中输出特定指令。这些指令为 .asg、.eval、.var、.sslist、.mlist、.fclist、.ssnolist、.mnolist、.fcnolist、.emsg、.wmsg、.mmsg、.length、.width 和 .break。

  对于指令列表，.drlist 是默认指令。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

6.9 使用递归和嵌套宏

宏语言支持递归和嵌套宏调用。即您可以在宏定义中调用其他宏。您最多可以嵌套 32 层宏。使用递归宏时，您可从它自有的定义中调用一个宏（宏调用其自身）。

您在创建递归宏或嵌套宏时，应密切关注传递到宏参数的参数，因为汇编器会使用参数的动态范围。这意味着调用的宏会使用它所调用宏的环境。

使用嵌套宏展示了嵌套宏。in_block 宏中的 y 将 out_block 宏中的 y 隐藏起来。但 in_block 宏可以访问 out_block 宏中的 x 和 z。

使用嵌套宏

```
in_block  .macro y,a
            .           ; visible parameters are y,a and x,z from the calling macro
          .endm
out_block .macrox,y,z
            .           ; visible parameters are x,y,z
            .
          in_block x,y  ; macro call with x and y as arguments
            .
            .
          .endm
          out_block     ; macro call
```

使用递归宏展示了递归宏和 fact 宏。fact 宏生成计算 n 的阶乘时必须用到的汇编代码，其中 n 是直接值。结果置于数据存储器地址位置。fact 宏实现这一目标的方式是调用 fact1，它会递归调用其自身。

使用递归宏

```
fact    .macro N, loc       ; N is an integer constant.Register loc address = N!
        .if    N < 2        ; 0! = 1! = 1
        MOV    loc, #1
        .else
        MOV    loc, #N      ; N >= 2 so, store N in loc.
        .eval  -1, N       ; Decrement N, and do the factorial of N - 1.
        fact1               ; Call fact with current environment.
        .endm
fact1   .macro
        .if    N > 1
        MOV    R0, #N       ; N > 1 so, store N in R0.
        MUL    loc, R0, loc ; Multiply present factorial by present position.
        .eval  N - 1, N     ; Decrement position.
        fact1               ; Recursive call.
        .endif
        .endm
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

6.10 宏指令摘要

表 6-2 至表 6-6 中列出的指令可与宏搭配使用。.macro、.mexit、.endm 和 .var 指令仅在与宏搭配使用时才有效；其余指令就是一般的汇编语言指令。

表 6-2. 创建宏

		请参阅	
助记符和语法	说明	宏的使用	指令
.endm	终止宏定义	节 6.2	.endm
macname **.macro** [*parameter*$_1$][,..., *parameter*$_n$]	通过 *macname* 定义宏	节 6.2	.macro
.mexit	转至 .endm	节 6.2	节 6.2
.mlib *filename*	识别包含宏定义的库	节 6.4	.mlib

表 6-3. 操作替代符号

		请参阅	
助记符和语法	说明	宏的使用	指令
.asg ["]*character string*["], *substitution symbol*	为替代符号分配字符串	节 6.3.1	.asg
.eval *well-defined expression, substitution symbol*	在数字替代符号上执行算法	节 6.3.1	.eval
.var *sym*$_1$ [, *sym*$_2$, ..., *sym*$_n$]	定义局部宏符号	节 6.3.6	.var

表 6-4. 条件汇编

		请参阅	
助记符和语法	说明	宏的使用	指令
.break [*well-defined expression*]	可选可重复块汇编	节 6.5	.break
.endif	终止条件汇编	节 6.5	.endif
.endloop	终止可重复块汇编	节 6.5	.endloop
.else	可选条件汇编代码块	节 6.5	.else
.elseif *well-defined expression*	可选条件汇编代码块	节 6.5	.elseif
.if *well-defined expression*	开始条件汇编	节 6.5	.if
.loop [*well-defined expression*]	开始可重复块汇编	节 6.5	.loop

表 6-5. 生成汇编时消息

		请参阅	
助记符和语法	说明	宏的使用	指令
.emsg	向标准输出发送错误消息	节 6.7	.emsg
.mmsg	向标准输出发送汇编时消息	节 6.7	.mmsg
.wmsg	向标准输出发送警告消息	节 6.7	.wmsg

表 6-6. 格式化列表

		请参阅	
助记符和语法	说明	宏的使用	指令
.fclist	允许列出错误条件代码块（默认）	节 6.8	.fclist
.fcnolist	禁止列出错误条件代码块	节 6.8	.fcnolist
.mlist	允许宏列表（默认）	节 6.8	.mlist
.mnolist	禁止宏列表	节 6.8	.mnolist
.sslist	允许列出展开的替代符号	节 6.8	.sslist
.ssnolist	禁止列出展开的替代符号（默认）	节 6.8	.ssnolist

Copyright © 2023 Texas Instruments Incorporated

This page intentionally left blank.

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

归档器说明

借助 ARM 归档器，您可以将多个单独的文件合并为一个存档文件。例如，您可以将多个宏收集到一个宏库中。汇编器会搜索库并使用被源文件作为宏调用的成员。您可以使用归档器将一组目标文件收集到一个对象库中。在链接阶段，链接器仅会在库中包含会解析外部引用的成员。归档器允许通过删除、替换、提取或添加成员来修改库。

在像 ARM 这样的体系结构上，通常需要具有相同目标文件库的多个版本，每个版本都使用不同的构建选项集构建。当单个库的多个版本可用时，库信息归档器可用于创建所有目标文件库版本的索引库。此索引库在链接步骤中用于代替目标文件库的特定版本。

7.1 归档器概述....170
7.2 归档器在软件开发流程中的作用....170
7.3 调用归档器....171
7.4 归档器示例....171
7.5 库信息归档器说明....173

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

7.1 归档器概述

您可以采用任何类型的文件来构建库。汇编器和链接器都接受归档库作为输入；汇编器可以使用包含单个源文件的库，链接器可以使用包含单个目标文件的库。

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

您还可以使用归档器来构建宏库。您可以创建多个源文件，每个源文件都包含一个宏，并使用归档器将这些宏收集到一个功能组中。您可以在汇编期间使用 .mlib 指令来指定要为您所调用的宏搜索的宏库。章节 6 详细讨论了宏和宏库，而本章介绍了如何使用归档器来构建库。

7.2 归档器在软件开发流程中的作用

图 7-1 展示了归档器在软件开发流程中的作用。阴影部分突出显示了最常用的归档器开发路径。汇编器和链接器均接受库作为输入。

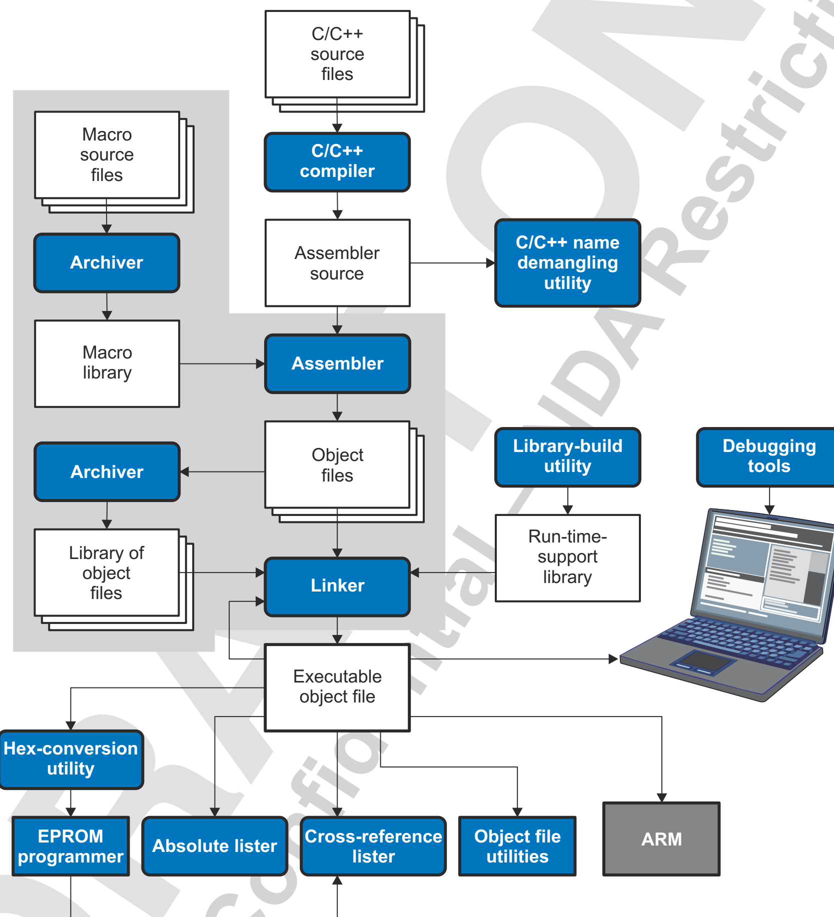


图 7-1. ARM 软件开发流程中的归档器

Copyright © 2023 Texas Instruments Incorporated

7.3 调用归档器

若要调用归档器，请输入：

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

armar 是用于调用归档器的命令。

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

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

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

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

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

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

备注

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

7.4 归档器示例

以下是典型的归档器操作示例：

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

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

  归档器的响应方式如下：

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

- 您可以使用 -t 命令列印 function.lib 的目录，输入：

```
armar -t function
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

归档器的响应方式如下：

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

- 如果您要向库中添加新成员，请输入：

```
armar -as function atan.obj
```

归档器的响应方式如下：

```
==> symbol defined: '_sin'
==> symbol defined: '$sin'
==> symbol defined: '_cos'
==> symbol defined: '$cos'
==> symbol defined: '_tan'
==> symbol defined: '$tan'
==> symbol defined: '_atan
==> symbol defined: '$atan'
==> building archive 'function.lib'
```

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

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

```
armar -x macros push.asm
```

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

```
armar -r macros push.asm
```

- 如果您要使用命令文件，请在 -@ 命令后指定命令文件名。例如：

```
armar -@modules.cmd
```

归档器的响应方式如下：

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

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

归档器命令文件

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


Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY
TI Confidential – NDA Restrictions

```
emsg.asm
end.asm
```

7.5 库信息归档器说明

节 7.1 至节 7.4 说明了如何使用归档器创建在一个或多个应用的链接器中使用的目标文件的库。您可以有相同目标文件库的多个版本，每个版本都使用不同的构建选项集构建。例如，您可为大端字节序和小端字节序、不同架构版本或不同 ABI 使用不同的目标文件库版本，具体取决于客户端应用程序的典型构建环境。但是，如果您有几个库版本，则在了解特定应用程序需要链接哪个版本的库时可能会很麻烦。

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

7.5.1 调用库信息归档器

若要调用库信息归档器，请输入以下命令：

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

armlibinfo 是用于调用库信息归档器的命令。

options 用于更改库信息归档器的默认行为。这些选项包括：

- **--output** *libname* 指定要创建或更新的索引库的名称。此选项为必备项。
- **--update** 更新使用 **--output** 选项指定的索引库中的任何现有信息，而不是创建新索引。

libnames 指定要操作的单个目标文件库的名称。输入库名时，必须输入完整文件名，包括扩展名（如适用）。

7.5.2 库信息归档器示例

我们来看看这些具有相同成员但使用不同构建选项构建的目标文件库：

目标文件库名称	构建选项
mylib_ARMv4_be.lib	--code_state=32 --silicon_version=4 --endian=big
mylib_ARMv4_le.lib	--code_state=32 --silicon_version=4 --endian=little
mylib_THUMBv4_be.lib	--code_state=16 --silicon_version=4 --endian=big
mylib_THUMBv4_le.lib	--code_state=16 --silicon_version=4 --endian=little
mylib_THUMBv7A8_le.lib	--code_state=16 --silicon_version=7A8 --endian=little

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

```
armlibinfo --output mylib.lib mylib_ARMv4_be.lib  mylib_THUMBv4_be.lib
    mylib_THUMBv7A8_le.lib mylib_ARMv4_le.lib  mylib_THUMBv4_le.lib
```

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

- 示例 **1**（ISA v7A8，小端字节序）：

```
armcl-mv7A8 -me --mylib_pruv3_be main.c -z -l lnk.cmd ./mylib.lib
<Linking>
remark: linking in "mylib_THUMBv7A8_le.lib" in place of "mylib.lib"
```

- 示例 **2**（ISAv5，大端字节序）：

```
armcl -mv5e --issue_remarks main.c -z -l lnk.cmd ./mylib.lib
<Linking>
remark: linking in "mylib_ARMv4_be.lib" in place of "mylib.lib"
```

在示例 2 中，没有针对 ISAv5 的库版本，但 ISAv4 库可用且兼容，所以使用了这个库。

Copyright © 2023 Texas Instruments Incorporated

7.5.3 列出索引库的内容

可以对索引库使用归档器的 -t 选项，以便列出索引库已索引的归档：

```
armar t mylib.lib
      SIZE   DATE                        FILE NAME
  --------   ------------------------    -----------------
       119   Mon Apr 23 12:45:22 2007    mylib_ARMv4_be.lib.libinfo
       119   Mon Apr 23 12:45:22 2007    mylib_ARMv4_le.lib.libinfo
       119   Mon Apr 23 12:45:22 2007    mylib_THUMBv4_be.lib.libinfo
       119   Mon Apr 23 12:45:22 2007    mylib_THUMBv4_le.lib.libinfo
       119   Mon Apr 23 12:45:22 2007    mylib_THUMBv7A8_le.lib.libinfo
         0   Mon Apr 23 12:45:22 2007    __TI_$$LIBINFO
```

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

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

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

7.5.4 要求

您必须遵循以下要求，才能使用二进制索引文件：

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

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

章节 8
链接器说明

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

段的概念是链接器操作的基础；章节 2 包含关于段的详细讨论。

8.1 链接器概述......176
8.2 链接器在软件开发流程中的作用......176
8.3 调用链接器......177
8.4 链接器选项......178
8.5 链接器命令文件......199
8.6 链接器符号......233
8.7 默认放置算法......238
8.8 使用由链接器生成的复制表......239
8.9 由链接器生成的 **CRC** 表......250
8.10 部分（增量）链接......254
8.11 链接 **C/C++** 代码......255
8.12 链接器示例......256

DRAFT TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.1 链接器概述

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

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

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

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

8.2 链接器在软件开发流程中的作用

图 8-1 展示了链接器在软件开发流程中的作用。链接器可接受若干种类型的文件作为输入，包括目标文件、命令文件、库和部分链接的文件。链接器会创建一个可执行目标模块，可下载到若干种开发工具中，或由 ARM 器件执行。

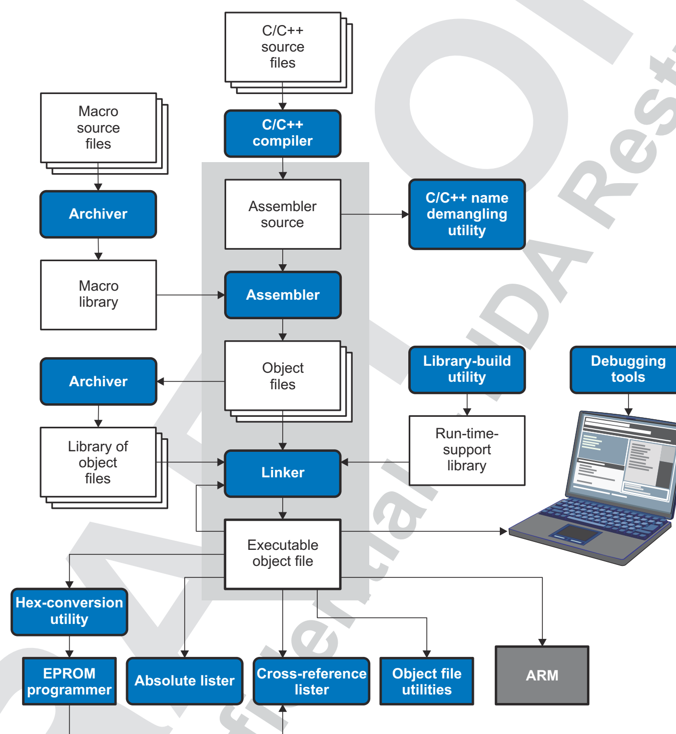


图 8-1. ARM 软件开发流程中的链接器

DRAFT ONLY
TI Confidential NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.3 调用链接器

调用链接器的一般语法如下：

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

armcl --run_linker	是用于调用链接器的命令。--run_linker 选项的短形式为 -z。
options	可出现在命令行或命令文件中的任意位置。（节 8.4 讨论了相关选项。）
filename $_1$, *filename* $_n$	可以是目标文件、链接器命令文件或存档库。输入文件的默认扩展名为 .c.obj（对于 C 源文件）和 .cpp.obj（对于 C++ 源文件）。必须显式指定任何其他扩展名。链接器可以确定输入文件是包含链接器命令的目标文件还是 ASCII 文件。除非使用 --output_file 选项来指定输出文件的名称，否则默认输出文件名为 *a.out*。

备注

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

有两种调用链接器的方法：

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

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

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

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

现在可以从命令行调用链接器；将命令文件名指定为输入文件：

```
armcl --run_linker linker.cmd
```

使用命令文件时，还可以在命令行中指定其他选项和文件。例如，可输入：

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

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

有关为 C/C++ 文件调用链接器的信息，请参阅节 8.11。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.4 链接器选项

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

表 8-1. 基本选项汇总

选项	别名	说明	段
--run_linker	-z	启用链接	节 8.3
--output_file	-o	为可执行输出模块命名。默认文件名为 a.out。	节 8.4.25
--map_file	-m	生成输入和输出段（包括空洞）的映射或列表，并将列表放置在 *文件名* 中。	节 8.4.20
--stack_size	-stack	将 C 系统栈大小设置为 *大小* 字节，并定义全局符号来指定栈大小。默认值 = 2K 字节	节 8.4.31
--heap_size	-heap	将堆大小（对于 C 中的动态存储器分配）设为 *大小* 字节，并定义全局符号来指定栈大小。默认值 = 2K 字节	节 8.4.16

表 8-2. 文件搜索路径选项汇总

选项	别名	说明	段
--library	-l	将归档库或链接命令*文件名*命名为链接器输入	节 8.4.18
--disable_auto_rts		禁止自动选择运行时支持库	节 8.4.9
--priority	-priority	满足由包含该符号定义的第一个库实现的未解析引用	节 8.4.18.3
--reread_libs	-x	强制重新读取库，以解析反向引用	节 8.4.18.3
--search_path	-i	在查找默认位置之前，更改库搜索算法以查找用*路径名*命名的目录。此选项必须出现在 --library 选项之前。	节 8.4.18.1

表 8-3. 命令文件预处理选项汇总

选项	别名	说明	段
--define		将*名称*预定义为预处理器宏。	节 8.4.11
--undefine		删除预处理器宏*名称*。	节 8.4.11
--disable_pp		禁用命令文件预处理	节 8.4.11

表 8-4. 诊断选项汇总

选项	别名	说明	段
--diag_error		将由 *num* 标识的诊断分类为错误	节 8.4.8
--diag_remark		将由 *num* 标识的诊断分类为备注	节 8.4.8
--diag_suppress		抑制由 *num* 标识的诊断	节 8.4.8
--diag_warning		将由 *num* 标识的诊断分类为警告	节 8.4.8
--display_error_number		显示诊断的标识符及其文本	节 8.4.8
--emit_references:file[=*file*]		发出包含段信息的文件。这些信息包括段大小、定义的符号和对符号的引用。	节 8.4.8
--emit_warnings_as_errors	-pdew	将警告视为错误	节 8.4.8
--issue_remarks		发出备注（非严重警告）	节 8.4.8
--no_demangle		禁止还原诊断中的符号名称	节 8.4.22
--no_warnings		抑制警告诊断（仍会发出错误）	节 8.4.8
--set_error_limit		将错误限值设置为 *num*。在达到此错误数量后，链接器将放弃链接。（默认为 100。）	节 8.4.8
--verbose_diagnostics		提供详细的诊断，以换行方式显示原始源代码	节 8.4.8
--warn_sections	-w	创建未定义的输出段时显示一条消息	节 8.4.35

表 8-5. 链接器输出选项汇总

选项	别名	说明	段
--absolute_exe	-a	生成绝对可执行模块。这是默认设置；如果 --absolute_exe 和 --relocatable 均未指定，链接器的行为就像指定了 --absolute_exe 一样。	节 8.4.3.1

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

表 8-5. 链接器输出选项汇总 (continued)

选项	别名	说明	段
--ecc={ on | off }		启用由链接器生成的错误校正码 (ECC)。默认为 off。	节 8.4.12 节 8.5.9
--ecc:data_error		将指定的错误注入到输出文件中进行测试	节 8.4.12 节 8.5.9
--ecc:ecc_error		将指定的错误注入到错误校正码 (ECC) 中进行测试	节 8.4.12 节 8.5.9
--mapfile_contents		控制映射文件中包含的信息。	节 8.4.21
--relocatable	-r	生成不可执行、可重定位的输出模块	节 8.4.3.2
--generate_dead_funcs_list		将由链接器删除的死函数列表写入文件名称。	节 8.4.15
--run_abs	-abs	生成绝对列表文件	节 8.4.29
--xml_link_info		生成结构良好的 XML *文件*，其中包含有关链接结果的详细信息	节 8.4.36

表 8-6. 符号管理选项汇总

选项	别名	说明	段
--entry_point	-e	定义一个全局符号，用于指定输出模块的主要入口点	节 8.4.13
--globalize		将与*模式*匹配的符号的符号链接更改为全局型	节 8.4.19
--hide		隐藏与*模式*匹配的全局符号	节 8.4.17
--localize		将与*模式*匹配的符号的符号链接更改为局部型	节 8.4.19
--make_global	-g	将*符号*设为全局型（覆盖 -h）	节 8.4.19.1
--make_static	-h	将所有全局符号设为静态型	节 8.4.19.1
--no_symtable	-s	从输出模块中去除符号表信息和行号条目	节 8.4.24
--retain		保留原本应丢弃的段列表	节 8.4.28
--scan_libraries	-scanlibs	扫描所有库中的重复符号定义	节 8.4.30
--symbol_map		将符号引用映射到不同名称的符号定义	节 8.4.32
--undef_sym	-u	将未解析的外部*符号*放入输出模块的符号表中	节 8.4.34
--unhide		显示（取消隐藏）与 *模式* 匹配的全局符号	节 8.4.17

表 8-7. 运行时环境选项汇总

选项	别名	说明	段
--arg_size	--args	分配可供加载程序传递参数之用的存储器	节 8.4.4
-be32		强制链接器生成 BE-32 目标代码。	节 8.4.5
-be8		强制链接器生成 BE-8 目标代码。	节 8.4.5
--cinit_hold_wdt={on|off}		在 cinit 自动初始化期间，保持（打开）或阻止（关闭）看门狗计时器。	节 8.11.5
--fill_value	-f	为输出段内的空洞设置默认填充值；*fill_value* 是 32 位常数	节 8.4.14
--ram_model	-cr	在加载时初始化变量	节 8.4.27
--rom_model	-c	在运行时自动初始化变量	节 8.4.27
--trampolines		生成 far call trampolines；默认开启	节 8.4.33

表 8-8. 链接时优化选项汇总

选项	别名	说明	段
--cinit_compression [=compression_kind]		指定应用于 C 自动初始化数据的压缩类型。如果在没有指定类型的情况下使用此选项，则默认为 lzss，表示 Lempel-Ziv-Storer-Szymanski 压缩。或者，指定 --cinit_compression=rle 以使用行程编码压缩，它提供的压缩效率通常较低。	节 8.4.6
--compress_dwarf		大力减少输入目标文件中 DWARF 信息的大小	节 8.4.7
--copy_compression [=compression_kind]		压缩由链接器复制表复制的数据	节 8.4.6
--unused_section_elimination		消除可执行模块中不需要的段；默认开启	节 8.4.10

Copyright © 2023 Texas Instruments Incorporated

表 **8-9.** 其他选项汇总

选项	别名	说明	段
--linker_help	-help	显示有关语法和可用选项的信息	-
--minimize_trampolines		放置段以最大限度地减少所需的 far trampolines 数量	节 8.4.33.2
--preferred_order		为函数放置设定优先级	节 8.4.26
--trampoline_min_spacing		当 trampoline 预留的间隔比指定的限值更近时，尝试使它们相邻	节 8.4.33.3
--zero_init		控制对未初始化变量的预初始化。默认为 on。如果使用了 --ram_model，则始终为 off。	节 8.4.37

8.4.1 文件、段和符号模式中的通配符

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

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

8.4.2 通过链接器选项指定 C/C++ 符号

链接时符号与高级语言名称相同。

有关引用符号名称的更多信息，请参阅 ARM 优化 C/C++ 编译器用户指南中的“目标文件符号命名规则（链接名称）”部分。

有关 C++ 符号命名的具体信息，请参阅本文中的节 13.3.1 和 ARM 优化 C/C++ 编译器用户指南 中的“C++ 名称还原器”一章。

有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 8.6。

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

链接器会执行重新定址，该过程即在符号的地址发生改变 (节 2.7) 时调整对符号的所有引用。

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

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

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY
TI Confidential – NDA Restrictions

8.4.3.1 生成绝对输出模块（--absolute_exe 选项）

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

- 由链接器定义的特殊符号（请参阅节 8.5.10.4）
- 介绍程序入口点等信息的标头
- *无*未解析的引用

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

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

备注

--absolute_exe 和 --relocatable 选项

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

8.4.3.2 生成可重定位输出模块（--relocatable 选项）

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

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

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

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

输出文件 a.out 可以在加载时与其他目标文件重新链接或进行重定位。（链接将与其他文件重新链接的文件被称为部分链接。如需更多信息，请参阅节 8.10。）

8.4.4 分配存储器供加载器使用以传递参数（--arg_size 选项）

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

--arg_size= *size*

size 是要在目标存储器中为命令行参数分配的字节数。

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

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

加载器和目标启动代码会使用 .args 段和 __c_args__ 符号来确定是否以及如何将参数从主机传递到目标程序。有关加载器的信息，请参阅《*ARM 优化 C/C++ 编译器用户指南*》

8.4.5 更改大端字节序指令的编码

在创建大端字节序可执行文件时，可确定指令编码是小端字节序还是大端字节序。-be8 选项生成带有小端字节序编码指令的大端字节序可执行模块。这是架构版本 6 及更高版本的默认行为。

-be32 选项生成带有大端字节序编码指令的大端字节序可执行模块。这是架构版本 5 及更低版本的默认行为。

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

默认情况下，链接器不压缩复制表（节 3.3.3 和节 8.8）源数据段。--cinit_compression 和 --copy_compression 选项指定通过链接器进行压缩。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

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

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

这些选项的语法为：

--cinit_compression[=*compression_kind*]

--copy_compression[=*compression_kind*]

compression_kind 可以是以下类型之一：

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

为了减少 .cinit 表中出现孔洞的几率，初始化表中的压缩段采用字节对齐方式。

更多有关压缩的信息，请参阅节 8.8.5。

8.4.7 压缩 DWARF 信息（--compress_dwarf 选项）

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

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

8.4.8 控制链接器诊断

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

--diag_error=*num*	将 *num* 标识的诊断分类为错误。若要查找诊断消息的数字标识符，请在单独的链接中首先使用 --display_error_number 选项。然后使用 --diag_error=*num* 将诊断重新归类为错误。只能更改任意诊断的严重性。
--diag_remark=*num*	将 *num* 标识的诊断分类为备注。若要查找诊断消息的数字标识符，请在单独的链接中首先使用 --display_error_number 选项。然后使用 --diag_remark=*num* 将诊断重新归类为备注。只能更改任意诊断的严重性。
--diag_suppress=*num*	抑制 *num* 标识的诊断。若要查找诊断消息的数字标识符，请在单独的链接中首先使用 --display_error_number 选项。然后使用 --diag_suppress=*num* 抑制诊断。只能抑制任意诊断。
--diag_warning=*num*	将 *num* 标识的诊断分类为警告。若要查找诊断消息的数字标识符，请在单独的链接中首先使用 --display_error_number 选项。然后使用 --diag_warning=*num* 将诊断重新分类为警告。只能更改任意诊断的严重性。
--display_error_number	显示诊断的数字标识符及其文本。使用此选项确定需要向诊断抑制选项（--diag_suppress、--diag_error、--diag_remark 和 --diag_warning）提供哪些参数。此选项还指示诊断是否是任意的。任意诊断是指其严重性可以被忽略的诊断。任意诊断包括后缀 -D；否则，不存在后缀。有关如何理解诊断消息的更多信息，请参阅 *ARM 优化 C/C++ 编译器用户指南*。
--emit_references:file [=*filename*]	发出包含段信息的文件。这些信息包括段大小、定义的符号和对符号的引用。此信息可用于确定为什么要将每个段包含在链接的应用中。输出文件是一个简单的 ASCII 文本文件。*filename* 用作创建的文件的基本名称。例如，--emit_references:file=myfile 在当前目录中生成一个名为 myfile.txt 的文件。
--emit_warnings_as_errors	将所有警告视为错误。此选项不能与 --no_warnings 选项一同使用。--diag_remark 选项优先于此选项。此选项优先于 --diag_warning 选项。
--issue_remarks	发出默认情况下被抑制的备注（非严重警告）。
--no_warnings	抑制警告诊断（仍会发出错误）。
--set_error_limit=*num*	将错误限制设置为 *num*，可以是任何十进制值。在出现此数量的错误后，链接器将放弃链接。（默认为 100。）
--verbose_diagnostics	提供详细的诊断，以换行方式显示原始源代码，并指示错误在源代码行中的位置

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

8.4.9 自动选择库（--disable_auto_rts 选项）

--disable_auto_rts 选项会禁用运行时支持 (RTS) 库的自动选择。有关自动选择过程的详细信息，请参阅《*ARM 优化 C/C++ 编译器用户指南*》

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

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

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

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

三个链接器选项控制着预处理器：

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

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

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

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

当一个命令文件包含 (#include) 另一个命令文件时，预处理上下文以常规方式从父级传递到子级（即，在父级定义的宏在子级中可见）。但是，当不是通过 #include 来调用命令文件时，无论是在命令行中还是通过在另一个命令文件中进行指定的常规方式，预处理上下文都不会传递到嵌套的文件中。 例外情况是 --define 和 --undefine 选项，这些选项是从遇到的位置开始全局应用。例如：

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

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

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

链接器按照以下顺序搜索 #include 文件，直到找到该文件：

1. 如果 #include 文件名位于引号中（而不是 <尖括号> 中），则在包含当前文件的目录中进行搜索。
2. 如果使用了 --include_path 编译器选项（在 --run_linker 或 -z 选项之前），请搜索使用该选项指定的路径。
3. 如果定义了 TI_ARM_C_DIR 环境变量，则搜索该定义指向的目录。请参阅节 8.4.18.2。

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

链接器提供了表 8-10 中列出的内置宏定义。链接器中这些宏的可用性取决于所使用的命令行选项，而不是所链接文件的构建属性。如果这些宏未按预期设置，请确认工程命令行使用了正确的编译器选项设置。

Copyright © 2023 Texas Instruments Incorporated

表 8-10. 预定义的 ARM 宏名称

宏名称	说明
__DATE__	扩展到 *mmm dd yyyy* 形式的编译日期
__FILE__	扩展到当前源文件名
__TI_COMPILER_VERSION__	已定义为 7-9 位整数，具体取决于 X 是 1、2 还是 3 位。该数字不包含小数。例如，版本 3.2.1 表示为 3002001。去掉前导零以防止数字被解释为八进制。
__TI_EABI__	如果启用了 EABI，则已定义为 1；否则未定义。
__TI_ ARM__	始终已定义
__TI_ ARM_V4__	如果目标是 v4 架构 (ARM7)（使用 -mv4 选项），则已定义为 1；否则未定义。
__TI_ ARM_V5__	如果目标是 v5E 架构 (ARM9E)（使用 -mv5e 选项），则已定义为 1；否则未定义。
__TI_ ARM_V6__	如果目标是 v6 架构 (ARM11)（使用了 -mv6 选项），则已定义为 1；否则未定义。
__TI_ ARM_V6M0__	如果目标是 v6M0 架构 (Cortex-M0)（使用了 -mv6M0 选项），则已定义为 1；否则未定义。
__TI_ ARM_V7__	如果目标是任意 v7 架构 (Cortex)，则定义为 1；否则未定义。
__TI_ ARM_V7A8__	如果目标是 v7A8 架构 (Cortex-A8)（使用 -mv7A8 选项），则已定义为 1；否则未定义。
__TI_ ARM_V7M__	如果目标是任何 Cortex-M 架构，则已定义为 1；否则未定义。
__TI_ ARM_V7M3__	如果目标是 v7M3 架构 (Cortex-M3)（使用了 -mv7M3 选项），则已定义为 1；否则未定义。
__TI_ ARM_V7M4__	如果目标是 v7M4 架构 (Cortex-M4)（使用了 -mv7M4 选项），则已定义为 1；否则未定义。
__TI_ ARM_V7R4__	如果目标是 v7R4 架构 (Cortex-R4)（使用 -mv7R4 选项），则已定义为 1；否则未定义。
__TIME__	扩展到“*hh:mm:ss*”形式的编译时间

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.4.12 纠错码测试（--ecc 选项）

可通过链接器命令文件生成纠错码 (ECC) 并将其放置在单独的段中。

若要启用 ECC 支持，应在命令行中添加 **--ecc=on** 链接器选项。默认情况下，ECC 生成功能已关闭，即使在链接器命令文件中使用了 ECC 指令和 ECC 限定符也是如此。因此，您可以在链接器命令文件中完全配置 ECC，同时仍然能够通过命令行快速打开和关闭代码生成功能。有关配置 ECC 支持功能的链接器命令文件语法的详细信息，请参阅节 8.5.9。

ECC 使用额外的位来允许器件检测和/或纠正错误。链接器提供的 ECC 支持与各种 TI 器件上 TI 闪存中的 ECC 支持兼容。TI 闪存使用修改后的汉明码 (72,64)，该代码为每 64 位使用 8 个奇偶校验位。请检查闪存相关文档以查看是否支持 ECC。（用于读写存储器的 ECC 在运行时完全在硬件中进行处理。）

使用 --ecc=on 选项启用 ECC 后，可使用以下命令行选项通过将位错误注入链接的可执行文件来测试 ECC。这些选项可以指定应出现错误的地址以及该地址处要翻转的代码/数据位的位掩码。可以使用绝对方式指定错误的地址，也可以指定为符号的偏移。在注入数据错误后，就会计算数据的 ECC 奇偶校验位，就好像错误不存在一样。这样便可以模拟实际可能发生的位错误，并测试 ECC 纠正不同级别错误的能力。

--ecc:data_error 选项可将错误注入位于指定位置的加载映像中。语法为：

```
--ecc:data_error=(symbol+offset|address)[,page],bitmask
```

address 是要注入错误的最小可寻址单元的位置。*symbol+offset* 可用于通过相对于该符号的有符号偏移来指定要注入错误的位置。如果地址出现在多个内存页上，则需要 *page* 编号来确保位置无歧义。*bitmask* 是要翻转的位掩码；其宽度应该是可寻址单元的宽度。

例如，以下命令行可以翻转地址 0x100 处字节中的最低有效位，使其与该字节的 ECC 奇偶校验位不一致：

```
armcl test.c --ecc:data_error=0x100,0x01 -z -o test.out
```

以下命令可以翻转 main() 代码的第三个字节中的两个位：

```
armcl test.c --ecc:data_error=main+2,0x42 -z -o test.out
```

--ecc:ecc_error 选项将错误注入与指定位置对应的 ECC 奇偶校验位中。请注意，ecc_error 选项因此只能指定 ECC 输入范围内的位置，而 data_error 选项也可以指定 ECC 输出存储器范围内的错误。语法为：

```
--ecc:ecc_error=(symbol+offset|address)[,page],bitmask
```

此选项的参数与 --ecc:data_error 的参数相同，只是 *bitmask* 必须正好为 8 位。受影响 ECC 字节的镜像副本也将包含相同的注入错误。

使用 --ecc:ecc_error 向一个 ECC 字节中注入错误后，可能会导致运行期间在该 ECC 字节覆盖的 8 个数据字节中的任何一个字节中检测到错误。

例如，以下命令可以翻转 0x200 处 ECC 字节中包含该字节奇偶校验信息的每个位：

```
armcl test.c --ecc:ecc_error=0x200,0xff -z -o test.out
```

链接器不允许将错误注入既不属于 ECC 范围也不属于 ECC 范围的输入范围的存储器范围。编译器只能将错误注入已初始化的段。

8.4.13 定义入口点（--entry_point 选项）

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

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

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY
TI Confidential – NDA Restrictions

- 由 --entry_point 选项指定的值。语法为：

  --entry_point= *global_symbol*

  其中，*global_symbol* 定义入口点，并且必须定义为输入文件的外部符号。C 或 C++ 对象的外部符号名称可能与源语言中声明的名称不同；请参阅《*ARM 优化 C/C++ 编译器用户指南*》。
- 符号 _c_int00 的值（如果存在）。如果要链接由 C 编译器生成的代码，则 _c_int00 符号*必须*作为入口点。
- 符号 _main 的值（如果存在）
- 0（默认值）

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

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

有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 8.6。

8.4.14 设置默认填充值（--fill_value 选项）

--fill_value 选项用于填充输出段中的孔洞。此选项的语法为：

--fill_value= *value*

参数 *value* 为 32 位常数（最多 8 个十六进制数字）。如果不使用 --fill_value，链接器会使用 0 作为默认填充值。

下面的示例使用十六进制值 ABCDABCD 来填充孔洞：

```
armcl --run_linker --fill_value=0xABCDABCD file1.c.obj file2.c.obj
```

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

--generate_dead_funcs_list 选项可创建一个从未被引用（死）的函数列表，并将该列表写入指定的文件。如果未指定文件名，则会使用默认文件名 dead_funcs.xml。此选项的语法为：

--generate_dead_funcs_list=*filename*

请参阅《*ARM 优化 C/C++ 编译器用户指南*》，了解有关 --generate_dead_funcs_list 选项以及相应 --use_dead_funcs_list 选项的详细信息。

8.4.16 定义堆大小（--heap_size 选项）

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

--heap_size= *size*

size 必须是一个常量。以下示例定义了一个 4K 字节的堆：

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

链接器创建 .sysmem 段的前提是输入文件中存在 .sysmem 段。

链接器还会创建全局符号 __TI_SYSMEM_SIZE，并为其分配一个等于堆大小的值。默认大小为 2K 字节。有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 8.6。

8.4.17 隐藏符号

“符号隐藏”可阻止符号在输出文件的符号表中被列出。局部化用于防止链接单元中出现名称空间冲突（请参阅节 8.4.19），而“符号隐藏”用于隐藏不应在链接单元外可见的符号。此类符号的名称在目标文件阅读器中仅显示为空字符串或“no name”。链接器通过 --hide 和 --unhide 选项支持符号隐藏。

这些选项的语法为：

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

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI confidential – NDA Restrictions

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

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

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

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

这些选项具有以下属性：

- 可在命令行上多次指定 --hide 和 --unhide 选项。
- --hide 和 --unhide 的顺序不重要。
- 一个符号只与一个由 --hide 或 --unhide 定义的模式相匹配。
- 一个符号与最严格的模式相匹配。如果模式 A 与比模式 B 更窄的集相匹配，则认为模式 A 比模式 B 更严格。
- 如果一个符号与来自 --hide 和 --unhide 的模式相匹配，但两个模式之间互不取代，那么这属于错误。如果模式 A 可以匹配模式 B 能够匹配的所有内容甚至更多内容，则认为 模式 A 取代模式 B。如果模式 A 取代模式 B，则认为模式 B 比模式 A 具有更严格的限制。
- 这些选项会影响最终链接和部分链接。

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

8.4.18 改变库搜索算法（--library、--search_path 和 TI_ARM_C_DIR）

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

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

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

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

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

当对象库的一个或多个成员被指定用作输出段的输入时，不需要 --library 选项。有关分配存档成员的更多信息，请参阅节 8.5.5.5。

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

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

备注

TI_ARM_C_DIR 环境变量优先于较旧的 TMS470_C_DIR 环境变量（如果两者均已定义）。如果只设置了 TMS470_C_DIR，则将继续使用它。同样，如果 TI_ARM_A_DIR 环境变量与较旧的 TMS470_A_DIR 环境变量都已定义，则前者优先于后者。。如果只设置了 TMS470_A_DIR，则将继续使用它。

8.4.18.1 指定备用库目录（--search_path 选项）

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

--search_path= *pathname*

DRAFT ONLY Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

pathname 指定一个包含输入文件的目录。

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

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

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

8.4.18.2 指定备用库目录（TI_ARM_C_DIR 环境变量）

环境变量是由您定义并为其分配字符串的系统符号。链接器使用名为 TI_ARM_C_DIR 的环境变量来指定包含对象库的备用目录。分配环境变量的命令语法是：

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

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

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

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

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

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

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

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

环境变量保持设置状态，直到您重新启动系统或通过输入以下命令来重置变量：

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

汇编器使用名为 TI_ARM_A_DIR 的环境变量来指定包含复制文件/头文件或宏库的备用目录。如果未设置 TI_ARM_C_DIR，则链接器会在使用 TI_ARM_A_DIR 指定的目录中搜索对象库。有关 TI_ARM_A_DIR 的信息，请参阅节 4.5.2。有关对象库的更多信息，请参阅节 8.6.6。

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

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

有两种方法可以详尽搜索未解析的符号：

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

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

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

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

或者也可以强制链接器为您执行该操作：

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

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

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

```
% armcl --run_linker objfile lib1 lib2
```

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

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

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

例如，假设您想覆盖 rtsv4_A_be_eabi.lib 中定义的 malloc 和 free 版本，而不提供对 rtsv4_A_be_eabi.lib 的完全替代。 在 rtsv4_A_be_eabi.lib 之前使用 --priority 并链接您的新库可确保所有对 malloc 和 free 的引用都解析为新库。

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

8.4.19 更改符号局部化

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

这些选项的语法为：

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

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

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

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

DRAFT ONLY
TI Confidential - NDA Restrictions

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

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

有关在链接器选项（如 --localize 和 --globalize）中使用 C/C++ 标识符的信息，请参阅节 8.4.2。

这些选项具有以下属性：

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

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

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

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

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

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

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

--make_global= *global_symbol*

8.4.20 创建映射文件（--map_file 选项）

--map_file 选项的语法为：

--map_file= *filename*

链接器映射会描述：

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

映射文件包含输出模块的名称以及入口点，并且最多还可以包含三个表：

- 一个表显示当 MEMORY 指令指定任何非默认配置时的新存储器配置。该表具有从链接器命令文件中的 MEMORY 指令生成的以下列。有关 MEMORY 指令的信息，请参阅节 8.5.4。
  - **Name**。这是使用 MEMORY 指令指定的存储器范围的名称。
  - **Origin**。这是存储器范围的起始地址。
  - **Length**。这是存储器范围的长度。
  - **Unused**。这是该存储器区域中未使用（可用）的存储器总量。

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential – NDA Restrictions

- **Attributes**。这是与指定范围相关联的一到四个属性：

  R 指定可以读取存储器。

  W 指定可以写入存储器。

  X 指定存储器可包含可执行代码。

  I 指示可以初始化存储器。

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

  有关 SECTIONS 指令的更多信息，请参阅节 8.5.5。
- 一个表显示每个外部符号及其地址（按符号名称排序）。
- 一个表显示每个外部符号及其地址（按符号地址排序）。

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

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

输出映射文件，demo.map 展示了一个映射文件的示例。

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

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

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

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

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

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

DRAFT ONLY TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

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

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

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

load_addr 和 sym_defs 属性默认都是禁用的。

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

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

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

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

8.4.22 禁用名称还原 (--no_demangle)

默认情况下，链接器在诊断中使用已还原的符号名称。例如：

未定义的符号	首次在文件中引用
ANewClass::getValue()	test.cpp.obj

--no_demangle 选项改为在诊断中显示符号的链接名称。例如：

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

有关引用符号名称的信息，请参阅《*ARM 优化 C/C++ 编译器用户指南*》中的“目标文件符号命名规则（链接名称）”一节。

有关 C++ 符号命名的具体信息，请参阅《*ARM 优化 C/C++ 编译器用户指南*》中的“C++ 名称还原器”一章。

8.4.23 禁止合并符号调试信息（--no_sym_merge 选项）

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

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

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

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential – NDA Restrictions

8.4.24 去除符号信息（--no_symtable 选项）

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

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

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

使用 --no_symtable 选项会限制之后对符号调试器的使用。

备注

去除符号信息

--no_symtable option 现已被弃用。若要去除符号表信息，请按照节 11.4 中所述，使用 armstrip 实用程序。

8.4.25 指定输出模块（--output_file 选项）

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

--output_file= *filename*

filename 是新输出模块的名称。

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

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

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

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

--preferred_order= *function specification*

请参阅《*ARM 优化 C/C++ 编译器用户指南*》了解程序缓存布局工具的详细信息，它会受 --preferred_option 影响。

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

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

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

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

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

如需更多信息，请参阅节 8.11、节 3.3.2.1 和节 3.3.2.2。

DRAFT ONLY
Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.4.28 保留丢弃的段（**--retain** 选项）

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

--retain=*sym_or_scn_spec*

--retain 选项采取以下形式之一：

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

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

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

  您无法指定 --retain='*'。
- **--retain=** *file_spec*(*scn_spec*[, *scn_spec*, ...]

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

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

  您可以指定 --retain='*(*)' 来保留所有输入文件中的所有段。不过，这不能阻止库成员中的段被优化掉。
- **--retain=** *ar_spec*<*mem_spec*, [*mem_spec*, ...>(*scn_spec*[, *scn_spec*, ...]

  指定该归档格式会保留以下位置与一个或多个 *scn_spec* 匹配的段：与 *ar_spec* 匹配的归档文件中与一个或多个 *mem_spec* 匹配的成员内。例如，以下代码会保留 rts32eabi.lib 库中 printf.c.obj 内的 .text 段：

  ```
  --retain=rts32eabi.lib<printf.c.obj>(.text)
  ```

  如果使用 --library 选项指定了库 (--library=rts32eabi.lib)，则会使用库搜索路径来搜索该库。您无法指定 '*<*>(*)'。

8.4.29 创建绝对列表文件（**--run_abs** 选项）

--run_abs 选项为每个已链接的文件生成一个输出文件。这些文件以输入文件名和 .abs 扩展名命名。但是，头文件不会生成相应的 .abs 文件。

8.4.30 扫描所有库中的重复符号定义（**--scan_libraries**）

--scan_libraries 选项用于在链接期间扫描所有库，以查找链接中实际所含那些符号的重复符号定义。该扫描不会考虑绝对符号和 COMDAT 段中定义的符号。--scan_libraries 选项有助于确定链接器实际所选的那些符号与库中相同符号的其他现有定义。

库扫描功能可用于在库中存在多个定义时，根据定义检查符号引用的意外解析。

8.4.31 定义栈大小（**--stack_size** 选项）

ARM C/C++ 编译器使用未初始化的段 .stack 来为运行时栈分配空间。可在链接时使用 --stack_size 选项设置此段的大小（以字节为单位）。--stack_size 选项的语法为：

--stack_size= *size*

size 必须是常量并以字节为单位。以下示例定义了一个 4K 字节的栈：

```
armcl --run_linker --stack_size=0x1000 /* defines a 4K heap (.stack section)*/
```

如果在输入段中指定了其他的栈大小，则输入段的栈大小将被忽略。输入段中定义的任何符号仍然有效；只是栈大小不同。

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential – NDA Restrictions

链接器在定义 .stack 段时，还会定义全局符号 __TI_STACK_SIZE，并为其分配一个等于该段大小的值。默认的软件栈大小为 2K 字节。有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 8.6。

8.4.32 符号映射（--symbol_map 选项）

符号映射允许由其他名称的符号来解析符号引用，这样便可使用替代定义来覆盖函数。这可用于在替代实现中打补丁以提供补丁（错误修复）或替代功能。--symbol_map 选项的语法为：

--symbol_map= *refname=defname*

例如，以下代码使链接器通过 foo_patch 定义来解析对 foo 的任何引用：

```
--symbol_map=foo=foo_patch
```

即使在编译时使用了 --opt_level=4，也支持 --symbol_map 选项。

使用 --symbol_map 选项传递的字符串不应包含空格，也不应在外侧使用引号。这样，可以在命令行、链接器命令文件和选项文件中使用相同的链接器选项语法。

8.4.33 生成 Far 调用 Trampoline（--trampolines 选项）

ARM 器件具有 PC 相关的调用指令和 PC 相关的分支指令，其范围小于整个地址空间。使用这些指令时，目标地址必须足够靠近指令，以便调用和目标之间的差异适应可用的编码位。如果被调用函数离调用函数太远，链接器会生成错误或生成 trampoline，具体取决于 --trampolines 选项的设置（on 还是 off）。

PC 相关调用的替代项是绝对调用，通常作为间接调用来实现：将被调用地址加载到寄存器中，然后调用该寄存器。这通常不是想要的结果，因为它需要更多的指令（依速度和大小）并且需要一个额外的寄存器来保存地址。

默认情况下，如果目标太远，编译器会生成可能需要 trampoline 的调用。在某些架构中，这种类型的调用被称为“near 调用”。

--trampolines 选项使用户能够控制 trampoline 的生成。当设置为“on”时，此选项会使链接器为其调用目标范围外链接的每个调用生成一个 trampoline 代码段。trampoline 代码段包含一个指令序列，该序列会执行到达原始调用地址的透明长分支。每个超出被调用函数范围的调用指令都会重定向到 trampoline。

此选项的语法为：

--trampolines[=on|off]

默认设置为 on。对于 ARM，默认启用 trampoline。

例如，在一段 C 代码中，bar 函数调用 foo 函数。编译器为函数生成以下代码：

```
bar:
      ...
      call    foo     ; call the function "foo"
      ...
```

如果 foo 函数置于 bar 内部对 foo 的调用范围之外，那么使用 --trampolines 时，链接器会将原来对 foo 的调用更改为对 foo_trampoline 的调用，如下所示：

```
bar:
      ...
      call    foo_trampoline   ; call a trampoline for foo
      ...
```

上面的代码会生成一个名为 foo_trampoline 的 trampoline 代码段，其中包含执行到达原始调用函数 foo 的长分支的代码。例如：

```
foo_trampoline:
      branch_long     foo
```


Copyright © 2023 Texas Instruments Incorporated

可在对同一被调用函数的调用之间共享 trampoline。唯一的要求是对被调用函数的所有调用都在被调用函数的 trampoline 附近进行链接。

当链接器生成一个映射文件（--map_file 选项）并且已生成一个或多个 trampoline 时，该映射文件将包含有关已生成的 trampoline 以及所到达的函数的统计信息。映射文件中还会提供每个 trampoline 的调用列表。

备注

链接器假定 R13 包含栈指针

汇编语言程序员必须知道链接器假定 R13 包含栈指针。链接器必须在由其生成的 trampoline 代码中保存和恢复栈上的值。如果用户不使用 R13 作为栈指针，则应使用禁用 trampoline 的链接器选项 --trampolines=off 。否则，trampoline 可能会破坏存储器并覆盖寄存器值。

8.4.33.1 使用 Trampoline 的优缺点

使用 trampoline 的优点是可以将所有的调用都当成 near 调用，这种调用的速度更快且更高效。您只需修改无法到达的调用。此外，几乎不需要考虑相互调用的函数的相对位置。调用必须通过 trampoline 的情况比 near 调用少见。

虽然生成 far 调用 trampoline 提供了更直接的解决方案，但 trampoline 的缺点是比直接调用函数要慢一些。trampoline 需要调用和分支。此外，虽然内联代码可以根据调用环境进行定制，但 trampoline 是以更通用的方式生成的，其效率可能略低于内联代码。

如果某个调用无法到达其被调用函数，为该调用创建 trampoline 代码段的另一种方法是实际修改该调用的源代码。在某些情况下，可在不影响代码大小的情况下完成此过程。但是，一般来说，这种方法是极其困难的，尤其是当代码的大小受变换影响时。

8.4.33.2 尽量减少所需的 Trampoline 数量（--minimize_trampolines 选项）

--minimize_trampolines 选项的目标是在尝试放置段时尽量减少所需的 far 调用 trampoline 数量，但可能的代价是存储器打包不再处于理想状态。语法为：

--minimize_trampolines=postorder

该参数选择要使用的启发法。postorder 启发法尝试将函数放置在函数调用方之前，以便在放置调用方时知道被调用方的 PC 相对偏移。首先放置被调用方，当放置调用方时被调用方的地址是已知的，因此链接器可以明确地知道是否需要 trampoline。

8.4.33.3 使 Trampoline 保留相邻（--trampoline_min_spacing 选项）

当发出调用并且被调用方的地址未知时，链接器必须临时为 far 调用 trampoline 保留空间，以防被调用方离得太远。即使最终表明被调用方足够接近，trampoline 预留也会对非常大的代码段的合理放置造成干扰。

当 trampoline 保留间隔比指定的限制值更近时，应使用 --trampoline_min_spacing 选项尝试使它们相邻。语法为：

--trampoline_min_spacing= *size*

较高的值可以更大限度地减少碎片，但可能会导致更多的 trampoline。较低的值可能会减少 trampoline，但代价是增加碎片和链接器运行时间。为此选项指定 0 值将禁用合并。默认为 16K。

8.4.33.4 将 Trampoline 从加载空间传送到运行空间

在存储器中的一个位置加载代码而在另一个位置运行代码的做法有时会很有用。链接器提供了为一个段指定不同的加载分配和运行分配的功能。实际将代码从加载空间复制到运行空间的重任将留给您完成。

必须先执行复制函数，然后才能在运行空间中执行实际函数。为了方便执行这个复制函数，汇编器提供了 .label 指令，允许您定义加载时地址。然后可以使用这些加载时地址来确定所要复制代码的起始地址和大小。但是，如果代码包含的某个调用需要 trampoline 才能到达其被调用函数，则此机制将*不*起作用。这是因为 trampoline 代码

Copyright © 2023 Texas Instruments Incorporated

是在链接时生成的，即在定义与 .label 指令关联的加载时地址之后。如果链接器在包含 trampoline 调用的输入段中检测到 .label 符号的定义，则会生成警告。

若要解决此问题，可使用 START()、END() 和 SIZE() 运算符（请参阅节 8.5.10.7）。通过这些运算符可以定义一些符号来表示链接器命令文件中的加载时起始地址和大小。这些符号可由复制代码引用，并且在分配 trampoline 段之后直到链接时才会解析符号的值。

以下示例说明了如何使用与输出段相关联的 START() 和 SIZE() 运算符来复制 trampoline 代码段以及含有需要 trampoline 的调用的代码：

```
SECTIONS
{   .foo : load = ROM, run = RAM, start(foo_start), size(foo_size)
          { x.obj(.text) }
    .text: {} > ROM
    .far : { --library=rts.lib(.text) } > FAR_MEM
}
```

x.c.obj 中的一个函数包含运行时支持调用。运行时支持库位于 far 存储器中，因此该调用超出范围。链接器会将一个 trampoline 段添加到 .foo 输出段。复制代码可以引用符号 foo_start 和 foo_size 作为整个 .foo 输出段的加载起始地址和大小的参数。因此，复制代码可以将 trampoline 段以及 .text 中的原始 x.c.obj 代码从其加载空间复制到其运行空间。

有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 8.6。

8.4.34 引入未解析的符号（--undef_sym 选项）

--undef_sym 选项将未解析符号的链接名称引入链接器的符号表。这会强制链接器搜索库并包含定义该符号的成员。链接器必须在链接定义符号的成员*之前*遇到 --undef_sym 选项。--undef_sym 选项的语法为：

--undef_sym= *symbol*

例如，假设名为 rtsv4_A_be_eabi.lib 的库包含一个定义符号 symtab 的成员；没有任何目标文件链接引用 symtab。但是，假设您计划重新链接输出模块，并希望在此链接中包含定义 symtab 的库成员。使用如下所示的 --undef_sym 选项会强制链接器在 rtsv4_A_be_eabi.lib 中搜索定义 symtab 的成员，并链接该成员。

```
armcl --run_linker --undef_sym=symtab file1.c.obj file2.c.obj rtsv4_A_be_eabi.lib
```

如果不使用--undef_sym，则不会包括该成员，因为在 file1.c.obj 和 file2.c.obj 中没有对该成员的显式引用。

8.4.35 创建未定义的输出段时显示一条消息 (--warn_sections)

在链接器命令文件中，您可以设置 SECTIONS 指令来描述如何将输入段组合成输出段。但是，如果链接器遇到一个或多个没有在 SECTIONS 指令中定义相应输出段的输入段，则链接器会将同名的输入段组合到该名称的输出段中。默认情况下，链接器不会显示一条消息来告知您发生了这种情况。

使用 --warn_sections 选项可以使链接器在创建新的输出段时显示消息。

有关 SECTIONS 指令的更多信息，请参阅节 8.5.5。有关链接器默认操作的更多信息，请参阅节 8.7。

8.4.36 生成 XML 链接信息文件（--xml_link_info 选项）

链接器支持通过 --xml_link_info=*file* 选项生成 XML 链接信息文件。此选项会使链接器生成一个格式良好的 XML 文件，其中包含有关链接结果的详细信息。这个文件中包含的信息包括由链接器生成的映射文件中当前生成的所有信息。有关生成的 XML 文件内容的具体信息，请参阅附录 B。

8.4.37 零初始化（--zero_init 选项）

C 和 C++ 标准要求，未显式初始化的全局变量和静态变量必须先设置为 0，然后才能执行程序。C/C++ 编译器默认支持对未初始化的变量执行预初始化。若要将其关闭，请指定链接器选项 --zero_init=off。

--zero_init 选项的语法为：

--zero_init[={*on*|*off*}]

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential - NDA Restrictions

只有使用 --rom_model 链接器选项（引发自动初始化）时，才发生零初始化。当您使用 --ram_model 链接选项时，链接器不会生成初始化记录，加载程序必须处理数据和零初始化。

备注

不建议禁用零初始化的情况：通常不建议禁用零初始化。如果关闭零初始化，将不会自动将未初始化的全局目标和静态目标初始化为零。然后您需要通过其他方式将这些变量初始化为零。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.5 链接器命令文件

借助链接器命令文件，您可以将链接器选项和指令放入一个文件中；在经常使用相同的选项和指令调用链接器时，这会很有用。在使用 MEMORY 和 SECTIONS 指令来自定义应用程序时，链接器命令文件也很有用。您必须在命令文件中使用这些指令；不能在命令行中使用它们。

链接器命令文件是包含以下一项或多项的 ASCII 文件：

- 输入文件名，指定了目标文件、归档库或其他命令文件。（如果一个命令文件调用另一个命令文件作为输入，则此语句必须是进行调用的命令文件中的*最后*一个语句。链接器不会从调用的命令文件返回。）
- 链接器选项在命令文件中的使用方式与在命令行中使用时相同
- MEMORY 和 SECTIONS 链接器指令。MEMORY 指令定义了目标内存配置（请参阅节 8.5.4）。SECTIONS 指令控制着如何构建和分配段（请参阅节 8.5.5）。
- 赋值语句为全局符号定义值和赋值

若要使用命令文件来调用链接器，请输入 armcl --run_linker 命令，后跟命令文件的名称：

armcl --run_linker *command_filename*

链接器按照遇到输入文件的顺序处理它们。如果链接器将一个文件识别为目标文件，则它会链接该文件。否则，它假定文件是命令文件，并开始从中读取和处理命令。不管使用何种系统，命令文件名都区分大小写。

连接器命令文件 展示了名为 link.cmd 的示例链接器命令文件。

连接器命令文件

```
a.c.obj                    /*  第一个输入文件名          */
b.c.obj                    /*  第二个输入文件名         */
--output_file=prog.out   /*  指定输出文件的选项 */
--map_file=prog.map      /*  指定映射文件的选项    */
```

连接器命令文件 中的示例文件仅包含文件名和选项。（您可以在命令文件中添加注释，使用 /* 和 */ 来分隔。）若要使用这个命令文件来调用链接器，请输入以下命令：

```
armcl --run_linker link.cmd
```

使用命令文件时，可以在命令行中放入其他参数：

```
armcl --run_linker --relocatable link.cmd x.c.obj y.c.obj
```

链接器在遇到文件名时立即处理命令文件，因此 a.c.obj 和 b.c.obj 在 x.c.obj 和 y.c.obj 之前链接至输出模块。

您可以指定多个命令文件。例如，如果您有一个包含文件名的文件 names.lst，还有另一个包含链接器指令的文件 dir.cmd，则您可以输入：

```
armcl --run_linker names.lst dir.cmd
```

一个命令文件可以调用另一个命令文件；这种类型的嵌套限制为 16 级。如果一个命令文件调用另一个命令文件作为输入，则此语句必须是进行调用的命令文件中的*最后*一个语句。

除了作为分隔符，空格和空白行在命令文件中没有其他意义。这也适用于命令文件中的链接器指令的格式。带有链接器指令的命令文件 展示了包含链接器指令的命令文件示例。

带有链接器指令的命令文件

```
a.obj b.obj c.obj          /* 输入文件名        */
--output_file=prog.out     /* 选项                */
--map_file=prog.map
MEMORY                     /* MEMORY 指令       */
{
```

DRAFT ONLY Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

```
  FAST_MEM:  origin = 0x0100     length = 0x0100
  SLOW_MEM:  origin = 0x7000     length = 0x1000
}
SECTIONS                     /* SECTIONS 指令     */
{
  .text:   > SLOW_MEM
  .data:   > SLOW_MEM
  .bss:    > FAST_MEM
}
```

有关 MEMORY 指令的更多信息，请参阅节 8.5.4，有关 SECTIONS 指令的更多信息，请参阅节 8.5.5。

8.5.1 链接器命令文件中的保留名称

以下名称（大小写都有）保留为链接器指令的关键字。请勿在命令文件中将其用作符号名称或段名。

ADDRESS_MASK	ECC	LAST	NOLOAD	RUN_START
ALGORITHM	END	LEN	o	SECTIONS
ALIAS	f	LENGTH	ORG	SIZE
ALIGN	FILL	LOAD	ORIGIN	START
ATTR	GROUP	LOAD_END	PAGE	TABLE
BLOCK	HAMMING_MASK	LOAD_SIZE	PALIGN	TYPE
COMPRESSION	HIGH	LOAD_START	PARITY_MASK	UNION
COPY	INPUT_PAGE	MEMORY	RUN	UNORDERED
CRC_TABLE	INPUT_RANGE	MIRRORING	RUN_END	VFILL
DSECT	l（小写 L）	NOINIT	RUN_SIZE	

此外，TI 工具使用的任何段名都是保留名称，不能用作其他名称的前缀，除非该段是 TI 工具所用段名的子段。例如，段名不能以 .debug 开头。

8.5.2 链接器命令文件中的常量

您可以使用两种语法方案中的任何一种来指定常量：一种方案用于指定汇编器中使用的十进制、八进制或十六进制常量（但不是二进制常量）（请参阅节 4.7），另一种方案用于指定 C 语法中的整数常量。

示例：

格式	十进制	八进制	十六进制
汇编器格式	32	40q	020h
C 格式	32	040	0x20

8.5.3 从链接器命令文件访问文件和库

许多应用程序使用自定义链接器命令文件（简称 LCF）来控制代码和数据在目标存储器中的放置。例如，您可能希望将特定文件中的特定数据对象放入目标存储器中的特定位置。这很容易实现，只需使用可用的 LCF 语法来引用所需的目标文件或库。但是，许多开发人员在尝试执行此操作时会遇到一个问题：从 LCF 内部访问先前在链接器的命令行调用中指定的目标文件或库时，链接器会生成“file not found”（文件未找到）错误。大多数情况下，发生此错误的原因是用于访问链接器命令行上文件的语法与用于访问 LCF 中相同文件的语法不一致。

让我们考虑一个简单的示例。假设一个应用程序需要在名为“DDR”的存储器区域中定义名为“app_coeffs”的常量表。此外，假设在位于目标文件 app_coeffs.c.obj 的 .data 段中定义“app_coeffs”数据对象。然后，app_coeffs.c.obj 文件包含在目标文件库 app_data.lib 中。在 LCF 中，可按如下方式控制“app_coeffs”数据对象的放置：

```
SECTIONS
{
   ...
   .coeffs: { app_data.lib<app_coeffs.c.obj>(.data) } > DDR
```


Copyright © 2023 Texas Instruments Incorporated

```
    ...
}
```

现在假设 app_data.lib 对象库位于一个名为“lib”的子目录中（相对于构建应用程序的位置）。为了从构建命令行访问 app_data.lib，可组合使用 –i 和 –l 选项来设置链接器可用于查找 app_data.lib 库的目录搜索路径：

```
%> armcl <compile options/files> -z -i ./lib -l app_data.lib mylnk.cmd <link options/files>
```

-i 选项将 lib 子目录添加到目录搜索路径，-l 选项指示链接器查看目录搜索路径中的目录以查找 app_data.lib 库。但是，如果不更新 mylnk.cmd 中对 app_data.lib 的引用，则链接器将无法找到 app_data.lib 库并会生成“file not found”（文件未找到）错误。原因是当链接器在 SECTIONS 指令中遇到对 app_data.lib 的引用时，引用前没有 -l 选项。因此，链接器会尝试打开当前工作目录中的 app_data.lib。

本质上，链接器有几种不同的打开文件的方式：

- 如果指定了路径，链接器将在指定位置查找文件。对于绝对路径，链接器将尝试打开指定目录中的文件。对于相对路径，链接器将进入从当前工作目录开始的指定路径，并尝试在该位置打开文件。
- 如果没有指定路径，链接器将尝试打开当前工作目录中的文件。
- 如果 -l 选项位于文件引用之前，则链接器将尝试在目录搜索路径下的目录之一中查找并打开所引用的文件。通过 –i 选项和环境变量（如 C_DIR 和 ）设置目录搜索路径。

只要在命令行上和所有适用的 LCF 中以一致的方式引用某个文件，链接器就能够找到并打开您的目标文件和库。

回到前面的示例，可在 mylnk.cmd 中对 app_data.lib 的引用前面插入一个 -l 选项，从而确保链接器在构建应用程序时会找到并打开 app_data.lib 库：

```
SECTIONS
{
    ...
    .coeffs: { -l app_data.lib<app_coeffs.c.obj>(.data) } > DDR
    ...
}
```

从 LCF 中引用文件时使用 –l 选项的另一个好处是，如果所引用文件的位置发生变化，则可修改目录搜索路径以合并文件的新位置（例如，在命令行中使用 –i 选项），而无需修改 LCF。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.5.4 MEMORY 指令

链接器确定输出段在存储器中的分配位置；它必须具有目标存储器的模型才能完成此任务。MEMORY 指令用于指定目标存储器的模型，使用户能够定义系统包含的存储器类型以及它们占用的地址范围。链接器在分配输出段时会保留此模型，并用它来确定目标代码可以使用的存储器位置。

ARM 系统的存储器配置因应用而异。MEMORY 指令用于指定各种配置。在使用 MEMORY 定义存储器模型后，用户可以使用 SECTIONS 指令将输出段分配到已定义的存储器中。如需了解更多信息，请参阅 节 2.5。

8.5.4.1 默认存储器型号

如果您不使用 MEMORY 指令，链接器会使用基于 ARM 架构的默认存储器型号。该型号会假设系统中存在完整的 32 位地址空间（2^{32} 个位置）并且可供使用。有关默认存储器型号的更多信息，请参阅节 8.7。

8.5.4.2 MEMORY 指令语法

MEMORY 指令可识别实际存在于目标系统中并可被程序使用的存储器范围。每个范围都有若干特性：

- 名称
- 起始地址
- 长度
- 可选属性集
- 可选的填充规格

使用 MEMORY 指令时，请务必确定程序在运行时可以访问的所有存储器范围。MEMORY 指令定义的存储器会被配置；而未使用 MEMORY 指令显式指定的任何存储器则不会被配置。链接器不会将程序的任何部分放入未配置的存储器中。用户可通过不在 MEMORY 指令语句中添加相应的地址范围来表示不存在的存储器空间。

在命令文件中指定 MEMORY 指令的方法是使用 MEMORY（大写）一词后跟用大括号括起来的内存范围规格列表。下面的示例中的 MEMORY 指令定义了一个系统，该系统在地址 0x0000 0000 处具有 4K 字节的快速外部存储器，在地址 0x0000 1000 处具有 2K 字节的慢速外部存储器，在地址 0x1000 0000 处具有 4K 字节的慢速外部存储器。它还演示了存储器范围表达式以及起始/结束/大小地址运算符的使用方式（请参阅表达式形式的原点和长度）。

MEMORY 指令

```
/****************************************************/
/*      Sample command file with MEMORY directive      */
/****************************************************/
file1.c.obj    file2.c.obj           /*    Input files     */
--output_file=prog.out            /*    Options          */
MEMORY
{
   FAST_MEM (RX): origin = 0x00000000  length = 0x00001000
   SLOW_MEM (RW): origin = 0x00001000  length = 0x00000800
   EXT_MEM (RX):  origin = 0x10000000  length = 0x00001000
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

MEMORY 指令的一般语法如下：

MEMORY

{

name 1 **[(** *attr* **)] : origin =** *expr* **, length =** *expr* [**, fill =** *constant*] [**LAST(** *sym* **)**]

.

.

name n **[(** *attr* **)] : origin =** *expr* **, length =** *expr* [**, fill =** *constant*] [**LAST(** *sym* **)**]

}

name 指定存储器范围的名称。存储器名称可包含 1 到 64 个字符；有效字符包括 A-Z、a-z、$、. 和 _。名称对链接器而言没有特殊意义，它们只用于标识存储器范围。存储器范围名称是链接器的内部名称，不会保留在输出文件或符号表中。所有存储器范围必须具有唯一名称且不得重叠。

attr 指定与命名的范围关联的一到四个属性。属性是可选的；使用时，必须将它们括在圆括号中。属性可以将输出段的分配限制在特定存储器范围内。如果不使用任何属性，则可以将任何输出段分配到任何范围内，不受限制。任何未指定属性的存储器（包括默认模型中的所有存储器）都具有所有这四个属性。有效属性为：

- **R** 指定可以读取存储器。
- **W** 指定可以写入存储器。
- **X** 指定存储器可包含可执行代码。
- **I** 指定可以初始化存储器。

origin 指定存储器范围的起始地址；输入形式为 *origin*、*org* 或 *o*。相应的值（以字节为单位指定）是一个 32 位的整数常量表达式，可以采用十进制、八进制或十六进制格式。

length 指定存储器范围的长度；输入形式为 *length*、*len* 或 *l*。相应的值（以字节为单位指定）是一个 32 位的整数常量表达式，可以采用十进制、八进制或十六进制格式。

fill 指定存储器范围的填充字符；输入形式为 *fill* 或 *f*。填充值是可选的。相应的值是一个 的整数常量，可以采用十进制、八进制或十六进制格式。填充值用于填充未分配给段的存储器范围区域。（请参阅节 8.5.9.3，了解使用纠错码（ECC）时存储器范围的虚拟填充。）

LAST （可选）指定一个符号，可在运行时将其用于查找存储器范围中上次分配字节的地址。请参阅节 8.5.10.8。

备注

填充存储器范围：如果为大存储器范围指定填充值，则输出文件将非常大，因为填充存储器范围（即使填充值为 0s）会导致为该范围内所有未分配的存储器块生成原始数据。

以下示例指定了一个具有 R 和 W 属性以及填充常量 0FFFFFFFFh 的存储器范围：

```
MEMORY
{
RFILE (RW) : o = 0x0020, l = 0x1000, f = 0xFFFF
}
```

通常将 MEMORY 指令与 SECTIONS 指令结合使用以控制输出段的放置。有关 SECTIONS 指令的更多信息，请参阅节 8.5.5。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.5.4.3 表达式和地址运算符

存储器范围原点和长度可以使用带有以下运算符的整数常量表达式：

二元运算符：	* / % + - << >> == = < <= > >= & | && ||
一元运算符：	- ~ !

表达式根据标准 C 运算符优先级规则进行求值。

不会检查上溢和下溢，但会使用更大的整数类型来计算表达式。

可使用预处理指令 #define 常量代替整数常量。不能在存储器指令表达式中使用全局符号。

三个地址运算符从先前的存储器范围条目中引用存储器范围属性：

START(MR)	返回先前定义的存储器范围 MR 的起始地址。
SIZE(MR)	返回先前定义的存储器范围 MR 的大小。
END(MR)	返回先前定义的存储器范围 MR 的结束地址。

表达式形式的原点和长度

```
/********************************************************/
/*     带有 MEMORY 指令的示例命令文件        */
/********************************************************/
file1.c.obj file2.c.obj              /*     输入文件     */
--output_file=prog.out           /*     选项          */
#define ORIGIN 0x00000000
#define BUFFER 0x00000200
#define CACHE  0x0001000
MEMORY
{
   FAST_MEM (RX): origin = ORIGIN + CACHE length = 0x00001000 + BUFFER
   SLOW_MEM (RW): origin = end(FAST_MEM)  length = 0x00001800 - size(FAST_MEM)
   EXT_MEM  (RX): origin = 0x10000000     length = size(FAST_MEM) - CACHE
```

8.5.4.4 ALIAS 语句

在 MSP432 Cortex M4 等部分器件中，一个 RAM 区域可以通过系统总线和指令总线这两种不同的存储器总线来进行寻址。这个 RAM 区域位于存储器映射的 DATA 区域（通常位于 0x20000000 处），在内部被另外命名为 CODE 区域（通常位于 0x01000000 处）。这个别名操作利用指令总线来从 RAM 获取代码，同时释放其他系统总线。在此类器件上，链接器命令文件应当使用 ALIAS 语句，以便 CODE 和 DATA 的放置不会发生冲突。

若要使用上述能力，链接器必须知道指向同一存储器的两个地址。在 MEMORY 指令中使用以下语法可以为存储器范围创建一个 ALIAS。各个 ALIAS 区域必须具有相同的长度。

```
MEMORY
{
    ...
    ALIAS
    {
      SRAM_CODE (RWX) : origin = 0x01000000
      SRAM_DATA (RW)  : origin = 0x20000000
    } length = 0x0001000
    ...
}
```

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.5.5 SECTIONS 指令

在使用 MEMORY 指令指定目标系统的存储器型号后，用户可以使用 SECTIONS 指令将输出段分配到具有特定名称的存储器范围中或者具有特定属性的存储器中。例如，用户可以将 .text 和 .data 段分配到名为 FAST_MEM 的区域中，将 .bss 段分配到名为 SLOW_MEM 的区域中。

SECTIONS 指令通过如下方式控制段：

- 描述输入段如何合并为输出段
- 在可执行程序中定义输出段
- 用户可以控制将输出段置于存储器中相对于其他段和相对于整个存储器空间的位置（请注意，存储器的放置顺序并*不是*段在 SECTIONS 指令中出现的顺序。）
- 允许对输出段重命名

如需更多信息，请参阅节 2.5 、节 2.7 和节 2.4.6。子段可使用户更精确地控制段。

如果用户不指定 SECTIONS 指令，链接器会使用默认算法组合并分配段。节 8.7 详细介绍了此算法。

8.5.5.1 SECTIONS 指令语法

SECTIONS 指令在命令文件中通过 SECTIONS（大写）一词后跟用大括号括起来的输出段规范列表来指定。

SECTIONS 指令的一般语法如下：

SECTIONS
{
name : [*property* **[, property]** [**, property**] ...]
name : [*property* **[, property]** [**, property**] ...]
name : [*property* **[, property]** [**, property**] ...]
}

每个段规范均以 *name* 开头，用于定义一个输出段。（输出段是输出文件中的一个段。）段名可以涉及段、子段或归档库成员。（有关多级子段的信息，请参阅节 8.5.5.4。）段名之后是一个属性列表，用于定义段的内容和段的分配方式。各个属性之间可以用可选逗号分隔。一个段的可能属性如下：

- **Load allocation** 定义段在存储器中的加载位置。请参阅节 3.5 、节 3.1.1 和节 8.5.6。

  语法： **load =** *allocation* 或
  > *allocation*

- **Run allocation** 定义段在存储器中的运行位置。

  语法： **run =** *allocation* 或
  run > *allocation*

- **Input sections** 定义构成输出段的各输入段（目标文件）。请参阅节 8.5.5.3。

  语法： **{** *input_sections* **}**

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

- **Section type** 定义特殊段类型的标志。请参阅节 8.5.8。

  语法： **type = COPY** 或
  type = DSECT 或
  type = NOLOAD

- **Fill value** 定义用于填充未初始化空洞的值。请参阅节 8.5.11。

  语法： **fill =** *value*

以下示例显示了示例链接器命令文件中的 SECTIONS 指令。

```
/****************************************************/
/*  Sample command file with SECTIONS directive   */
/****************************************************/
file1.c.obj    file2.c.obj           /* Input files   */
--output_file=prog.out           /* Options       */
SECTIONS
{
   .text:       load = EXT_MEM, run = 0x00000800
   .const:      load = FAST_MEM
   .bss:        load = SLOW_MEM
   .vectors:    load = 0x00000000
      {
         t1.c.obj(.intvec1)
         t2.c.obj(.intvec2)
         endvec = .;
      }
   .data:alpha: align = 16
   .data:beta:  align = 16
}
```

图 8-2 显示了上面的示例中由 SECTIONS 指令定义的输出段（.vectors、.text、.const、.bss、.data:alpha 和 .data:beta），并显示了如何使用节 8.5.4.2 中提供的 MEMORY 指令在存储器中分配这些段。

图 8-2. 由 SECTIONS 指令定义的段放置示例

DRAFT ONLY
Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.5.5.2 段分配和放置

链接器在目标存储器中为每个输出段分配两个位置：即加载位置和运行位置。它们通常是相同的，用户也可以认为每个段只有一个地址。在目标存储器中放入输出段并向其分配地址的过程被称为放置。有关加载位置和运行位置不同的更多信息，请参阅节 8.5.6。

如果用户不指示链接器如何分配某个段，它会使用默认算法放置该段。通常，链接器会将段置于所配置存储器中的任何合适位置。用户可以在 SECTIONS 指令中进行定义，覆盖某个段的默认放置位置，并提供有关如何分配的指令。

用户可通过指定一个或多个分配参数来控制放置。每个参数均包含一个关键字、一个等号或大于号（可选）和一个值（可选择置于括号中）。如果加载位置和运行位置不同，则关键字 LOAD 之后的所有参数均适用于加载位置，关键字 RUN 之后的所有参数均适用于运行位置。分配参数包括：

绑定 为段分配一个具体地址。

```
.text: load = 0x1000
```

指定的存储器 将段分配到在具有指定名称（如 SLOW_MEM）或属性的 MEMORY 指令中定义的一个范围内。

```
.text: load > SLOW_MEM
```

对齐 使用 align 或 palign 关键字指定，该段必须始于一个地址边界。

```
.text: align = 0x100
```

分块 使用 block 关键字指定，该段必须置于符合分块系数的两个地址之间。如果段太大，则始于地址边界。

```
.text: block(0x100)
```

对于负载（通常是唯一的）分配，请使用大于号，并省略负载关键字：

```
    .text: > SLOW_MEM
    .text: {...}> SLOW_MEM
    .text: > 0x4000
```

如果使用多个参数，则可以按如下方式将它们串到一起：

```
    .text: > SLOW_MEM align 16
```

也可以选择使用圆括号来提高可读性：

```
    .text: load = (SLOW_MEM align(16))
```

用户也可以根据输入段规范，确定输入文件中可合并构成输出段的段。请参阅节 8.5.5.3。

8.5.5.2.1 示例：在 RAM 放置在函数中

--ramfunc 编译器选项和 ramfunc 函数属性允许编译器指定将函数放入 RAM 中并从 RAM 执行。大多数较新的 TI 链接器命令文件通过将这些函数放入 .TI.ramfunc 段来支持 ramfunc 选项和函数属性。如果您看到与此段相关的链接器错误，应将 .TI.ramfunc 段添加到您的 SECTIONS 指令中，如下所示。在以下示例中，RAM 和 FLASH 是 RAM 存储器和闪存的 MEMORY 区域的名称；这些名称在您的链接器命令文件中可能不同。

对于基于 RAM 的器件：

```
.TI.ramfunc : {} > RAM
```

对于基于闪存的器件：

```
.TI.ramfunc : {} load=FLASH, run=RAM, table(BINIT)
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.5.5.2.2 绑定

用户可以在段名称后面加上地址来设置输出段的起始地址：

```
    .text: 0x00001000
```

此示例指定 .text 段必须从位置 0x1000 开始。绑定地址必须是 32 位常量。

输出段可以绑定到所配置的存储器中的任何位置（假设有足够的空间），但不能重叠。如果没有足够的空间将段绑定到指定地址，链接器会发出错误消息。

备注

绑定与对齐和已命名的存储器不兼容：如果用户使用对齐或已命名的存储器，则无法将段绑定到地址。如果用户尝试这样做，链接器会发出错误消息。

8.5.5.2.3 指定的存储器

可将一个段分配到由 MEMORY 指令定义的存储器范围中（请参阅节 8.5.4）。以下示例将指定范围并将段链接到这些范围中：

```
MEMORY
{
   SLOW_MEM (RIX)  : origin = 0x00000000,  length = 0x00001000
   FAST_MEM (RWIX) : origin = 0x03000000,  length = 0x00000300
}
SECTIONS
{
   .text  :    > SLOW_MEM
   .data  :    > FAST_MEM ALIGN(128)
   .bss   :    > FAST_MEM
}
```

在此示例中，链接器将 .text 放置到名为 SLOW_MEM 的区域中。.data 和 .bss 输出段将分配到 FAST_MEM 中。可在指定的存储器范围内对齐某个段；.data 段在 FAST_MEM 范围内的 128 字节边界上对齐。

同样，可将一个段链接到具有特定属性的存储器区域中。为此，请指定一组属性（括在圆括号中）而不是存储器名称。使用相同的 MEMORY 指令声明，可指定：

```
SECTIONS
{
   .text: > (X)    /* .text --> executable memory     */
   .data: > (RI)   /* .data --> read or init memory    */
   .bss : > (RW)   /* .bss --> read or write memory    */
}
```

在此示例中，.text 输出段可以链接到 SLOW_MEM 或 FAST_MEM 区域中，因为这两个区域都具有 X 属性。.data 段也可以进入 SLOW_MEM 或 FAST_MEM 中，因为这两个区域都具有 R 和 I 属性。但是，.bss 输出段必须进入 FAST_MEM 区域，因为只有 FAST_MEM 的声明中具有 W 属性。

尽管链接器首先使用较低的存储器地址并在可能的情况下避免碎片化，但您无法控制将段分配在指定存储器范围中的何处。在前面的示例中，假设不存在冲突性分配，.text 段将从地址 0 开始。如果某个段必须从特定地址开始，请使用绑定而不是指定的存储器。

8.5.5.2.4 使用 HIGH 位置说明符来控制放置

默认情况下，链接器在指定的存储器范围内从低地址到高地址分配输出段。或者，用户可以通过使用 SECTION 指令声明中的 HIGH 位置说明符，使链接器在存储器范围内从高地址到低地址分配某个段。用户可使用 HIGH 位置说明符将 RTS 代码与应用程序代码分开，这样一来应用程序中的微小变化就不会导致存储器映射发生较大变化。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

例如，给定以下 MEMORY 指令：

```
MEMORY
{
    RAM              : origin = 0x0200, length = 0x0800
    FLASH            : origin = 0x1100, length = 0xEEE0
    VECTORS          : origin = 0xFFE0, length = 0x001E
    RESET            : origin = 0xFFFE, length = 0x0002
}
```

以及附带的 SECTIONS 指令：

```
SECTIONS
{
    .bss     : {} > RAM
    .sysmem  : {} > RAM
    .stack   : {} > RAM (HIGH)
}
```

用于放置 .stack 段的 HIGH 说明符会使链接器尝试将 .stack 分配到 RAM 存储器范围内的更高地址。.bss 和 .sysmem 段将分配到 RAM 中的较低地址。示例 8-1 所示为映射文件的一部分，其中展示了在 RAM 中为典型程序分配给定段的位置。

示例 8-1. 使用 HIGH 说明符进行链接器放置

```
.bss        0     00000200    00000270     UNINITIALIZED
                    00000200    0000011a     rtsxxx.lib   : defs.c.obj (.bss)
                    0000031a    00000088                  : trgdrv.c.obj (.bss)
                    000003a2    00000078                  : lowlev.c.obj (.bss)
                    0000041a    00000046                  : exit.c.obj (.bss)
                    00000460    00000008                  : memory.c.obj (.bss)
                    00000468    00000004                  : _lock.c.obj (.bss)
                    0000046c    00000002                  : fopen.c.obj (.bss)
                    0000046e    00000002     hello.c.obj (.bss)
.sysmem     0     00000470    00000120     UNINITIALIZED
                    00000470    00000004     rtsxxx .lib : memory.c.obj (.sysmem)
.stack      0     000008c0    00000140     UNINITIALIZED
                    000008c0    00000002     rtsxxx .lib : boot.c.obj (.stack)
```

如示例 8-1 所示，.bss 和 .sysmem 段被分配至 RAM 的较低地址 (0x0200 - 0x0590)，而 .stack 段被分配至地址 0x08c0（虽然较低的地址也可用）。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

如果不使用 HIGH 说明符，链接器分配将产生示例 8-2 中所示的代码

如果 HIGH 说明符与特定地址绑定操作或自动段拆分操作（>> 运算符）搭配使用，则会忽略该说明符。

示例 8-2. 在没有 HIGH 说明符的情况下进行链接器放置

```
.bss        0     00000200     00000270      UNINITIALIZED
                    00000200     0000011a      rtsxxx.lib   : defs.c.obj (.bss)
                    0000031a     00000088                   : trgdrv.c.obj (.bss)
                    000003a2     00000078                   : lowlev.c.obj (.bss)
                    0000041a     00000046                   : exit.c.obj (.bss)
                    00000460     00000008                   : memory.c.obj (.bss)
                    00000468     00000004                   : _lock.c.obj (.bss)
                    0000046c     00000002                   : fopen.c.obj (.bss)
                    0000046e     00000002      hello.c.obj (.bss)
.stack      0     00000470     00000140      UNINITIALIZED
                    00000470     00000002      rtsxxx.lib   : boot.c.obj (.stack)
.sysmem     0     000005b0     00000120      UNINITIALIZED
                    000005b0     00000004      rtsxxx.lib   : memory.c.obj (.sysmem)
```

8.5.5.2.5 对齐和分块

使用 align 关键字可以告诉链接器将输出段放置在位于 n 字节边界上的地址处，其中 n 是 2 的幂。例如，以下代码分配 .text，使其落在 32 字节边界上：

```
     .text: load = align(32)
```

分块是一种较弱的对齐形式，它将一个段分配到大小为 n 的块*内*的任何位置。指定的块大小必须是 2 的幂。例如，以下代码分配 .bss，使整个段包含在单个 128 字节的块中，或从该边界开始：

```
bss: load = block(0x0080)
```

对齐或分块可单独使用，也可与存储器区域结合使用，但对齐和分块不能同时使用。

8.5.5.2.6 对齐和填充

与 align 关键字一样，使用 palign 关键字可以告诉链接器将输出段放置在位于 n 字节边界上的地址处，其中 n 是 2 的幂。此外，palign 确保段的大小是其放置对齐限制值的倍数，并会根据需要将段大小填充到这样的边界。

例如，以下代码行在 PMEM 区域内的 2 字节边界上分配 .text。.text 段的大小需保证是 2 字节的倍数。以下两条语句是等效的：

```
    .text: palign(2) {} > PMEM
    .text: palign = 2 {} > PMEM
```

如果链接器向初始化的输出段添加填充，则填充空间也会被初始化。默认情况下，填充空间的填充值为 0（零）。但是，如果为输出段指定一个填充值，则该段的任何填充也将使用该填充值进行填充。例如，考虑以下段规范：

```
    .mytext: palign(8), fill = 0xffffffff {} > PMEM
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

在此示例中，在应用 palign 运算符之前，.mytext 段的长度为 6 个字节。.mytext 的内容如下：

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
```

应用 palign 运算符后，.mytext 的长度为 8 个字节，且其内容如下：

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
0006 0xffff
```

.mytext 的大小已提升到 8 字节的倍数，并且由链接器创建的填充已填充为 0xff。

链接器命令文件中指定的填充值被解释为 16 位常量。如果指定以下代码：

```
    .mytext: palign(8), fill = 0xff {} > PMEM
```

链接器假定的填充值为 0x00ff，然后 .mytext 将具有以下内容：

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
0006 0x00ff
```

如果 palign 运算符应用于未初始化的段，则该段的大小会根据需要提升到相应的边界，但不会初始化创建的任何填充。

palign 运算符也可以采用 *power2* 参数。此参数告诉链接器添加填充以将段的大小增加到下一个“2 的幂”边界。此外，该段也在这个“2 的幂”边界上对齐。例如，考虑以下段规范：

```
     .mytext: palign(power2) {} > PMEM
```

假设 .mytext 段的大小为 120 字节，PMEM 从地址 0x10020 开始。在应用 palign(power2) 运算符后，.mytext 输出段将具有以下属性：

```
  name      addr        size    align
-------   ----------   -----    -----
.mytext   0x00010080   0x80     128
```

8.5.5.3 指定输入段

输入段规范确定输入文件中组合在一起构成输出段的各段。一般而言，链接器会按照指定的顺序连接各输入段以将这些段组合在一起。不过，如果为输入段指定了对齐或阻塞，输出段内的所有输入段都会按如下方式排序：

- 所有对齐的段从大到小排列
- 所有阻塞的段从大到小排列
- 所有其他段从大到小排列

输出段的大小为所含各输入段的大小之和。

示例 8-3 显示了段规范的最常见类型；请注意，这里没有列出任何输入段。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

示例 8-3. 指定段内容的最常用方法

```
SECTIONS
{
   .text:
   .data:
   .bss:
}
```

在示例 8-3 中，链接器从输入文件中获取所有 .text 段，并将它们组合到 .text 输出段中。链接器按照其在输入文件中遇到 .text 输入段的顺序连接这些输入段。链接器对 .data 和 .bss 段执行类似的操作。用户可以将这种类型的规范用于任何输出段。

用户可以显式指定构成输出段的输入段。每个输入段由其文件名和段名标识。如果文件名带有连字符（或包含特殊字符），请将其放入引号中：

```
SECTIONS
{
  .text :              /* 构建 .text 输出段                           */
  {
    f1.c.obj(.text)      /* 链接来自 f1.c.obj 的 .text 段                     */
    f2.c.obj(sec1)       /* 链接来自 f2.c.obj 的 sec1 段                      */
    "f3-new.c.obj"       /* 链接来自 f3-new.c.obj 的所有段                  */
    f4.c.obj(.text,sec2) /* 链接来自 f4.c.obj 的 .text 段和 sec2 段              */
    f5.c.obj(.task??)    /* Link .task00, .task01, .taskXX, etc. from f5.c.obj */
    f6.c.obj(*_ctable)   /* 链接来自 f6.c.obj 并以 "_ctable" 结尾的各段     */
    X*.c.obj(.text)      /* 链接以   */
                       /*  "X" 开头并以 ".c.obj" 结尾的所有文件的 .text 段                    */
   }
}
```

输入段之间不必使用相同名称，也不必与它们所属的输出段具有相同名称。如果列出某个文件时未提供段，则其*所有*的段都将包含在输出段中。如果任何其他输入段与输出段同名，但未通过 SECTIONS 指令显式指定，则其将在输出段的末尾自动链接进来。例如，如果链接器在前面的示例中发现更多 .text 段，并且这些 .text 段*没有*在 SECTIONS 指令中的任何位置指定，则链接器将在 f4.c.obj(sec2) 之后连接这些额外的段。

示例 8-3 中的规范实际上是以下内容的简略表达：

```
SECTIONS
{
  .text: { *(.text) }
  .data: { *(.data) }
  .bss:  { *(.bss)  }
}
```

规范 *(.text) 表示*所有输入文件中未分配的 .text 段*。这种格式在以下情况下很有用：

- 用户希望输出段包含具有指定名称的所有输入段，但输出段名称与输入段的名称不同。
- 用户希望链接器在处理大括号内的其他输入段或命令之前分配输入段。

Copyright © 2023 Texas Instruments Incorporated

以下示例展示了上述两种用途：

```
SECTIONS
{
    .text  :  {
                  abc.c.obj(xqt)
                  *(.text)
              }
    .data  :  {
                  *(.data)
                  fil.c.obj(table)
              }
}
```

在此示例中，.text 输出段包含来自文件 abc.c.obj 的指定段 xqt，后跟所有 .text 输入段。.data 段包含所有 .data 输入段，后跟来自文件 fil.c.obj 的指定段表。此方法包括所有未分配的段。 例如，如果当链接器遇到 *(.text) 时其中一个 .text 输入段已包含在另一个输出段中，则链接器无法将第一个 .text 输入段添加到第二个输出段中。

每个输入段充当一个前缀，以收集名称更长的段。例如，模式 *(.data) 与 .dataspecial 匹配。因此，前缀启用子段（在下一节中将介绍这些子段）。

8.5.5.4 使用多级子段

基础段名与一个或多个子段名（由冒号分隔）可识别为子段。例如，名为 A:B 和 A:B:C 的子段为基础段 A 的子段。在链接器命令文件的特定位置会指定一个基础名称，例如 A，选择段 A 以及 A 的任何子段，例如 A:B 或 A:C:D。

名称（例如 A:B）可指定该名称的（子）段以及以该名称起始的（多级）子段，例如 A:B:C、A:B:OTHER 等。A:B 的所有子段同样也是 A 的子段。A 和 A:B 都是 A:B:C 的超段。在一个子段的一组超段中，最近的超段是名称最长的超段。因此，在 {A, A:B} 之间，A:B:C:D 最近的超段是 A:B。多级子段具有如下限制：

1. 在一个文件（或库单元）中指定**输入**段时，段名会选择同名输入段以及该名称的任何子段。
2. 未显式分配的输入段会在同名的现有**输出**段或这样的输出段的现有最近超段中分配。此规则的例外是，在部分链接（由 --relocatable 链接器选项指定）期间，子段只会分配到同名的现有输出段中。
3. 如果未定义 2) 中介绍的输出段，输入段将置于与输入段的基本名称同名的**新创建的输出**段中。

请考虑具有以下名称的链接输入段：

europe:north:norway
europe:north:sweden
europe:north:finland
europe:north:iceland
europe:central:france
europe:central:germany
europe:central:denmark
europe:south:spain
europe:south:italy
europe:south:malta

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

此 SECTIONS 规范分配输入段的方式如注释中所示：

```
SECTIONS {
  nordic:   {*(europe:north)
             *(europe:central:denmark)} /* the nordic countries */
  central: {*(europe:central)}         /* france, germany      */
  therest: {*(europe)}                 /* spain, italy, malta  */
}
```

此 SECTIONS 规范分配输入段的 方式如注释中所示：

```
SECTIONS {
  islands: {*(europe:south:malta)
             *(europe:north:iceland)}  /* malta, iceland   */
  europe:north:finland : {}            /* finland          */
  europe:north         : {}            /* norway, sweden   */
  europe:central       : {}            /* germany, denmark */
  europe:central:france: {}            /* france           */
  /* (italy, spain) go into a linker-generated output section "europe" */
}
```

备注

多级子段的向上兼容性

使用现有单级子段功能的现有链接器命令，以及不包含具有多个冒号字符的段名的命令，其运行方式与原来一样。但如果链接器命令文件中的段名，或提供给链接器的输入段中的段名包含多个冒号字符，运行方式可能会有所变化。您应仔细考虑多级规则产生的影响，看它是否会影响特定系统链接。

8.5.5.5 指定库或存档成员作为输出段的输入

您可以指定对象库或存档的一个或多个成员作为输出段的输入。可考虑以下 SECTIONS 指令：

示例 8-4. 将成员存档至输出段

```
SECTIONS
{
boot>BOOT1
{
-l rtsXX.lib<boot.c.obj> (.text)
-l rtsXX.lib<exit.c.obj strcpy.c.obj> (.text)
}
.rts>BOOT2
{
-l rtsXX.lib (.text)
}
.text>RAM
{
* (.text)
}
}
```

在示例 8-4 中，boot.c.obj、exit.c.obj 和 strcpy.c.obj 的 .text 段是从运行时支持库中提取的，并放置在 .boot 输出段中。引用的运行时支持库对象的其余部分将分配给 .rts 输出段。最后，所有其他 .text 段的其余部分将放置在 .text 段中。

为了指定一个存档成员或成员列表，需要在库名称后用尖括号 < 和 > 将成员名称括起来。指定的存档文件中以逗号或空格分隔的任何目标文件在尖括号内都是合法的。

在 < > 中列出特定存档成员时，在示例 8-4 中的每个库之前列出的 --library 选项（通常意味着对选项后面的指定文件进行库路径搜索）是可选项。使用 < > 意味着要引用某个库。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

若要在一个地方收集库中的一组输入段，请在 SECTIONS 指令中使用 --library 选项。例如，以下代码会将 rtsv4_A_be_eabi.lib 中的所有 .text 段收集到 .rtstest 段中：

```
SECTIONS
{
   .rtstest { -l rtsv4_A_be_eabi.lib(.text) } > RAM
}
```

备注

SECTIONS 指令对 --priority 的影响：在 SECTIONS 指令中指定某个库会导致这个库被输入到库列表中，该库列表由链接器进行搜索以解析引用。如果使用 --priority 选项，则会首先搜索在命令文件中指定的第一个库。

8.5.5.6 使用多个存储器范围进行分配

链接器可使用户指定一个显式的存储器范围列表，可以向其中分配输出段。考虑以下示例：

```
MEMORY
{
   P_MEM1 :  origin = 0x02000,  length = 0x01000
   P_MEM2 :  origin = 0x04000,  length = 0x01000
   P_MEM3 :  origin = 0x06000,  length = 0x01000
   P_MEM4 :  origin = 0x08000,  length = 0x01000
}
SECTIONS
{
   .text   : { } > P_MEM1 | P_MEM2 | P_MEM4
}
```

运算符 **|** 用于指定多个存储器范围。输出段 .text 作为一个整体分配给与之适应的第一个存储器范围。存储器范围按指定顺序访问。在此示例中，链接器首先会试在 P_MEM1 中分配段。如果该尝试失败，链接器会尝试将该段置于 P_MEM2 中，依此类推。如果未在任何指定的存储器范围中成功分配输出段，则链接器会发出错误消息。

借助这种类型的 SECTIONS 指令规范，链接器可以无缝处理超出最初分配的存储器范围可用空间的输出段。用户可以让链接器将段移动到其他某个区域，而不是修改链接器命令文件。

8.5.5.7 在非连续存储器范围之间自动拆分输出段

链接器可以在多个存储器范围之间拆分输出段以提高分配效率。使用 **>>** 运算符来指示如有必要，可将输出段拆分成指定的存储器范围：

```
MEMORY
{
   P_MEM1 :  origin = 0x2000,  length = 0x1000
   P_MEM2 :  origin = 0x4000,  length = 0x1000
   P_MEM3 :  origin = 0x6000,  length = 0x1000
   P_MEM4 :  origin = 0x8000,  length = 0x1000
}
SECTIONS
{
  .text: { *(.text) } >> P_MEM1 | P_MEM2 | P_MEM3 | P_MEM4
}
```

在此示例中，**>>** 运算符指示可在任何列出的存储器区域之间拆分 .text 输出段。如果 .text 段增长到超出 P_MEM1 中的可用存储器，则会在输入段边界上拆分 .text 段，而输出段的其余部分将分配给 P_MEM2 | P_MEM3 | P_MEM4。

| 运算符用于指定多个存储器范围的列表。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

还可以使用 >> 运算符来指示可在单个存储器范围内拆分输出段。当多个输出段必须分配到同一存储器范围，但一个输出段的限制会导致存储器范围被分区时，此功能会很有用。考虑以下示例：

```
MEMORY
{
   RAM :  origin = 0x1000,  length = 0x8000
}
SECTIONS
{
  .special: { f1.c.obj(.text) } load = 0x4000
  .text: { *(.text) } >> RAM
}
```

.special 输出段被分配在靠近 RAM 存储器范围中间的地方。这会在 RAM 中留下两个未使用的区域：从 0x1000 到 0x4000，以及从 f1.c.obj(.text) 的末尾到 0x8000。.text 段的规格允许链接器围绕 .special 段拆分 .text 段，并使用 RAM 中 .special 两侧的可用空间。

>> 运算符还可用于在匹配指定属性组合的所有存储器范围之间拆分输出段。例如：

```
MEMORY
{
  P_MEM1 (RWX) : origin = 0x1000,  length = 0x2000
  P_MEM2 (RWI) : origin = 0x4000,  length = 0x1000
}
SECTIONS
{
  .text: { *(.text) } >> (RW)
}
```

链接器会尝试将全部或部分输出段分配至属性与 SECTIONS 指令中指定的属性匹配的任何存储器范围。

此 SECTIONS 指令与以下指令具有相同的效果：

```
SECTIONS
{
.text: { *(.text) } >> P_MEM1 | P_MEM2}
}
```

某些段不应拆分：

- 编译器创建的某些段，包括
  - .cinit 段，其中包含 C/C++ 程序的自动初始化表
  - .pinit 段，其中包含 C++ 程序的全局构造函数列表
- 具有输入段规格（其中包含要计算的表达式）的输出段。该表达式可能会定义一个符号。该符号在程序中用于在运行时管理输出段。
- 应用了 START()、END() 或 SIZE() 运算符的输出段。这些运算符提供有关段的加载或运行地址以及大小的信息。拆分该段可能会损害运算的完整性。
- UNION 的运行分配。（允许拆分 UNION 的加载分配。）

如果对这些段中的任何一个段使用 >> 运算符，则链接器将发出警告并忽略该运算符。

8.5.6 在不同的加载和运行地址放置段

有时您可能希望将代码加载到存储器的一个区域，而在另一个区域运行。例如，慢速外部存储器中可能有对性能至关重要的代码。代码必须加载至慢速外部存储器，但在快速外部存储器中能够以更快的速度运行。

链接器提供了实现此目标的简单方式。您可以使用 SECTIONS 指令使链接器分配一个段两次：第一次设置其加载地址，第二次设置其运行地址。例如：

```
    .fir: load = SLOW_MEM, run = FAST_MEM
```

加载地址使用 *load* 关键字，运行地址使用 *run* 关键字。

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

DRAFT ONLY TI Confidential – NDA Restrictions

请参阅节 3.5，了解运行时重定址的概述。

应用必须将段从其加载地址复制到其运行地址；如果您另外指定了运行地址，这*不会*自动发生。（TABLE 操作符指示链接器生成复制表；请参阅节 8.8.4.1。）

8.5.6.1 指定加载和运行地址

加载地址决定了加载程序将段的原始数据放在何处。对该段的任何引用（例如其中的标号）都会引用其运行地址。请参阅节 3.1.1，以简要了解加载和运行地址。

如果您只为某个段提供了一个分配（加载或运行），该段只会分配一次，并会在相同的地址加载和运行。如果您提供了这两个分配，该段会被视作两个大小相同的段来进行分配。这意味着，这两个分配都会占用存储器映射中的空间，并且彼此不能叠加，也不能与其他段叠加。（UNION 提供了一种叠加段的方式；请参阅节 8.5.7.2。）

如果加载地址或运行地址包含额外的参数，例如对齐或阻塞，请在相应的关键字后列出。在遇到关键字 *load* 后，与分配相关的一切内容都会影响加载地址，直到遇到关键字 *run*，在此之后，一切内容都会转而影响运行地址。加载和运行分配完全独立，因此其中任何一个的资格（例如对齐）都不会对另一个产生影响。您还可以先指定运行地址，再指定加载地址。可以使用括号来提高可读性。

下面的示例指定了加载和运行地址。

本例中，对齐只会应用到加载：

```
    .data: load = SLOW_MEM, align = 32, run = FAST_MEM
```

下例中使用了括号，但作用与上例完全相同：

```
    .data: load = (SLOW_MEM align 32), run = FAST_MEM
```

下例会针对运行分配将 FAST_MEM 对齐到 32 位并将所有加载分配对齐到 16 位：

```
    .data: run  = FAST_MEM, align 32, load = align 16
```

有关运行时重定位的更多信息，请参阅节 3.5。

未初始化的段（例如 .bss）不会被加载，因此它们唯一重要的地址是运行地址。链接器只分配一次未初始化的段：如果您同时指定运行地址和加载地址，链接器会向您发出警告并忽略加载地址。另一方面，如果只指定一个地址，无论您称其为加载地址还是运行地址，链接器都会将其视为运行地址。

以下示例指定了未初始化的段的加载和运行地址：

```
    .bss: load = 0x1000, run = FAST_MEM
```

链接器会发出警告，忽略加载，并在 FAST_MEM 中分配空间。以下所有示例都具有相同的效果。.bss 段将分配到 FAST_MEM 中。

```
    .dbss: load = FAST_MEM
    .bss: run = FAST_MEM
    .bss: > FAST_MEM
```

8.5.6.2 使用 .label 指令引用加载地址

通常情况下，引用符号时会引用其运行时地址。但在运行时可能需要引用加载时地址。具体来说，将段从其加载地址复制到其运行地址的代码必须能够访问加载地址。.label 指令可定义一个引用段的加载地址的特殊符号。因此，如果正常符号相对于运行地址进行重定位，.label 符号也会相对于加载地址进行重定位。请参阅创建加载时地址标签，了解有关 .label 指令的更多信息。

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

在运行时将一个函数从慢速存储器移动到快速存储器和 的链接器命令文件展示了 .label 指令的用法：将段从 SLOW_MEM 中的加载地址复制到 FAST_MEM 中的运行地址。图 8-3 演示了在运行时将一个函数从慢速存储器移动到快速存储器的运行时执行。

如果使用表操作符，则不需要 .label 指令。请参阅节 8.8.4.1。

在运行时将一个函数从慢速存储器移动到快速存储器

```
;-----------------------------------------------------------
;   define a section to be copied from SLOW_MEM to FAST_MEM
;-----------------------------------------------------------
        .sect  ".fir"
        .label fir_src      ; load address of section
fir:                        ; run address of section
        <code here>         ; code for section
        .label fir_end      ; load address of section end
;-----------------------------------------------------------
;   copy .fir section from SLOW_MEM to FAST_MEM
;-----------------------------------------------------------
        .text
        LDR    r4, fir_s    ; get fir load address start
        LDR    r5, fir_e    ; get fir load address stop
        LDR    r3, fir_a    ; get fir run address
$1:     CMP    r4, r5
        LDRCC  r0, [r4], #4 ; copy fir routine to its
                            ; run address
        STRCC  r0, [r3], #4
        BCC    $1
;-----------------------------------------------------------
;   jump to fir routine, now in FAST_MEM
;-----------------------------------------------------------
        B      fir
fir_a   .word  fir
fir_s   .word  fir_start
fir_e   .word  fir_end
```

在运行时将一个函数从慢速存储器移动到快速存储器 的链接器命令文件

```
/*******************************************************/
/*    FIR 示例的部分链接器命令文件      */
/*******************************************************/
MEMORY
{
   FAST_MEM :  origin = 0x00001000, length = 0x00001000
   SLOW_MEM :  origin = 0x10000000, length = 0x00001000
}
SECTIONS
{
   .text: load = FAST_MEM
   .fir:  load = SLOW_MEM, run FAST_MEM
}
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

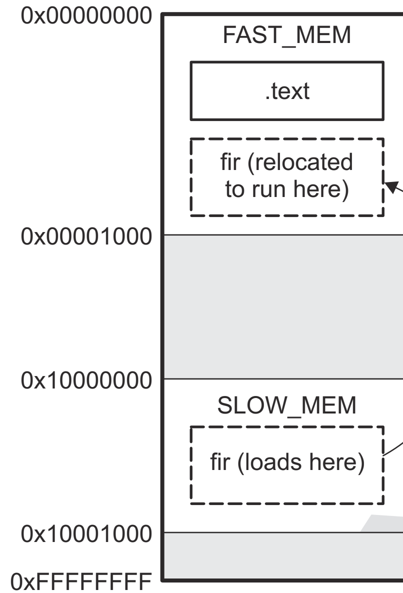


图 8-3. 在运行时将一个函数从慢速存储器移动到快速存储器 **的运行时执行**

有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 8.6。

8.5.7 使用 GROUP 和 UNION 语句

两个 SECTIONS 语句可用于组织或保留存储器：GROUP 和 UNION。将段分组可使链接器在存储器中连续分配这些段。将段合并可使链接器为它们分配相同的运行地址。

8.5.7.1 将输出段一同进行分组

除非使用 UNORDERED 运算符，否则 SECTIONS 指令的 GROUP 选项会强制按列出的顺序连续分配多个输出段。例如，假设一个名为 term_rec 的段包含 .data 段中某个表的终止记录。可强制链接器将 .data 和 term_rec 分配在一起：

将段分配在一起

```
SECTIONS
{
   .text              /*  正常的输出段                  */
   .bss               /*  正常的输出段                  */
   GROUP 0x00001000 : /*  指定一组段      */
   {
      .data           /*  组中的第一个段           */
      term_rec        /*  在 .data 之后立即分配 */
   }
}
```

可使用绑定、对齐或指定的存储器通过与单个输出段相同的方式分配 GROUP。在前面的示例中，GROUP 绑定到地址 0x1000。这意味着 .data 分配到存储器中的 0x1000 位置，而 term_rec 紧跟其后。

备注

不能为 GROUP 内的段指定地址：使用 GROUP 选项时，只能为组指定绑定、对齐或分配到指定的存储器中。不能对组内的段使用绑定、指定的存储器或对齐。

8.5.7.2 利用 UNION 语句叠加段

对于某些应用，用户可能希望在运行时期间分配占用同一地址的多个段。例如，用户可能希望在不同执行阶段使用快速外部存储器中的若干例程。用户也可能希望若干不同时处于活动状态的数据对象共享一个存储器块。SECTIONS 指令中的 UNION 语句可在同一运行时地址分配多个段。

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY
TI Confidential – NDA Restrictions

在 UNION 语句 中（ ），file1.c.obj 和 file2.c.obj 的 .bss 段在 FAST_MEM 中的同一地址进行分配。在存储器映射中，union 占用的空间为其最大组件所占的空间。union 的组件仍是独立段；它们只是作为一个单元一同被分配。

UNION 语句

```
SECTIONS
{
   .text: load = SLOW_MEM
   UNION: run  = FAST_MEM
   {
      .bss:part1: { file1.c.obj(.bss) }
      .bss:part2: { file2.c.obj(.bss) }
   }
      .bss:part3: run = FAST_MEM { globals.c.obj(.bss) }
}
```

分配一个作为 union 一部分的段仅影响其*运行地址*。加载时的段*永远不会*叠加。如果经过初始化的段是 union 成员（经过初始化的段具有原始数据，例如 .text），则*必须*分别指定其负载分配。请参阅 UNION 段的单独加载地址。（在将经过初始化的段与未经初始化的段相结合时，此规则有一个例外；请参阅节 8.5.7.3。）

UNION 段的单独加载地址

```
UNION run = FAST_MEM
{
   .text:part1: load = SLOW_MEM, { file1.c.obj(.text) }
   .text:part2: load = SLOW_MEM, { file2.c.obj(.text) }
}
```

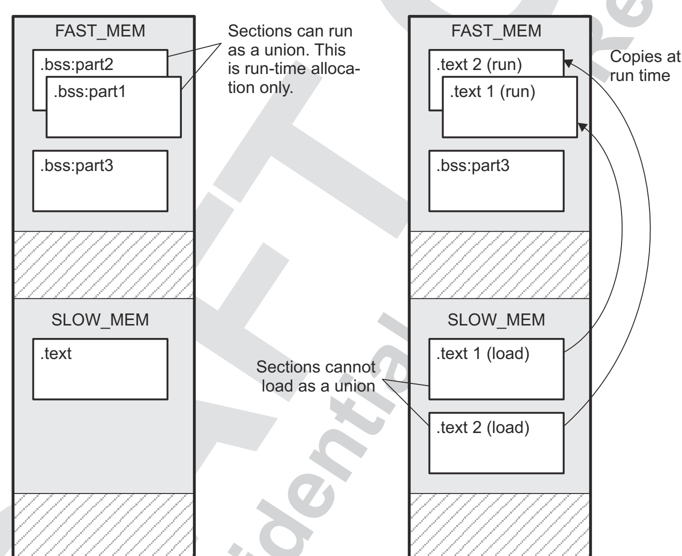


图 8-4. 存储器分配如 UNION 语句和 UNION 段的单独加载地址所示

.text 段包含原始数据，因此无法作为 union *加载*，但可以作为 union *运行*。因此，每个 .text 段都需要有自己的加载地址。如果用户未能为 UNION 中的初始化段提供加载分配，链接器会发出警告，并将所配置存储器中的任意位置作为加载空间分配。

未初始化的段未加载，不需要加载地址。

UNION 语句仅适用于运行地址的分配，因此无需为 union 本身指定加载地址。如用于分配，union 可被视为未初始化的段：指定的任何分配均被视为运行地址，如果同时指定了运行和加载地址，链接器会发出警告，并忽略加载地址。

Copyright © 2023 Texas Instruments Incorporated

8.5.7.3 将存储器用于多种用途

减少应用对存储器需求的一种方式是将存储器的一部分空间用于多种用途。用户可以首先使用存储器中的一部分空间进行系统初始化和启动。该阶段完成后，相同的存储器空间可改变用途，作为一组未初始化的数据变量或一个堆。若要实施此方案，请使用以下 UNION 语句的变化形式，它允许初始化一个段，而使其他段保留未初始化状态。

通常，一个 union 中经过初始化的段（具有原始数据，例如 .text）必须拥有其单独指定的负载分配。但一个 union 中有且只有一个初始化的段可分配至该 union 的运行地址。在 UNION 语句中列出，但没有负载分配，该段就会使用该 union 的运行地址作为它自己的负载地址。

例如：

```
UNION run = FAST_MEM
{ .cinit .bss }
```

在本例中，.cinit 段是初始化的段。它会在 union 的运行地址被加载到 FAST_MEM 中。相反，.bss 是一个未初始化的段。它的运行地址也是该 union 的运行地址。

8.5.7.4 嵌套 UNION 和 GROUP

链接器允许使用 SECTIONS 指令对 GROUP 和 UNION 语句进行任意嵌套。通过嵌套 GROUP 和 UNION 语句，可以表示段的分层重叠和分组。嵌套 GROUP 和 UNION 语句显示了如何将两个重叠层组合在一起。

嵌套 GROUP 和 UNION 语句

```
SECTIONS
{
   GROUP 0x1000 : run = FAST_MEM
   {
      UNION:
      {
         mysect1: load = SLOW_MEM
         mysect2: load = SLOW_MEM
      }
      UNION:
      {
         mysect3: load = SLOW_MEM
         mysect4: load = SLOW_MEM
      }
   }
}
```

对于此示例，链接器执行以下分配：

- 四个段（mysect1、mysect2、mysect3、mysect4）被分配唯一且不重叠的加载地址。由 .label 指令定义的名称将用于 SLOW_MEM 存储器区域。该分配取决于为每个段提供的特定加载分配。
- mysect1 和 mysect2 段会在 FAST_MEM 中被分配相同的运行地址。
- mysect3 和 mysect4 段会在 FAST_MEM 中被分配相同的运行地址。
- mysect1/mysect2 和 mysect3/mysect4 的运行地址按照 GROUP 语句的指示进行连续分配（受限于对齐和分块限制）。

为了引用组和联合体，链接器诊断消息使用以下表示法：

GROUP_*n* UNION_*n*

其中 *n* 是一个序号（从 1 开始），表示链接器控制文件中该组或联合体的词法顺序，不考虑嵌套。组和联合体都自带计数器。

8.5.7.5 检查分配器的一致性

链接器会检查为联合体、组和段指定的加载和运行分配的一致性。使用的规则如下：

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

- 仅允许对顶级段、组或联合体（即未嵌套在任何其他组或联合之下的段、组或联合体）运行分配。链接器使用顶级结构的运行地址来计算组和联合体内组件的运行地址。
- 对于联合体，链接器不接受加载分配。
- 对于未初始化的段，链接器不接受加载分配。
- 在大多数情况下，必须为已初始化的段提供加载分配。但是，如果一个已初始化的段位于已定义加载分配器的组内，对于该段，链接器不接受加载分配。
- 作为一种捷径，您可以为整个组指定加载分配，从而确定嵌套在该组中的每个已初始化段或子段的加载分配。但是，仅当以下所有条件都成立时，才接受整个组的加载分配：
  - 该组已初始化（即，它至少有一个已初始化的成员）。
  - 该组未嵌套在具有加载分配器的另一个组中。
  - 该组不包含具有已初始化段的联合体。
- 如果该组包含一个具有已初始化段的联合体，则需要为嵌套在该组中的每个已初始化段指定加载分配。考虑以下示例：

```
SECTIONS
{
  GROUP: load = SLOW_MEM, run = SLOW_MEM
  {
   .text1:
   UNION:
   {
    .text2:
    .text3:
   }
  }
}
```

为组提供的加载分配器不唯一指定联合体中元素（.text2 和 .text3）的加载分配。在这种情况下，链接器会发出诊断消息以请求显式指定这些加载分配。

8.5.7.6 为 UNION 和 GROUP 命名

可通过在位于声明之后的圆括号中输入名称来为 UNION 或 GROUP 命名。例如：

```
GROUP(BSS_SYSMEM_STACK_GROUP)
{
   .bss    :{}
   .sysmem :{}
   .stack  :{}
} load=D_MEM,  run=D_MEM
```

您定义的名称将用于在诊断中轻松识别问题 LCF 区域。例如：

```
warning: LOAD placement ignored for "BSS_SYSMEM_STACK_GROUP": object is uninitialized
   UNION(TEXT_CINIT_UNION)
   {
      .const :{}load=D_MEM, table(table1)
      .pinit :{}load=D_MEM, table(table1)
   }run=P_MEM
warning:table(table1) operator ignored: table(table1) has already been applied to a section
in the "UNION(TEXT_CINIT_UNION)" in which ".pinit" is a descendant
```

8.5.8 特殊段类型（DSECT、COPY、NOLOAD 和 NOINIT）

您可以为输出段分配以下特殊类型：DSECT、COPY、NOLOAD 和 NOINIT。这些类型会影响链接和加载程序时处理程序的方式。您可以通过在段定义后加上类型来为段分配类型。例如：

```
SECTIONS
   {
   sec1: load = 0x00002000, type =  DSECT  {f1.c.obj}
   sec2: load = 0x00004000, type =  COPY   {f2.c.obj}
   sec3: load = 0x00006000, type =  NOLOAD {f3.c.obj}
```


English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

DRAFT ONLY Confidential – NDA Restrictions

```
    sec4: load = 0x00008000, type =  NOINIT {f4.c.obj}
    }
```

- DSECT 类型会创建具有以下特性的虚拟段：
  - 它不会包含在输出段存储器分配中。它不会占用存储器，也不会包含在存储器映射列表中。
  - 它可以与其他输出段、其他 DSECT 和未配置的存储器重叠。
  - 虚拟段中定义的全局符号会正常进行重定位。它们会出现输出模块的符号表中，并且值与实际加载 DSECT 时原本该有的值相同。其他输入段可以引用这些符号。
  - 如果在 DSECT 中找到任何未定义的外部符号，则会搜索指定的归档库。
  - 该段的内容、重定位信息以及行号信息不会放到输出模块中。

  在上述示例中，f1.c.obj 中的段都不会被分配，但所有符号都会进行重定位，就像这些段在地址 0x2000 处进行链接。其他段可以引用 sec1 中的任何全局符号。
- COPY 段与 DSECT 段相似，不同的是它的内容及关联信息会被写入输出模块。包含 ARM C/C++ 编译器初始化表的 .cinit 段在运行时初始化模型下具有此属性。
- NOLOAD 段在一点上与普通输出段不同，那就是：该段的内容、重定位信息和行号信息不会放到输出模块中。链接器会为该段分配空间，并且它会出现在存储器映射列表中。
- NOINIT 段不会由链接器在 C 自动初始化过程中进行初始化。您需要自行根据需要对此段进行初始化。

8.5.9 使用链接器配置纠错码 (ECC)

可通过链接器命令文件生成纠错码 (ECC) 并将其放置在单独的段中。ECC 使用额外的位来允许器件检测和/或纠正错误。若要启用 ECC 生成功能，必须在命令行中包含 **--ecc=on** 链接器选项。默认情况下，ECC 生成功能已关闭，即使在链接器命令文件中使用了 ECC 指令和 ECC 限定符也是如此。因此，您可以在链接器命令文件中完全配置 ECC，同时仍然能够通过命令行快速打开和关闭代码生成功能。

链接器提供的 ECC 支持与各种 TI 器件上 TI 闪存中的 ECC 支持兼容。TI 闪存使用修改后的汉明码 (72,64)，该代码为每 64 位使用 8 个奇偶校验位。请检查闪存相关文档以查看是否支持 ECC。（用于读写存储器的 ECC 在运行时完全在硬件中处理。）

可使用存储器映射中的 ECC 限定符（节 8.5.9.1）和 ECC 指令（节 8.5.9.2）来控制 ECC 生成的详细信息。

请参阅节 8.4.12，了解哪些命令行选项会将位错误引入具有相应 ECC 段的代码或引入 ECC 奇偶校验位本身。使用这些选项可以测试 ECC 错误处理代码。

可在链接期间生成 ECC。ECC 数据随代码和数据一同包含在生成的目标文件中，作为位于相应地址的数据段。编译后不需要额外的 ECC 生成步骤，并且 ECC 可以与其他所有内容一同上传至器件。

8.5.9.1 在存储器映射中使用 ECC 说明符

若要生成 ECC，请向存储器映射添加一个单独的存储器范围，用于存放 ECC 数据并指示哪个存储器范围包含与此 ECC 数据对应的闪存数据。如果闪存数据有多个存储器范围，用户应为每个闪存数据范围分别添加 ECC 存储器范围。

ECC 存储器范围的定义也可为生成 ECC 数据的方式提供参数。

支持闪存 ECC 的器件的存储器映射与如下示例类似：

```
MEMORY {
   VECTORS  : origin=0x00000000 length=0x000020
   FLASH0   : origin=0x00000020 length=0x17FFE0
   FLASH1   : origin=0x00180000 length=0x180000
   STACKS   : origin=0x08000000 length=0x000500
   RAM      : origin=0x08000500 length=0x03FB00
   ECC_VEC  : origin=0xf0400000 length=0x000004 ECC={ input_range=VECTORS }
   ECC_FLA0 : origin=0xf0400004 length=0x02FFFC ECC={ input_range=FLASH0  }
   ECC_FLA1 : origin=0xf0430000 length=0x030000 ECC={ input_range=FLASH1  }
}
```


English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

ECC 存储器范围的规范语法如下：

```
MEMORY {
   <memory specifier1> : <memory attributes> [ vfill=<fill value> ]
   <memory specifier2> : <memory attributes> ECC = {
       input_range = <memory specifier1>
       [ algorithm   = <algorithm name> ]
       [ fill        = [ true, false ]  ]
   }
}
```

附加到 ECC 存储器范围的“ECC”说明符指示 ECC 范围涵盖的数据存储器范围。ECC 说明符支持以下参数：

input_range = *<range>*	此 ECC 数据范围涵盖的数据存储器范围。必需。
algorithm = *<ECC alg name>*	稍后在命令文件中使用 ECC 指令定义的 ECC 算法名称。如果只定义了一种算法，此参数则为可选项。（请参阅节 8.5.9.2。）
fill = true | false	对于输入范围的初始化数据中的空洞，是否生成 ECC 数据。默认值为“true”。使用 fill=false 生成的行为与 nowECC 工具类似。输入范围可正常填充，或使用虚拟填充（请参阅节 8.5.9.3）。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.5.9.2 使用 ECC 指令

除了在存储器映射中指定 ECC 存储器范围，链接器命令文件必须为生成 ECC 数据的算法指定参数。如果有多个闪存器件，可能需要多个 ECC 算法规范。

支持闪存 ECC 的每个 TI 器件对于这些参数均有一组精确的有效值。Code Composer Studio 提供的链接器命令文件包括在闪存上提供 ECC 支持所必要的 ECC 参数，可由器件访问。此处提供的文档旨在确保完整性。

您在链接器命令文件中使用顶级 ECC 指令来指定算法参数。规范语法如下：

```
ECC {
   <algorithm name> : parity_mask  = <8-bit integer>
                      mirroring    = [ F021, F035 ]
                      address_mask = <32-bit mask>
}
```

例如：

```
MEMORY {
   FLASH0 : origin=0x00000020 length=0x17FFE0
   ECC_FLA0 : origin=0xf0400004 length=0x02FFFC ECC={ input_range=FLASH0 algorithm=F021 }
}
ECC { F021 : parity_mask = 0xfc
             mirroring = F021 }
```

此 ECC 指令接受以下属性：

algorithm_name	指定希望用来引用算法的名称。
address_mask = *<32-bit mask>*	此掩码确定每个 64 位存储器段的地址中的哪些位会用于计算该存储器的 ECC 字节。默认为 0xffffffff，使用了该地址的所有位。（请注意，ECC 算法本身会忽略最低位，对于正确对齐的输入块来说，最低位始终为零。）
parity_mask = *<8-bit mask>*	此掩码确定哪些 ECC 位会编码偶校验，哪些位会编码奇校验。默认为 0，即所有位均编码偶校验。
mirroring = F021 | F035	此设置确定 ECC 字节的顺序，及其冗余复制模式。默认为 F021。

8.5.9.3 在存储器映射中使用 VFILL 限定符

通常情况下，为 MEMORY 范围指定填充值可创建初始化数据段，以涵盖之前未初始化的存储器区域。若要为整个存储器范围生成 ECC 数据，链接器的整个范围中需要有初始化数据，或者需要知道在运行时未初始化的存储器区域将有何种值。

如果要为整个存储器范围生成 ECC，但不希望通过指定填充值来初始化整个范围，可以使用“vfill”限定符来代替“fill”限定符，从而以虚拟方式填充范围：

```
MEMORY {
   FLASH : origin=0x0000  length=0x4000  vfill=0xffffffff
}
```

vfill 限定符的功能与省略 fill 限定符相当，只是它允许针对未初始化的输入存储器范围区域生成 ECC 数据。它的优势在于可减少生成的目标文件的大小。

vfill 限定符只对 ECC 数据生成产生影响。它无法与 fill 限定符同时指定，因为会产生歧义。

如果在 ECC 限定符中指定了 fill，但未指定 vfill，则 vfill 默认为 0xff。

8.5.10 在链接时分配符号

链接器赋值语句允许定义外部（全局）符号并在链接时为其赋值。此功能可用于初始化变量，或针对取决于赋值的值来初始化其指针。有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 8.6。

8.5.10.1 赋值语句的语法

链接器中赋值语句的语法与 C 语言中赋值语句的语法类似：

Copyright © 2023 Texas Instruments Incorporated

symbol	=	*expression;*	为符号分配表达式的值
symbol	+ =	*expression;*	将表达式的值加到符号
symbol	- =	*expression;*	从符号减去表达式的值
symbol	* =	*expression;*	将符号和表达式相乘
symbol	/ =	*expression;*	符号除以表达式

符号应是在外部定义的。如果不是，链接器会定义一个新符号，并将其加入符号表。表达式必须遵循节 8.5.10.3 中定义的规则。赋值语句*必须*以分号结束。

链接器在分配所有输出段*后*处理赋值语句。因此，如果表达式包含一个符号，该符号使用的地址会反映可执行输出文件中该符号的地址。

例如，假设程序从由两个外部符号标识的两个表（表 1 和表 2）中的一个读取数据。程序使用 cur_tab 符号作为当前表的地址。cur_tab 符号必须指向表 1 或表 2。用户可以在汇编代码中实现此目标，但需要重新汇编程序以更改表。相反，用户可以使用链接器赋值语句在链接时分配 cur_tab：

```
prog.c.obj           /* 输入文件                          */
cur_tab = Table1;   /* 将 cur_tab 分配至其中一个表 */
```

8.5.10.2 *向符号分配 SPC*

一个由点 (**.**) 表示的特殊符号，表示分配期间段程序计数器 (SPC) 的当前值。SPC 会跟踪某个段中的当前位置。链接器的 . 符号类似于汇编器的 $ 符号。 **.** 符号只能在 SECTIONS 指令的赋值语句中使用，因为 . 仅在分配期间有意义，并且 SECTIONS 会控制分配过程。（请参阅节 8.5.5。）

. 符号指段的当前运行地址，而不是当前加载地址。

例如，假设一个程序需要知道 .data 段开头的地址。通过使用 .global 指令（请参阅识别全局符号），可在程序中创建一个名为 Dstart 的外部未定义变量。然后，向 Dstart 赋予 **.** 的值：

```
SECTIONS
{
   .text:    {}
   .data:    {Dstart = .;}
   .bss :    {}
}
```

此代码会将 Dstart 定义为 .data 段的第一个已链接地址。（在分配 .data *之前* 先分配 Dstart。）链接器会重新定位对 Dstart 的所有引用。

一种特殊类型的赋值语句为 **.**（点）符号赋值。这会调整输出段内的 SPC 并在两个输入段之间创建一个空洞。分配给 **.** 以创建空洞的任何值都是相对于段的开头，而不是相对于 **.** 符号实际表示的地址。空洞和 **.** 的赋值语句详见节 8.5.11。

8.5.10.3 *赋值表达式*

链接器表达式适用以下规则：

- 表达式可以包含全局符号、常量和表 8-11 中列出的 C 语言运算符。
- 所有数字都被视为长（32 位）整数。
- 链接器使用与汇编器相同的方式来识别常量。也就是说，除非有后缀（H 或 h 表示十六进制，Q 或 q 表示八进制），否则数字将识别为十进制。还会识别 C 语言前缀（0 表示八进制，0x 表示十六进制）。十六进制常量必须以数字开头。不允许使用二进制常量。
- 表达式中的符号只有符号*地址* 的值。不会执行类型检查。
- 链接器表达式可以是绝对的或可重定位的表达式。如果表达式包含*任何* 可重定位符号（以及 0 个或多个常量或绝对符号），则该表达式是可重定位的表达式。否则，就是绝对表达式。如果一个符号被赋予了可重定位表达式的值，则该符号是可重定位的符号；如果被赋予了绝对表达式的值，则是绝对符号。

Copyright © 2023 Texas Instruments Incorporated

链接器支持表 8-11 中按优先顺序列出的 C 语言运算符。同一组中的运算符具有相同的优先级。除了表 8-11 中列出的运算符之外，链接器还有一个 align 运算符，该运算符允许符号在输出段内的 n 字节边界上对齐（n 是 2 的幂）。例如，以下表达式使当前段内的 SPC 在下一个 16 字节边界上对齐。align 运算符是当前 SPC 的函数，因此它只能在与 . 相同的上下文中使用，也就是说，在 SECTIONS 指令中使用。

```
    .= align(16);
```

表 8-11. 表达式中使用的运算符组（优先级）

组 1（最高优先级）		组 6	
!	逻辑非	且	按位与
~	按位非		
-	否定		
组 2		**组 7**	
*	乘法	|	按位或
/	除法		
%	模数		
组 3		**组 8**	
+	加法	&&	逻辑与
-	减法		
组 4		**组 9**	
>>	算术右移	||	逻辑或
<<	算术左移		
组 5		**组 10（最低优先级）**	
==	等于	=	赋值
! =	不等于	+ =	A + = B 等效于 A = A + B
>	大于	- =	A - = B 等效于 A = A - B
<	小于	* =	A * = B 等效于 A = A * B
< =	小于等于	/ =	A / = B 等效于 A = A / B
> =	大于等于		

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.5.10.4 由链接器自动定义的符号

.text	指定 .text 输出段的第一个地址。 （标志着可执行代码的*开始*。）
etext	指定 .text 输出段后的第一个地址。 （标志着可执行代码的*结束*。）
.data	指定 .data 输出段的第一个地址。 （标志着已初始化数据表的*开始*。）
edata	指定 .data 输出段后的第一个地址。 （标志着已初始化数据表的*结束*。）
.bss	指定 .bss 输出段的第一个地址。 （标志着未初始化数据的*开始*。）
end	指定 .bss 输出段后的第一个地址。 （标志着未初始化数据的*结束*。）

如果使用 --ram_model 或 --rom_model 选项，链接器会自动定义以下符号，以支持 C/C++。

__TI_STACK_SIZE	指定 .stack 段的大小。
__TI_STACK_END	指定 .stack 段的结束。
__TI_SYSMEM_SIZE	指定 .sysmem 段的大小。

如果使用 .global 指令声明，可在任何汇编语言模块中访问这些链接器定义的符号（请参阅识别全局符号）。

有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 8.6。

8.5.10.5 向符号分配一个段的确切开始值、结束值和大小值

代码生成工具目前支持在存储器的一个（慢）区域加载程序代码而在另一个（更快）区域运行程序代码的功能。实现此目标的方法是为链接器命令文件中的输出段或组指定单独的加载和运行地址。然后执行一系列指令（在运行时将一个函数从慢速存储器移动到快速存储器中的复制代码），从而预先将程序代码从其加载区域移动到其运行区域以满足需要。

在使用此功能设置系统时，程序员必须承担一些责任。其中一项责任是确定要移动的程序代码的大小和运行时地址。当前执行此过程的机制涉及在复制代码中使用 .label 指令。在运行时将一个函数从慢速存储器移动到快速存储器中展示了一个简单示例。

这种指定程序代码大小和加载地址的方法存在局限性。虽然适用于完全包含在单个源文件中的单个输入段，但如果程序代码分布在多个源文件中，或者如果程序员想要将整个输出段从加载空间复制到运行空间，那么这种方法就会变得更加复杂。

这种方法存在的另一个问题是，它没有考虑到移动的段可能有一个关联的 far 调用 trampoline 段需要随之移动。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.5.10.6 为什么点运算符有时不起作用

点运算符 (.) 用于在链接时用输出段中的特定地址定义符号。它的解释方式与 PC 类似。无论当前段中的当前偏移量是多少，都是与点关联的值。考虑在 SECTIONS 指令内使用输出段规范：

```
outsect:
{
        s1.c.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.c.obj(.text)
        end_of_s2 = .;
}
```

此语句创建了三个符号：

- end_of_s1 — s1.c.obj 中 .text 的结束地址
- start_of_s2 — s2.c.obj 中 .text 的开始地址
- end_of_s2 — s2.c.obj 中 .text 的结束地址

假设 s1.c.obj 和 s2.c.obj 之间由于对齐而产生了边界填充。那么 start_of_s2 并不是 .text 段真正的起始地址，但却是在 s2.c.obj 中对齐 .text 段所需的边界填充之前的地址。这是由于链接器将点运算符解释为当前 PC。这也是成立的，因为点运算符是独立于其周围的输入段进行评估的。

以上示例中的另一个潜在问题是 end_of_s2 可能不会考虑输出段结尾所需的任何边界填充。end_of_s2 不能可靠地作为输出段的结尾地址。避开这个问题的一种方式是，接在有问题的输出段之后创建一个虚拟段。例如：

```
GROUP
{
   outsect:
   {
      start_of_outsect = .;
      ...
   }
      dummy: { size_of_outsect = .- start_of_outsect; }
}
```

8.5.10.7 地址和维度运算符

使用以下六个运算符可以定义加载时和运行时地址和大小的符号：

LOAD_START(*sym* **)**
START(*sym* **)** 用相关分配单元的加载时起始地址定义 *sym*

LOAD_END(*sym* **)**
END(*sym* **)** 用相关分配单元的加载时结束地址定义 *sym*

LOAD_SIZE(*sym* **)**
SIZE(*sym* **)** 用相关分配单元的加载时大小定义 *sym*

RUN_START(*sym* **)** 用相关分配单元的运行时起始地址定义 *sym*

RUN_END(*sym* **)** 用相关分配单元的运行时结束地址定义 *sym*

RUN_SIZE(*sym* **)** 用相关分配单元的运行时大小定义 *sym*

LAST(*sym* **)** 用相关存储器范围中上次分配的字节的运行时地址定义 *sym*。

备注

链接器命令文件运算符等效性：LOAD_START() 和 START() 是等效的，LOAD_END()/END() 和 LOAD_SIZE()/SIZE() 也是如此。为清楚起见，建议使用 LOAD 名称。

这些地址和维度运算符可与几种不同类型的分配单元（包括输入项、输出段、GROUP 和 UNION）相关联。以下几节提供了一些示例来说明如何在每种情况下使用运算符。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

链接器定义的这些符号可在运行时通过 _symval 运算符予以访问，这本质上是一个强制转换操作。例如，假设链接器命令文件包含以下内容：

```
.text: RUN_START(text_run_start), RUN_SIZE(text_run_size) { *(.text) }
```

您的 C 程序可以按如下方式访问这些符号：

```
extern char text_run_start, text_run_size;
printf(".text load start is %lx\n", _symval(&text_run_start));
printf(".text load size is %lx\n",  _symval(&text_run_size));
```

更多有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 8.6。

8.5.10.7.1 输入项

考虑在 SECTIONS 指令内使用输出段规范：

```
outsect:
{
        s1.c.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.c.obj(.text)
        end_of_s2 = .;
}
```

这可以通过使用 START 和 END 运算符重写为以下格式：

```
outsect:
{
        s1.c.obj(.text) { END(end_of_s1) }
        s2.c.obj(.text) { START(start_of_s2), END(end_of_s2) }
}
```

end_of_s1 和 end_of_s2 的值将会相同，就像在原始示例中使用了点运算符一样，但 start_of_s2 将在两个 .text 段之间需要添加的任何必需填充之后进行定义。请记住，点运算符会导致 start_of_s2 在两个输入段之间插入的任何必需填充之前进行定义。

将这些运算符与输入段相关联的语法需要使用大括号 { } 将运算符列表括起来。列表中的运算符应用于紧接在列表之前出现的输入项。

8.5.10.7.2 输出段

START、END 和 SIZE 操作符也可与输出段关联。下面我们举例说明：

```
outsect: START(start_of_outsect), SIZE(size_of_outsect)
{
        <list of input items>
}
```

在此例中，SIZE 操作符定义 size_of_outsect 在输出段中加入必要的边界填充，以符合任何对齐要求。

为输出段指定操作符的语法不需要用括号括住操作符列表。操作符列表只需包含在输出段的分配规范中。

8.5.10.7.3 GROUP

以下是 START 和 SIZE 运算符在 GROUP 规范上下文中的另一种用法：

```
GROUP
{
        outsect1: { ...}
        outsect2: { ...}
} load = ROM, run = RAM, START(group_start), SIZE(group_size);
```


English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential – NDA Restrictions

如果要在一个位置加载整个 GROUP 并在另一个位置运行，这会很有用。复制代码可以使用 group_start 和 group_size 作为复制起始地址和复制大小参数。这样一来就没必要在源代码中使用 .label。

8.5.10.7.4 UNION

RUN_SIZE 和 LOAD_SIZE 操作符提供了一种机制，区别 UNION 负载空间的大小及其组成部分在运行前将要复制到的空间的大小。下面我们举例说明：

```
UNION: run = RAM, LOAD_START(union_load_addr),
       LOAD_SIZE(union_ld_sz), RUN_SIZE(union_run_sz)
{
        .text1: load = ROM, SIZE(text1_size) { f1.c.obj(.text) }
        .text2: load = ROM, SIZE(text2_size) { f2.c.obj(.text) }
}
```

此处 union_ld_sz 将等于 union 中放置的所有输出段的大小之和。union_run_sz 值等于 union 中最大的输出段。这些符号均会根据分块或对齐要求加入任意边界填充。

8.5.10.8 LAST 运算符

LAST 运算符类似于前面描述的 START 和 END 运算符。但是，LAST 适用于存储器范围而不是段。在 MEMORY 指令中使用该运算符可以定义一个符号，而这个符号可以在运行时用于了解链接程序时分配了多少存储器空间。有关语法的详细信息，请参阅节 8.5.4.2。

例如，一个存储器范围可能定义如下：

```
D_MEM : org = 0x20000020 len = 0x20000000 LAST(dmem_end)
```

然后，您的 C 程序可以在运行时使用 _symval 运算符访问此符号。例如：

```
extern char dmem_end;
printf("End of D_MEM memory is %lx\n", _symval(&dmem_end));
```

更多有关在 C/C++ 代码中引用链接器符号的信息，请参阅节 8.6。

8.5.11 创建和填充孔洞

链接器支持*在输出段内*创建没有任何链接的区域。这些区域被称为*孔洞*。在特殊情况下，未初始化的段也可以被视为孔洞。本节介绍了链接器如何处理孔洞以及如何用值填充孔洞（和未初始化的段）。

8.5.11.1 已初始化和未初始化的段

关于输出段的内容，有两条规则需要注意。输出段符合以下条件之一：

- 包含*整个*段的原始数据
- *没有*原始数据

具有原始数据的段被称为*已初始化*的段。这意味着目标文件包含段的实际存储器映像内容。加载段时，此映像将加载到位于段的指定起始地址处的存储器中。如果有任何内容汇编到 .text 和 .data 段中，那么它们*始终*具有原始数据。用 .sect 汇编器指令定义的指定段也具有原始数据。

默认情况下，.bss 段（请参阅保留 .bss 段中的空间）和由 .usect 指令定义的段（请参阅保留未初始化的段）没有原始数据（它们*未初始化*）。它们在存储器映射中占据空间但没有实际内容。未初始化的段通常会在快速外部存储器中为变量保留空间。在目标文件中，一个未初始化的段具有一个正常的段标头并且可以在其中定义符号；但是，该段中没有存储任何存储器映像。

8.5.11.2 创建空洞

用户可以在已初始化的输出段中创建一个空洞。当用户强制链接器在输出段内的输入段之间留出额外空间时，就会创建空洞。创建此类空洞时，*链接器必须为该空洞提供原始数据*。

DRAFT ONLY
TI Confidential – NDA Restrictions

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

只能在输出段*内* 创建空洞。输出段*之间* 可以存在空间，但这种空间不是空洞。若要填充输出段之间的空间，请参阅节 8.5.4.2。

若要在输出段中创建空洞，必须在输出段定义中使用特殊类型的链接器赋值语句。赋值语句会修改 SPC（由 **.** 表示），修改方法是向其添加、为其分配更大的值或在地址边界上将其对齐。有关赋值语句的运算符、表达式和语法的说明，请参阅节 8.5.10。

以下示例使用了赋值语句在输出段中创建空洞：

```
SECTIONS
{
   outsect:
   {
      file1.c.obj(.text)
     .+= 0x0100     /* 按大小 0x0100 创建一个空洞 */
      file2.c.obj(.text)
    .= align(16);   /* 创建一个空洞以对齐 SPC */
      file3.c.obj(.text)
    }
}
```

输出段 outsect 的构建方式如下：

1. 将 file1.c.obj 中的 .text 段链接进来。
2. 链接器创建一个 256 字节的空洞。
3. 在空洞之后将 file2.c.obj 中的 .text 段链接进来。
4. 链接器通过在 16 字节边界上对齐 SPC 来创建另一个空洞。
5. 最后，将 file3.c.obj 中的 .text 段链接进来。

在一个段内分配给 **.** 符号的所有值指代*该段内的相对地址*。链接器处理 **.** 符号的赋值语句时会认为该段从地址 0 开始（即使已指定绑定地址也是如此）。请考虑示例中的赋值语句 **.**= align(16)。该语句实际上会将 file3.c.obj .text 段与 outsect 内的 16 字节边界上的起点对齐。如果 outsect 最终分配至未对齐的地址起点，则 file3.c.obj .text 段也不会对齐。

. 符号指段的当前运行地址，而不是当前加载地址。

减小 **.** 符号的表达式是非法的。例如，在 **.** 符号的赋值语句中使用 -= 运算符是无效的。**.** 符号的赋值语句中最常用的运算符是 += 和 align。

如果输出段包含某种类型的所有输入段（例如 .text），则可使用以下语句在输出段的开头或结尾创建一个空洞。

```
    .text:   {  .+= 0x0100; }      /* 位于开头的空洞 */
    .data:   {  *(.data)
                .+= 0x0100; }     /* 位于结尾的空洞       */
```

在输出段中创建空洞的另一种方法是，将未初始化的段与已初始化的段组合形成单个输出段。*在这种情况下，链接器会将未初始化的段视为空洞并为其提供数据*。以下示例说明了此方法：

```
SECTIONS
{
   outsect:
   {
      file1.c.obj(.text)
     file1.c.obj(.bss)           /* 这成为了一个空洞 */
   }
}
```

由于 .text 段有原始数据，整个 outsect 也必须包含原始数据。因此，未初始化的 .bss 段将成为空洞。

未初始化的段只有在与已初始化的段组合时才会成为空洞。如果将多个未初始化的段链接在一起，则生成的输出段也未经初始化。

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

DRAFT ONLY TI Confidential - NDA Restrictions

8.5.11.3 填充孔洞

当已初始化的输出段中存在孔洞时，链接器必须提供原始数据来填充该孔洞。链接器会使用 32 位填充值来填充孔洞。该填充值在存储器中持续复制，直到填满孔洞。链接器按如下方式确定填充值：

1. 如果孔洞是由于将未初始化的段与已初始化的段组合在一起而形成的，则可以为未初始化的段指定填充值。在段名后添加一个 = sign（符号）和一个 32 位常量。例如：

```
SECTIONS
{  outsect:
   {
      file1.c.obj(.text)
      file2.c.obj(.bss)= 0xFF00FF00  /* 用 0xFF00FF00 填充该孔洞 */
   }
}
```

2. 还可以通过在段定义之后提供填充值来为输出段中的所有孔洞指定填充值：

```
SECTIONS
{  outsect:fill = 0xFF00FF00       /* 用 0xFF00FF00 填充孔洞 */
   {
      .+= 0x0010;                 /* 这会形成一个孔洞         */
      file1.c.obj(.text)
      file1.c.obj(.bss)             /* 这会形成另一个孔洞   */
   }
}
```

3. 如果没有为某一个孔洞指定初始化值，则链接器将使用通过 --fill_value 选项指定的值来填充该孔洞（请参阅节 8.4.14）。例如，假设命令文件 link.cmd 包含以下 SECTIONS 指令：

```
SECTIONS { .text: { .= 0x0100; } /* Create a 100 word hole */ }
```

现在使用 --fill_value 选项来调用链接器：

```
armcl --run_linker --fill_value=0xFFFFFFFF link.cmd
```

这会用 0xFFFFFFFF 填充孔洞。

4. 如果不使用 --fill_value 选项调用链接器，或以其他方式指定填充值，则链接器将用 0 填充孔洞。

每当在已初始化的输出段中创建并填充孔洞时，都会在链接映射中标识该孔洞以及链接器用来填充该孔洞的值。

8.5.11.4 对未初始化的段进行显式初始化

您可以强制链接器对未初始化的段进行初始化，只需在 SECTIONS 指令中为其指定显式填充值即可实现这一点。这种做法会使整个段具有原始数据（填充值）。例如：

```
SECTIONS
{
   .bss: fill = 0x12341234 /* Fills .bss with 0x12341234 */
}
```

备注

填充段

填充一个段（即使是填充 0）会导致在输出文件中为整个段生成原始数据，因此如果您为很大的段或孔洞指定填充值，则输出文件将非常大。

8.6 链接器符号

C/C++ 源代码可能需要引用由链接器定义的符号，而不是 C/C++ 源代码中的符号。对于充当函数或数组的由链接器定义的符号，您通常可以在 C/C++ 代码中直接引用此类符号。对于 C/C++ 代码中的其他用途，通常需要使用其他技术（例如 _symval 运算符）来访问由链接器定义的符号。后续各小节将介绍这些技术。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.6.1 链接器定义的函数和数组

在大多数情况下，您可以像访问 C/C++ 函数一样访问链接器定义的函数。只需为这类函数提供外部声明（原型），然后正常访问这类函数即可：

```
extern int linker_defined_function(void);
printf("value is %d\n", linker_defined_function());
```

在大多数情况下，您可以像访问 C/C++ 数组一样访问链接器定义的数组。 只需为这类数组提供外部声明（可以省略第一个维度），然后正常访问数组即可：

```
extern int linker_defined_data[][10][10];
printf("value is %d\n", linker_defined_data[2][3][4]);
```

如果因为函数或数组超出正常地址范围而收到重定位错误，请按照 节 8.6.4 中所述使用 _symval 运算符。

8.6.2 链接器定义的整数值

要访问表示整数值的链接器符号，请使用 _symval 内置运算符，这本质上是一个强制转换操作。

例如，链接器符号 __TI_STACK_SIZE 表示普通整数。要在 C/C++ 代码中获取符号的值作为整数，请使用以下语法：

```
extern void __TI_STACK_SIZE;
size_t get_stack_size() { return _symval(&__TI_STACK_SIZE); }
```

此类外部声明中的类型无关紧要，因为只需要符号的地址。在严格的 ANSI 模式下，无法使用 void 类型来声明此变量，因此请改用 unsigned char。

备注

请勿尝试在 C/C++ 中单独使用 _TI_STACK_SIZE 符号。符号的值未定义，因此符号的地址可能是无效的存储器地址。

要了解为何需要使用 _symval 运算符来访问链接器定义的整数值，请参阅 节 8.6.4，了解链接器符号与 C 标识符的区别。

8.6.3 链接器定义的地址

要访问表示地址的由链接器定义的符号，请使用 _symval 内置运算符获取符号的值来作为指针值。

例如，链接器符号 _TI_STACK_END 表示一个地址。要在 C/C++ 代码中获取符号的值来作为指针值，请使用以下语法：

```
extern void __TI_STACK_END;
void *get_stack_end() { return (void*)_symval(&__TI_STACK_END); }
```

尽管链接器符号 __TI_STACK_END 是一个地址，因此看起来非常像 C/C++ 指针值，但您无法直接将 C/C++ 变量 __TI_STACK_END 定义为指针变量并省略获取符号的地址。有关详细信息，请参阅节 8.6.4。下面是一个*错误*示例。

```
extern void * __TI_STACK_END;
void *get_stack_end() { return __TI_STACK_END; }  // wrong, missing &
```

8.6.4 有关 _symval 运算符的更多信息

当您在 C/C++ 中声明 `int x = 1234;` 等变量时，系统将在地址 &x 处创建标识符（名称）为“x”且内容为 1234 的对象。此外，系统还会创建一个名为 x 的相关链接器符号。该链接器符号仅表示地址，而不是对象本身。引用链接器符号 x 会得到链接器符号的值，即地址。但是，对 C/C++ 标识符 x 的引用会得到内容 1234。如果您

Copyright © 2023 Texas Instruments Incorporated

需要此 C/C++ 标识符的地址，则需要使用 `&x`。因此，C/C++ 表达式 `&x` 与链接器表达式 `x` 具有相同的值，单链接器符号没有关联类型。

假设链接器定义的符号表示整数而不是地址，例如 __TI_STACK_SIZE。无法在编译器中直接引用整数链接器符号，因此我们使用了一个技巧。首先，假设此链接器符号代表一个地址。在 C/C++ 代码中声明名称相同的假变量。现在，请参考 &__TI_STACK_SIZE，这是一个与链接器符号 __TI_STACK_SIZE 具有相同值的表达式。该值的类型不正确，您可以通过转换更改该类型，如下所示：

```
extern unsigned char __TI_STACK_SIZE;
size_t stack_size = (size_t) &__TI_STACK_SIZE;
```

如以上示例所示，省略 _symval 在大部分时间中都会起作用，但并非总是如此。在某些情况下，指针值不足以表示链接器符号值。例如，一些目标具有 16 位地址空间，因此为 16 位指针。TI 链接器符号为 32 位，因此链接器符号的值可大于可由目标指针表示的值。在这种情况下，表达式 &v 仅反映链接器符号 “v” 实际值的较低 16 位。要解决此问题，请使用内置的 _symval 运算符，它会复制链接器符号的所有值位：

```
extern void v;
unsigned long value = (unsigned long)_symval(&v);
```

对于每种链接器符号，请使用此模式：

```
extern void name;
desired_type name = (desired_type)_symval(&name);
```

例如，

```
extern void farfunc;

void foo()
{
    void (*func)(void) = (void (*)(void))_symval(&farfunc);
    func();
}
```

8.6.5 弱符号

弱符号是可能定义，也可能不定义的符号。

有关链接器如何处理弱符号的详细信息，请参阅节 2.6.3。

8.6.5.1 弱符号引用

在执行最终链接后，弱符号引用可能具有定义，也可能没有定义。如果符号未定义，则其地址被视为 0。在尝试使用该变量的内容之前，C/C++ 代码必须检查弱引用的地址，以确保该值不为 0。

```
extern __attribute__((weak)) unsigned char * foo;
if (&foo != 0)
    *foo = 1;
```

如果对应于 foo 的链接器符号可能没有有效地址（例如，因为该符号包含整数值而非地址），或者可能超出 2GB 范围 PC 相对寻址，请使用 _symval 内置运算符，如下所示：

```
extern __attribute__((weak)) unsigned char * foo;
if (_symval(&foo) != 0)
    *foo = 1;
```

8.6.5.2 弱符号定义

弱符号定义是有效的定义，但如果在链接时找到此类定义，则会丢弃该定义，取而代之的是非弱定义。您可以在链接器命令文件中定义弱符号 C/C++。

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY
TI Confidential – NDA Restrictions

在 C/C++ 中，按如下所示定义弱符号：

```
__attribute__((weak)) int bar;
```

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

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

当使用链接器命令文件执行最终链接时，“ext_addr_sym”会作为弱绝对符号呈现给链接器。如果未引用此符号，它便不会包含在生成的输出文件中。

8.6.6 利用对象库解析符号

对象库属于分区存档文件，其中包含目标文件。通常，一组相关模块被组合起来构成库。将对象库指定为链接器输入时，链接器会包含定义现有未解析符号引用的所有库成员。您可以使用归档器来构建和维护库。节 7.1 介绍了关于归档器的更多信息。

使用对象库可以缩减可执行模块的链接时间和大小。通常，如果在链接时指定了包含函数的目标文件，则无论是否会使用对应函数，都会链接该文件；不过，如果存档库中存在相同的函数，那么只有引用了对应函数时，才会包含该文件。

指定库的顺序非常重要，因为链接器在搜索库时仅包含那些会解析未定义符号的成员。您可以根据需要多次指定同一个库；只要包含库，链接器就会搜索库。此外，您可以使用 --reread_libs 选项来重新读取库，直到没有其他引用可供解析（请参阅节 8.4.18.3）。库包含一个表格，其中列出了该库中定义的所有外部符号；链接器会通过该表格进行搜索，直到确定无法使用该库来解析任何其他引用。

以下示例会链接多个文件和库并做下列假设：

- 输入文件 f1.c.obj 和 f2.c.obj 都引用一个名为 *clrscr* 的外部函数。
- 输入文件 f1.c.obj 引用了符号 *origin*。
- 输入文件 f2.c.obj 引用了符号 *fillclr*。
- 库 libc.lib 的成员 0 包含 *origin* 的定义。
- 库 liba.lib 的成员 3 包含 *fillclr* 的定义。
- 这两个库的成员 1 都定义了 *clrscr*。

如果您输入：

```
armcl --run_linker f1.c.obj f2.c.obj liba.lib libc.lib
```

那么：

- liba.lib 的成员 1 会满足 f1.c.obj 和 f2.c.obj 对 *clrscr* 的引用，因为链接器通过搜索该库可以找到 *clrscr* 的定义。
- libc.lib 的成员 0 会满足对 *origin* 的引用。
- liba.lib 的成员 3 会满足对 *fillclr* 的引用。

不过，如果您输入：

```
armcl --run_linker f1.c.obj f2.c.obj libc.lib liba.lib
```

那么，对 *clrscr* 的引用会由 libc.lib 的成员 1 满足。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

如果库中没有定义任何链接的文件引用符号，您可以使用 --undef_sym 选项来强制链接器包含库成员。（请参阅节 8.4.34。）下例会在链接器的全局符号表中创建一个未定义的符号 rout1：

```
armcl --run_linker --undef_sym=rout1 libc.lib
```

如果 libc.lib 的任何成员定义了 rout1，链接器会包含该成员。

会根据 SECTIONS 指令的默认分配算法来分配库成员；请参阅节 8.5.5。

节 8.4.18 介绍了几种指定对象库所在目录的方法。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.7 默认放置算法

MEMORY 和 SECTIONS 指令提供了灵活的方法来进行段的构建、组合和分配。但是，任何*未*指定的存储器位置或段仍必须由链接器处理。链接器使用相应的算法并根据您提供的任何规格来构建和分配段。

如果不使用 MEMORY 和 SECTIONS 指令，则链接器会按照 ARM 器件的默认分配中所示的存储器映射和段定义，进行输出段的分配。

ARM 器件的默认分配

```
{
   RAM      : origin = 0x00000000, length = 0xFFFFFFFF
}
SECTIONS
{
   .text :  ALIGN(4)    {} > RAM
   .const:  ALIGN(4)    {} > RAM
   .data :  ALIGN(4)    {} > RAM
   .bss  :  ALIGN(4)    {} > RAM
   .cinit:  ALIGN(4)    {} > RAM      /* -c option only */
   .pinit:  ALIGN(4)    {} > RAM      /* -c option only */
}
```

有关默认存储器分配的信息，请参阅节 2.5.1。

在可执行输出文件中，所有 .text 输入段被连接起来形成一个 .text 输出段，所有 .data 输入段被组合起来形成一个 .data 输出段。

如果使用 SECTIONS 指令，则链接器*不*执行此默认分配的任何部分。实际上，会根据 SECTIONS 指令指定的规则和接下来在节 8.7.1 中介绍的通用算法来执行分配。

8.7.1 分配算法如何创建输出段

可通过以下两种方式之一构建输出段：

方法 1	作为 SECTIONS 指令定义的结果
方法 2	通过将同名的输入段组合成一个未在 SECTIONS 指令中定义的输出段

如果作为 SECTIONS 指令的结果构成输出段，则段的内容完全由此定义确定。（如需查看如何定义输出段内容的示例，请参阅节 8.5.5。）

如果输出段是通过组合未由 SECTIONS 指令指定的输入段构成的，则链接器会将所有同名的此类输入段组合到该名称的输出段中。例如，假设文件 f1.c.obj 和 f2.c.obj 都包含名为 Vectors 的指定段，并且 SECTIONS 指令没有为它们定义输出段。链接器会将输入文件中的两个 Vectors 段组合成名为 Vectors 的单个输出段，将这个输出段分配到存储器中，并将其包含在输出文件中。

默认情况下，链接器在创建未在 SECTIONS 指令中定义的输出段时不显示消息。使用 --warn_sections 链接器选项（请参阅节 8.4.35）可以使链接器在创建新的输出段时显示消息。

链接器确定所有输出段的组成方式后，必须将它们分配到配置的存储器中。MEMORY 指令可以指定配置存储器的哪些部分。如果没有 MEMORY 指令，链接器将使用默认配置，如 ARM 器件的默认分配中所示。（有关配置存储器的更多信息，请参阅节 8.5.4。）

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.7.2 减少存储器碎片

链接器的分配算法会尝试尽量减少存储器碎片。这样可以更高效地使用存储器并增加程序纳入存储器的概率。算法分以下几步：

1. 每个提供具体绑定地址的输出段将置于存储器中的该位置。
2. 将分配包含在具体的已命名存储器范围中的每个输出段，或具有存储器属性限制的每个输出段。每个输出段将置于已命名区域中的第一个可用空间，必要时考虑对齐。
3. 任何其余段的分配顺序与定义顺序相同。未在 SECTIONS 指令中定义的段的分配顺序与遇到段的顺序相同。每个输出段将置于第一个可用存储器空间中，必要时考虑对齐。

8.8 使用由链接器生成的复制表

链接器支持对链接器命令文件语法进行扩展，使您能够：

- 在引导时更轻松地将对象从加载空间复制到运行空间
- 在运行时更轻松地管理存储器叠加
- 拆分具有不同加载和运行地址的 GROUP 和输出段

有关复制表及其使用的介绍，请参阅节 3.3.3。

8.8.1 使用复制表进行引导加载

在某些嵌入式应用中，在引导时需要将代码和/或数据从一个位置复制或下载到另一个位置，然后应用才会开始其主执行线程。例如，应用的代码和/或数据可能位于闪存存储器中，需要将其复制到片上存储器后应用才能开始执行线程。

开发此类应用的一种方式是在汇编代码中创建复制表，其中包含每个代码块或数据块的三个元素，需要在引导时将它们从闪存移至片上存储器：

- 加载地址
- 运行地址
- 大小

开发此类应用的流程应该与以下流程类似：

1. 编译应用以生成 .map 文件，其中包含每个段的加载和运行地址，在加载和运行时会分别放置。
2. 编辑复制表（由引导加载程序使用）更正需要在引导时移动的每个代码块或数据块的加载和运行地址以及大小。
3. 再次编译应用，加入更新的复制表。
4. 运行应用。

此流程会为您带来维护复制表的沉重负担（仍然需要手动完成）。应用每次添加或删除一些代码或数据，您都必须重复此流程，以保持复制表的内容是最新的。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.8.2 在复制表中使用内置链接运算符

使用链接器命令文件句法中包含的 LOAD_START()、RUN_START() 和 SIZE() 操作符，可避免一些维护负担。例如，链接器命令文件可标注为以下形式，而不必编译应用以生成 .map 文件：

```
SECTIONS
{
   .flashcode: { app_tasks.c.obj(.text) }
      load = FLASH, run = PMEM,
      LOAD_START(_flash_code_ld_start),
      RUN_START(_flash_code_rn_start),
      SIZE(_flash_code_size)
   ...
}
```

在本例中，LOAD_START()、RUN_START() 和 SIZE() 运算符指示链接器创建三个符号：

符号	说明
_flash_code_ld_start	.flashcode 段的加载地址
_flash_code_rn_start	.flashcode 段的运行地址
_flash_code_size	.flashcode 段的大小

然后可在复制表中引用这些符号。每次链接应用时，复制表中的实际数据将自动更新。此方法省去了节 8.8.1 中所述流程的步骤 1。

虽然维护复制表的工作显著减少，但您还有责任使复制表内容与链接器命令文件中定义的符号保持同步。理想情况下，链接器会自动生成引导复制表。这样可避免两次编译应用，*而且* 无需再显式管理引导复制表的内容。

有关 LOAD_START()、RUN_START() 和 SIZE() 运算符的更多信息，请参阅节 8.5.10.7。

8.8.3 重叠管理示例

假设应用程序包含存储器重叠区，而必须在运行时管理重叠。存储器重叠区在链接器命令文件中使用 UNION 来定义，如使用 UNION 定义存储器重叠区 中所示：

使用 UNION 定义存储器重叠区

```
SECTIONS
{
   ...
   UNION
   {
      GROUP
      {
         .task1: { task1.c.obj(.text) }
         .task2: { task2.c.obj(.text) }
      } load = ROM, LOAD_START(_task12_load_start), SIZE(_task12_size)
      GROUP
      {
         .task3: { task3.c.obj(.text) }
         .task4: { task4.c.obj(.text) }
      } load = ROM, LOAD_START(_task34_load_start), SIZE(_task_34_size)
   } run = RAM, RUN_START(_task_run_start)
 ...
}
```

应用程序必须在运行时管理存储器重叠区的内容。也就是说，当需要来自 .task1 或 .task2 的任何服务时，应用程序必须首先确保 .task1 和 .task2 驻留在存储器重叠区中。.task3 和 .task4 与此类似。

若要在运行时影响 .task1 和 .task2 从 ROM 到 RAM 的复制，应用程序必须先获取各任务的加载地址 (_task12_load_start)，再获取运行地址 (_task_run_start) 和大小 (_task12_size)。然后使用这些信息来执行实际的代码复制。

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

8.8.4 使用 table() 运算符生成复制表

链接器支持链接器命令文件语法的扩展，使您能够进行以下操作：

- 确定在应用程序运行期间的某个时刻可能需要从加载空间复制到运行空间的任何对象组件
- 指示链接器自动生成一个复制表，其中（至少）包含需要复制的组件的加载地址、运行地址和大小
- 指示链接器生成您指定的符号，该符号提供由链接器生成的复制表的地址。例如，使用 UNION 定义存储器重叠区 可重新写成为由链接器生成的复制表生成地址所示的代码：

为由链接器生成的复制表生成地址

```
SECTIONS
{
   ...
   UNION
   {
      GROUP
      {
         .task1: { task1.c.obj(.text) }
         .task2: { task2.c.obj(.text) }
      } load = ROM, table(_task12_copy_table)
      GROUP
      {
         .task3: { task3.c.obj(.text) }
         .task4: { task4.c.obj(.text) }
      } load = ROM, table(_task34_copy_table)

   } run = RAM
   ...
}
```

使用链接器命令文件为由链接器生成的复制表生成地址中所示的 SECTIONS 指令，链接器可以生成两个名称如下的复制表：_task12_copy_table 和 _task34_copy_table。每个复制表提供与复制表关联的 GROUP 的加载地址、运行地址和大小。可使用链接器生成的符号 _task12_copy_table 和 _task34_copy_table（它们分别提供两个复制表的地址）从应用程序源代码访问此信息。

通过使用此方法，无需担心复制表的创建或维护。您可以在 C/C++ 或汇编源代码中引用由链接器生成的任何复制表的地址，从而将该值传递给通用复制例程，然后该例程将处理复制表并影响实际复制。

8.8.4.1 table() 操作符

您可以使用 table() 操作符指示链接器生成复制表。table() 操作符可应用于输出段、GROUP 或 UNION 成员。为特定 table() 规范生成的复制表可通过一个符号访问，该符号由您指定，作为一个参数提供给 table() 操作符。链接器创建一个具有此名称的符号，为其指定复制表的地址，作为此符号的值。然后可从应用中使用由链接器生成的符号访问此复制表。

您针对已知 UNION 的成员应用的每个 table() 规范必须包含唯一名称。如果将 table() 操作符应用于 GROUP，该 GROUP 的成员不会标记 table() 规范。链接器会检测违反这些规则的情况，并作为警告报告，忽略每次违规使用 table() 规格的情况。链接器不会为错误的 table() 操作符规范生成复制表。

复制表可自动生成；请参阅节 8.8.4。表操作符可使用压缩功能；请参阅节 8.8.5。

8.8.4.2 启动时复制表

链接器支持特殊的复制表名称 BINIT（或 binit）；您可以使用该名称来创建启动时复制表。使用 .cinit 段在启动时初始化变量之前该表会被处理。例如，节 8.8.2 中所述的启动加载应用程序的链接器命令文件可以重写如下：

```
SECTIONS
{
   .flashcode: { app_tasks.c.obj(.text) }
   load = FLASH, run = PMEM,
      table(BINIT)
   ...
}
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

对于此示例，链接器会创建一个复制表，可通过链接器生成的一个特殊符号 __binit__ 来访问该表，其中包含需要在启动时从加载位置复制到运行位置的所有对象组件的列表。如果链接器命令文件未在任何情况下使用 table(BINIT)，则会向 __binit__ 符号赋值 -1 以指示某个特定应用程序不存在启动时复制表。

您可以将 table(BINIT) 规格应用于输出段、GROUP 或 UNION 成员。如果是在 UNION 的上下文中使用，则只能使用 table(BINIT) 指定 UNION 的一个成员。如果是应用于 GROUP，则该 GROUP 的任何成员都不能用 table(BINIT) 进行标记。链接器会检测违反这些规则的情况，并作为警告报告，但会忽略每次违规使用 table(BINIT) 规格的情况。

8.8.4.3 使用 table() 操作符管理目标组件

如果您需要共同管理多个代码段，可以针对多个不同的目标组件应用同一 table() 操作符。此外，如果您要通过多种方式管理特定的目标组件，则可以对其应用多个 table() 操作符。请考虑用于管理目标组件的链接器命令文件中的链接器命令文件片段：

用于管理目标组件的链接器命令文件

```
SECTIONS
{
   UNION
   {
      .first: { a1.c.obj(.text), b1.c.obj(.text), c1.c.obj(.text) }
            load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)
      .second: { a2.c.obj(.text), b2.c.obj(.text) }
            load = EMEM, run = PMEM, table(_second_ctbl)
   }
   .extra: load = EMEM, run = PMEM, table(BINIT)
   ...
}
```

在本例中，输出段 .first 和 .extra 在引导时从外部存储器 (EMEM) 复制到程序存储器 (PMEM)，同时处理 BINIT 复制表。应用开始执行其主线程后，可以使用两个叠加复制表管理叠加的内容，这两个复制表分别命名为：_first_ctbl 和 _second_ctbl。

8.8.4.4 由链接器生成的复制表段和符号

链接器会为由其生成的每个复制表创建并分配一个单独的输入段。每个复制表符号都用包含相应复制表的输入段的地址值进行定义。

链接器会为每个重叠复制表输入段生成一个唯一的名称。例如，table(_first_ctbl) 会将 .first 段的复制表放置在名为 .ovly:_first_ctbl 的输入段中。链接器会创建单个输入段 .binit 以包含整个启动时复制表。

控制由链接器生成的复制表段的放置说明了如何使用链接器命令文件中的输入段名称来控制由链接器生成的复制表段的放置。

控制由链接器生成的复制表段的放置

```
SECTIONS
{
   UNION
   {
      .first: { a1.c.obj(.text), b1.c.obj(.text), c1.c.obj(.text) }
             load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)
      .second: { a2.c.obj(.text), b2.c.obj(.text) }
             load = EMEM, run = PMEM, table(_second_ctbl)
   }
   .extra: load = EMEM, run = PMEM, table(BINIT)
   ...
   .ovly: { } > BMEM
   .binit: { } > BMEM
}
```

对于控制由链接器生成的复制表段的放置中的链接器命令文件，启动时复制表会生成到 .binit 输入段中，该输入段收集到 .binit 输出段中，该输出段映射到 BMEM 存储器区域中的某个地址。_first_ctbl 生成到 .ovly:_first_ctbl 输

Copyright © 2023 Texas Instruments Incorporated

入段中，_second_ctbl 生成到 .ovly:_second_ctbl 输入段中。这些输入段的基本名称与 .ovly 输出段的名称匹配，因此这些输入段会收集到 .ovly 输出段中，然后该输出段映射到 BMEM 存储器区域中的某个地址。

如果没有为由链接器生成的复制表段提供显式的放置指令，则会根据链接器的默认放置算法来分配它们。

链接器不允许将其他类型的输入段与同一输出段中的复制表输入段进行组合。链接器不允许将从部分链接会话创建的复制表段用作后续链接会话的输入。

8.8.4.5 拆分对象组件和重叠管理

可以拆分包含单独加载和运行放置指令的段。链接器可以访问拆分对象组件每个部分的加载地址和运行地址。使用 table() 操作符，用户可以指示链接器生成此信息并将其放入复制表。链接器会在复制表对象中为拆分对象组件的每个部分提供一个 COPY_RECORD 条目。

例如，设想一个具有七个任务的应用。任务 1 至 3 与任务 4 至 7 重叠（使用 UNION 指令）。所有这些任务的加载放置都会在四个不同的存储器区域（LMEM1、LMEM2、LMEM3 和 LMEM4）之间分配。重叠定义为存储器区域 PMEM 的一部分。用户必须在运行时将每组任务移入重叠区，才能使用该组中的服务。

用户可以将 table() 操作符与分拆操作符 >> 结合使用来创建包含所有所需信息的复制表，以将任一组任务移入存储器重叠区，如创建复制表来访问拆分对象组件中所示。

创建复制表来访问拆分对象组件

```
SECTIONS
{
   UNION
   {
      .task1to3: { *(.task1), *(.task2), *(.task3) }
              load >> LMEM1 | LMEM2 | LMEM4, table(_task13_ctbl)
      GROUP
      {
         .task4: { *(.task4) }
         .task5: { *(.task5) }
         .task6: { *(.task6) }
         .task7: { *(.task7) }
      } load >> LMEM1 | LMEM3 | LMEM4, table(_task47_ctbl)
   } run = PMEM
   ...
   .ovly: > LMEM4
}
```

拆分对象组件驱动程序显示了此类应用可能的驱动程序。

拆分对象组件驱动程序

```
#include <cpy_tbl.h>
extern far COPY_TABLE task13_ctbl;
extern far COPY_TABLE task47_ctbl;
extern void task1(void);
...
extern void task7(void);
main()
{
   ...
   copy_in(&task13_ctbl);
   task1();
   task2();
   task3();
   ...
   copy_in(&task47_ctbl);
   task4();
   task5();
   task6();
   task7();
   ...
}
```

用户必须将 COPY_TABLE 对象声明为 *far*，以便重叠复制表段独立于包含数据对象的其他段（例如 .bss）放置。

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

.task1to3 段的内容会分拆到该段的加载空间并且在其运行空间中是连续的。由链接器生成的复制表 _task13_ctbl 会针对分拆段 .task1to3 的每个部分包含一个单独的 COPY_RECORD。当 _task13_ctbl 的地址被传递到 copy_in() 时，.task1to3 的每个部分都会从其加载位置复制到运行位置。

包含任务 4 至 7 的 GROUP 的内容也会拆分到加载空间中。链接器会按顺序为 GROUP 中每个成员应用分拆操作符，从而执行 GROUP 分拆。然后，该 GROUP 的复制表会针对该 GROUP 中每一个成员的每个部分包含一个 COPY_RECORD 条目。当 copy_in() 对 _task47_ctbl 进行处理时，这些部分都会复制到存储器重叠区。

分拆操作符可应用于输出段、GROUP、UNION 或 UNION 成员的加载放置。链接器不允许将分拆操作符应用于 UNION 或 UNION 成员的运行放置。链接器会检测此类违规情况，发出警告，并忽略违规使用分拆操作符的情况。

8.8.5 压缩

自动生成复制表时，链接器提供了一种压缩加载空间数据的方法。这可以减少只读存储器占用空间。在将压缩的数据从加载空间复制到运行空间时可以解压缩此数据。

可通过两种方式指定压缩：

- 可使用链接器命令行选项 --copy_compression=compression_kind 将指定的压缩应用于任何应用了 table() 运算符的输出段。
- table() 运算符接受一个可选的压缩参数。语法为：.

  table(*name* **, compression=** *compression_kind* **)**

  compression_kind 可以是以下类型之一：
  - **off**。不要压缩数据。
  - **rle**。使用行程编码格式压缩数据。
  - **lzss**。使用 Lempel-Ziv-Storer-Szymanski 压缩格式压缩数据。

不带 compression 关键字的 table() 运算符使用通过命令行选项 --copy_compression 指定的压缩类型。

选择压缩时，不能保证链接器将会压缩加载数据。仅当压缩会减小加载空间的整体大小时，链接器才会压缩加载数据。在某些情况下，即使压缩会使加载段大小变小，如果解压缩例程会抵消节省量，那么链接器也不会压缩数据。

例如，假设 RLE 压缩将 section1 的大小减少 30 个字节。此外，还假设 RLE 解压缩例程在加载空间中占用 40 字节。通过选择对 section1 进行压缩，加载空间将增加 10 个字节。因此，链接器不会压缩 section1。另一方面，如果有另一个段（比如 section2）可以从应用相同的压缩中受益超过 10 个字节，则可以压缩这两个段，使得整体加载空间减小。在此类情况下，链接器会压缩这两个段。

当这样做无法实现节省时，不能强制链接器压缩数据。

不能对解压缩例程或包含 .cinit 的 GROUP 中的任何成员进行压缩。

8.8.5.1 压缩的复制表格式

无论 *compression_kind* 如何，复制表格式都是相同的。复制记录的大小字段将被重载以支持压缩。图 8-5 显示了压缩的复制表布局。

Rec size	Rec cnt		
Load address		Run address	Size (0 if load data is compressed)

图 8-5. 压缩的复制表

在图 8-5 中，如果复制记录中的大小不为零，则此大小表示要复制的数据的大小，也意味着加载数据的大小与运行数据的大小相同。此大小为 0 时，表示加载数据被压缩。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.8.5.2 目标文件中的压缩段表示

链接器会创建一个单独的输入段来保存压缩数据。请考虑链接器命令文件中的以下 table() 操作。

```
SECTIONS
{
    .task1: load = ROM, run = RAM, table(_task1_table)
}
```

输出目标文件有一个名为 .task1 的输出段，其中具有不同的加载地址和运行地址。这是可行的，因为当段未被压缩时，加载空间和运行空间具有相同的数据。

或者，请考虑以下代码：

```
SECTIONS
{
    .task1: load = ROM, run = RAM, table(_task1_table, compression=rle)
}
```

如果链接器对 .task1 段进行压缩，则加载空间数据和运行空间数据会不同。链接器会创建以下两个段：

- **.task1**：此段未初始化。此输出段表示 task1 段的运行空间映像。
- **.task1.load**：此段已初始化。此输出段表示 task1 段的加载空间映像。此段的大小通常比 .task1 输出段小得多。

在为 .task1 段的加载放置指定的存储器区域中，链接器为 .task1.load 输入段分配加载空间。只有一个加载段用于表示 .task1 的加载放置，也就是 .task1.load 段。如果尚未压缩 .task1 数据，则 .task1 输入段将有两个分配：一个用于该段的加载放置，另一个用于该段的运行放置。

8.8.5.3 压缩的数据布局

压缩的加载数据具有以下布局：

8 位索引	压缩的数据

加载数据的前 8 位是处理程序索引。此处理程序索引用于索引到处理程序表中，以获取知道如何解码后续数据的处理程序函数的地址。处理程序表是 32 位函数指针的列表，如图 8-6 所示。

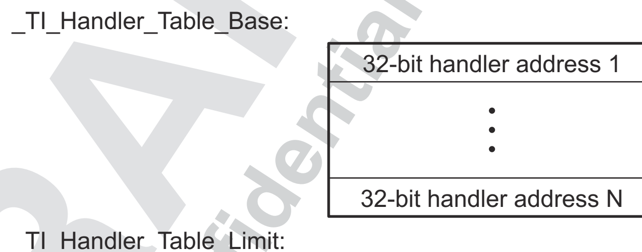


图 8-6. 处理程序表

链接器为加载和运行空间创建一个单独的输出段。例如，如果使用了 RLE 来压缩 .task1.load，则处理程序索引会指向处理程序表中具有运行时支持例程 __TI_decompress_rle() 地址的条目。

8.8.5.4 运行时解压缩

在运行时期间，用户可以调用运行时支持例程 copy_in() 来将数据从加载空间复制到运行空间。复制表的地址会传递给此例程。首先，该例程读取记录计数。然后，它会针对每条记录重复以下步骤：

1. 从记录中读取加载地址、运行地址和大小。
2. 如果大小为零，则转到第 5 步。
3. 调用 memcpy 来传递运行地址、加载地址和大小。
4. 如果要读取其他记录，则转到第 1 步。

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

5. 读取加载地址的。调用此索引。
6. 从 (&__TI_Handler_Base)[index] 读取处理程序地址。
7. 调用处理程序并传递加载地址 + 1 和运行地址。
8. 如果要读取其他记录，则转到第 1 步。

运行时支持库中提供了用于处理加载数据解压缩的例程。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.8.5.5 压缩算法

以下几个小节提供了有关 RLE 和 LZSS 格式的解压缩算法的信息。若要查看解压缩算法的示例，请参阅运行时支持库中的以下函数：

- **RLE**：copy_decompress_rle.c 文件中的 __TI_decompress_rle() 函数。
- **LZSS**：copy_decompress_lzss.c 文件中的 __TI_decompress_lzss() 函数。

行程编码 (RLE)：

8 位索引	使用行程编码压缩的初始化数据

8 位索引之后的数据使用行程编码 (RLE) 格式进行压缩。ARM 使用一种可以使用以下算法解压缩的简单行程编码：请查看 copy_decompress_rle.c 以了解详细信息。

1. 读取第一个字节，即分隔符 (D)。
2. 读取下一个字节 (B)。
3. 如果 B != D，则将 B 复制到输出缓冲区并转到步骤 2。
4. 读取下一个字节 (L)。
   a. 如果 L == 0，则长度要么是 16 位或 24 位值，要么已经到达数据的末尾，读取下一个字节 (L)。
      i. 如果 L == 0，则长度为 24 位值，或者已经到达数据的末尾，读取下一个字节 (L)。
         1. 如果 L == 0，则已经到达数据的末尾，转到步骤 7。
         2. 否则 L <<= 16，将接下来的两个字节读入 L 的低 16 位以完成 L 的 24 位值。
      ii. 否则 L <<= 8，将下一个字节读入 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。

ARM 运行时支持库有一个例程 __TI_decompress_rle24() 可以解压缩使用 RLE 压缩的数据。此函数的第一个参数是指向字节（位于 8 位索引后）的地址，第二个参数是 C 自动初始化记录的运行地址。

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。

Lempel-Ziv-Storer-Szymanski 压缩 (LZSS)：

8 位索引	使用 LZSS 压缩的数据

8 位索引之后的数据使用 LZSS 压缩格式进行压缩。ARM 运行时支持库有一个例程 __TI_decompress_lzss() 可以解压缩使用 LZSS 压缩的数据。此函数的第一个参数是指向字节（位于 8 位索引后）的地址，而第二个参数是 C 自动初始化记录的运行地址。

请查看 copy_decompress_lzss.c 以了解 LZSS 算法的详细信息。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.8.6 复制表内容

要使用由链接器生成的复制表，用户必须知道复制表的内容。此信息包含在一个运行时支持库头文件 cpy_tbl.h 中。该文件包含由链接器生成的复制表数据结构的 C 源代码表示。ARM cpy_tbl.h 文件展示了该复制表头文件。

ARM cpy_tbl.h 文件

```
/*****************************************************************************/
/* cpy_tbl.h  v#####                                                        */
/* 2003 德州仪器 版权所有                              */
/*                                                                          */
/* 可由链接器自动生成的复制表数据结构的规范     */
/* （使用 LCF 中的 table() 运算符）。          */
/*****************************************************************************/
#ifndef _CPY_TBL
#define _CPY_TBL
#ifdef __cplusplus
extern "C" namespace std {
#endif /* __cplusplus */
/*****************************************************************************/
/* 复制记录数据结构                                                  */
/*****************************************************************************/
typedef struct copy_record
{
   unsigned int load_addr;
   unsigned int run_addr;
   unsigned int size;
} COPY_RECORD;
/*****************************************************************************/
/* 复制表数据结构                                                   */
/*****************************************************************************/
typedef struct copy_table
{
   unsigned short rec_size;
   unsigned short num_recs;
   COPY_RECORD    recs[1];
} COPY_TABLE;
/*****************************************************************************/
/* 通用复制例程原型。                                  */
/*****************************************************************************/
extern void copy_in(COPY_TABLE *tp);
#ifdef __cplusplus
} /* extern "C" namespace std */
#ifndef _CPP_STYLE_HEADER
using std::COPY_RECORD;
using std::COPY_TABLE;
using std::copy_in;
#endif /* _CPP_STYLE_HEADER */
#endif /* __cplusplus */
#endif /* !_CPY_TBL */
```

对于已标记要进行复制的每个对象组件，链接器会为其创建一个 COPY_RECORD 对象。每个 COPY_RECORD 至少包含对象组件的以下信息：

- 加载地址
- 运行地址
- 大小

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

链接器会将与同一复制表关联的所有 COPY_RECORD 收集到一个 COPY_TABLE 对象中。COPY_TABLE 对象包含给定 COPY_RECORD 的大小、表中 COPY_RECORD 的数量以及表中 COPY_RECORD 的数组。例如，在节 8.8.4.2 的 BINIT 示例中，各输出段 .first 和 .extra 将在 BINIT 复制表中拥有各自的 COPY_RECORD 条目。BINIT 复制表将如下所示：

```
COPY_TABLE __binit__ = { 12, 2,
                         { <load address of .first>,
                           <run address of .first>,
                           <size of .first> },
                         { <load address of .extra>,
                           <run address of .extra>,
                           <size of .extra> } };
```

8.8.7 通用复制例程

ARM cpy_tbl.h 文件中的 cpy_tbl.h 文件还包含运行时支持库中提供的通用复制例程 copy_in() 的原型。copy_in() 例程只接受一个参数：由链接器生成的复制表的地址。该例程随后会处理复制表数据对象，并执行复制表中指定的每个对象组件的复制。

运行时支持 cpy_tbl.c 文件显示的 cpy_tbl.c 运行时支持源文件中提供了 copy_in() 函数定义。

运行时支持 cpy_tbl.c 文件

```
/****************************************************************************/
/* cpy_tbl.c  v#####                                                        */
/*                                                                          */
/* 通用复制例程。  给定由链接器生成的     */
/* COPY_TABLE 数据结构的地址，通过        */
/* 相应的 LCF table() 运算符复制指定用于复制的所有对象组件。  */
/****************************************************************************/
#include <cpy_tbl.h>
#include <string.h>
typedef void (*handler_fptr)(const unsigned char *in, unsigned char *out)
/****************************************************************************/
/* COPY_IN()                                                                */
/****************************************************************************/
void copy_in(COPY_TABLE *tp)
{
   unsigned short I;
   for (I = 0; I < tp->num_recs; I++)
   {
      COPY_RECORD crp = tp->recs[i];
      unsigned char *ld_addr = (unsigned char *)crp.load_addr;
      unsigned char *rn_addr = (unsigned char *)crp.run_addr;
      if (crp.size)
      {
         /*------------------------------------------------------------------*/
         /* 复制记录具有非零大小，因此数据不会被压缩。   */
         /* 仅复制数据。                                                     */
         /*------------------------------------------------------------------*/
         memcpy(rn_addr, ld_addr, crp.size);
      }
   }
}
```

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.9 由链接器生成的 CRC 表

链接器支持扩展链接器命令文件的语法，以便通过循环冗余校验 (CRC) 来验证代码或数据。链接器在链接时计算指定区域的 CRC 值，并将该值存储在目标存储器中，以便在启动或运行时可以访问该值。然后，应用程序代码可以计算这个区域的 CRC，并确保该值与链接器计算的值匹配。

在链接器命令文件中，可生成以下 CRC 值：

- **段的 CRC**：在 SECTIONS 指令中使用 crc_table() 运算符。请参阅节 8.9.1。

运行时支持库不提供在启动或运行时计算 CRC 值的例程。TI E2E 社区的 Tools Insider 博客提供了使用由链接器生成的 CRC 表执行循环冗余校验的示例。

8.9.1 在 SECTIONS 指令中使用 crc_table() 运算符

对于应使用 CRC 验证的任何段，都必须修改链接器命令文件，以包含 crc_table() 运算符。指定 CRC 算法是可选的。语法为：

crc_table(*user_specified_table_name*[**, algorithm=** *xxx*]**)**

链接器使用 crc_table() 运算符中指定的任何规范中的 CRC 算法。如果省略此规范，将使用 TMS570_CRC64_ISO 算法。链接器在映射文件中包含 CRC 表。其中包括 CRC 值以及计算使用的算法。

为特定 crc_table() 实例生成的 CRC 表可通过表名称访问，表名称作为 crc_table() 运算符的参数提供。链接器创建了一个具有此名称的符号，为其指定 CRC 表的地址，作为此符号的值。然后可从应用中使用由链接器生成的符号访问此 CRC 表。

crc_table() 运算符可应用于输出段、GROUP、GROUP 成员、UNION 或 UNION 成员。在 GROUP 或 UNION 中，运算符应用于每个成员。

您可以在应用中包含对例程的调用，以验证相关段的 CRC 值。您必须提供此例程。请参阅以下内容，详细了解数据结构和建议的接口。

8.9.1.1 使用 crc_table() 运算符时的限制

需要注意的是，链接器使用的 CRC 生成器会按照 crc_tbl.h 头文件中所述进行参数化（请参阅示例 8-9）。在链接器之外采用的任何 CRC 计算例程都必须以相同的方式工作，以确保匹配 CRC 值。链接器无法检测参数中的不匹配情况。若要了解这些参数，请参阅 Ross Williams 编写的“A Painless Guide to CRC Error Detection Algorithms”（CRC 错误检测算法简易指南），对应的网址可能为 http://www.ross.net/crc/download/crc_v3.txt。

链接器仅支持 crc_tbl.h 中的 CRC 算法名称和标识符。所有其他名称和 ID 值均保留以供未来使用。系统可能不包含用于计算这些 CRC 算法的内置硬件。相关详细信息，请参阅硬件文档。受支持的 CRC 算法如下：

- CRC8_PRIME
- CRC16_ALT
- CRC16_802_15_4
- CRC_CCITT
- CRC24_FLEXRAY
- CRC32_PRIME
- CRC32_C
- CRC64_ISO

如果没有指定任何算法，则默认算法为 TMS570_CRC64_ISO。

TMS570_CRC64_ISO 算法采用初始值 0。关于该算法的详细信息可在 MCRC 文档中找到。

另外，链接器也会强制执行一些限制：

- 只能在最终链接期间请求 CRC。
- CRC 只能应用到已初始化的段。
- 只能为加载地址请求 CRC。

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential - NDA Restrictions

- CRC 表名也受到一些限制。例如，BINIT 不能用作 CRC 表名。

8.9.1.2 示例

crc_table() 运算符在语法上类似于用于复制表的 table() 运算符。下面提供了链接器命令文件的几个简单示例。

示例 8-5. 使用 crc_table() 操作符计算 .text 数据的 CRC 值

```
SECTIONS
{
   ...
   .section_to_be_verified: {a1.c.obj(.text)} crc_table(_my_crc_table_for_a1)
}
```

示例 8-5 定义了一个名为“.section_to_be_verified”的段，其中包含来自 a1.c.obj 文件的 .text 数据。crc_table() 运算符请求链接器计算 .text 数据的 CRC 值，并将该值存储在名为“my_crc_table_for_a1”的表中。此表将包含调用用户提供的 CRC 计算例程所需的所有信息，并验证在运行时计算的 CRC 是否与链接器生成的 CRC 匹配。可使用符号 my_crc_table_for_a1 从应用程序代码访问此表，该符号应声明为“extern CRC_TABLE”类型。该符号将由链接器定义。应用程序代码可能类似于以下代码。

```
#include "crc_tbl.h"
extern CRC_TABLE my_crc_table_for_a1;
verify_a1_text_contents()
{
   ...
   /* Verify CRC value for .text sections of a1.c.obj.*/
   if (my_check_CRC(&my_crc_table_for_a1)) puts("OK");
}
```

示例 8-10 中详细显示了 my_check_CRC() 例程。

示例 8-6. 在 crc_table() 操作符中指定算法

```
SECTIONS
{
   ...
   .section_to_be_verified_2:   {b1.c.obj(.text)} load=SLOW_MEM, run=FAST_MEM,
       crc_table(_my_crc_table_for_b1, algorithm=TMS570_CRC64_ISO)
.TI.crctab:   > CRCMEM
}
```

在示例 8-6 中，CRC 算法在 crc_table() 运算符中指定。指定的算法用于计算 b1.c.obj 中文本数据的 CRC。由链接器生成的 CRC 表在特殊段 .TI.crctab 中创建，而这个段可采用与其他段相同的方式放置。在本例中，CRC 表 _my_crc_table_for_b1 是在 .TI.crctab:_my_crc_table_for_b1 段中创建的，并且该段被放置在 CRCMEM 存储器区域中。

示例 8-7. 多个段使用单一表

```
SECTIONS
{
   .section_to_be_verified_1: {a1.c.obj(.text)}
      crc_table(_my_crc_table_for_a1_and_c1)
   .section_to_be_verified_3: {c1.c.obj(.text)}
      crc_table(_my_crc_table_for_a1_and_c1, algorithm=TMS570_CRC64_ISO)
}
```

示例 8-7 中为 a1.c.obj 和 c1.c.obj 指定了相同的标识符 _my_crc_table_for_a1_and_c1。链接器会创建单个包含两个文本段条目的表。

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

示例 8-8. 将 crc_table() 运算符应用于 GROUP 或 UNION

```
SECTIONS
{
   UNION
   {
      section1: {} crc_table(table1)
      section2:
   }  crc_table(table2)
}
```

将 crc_table() 操作符应用于 GROUP 或 UNION 时，链接器会将对应的表规格应用于 GROUP 或 UNION 的成员。

在示例 8-8 中，链接器会创建两个 CRC 表——table1 和 table2。table1 包含 section1 的条目并。这两个段都是 UNION 的成员，因此 table2 包含 section1 和 section2 的条目并。table2 中条目的顺序并未指定。

8.9.1.3 使用 crc_table() 运算符时的接口

CRC 生成功能使用一种类似于复制表功能的机制。在链接器命令文件中使用上面显示的语法可以指定计算并存储在运行时映像中的 CRC 值的代码/数据段。本节介绍了由链接器创建的表数据结构，以及如何从应用程序代码访问此信息。

CRC 表包含运行时支持头文件 crc_tbl.h 中详述的条目，如图 8-7 所示。

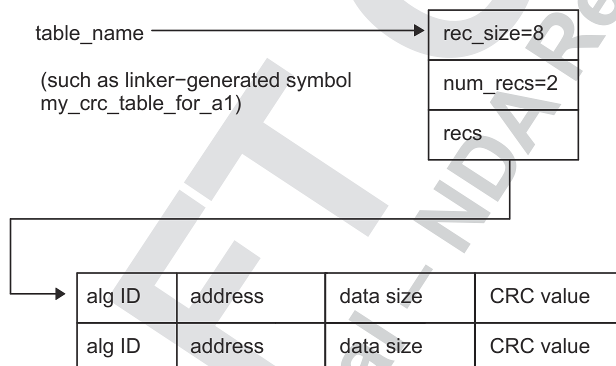


图 8-7. CRC_TABLE 概念模型

示例 8-9 提供了 crc_tbl.h 头文件。该文件指定了由链接器创建的用于管理 CRC 信息的 C 结构。该文件还包含受支持的 CRC 算法的规范。CRC 算法的详细讨论超出了本文档的范围，感兴趣的读者应查阅参考文档以了解表中所示字段的说明。以下字段与本文档相关。

- Name – 算法的文本标识符，由程序员在链接器命令文件中使用。
- ID – 算法的数字标识符，由链接器存储在每个表条目的 crc_alg_ID 成员中。
- Order – 用于 CRC 计算的位数。
- Polynomial – 由 CRC 计算引擎使用。
- Initial Value – 指定给 CRC 计算引擎的初始值。

示例 8-9. CRC 表头，crc_tbl.h

```
/****************************************************************************/
/* crc_tbl.h                                                               */
/*                                                                         */
/* 可由链接器自动生成的 CRC 表数据结构的规范      */
/* （使用链接器命令文件中的 crc_table() 操作符）。      */
/*                                                    */
```


Copyright © 2023 Texas Instruments Incorporated

```
/****************************************************************************/
/*                                                                          */
/*  链接器使用的 CRC 生成器基于以下文档中的概念：        */
/*                                                                   */
/*     "A Painless Guide to CRC Error Detection Algorithms"                 */
/*                                                                          */
/* Author : Ross Williams (ross@guest.adelaide.edu.au.).                    */
/* Date   : 3 June 1993.                                                    */
/* Status : Public domain (C code).                                         */
/*                                                                          */
/* 说明：有关 Rocksoft^tm 模型 CRC 算法的更多信息，            */
/* 请参阅 Ross Williams (ross@guest.adelaide.edu.au.) 撰写的         */
/* 文档“A Painless Guide to CRC Error Detection Algorithms”。     */
/* 该文档可能位于 "ftp.adelaide.edu.au/pub/rocksoft" 或    */
/* http:www.ross.net/crc/download/crc_v3.txt。                            */
/*                                                                          */
/* 注：Rocksoft 是澳大利亚阿德莱德 Rocksoft Pty Ltd 公司的商标。    */
/****************************************************************************/
#include <stdint.h>        /* For uintXX_t */
/****************************************************************************/
/* CRC 算法说明符                                                  */
/*                                                                          */
/* 链接器使用基于上述引用的文档的下述规范 */
/* 来生成 CRC 值。                                    */
/*
 ID   Name              Order Polynomial  Initial      Ref Ref  CRC XOR    Zero
                                          Value        In  Out  Value      Pad
-----------------------------------------------------------------------------------
 10  "TMS570_CRC64_ISO", 64, 0x0000001b, 0x00000000,  0,  0,  0x00000000,  1
                                                                          */
/* 用户应在链接器命令文件中指定名称，例如 TMS570_CRC64_ISO。     */
/*   生成的 CRC_RECORD 结构将在        */
/*  crc_alg_ID 字段中包含相应的 ID 值。                */
/****************************************************************************/
#define TMS570_CRC64_ISO  10
/*********************************************************/
/* CRC 记录数据结构                                */
/* 注：字段列表和每个字段的大小    */
/*       因目标和存储器模型而异。                */
/*********************************************************/
typedef struct crc_record
{
uint64_t        crc_value;
uint32_t        crc_alg_ID;     /* CRC 算法 ID */
uint32_t        addr;           /* 起始地址     */
uint32_t        size;           /* 数据大小（字节） */
uint32_t        padding;        /* 显式填充，因此 ELF 的布局相同 */
                                /*                    */
} CRC_RECORD;
```

示例 8-10. 通用 CRC 校验例程

在 CRC_TABLE 结构中，数组 recs[1] 由链接器动态调整大小以适应表中包含的记录数 (num_recs)。用户提供的用于验证 CRC 值的例程应采用表名并校验表中所有条目的 CRC 值。以下代码中显示了一个此类例程的概要。

```
/************************************************************/
/* General purpose CRC check routine.  Given the address of a */
/* linker-generated CRC_TABLE data structure, verify the CRC  */
/* of all object components that are designated with the      */
/* corresponding LCF crc_table() operator.                    */
/************************************************************/
#include <crc_tbl.h>
/*********************************************************/
/* MY_CHECK_CRC() - returns 1 if CRCs match, 0 otherwise */
/*********************************************************/
unsigned int my_check_CRC(CRC_TABLE *tp)
{
   int i;
   for (i = 0; i < tp-> num_recs; i++)
   {
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

```
        CRC_RECORD crc_rec = tp->recs[i];

        /****************************************************/
        /* COMPUTE CRC OF DATA STARTING AT crc_rec.addr    */
        /* FOR crc_rec.size UNITS.  USE                     */
        /* crc_rec.crc_alg_ID to select algorithm.*/
        /* COMPARE COMPUTED VALUE TO crc_rec.crc_value.    */
        /****************************************************/
    }
    if all CRCs match, return 1;
    else return 0;
}
```

8.9.2 关于 TMS570_CRC64_ISO 算法的注意事项

MCRC 模块会计算 64 位数据块的 CRC。这是通过将 long long 值写入两个存储器映射的寄存器来实现的。在 C 中，这看起来像是向存储器写入 long long 的正常操作。为在存储器中读取/写入 long long 而生成的代码如下所示，其中 R2 包含最高有效字，R3 包含最低有效字。因此，最高有效字写入到低地址，最低有效字写入到高地址：

```
    LDM R0, {R2, R3}
    STM R1, {R2, R3}
```

CRC 存储器映射的寄存器的顺序与编译器执行存储的顺序相反。最低有效字映射到低地址，最高有效字映射到高地址。

这意味着在执行 CRC 计算之前实际上交换了字。此外，也意味着计算出的 CRC 值交换了字。TMS570_CRC64_ISO 算法会考虑这些问题，并在计算 CRC 值时执行交换。在表中存储的 CRC 计算值交换了字，因此该值与存储器中的值相同。

对于最终用户而言，这些细节应该是透明的。如果运行时 CRC 例程是用 C 编写的，则将会正确生成 long long 加载和存储。MCRC 模块的 DMA 模式也将正常工作。

该算法的另一个问题是它需要使用 64 位块来完成运行时的 CRC 计算。MCRC 模块允许使用更小的数据块，但值会填充为 64 位。TMS570_CRC64_ISO 算法不会执行任何填充，因此所有 CRC 计算都必须使用 64 位值来完成。如果数据不是在 64 位边界上结束，该算法将自动用零填充数据的末尾。

8.10 部分（增量）链接

已链接的输出文件可以继续与其他模块链接。这称为*部分链接*或*增量链接*。通过部分链接，可对较大的应用程序进行分区，分别链接每个部分，然后将所有部分链接在一起，以创建最终的可执行程序。请遵循以下指导原则来生成要重新链接的文件：

- 由链接器生成的中间文件*必须*具有重定位信息。首次链接文件时使用 --relocatable 选项。（请参阅节 8.4.3.2。）
- 中间文件*必须*包含符号信息。默认情况下，链接器会在其输出中保留符号信息。如果用户打算重新链接文件，请勿使用 --no_sym_table 选项，因为 --no_sym_table 会从输出模块中去除符号信息。（请参阅节 8.4.24。）
- 只有构建输出段且未进行分配时，才应考虑使用中间链接操作。所有分配、绑定和 MEMORY 指令都应在最终链接中执行。由于 ELF 目标文件格式 搭配使用，部分链接期间不会将各输入段合并成输出段，除非链接步骤命令文件中指定了匹配的 SECTIONS 指令。
- 如果中间文件中的全局符号与其他文件中的全局符号具有相同的名称，并且用户想要将它们视为静态符号（仅在该中间文件中可见），则必须使用 --make_static 选项链接各文件（请参阅 节 8.4.19.1）。
- 如果是链接 C 代码，在最终链接器之前，请勿使用 --ram_model 或 --rom_model。每次使用 --ram_model 或 --rom_model 选项调用链接器时，链接器都会尝试创建一个入口点。（请参阅节 8.4.27 、节 3.3.2.1 和 节 3.3.2.2 。）

以下示例展示了如何使用部分链接：

Copyright © 2023 Texas Instruments Incorporated

DRAFT ONLY
Confidential - NDA Restrictions

步骤 1： 链接文件 file1.com；使用 --relocatable 选项在输出文件 out1.out 中保留重定位信息。

```
armcl --run_linker --relocatable --output_file=out1 file1.com
```

file1.com 包含：

```
SECTIONS
{
   ss1:    {
           f1.c.obj
           f2.c.obj
            . . .
           fn.c.obj
           }
}
```

步骤 2： 链接文件 file2.com；使用 --relocatable 选项以在输出文件 out2.out 中保留重定位信息。

```
armcl --run_linker --relocatable --output_file=out2 file2.com
```

file2.com 包含：

```
SECTIONS
{
   ss2:    {
           g1.c.obj
           g2.c.obj
            . . .
           gn.c.obj
           }
}
```

步骤 3： 链接 out1.out 和 out2.out。

```
armcl --run_linker --map_file=final.map --output_file=final.out out1.out out2.out
```

8.11 链接 C/C++ 代码

C/C++ 编译器生成可进行汇编和链接的汇编语言源代码。例如，可以汇编包括模块 prog1、prog2 等的 C 程序，然后进行链接以生成名为 prog.out 的可执行文件：

```
armcl --run_linker --rom_model --output_file prog.out prog1.c.obj prog2.c.obj ...
rtsv4_A_be_eabi.lib
```

--rom_model 选项告诉链接器使用由 C/C++ 环境定义的特殊约定。

TI 提供的归档库包含 C/C++ 运行时支持函数。

C、C++ 以及混合的 C 和 C++ 程序可以使用相同的运行时支持库。可以从 C 和 C++ 调用和引用的运行时支持函数和变量将具有相同的链接。

有关 ARM C/C++ 语言的更多信息，包括运行时环境和运行时支持函数，请参阅 *ARM 优化 C/C++ 编译器用户指南*。

8.11.1 运行时初始化

C/C++ 程序必须与用于初始化和执行程序的代码链接起来，这个代码被称为 *bootstrap* 例程（*boot.c.obj* 对象模块）。_c_int00 符号定义为程序入口点，是 boot.c.obj 中 C 引导例程的起点。引用 _c_int00 可确保自动从运行时支持库将 boot.c.obj 链接进来。程序在运行时会首先执行 boot.c.obj。boot.c.obj 符号包含用于初始化运行时环境的代码和数据；该符号执行以下任务：

- 从系统模式更改为用户模式
- 设置用户模式堆栈
- 处理运行时 *.cinit* 初始化表并自动初始化全局变量（如果使用了 --rom_model）
- 调用 main

运行时支持对象库中包含 boot.c.obj。用户可以：

- 使用归档器以从库中提取 boot.c.obj，然后直接链接该模块。
- 添加合适的运行时支持库作为输入文件（当用户使用 --ram_model 或 --rom_model 选项时，链接器会自动提取 boot.c.obj）。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

8.11.2 对象库和运行时支持

《*ARM 优化 C/C++ 编译器用户指南*》介绍了 rts.src 中包含的额外运行时支持函数。如果用户的程序使用这些函数，则必须将适当的运行时支持库与目标文件链接。

用户还可以自行创建对象库并链接它们。链接器仅包含和链接那些会解析未定义引用的库成员。

如果用户想将通过 TI CodeGen 工具创建的目标文件与通过其他编译器工具链生成的目标文件链接起来，根据 ARM 标准的要求，用户应先定义 _AEABI_PORTABILITY_LEVEL 预处理器符号（如下所示），然后再#包含任何标准头文件，如 <stdlib.h>。

```
#define _AEABI_PORTABILITY_LEVEL 1
```

此定义可实现完全可移植性。将符号定义为 0 明确指出，应使用“C 标准”可移植性级别。

8.11.3 设置堆栈段的大小

C/C++ 语言使用两个未初始化段，被称为 .sysmem 和 .stack 分别用于设置 malloc( ) 函数和运行时堆栈使用的存储器池。用户可以通过使用 --heap_size 或 --stack_size 选项并紧跟该选项后指定段大小作为 4 字节常量来设置这些存储器池的大小。如果不使用上述选项，堆的默认大小为 2K 字节，而栈的默认大小为 2K 字节。

请参阅节 8.4.16 来设置堆大小参阅节 8.4.31 来设置栈大小。

8.11.4 在运行时初始化和自动初始化变量

在运行时自动初始化变量是默认的自动初始化方法。若要使用此方法，请使用 --rom_model 选项调用链接器。详细信息请参阅 节 3.3.2.1 。

在加载时初始化变量可通过缩短启动时间和节省初始化表使用的存储器来提高性能。若要使用此方法，请使用 --ram_model 调用链接器。详细信息请参阅 节 3.3.2.2 。

请参阅节 3.3.2.3，了解使用 --ram_model 或 --rom_model 调用链接器时需执行的步骤。

8.11.5 Cinit 的初始化和看门狗计时器保持

可在某些器件上使用 --cinit_hold_wdt 选项来指定在 cinit 自动初始化期间是否应保持 (on) 或不保持 (off) 看门狗计时器。设置此选项会使 RTS 自动初始化例程与程序链接，从而处理所需的看门狗计时器行为。

8.12 链接器示例

此示例链接了名为 demo.c.obj、ctrl.c.obj 和 tables.c.obj 的三个目标文件，并创建了名为 demo.out 的程序。

假定目标存储器具有以下程序存储器配置：

地址范围	内容
0x0080 至 0x7000	片上 RAM_PG
0xC000 至 0xFF80	片上 ROM

地址范围	内容
0x0080 至 0x0FFF	RAM 块 ONCHIP
0x0060 至 0xFFFF	映射的外部地址 EXT

地址范围	内容
0x00000000 至 0x00001000	SLOW_MEM
0x00001000 至 0x00002000	FAST_MEM
0x08000000 至 0x08000400	EEPROM

输出段的构造方式如下：

Copyright © 2023 Texas Instruments Incorporated

- demo.c.obj、ctrl.c.obj 和 tables.c.obj 中 .text 段包含的可执行代码必须链接到 FAST_MEM。
- tables.c.obj 中 .intvecs 段包含的一组中断矢量必须在地址 FAST_MEM 处链接。
- tables.c.obj 中 .data 段包含的系数表必须链接到 EEPROM。FLASH 块的剩余部分必须初始化为值 0xFF00FF00。
- ctrl.c.obj 中 .bss 段包含的一组变量必须链接到 SLOW_MEM 并预初始化为 0x00000100。
- demo.c.obj 和 tables.c.obj 中的 .bss 段必须链接到 SLOW_MEM。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

链接器命令文件 demo.cmd 展示了此示例的链接器命令文件。输出映射文件，demo.map 展示了映射文件。

链接器命令文件 demo.cmd

```
/****************************************************************************/
/***                         指定链接选项                                 ***/
/****************************************************************************/
--entry_point SETUP                         /* 定义程序入口点   */
--output_file=demo.out                      /* 命名输出文件              */
--map_file=demo.map                         /* 创建输出映射文件        */
/****************************************************************************/
/***                         指定输入文件                              ***/
/****************************************************************************/
demo.c.obj
ctrl.c.obj
tables.c.obj
/****************************************************************************/
/***                     指定存储器配置                           ***/
/****************************************************************************/
MEMORY
{
   FAST_MEM : org = 0x00000000   len = 0x00001000 /* PROGRAM MEMORY    (ROM)   */
   SLOW_MEM : org = 0x00001000   len = 0x00001000 /* DATA MEMORY       (RAM)   */
   EEPROM   : org = 0x08000000   len = 0x00000400 /* COEFFICIENTS   (EEPROM)   */
}
/****************************************************************************/
/*                         指定输出段                              */
/****************************************************************************/
SECTIONS
{
   .text    : {} > FAST_MEM   /* 将所有 .text 段链接到 ROM            */
   .intvecs : {} > 0x0        /* 链接 0x0 处的中断矢量                */
   .data    :                 /* 链接 .data 段                              */
   {
      tables.c.obj(.data)
      .= 0x400;                   /* 在块的末尾创建空洞             */
   } > EEPROM, fill = 0xFF00FF00   /* 填充并链接至 EEPROM                 */
   ctrl_vars:                      /* 为 ctrl 变量创建新段 */
   {
      ctrl.c.obj(.bss)
   } > SLOW_MEM, fill = 0x00000100 /* 用 0x100 填充并链接至 RAM       */
   .bss     : {} > SLOW_MEM        /* 将剩余 .bss 段链接到 RAM  */
}
/****************************************************************************/
/***                          命令文件结束                               ***/
/****************************************************************************/
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

输入以下命令来调用链接器：

```
armcl --run_linker demo.cmd
```

这会创建一个如输出映射文件，demo.map 中所示的映射文件，以及一个名为 demo.out 的输出文件，可以在 ARM 器件 上运行。

输出映射文件，demo.map

```
OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: "SETUP"  address: 000000d4
MEMORY CONFIGURATION
           name       origin      length      attributes     fill
         --------   --------   ---------    ----------   --------
         FAST_MEM   00000000   000001000      RWIX
         SLOW_MEM   00001000   000001000      RWIX
         EEPROM     08000000   000000400      RWIX
SECTION ALLOCATION MAP
 output                                   attributes/
section    page    origin      length       input sections
--------   ----  ----------  ----------   ----------------
.text       0    00000020    00000138
                   00000020    000000a0     ctrl.c.obj (.text)
                   000000c0    00000000     tables.c.obj (.text)
                   000000c0    00000098     demo.c.obj (.text)
.intvecs    0    00000000    00000020
                   00000000    00000020     tables.c.obj (.intvecs)
.data       0    08000000    00000400
                   08000000    00000168     tables.c.obj (.data)
                   08000168    00000298     --HOLE-- [fill = ff00ff00]
                   08000400    00000000     ctrl.c.obj (.data)
                   08000400    00000000     demo.c.obj (.data)
ctrl_var    0    00001000    00000500
                   00001000    00000500     ctrl.c.obj (.bss) [fill = 00000100]
.bss        0    00001500    00000100     UNINITIALIZED
                   00001500    00000100     demo.c.obj (.bss)
                   00001600    00000000     tables.c.obj (.bss)
GLOBAL SYMBOLS
address  name                              address  name
-------- ----                              -------- ----
000000d4 SETUP                             00000020 clear
00000020 clear                             000000b8 set
000000b8 set                               000000c0 x42
000000c0 x42                               000000d4 SETUP
[4 symbols]
```


Copyright © 2023 Texas Instruments Incorporated

DRAFT ONLY
TI Confidential – NDA Restrictions

This page intentionally left blank.

Copyright © 2023 Texas Instruments Incorporated

章节 9

绝对列表器说明

ARM 绝对列表器是一款调试工具，支持将链接的目标文件作为输入并创建 .abs 文件作为输出。这些 .abs 文件经汇编可产生显示对象代码绝对地址的列表。如果手动，这可能是一个繁琐的过程，需要许多操作；但是，绝对列表器实用程序会自动执行这些操作。

9.1 生成绝对列表........ 262
9.2 调用绝对列表器........ 263
9.3 绝对列表器示例........ 264

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

9.1 生成绝对列表

图 9-1 展示了生成绝对列表所需的步骤。

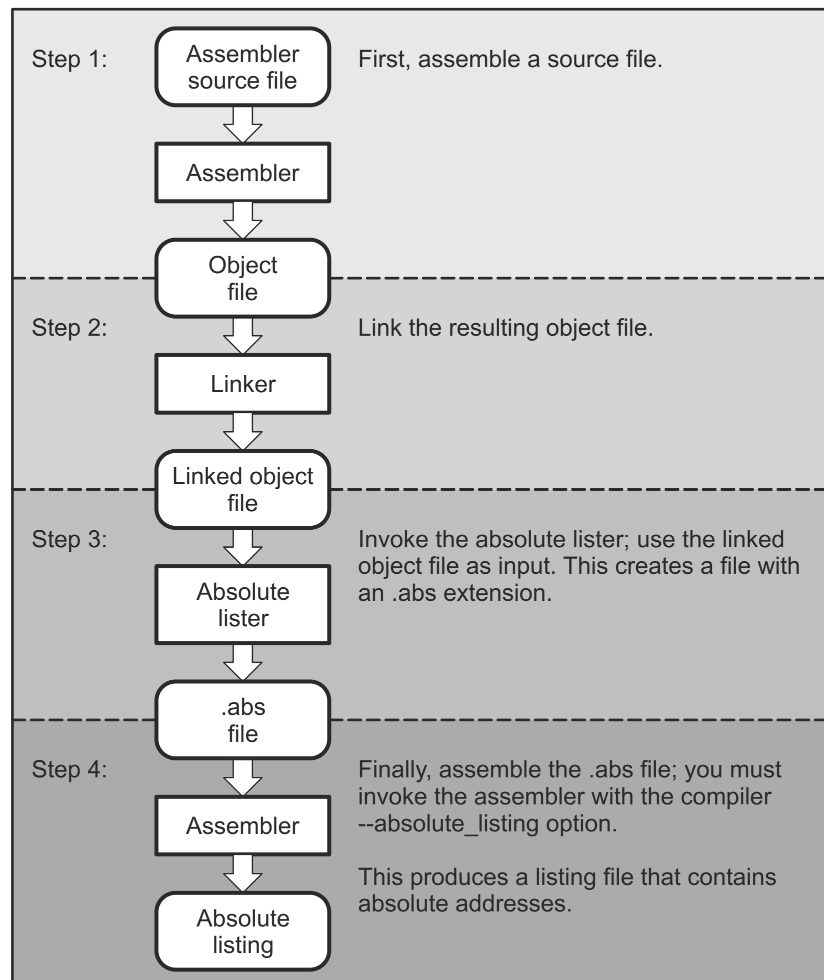


图 9-1. 绝对列表器开发流程

DRAFT TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

9.2 调用绝对列表器

调用绝对列表器的语法如下：

armabs [-*options*] *input file*

armabs 是调用绝对列表器的命令。

options 标识要使用的绝对列表器选项。选项不区分大小写，可出现在命令行中的命令之后的任意位置。在每个选项前面加连字符 (-)。绝对列表器选项如下：

-e 使您能够更改汇编文件、C 源文件和 C 头文件中的文件扩展名的默认命名规则。有效选项包括：

- ea [.]*asmext*，用于汇编文件（默认为 .asm）
- ec [.]*cext*，用于 C 源文件（默认为 .c）
- eh [.]*hext* 用于 C 头文件（默认为 .h）
- ep [.]*pext*，用于 CPP 源文件（默认为 cpp）

表达式中的 . 以及选项与扩展名之间的空格是可选的。

-q （静默）不显示横幅和所有进度信息。

input file 为链接的目标文件命名。如果不提供扩展名，则绝对列表器假定输入文件使用默认扩展名 .out。如果在调用绝对列表器时不提供输入文件名，则绝对列表器会提示您提供文件名。

绝对列表器为链接的每个文件生成一个输出文件。这些文件以输入文件名和 .abs 扩展名命名。但是，头文件不会生成相应的 .abs 文件。

按如下所示使用 --absolute_listing 汇编器选项汇编这些文件，以创建绝对列表：

armcl --absolute_listing *filename* **.abs**

-e 选项会影响对命令行上的文件名进行的解释和输出文件的名称。它们在命令行上应始终位于任何文件名之前。

当链接的目标文件通过用调试选项（--symdebug:dwarf 编译器选项）编译的 C 文件创建时，-e 选项很有用。设置了调试选项时，生成的链接目标文件包含用于构建目标文件的源文件的名称。在这种情况下，绝对列表器不会为 C 头文件生成相应的 .abs 文件。此外，与 C 源文件对应的 .abs 文件使用从 C 源文件生成的汇编文件，而不是使用 C 源文件本身。

例如，假设 C 源文件 hello.csr 使用调试选项集进行编译；则调试选项会生成汇编文件 hello.s。hello.csr 文件包括 hello.hsr。假设创建的可执行文件被称为 hello.out，则以下命令会生成适合的 .abs 文件：

```
armabs -ea s -ec csr -eh hsr hello.out
```

不会为 hello.hsr（头文件）创建 .abs 文件，并且 hello.abs 包括汇编文件 hello.s，而不包括 C 源文件 hello.csr。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

9.3 绝对列表器示例

此示例使用三个源文件：module1.asm 和 module2.asm，这两个文件都包含 globals.def。

module1.asm

```
         .text
         .bss    dflag, 1
         .bss    array, 100
dflag_a  .word   dflag
array_a  .word   array
offst_a  .word   offst
         .copy   globals.def
         LDR     r4, array_a
         LDR     r5, offst_a
         LDR     r3, dflag_a
         LDR     r0, [r4, r5]
         STR     r0, [r3]
```

module2.asm

```
         .text
         .bss    offst, 1
offst_a  .word   offst
         .copy   globals.def
         LDR     r4, offst_a
         STR     r0, [r4]
```

globals.def

```
         .global array
         .global offst
         .global dflag
```

以下步骤会为 **module1.asm** 和 **module2.asm** 文件创建绝对列表：

步骤 1： 首先汇编 **module1.asm** 和 **module2.asm**：

```
armcl module1
armcl module2
```

这将创建名为 module1.obj 和 module2.obj 的两个目标文件。

步骤 2： 接下来，使用以下名为 bttest.cmd 的链接器命令文件链接 module1.obj 和 module2.obj：

```
--output_file=bttest.out
--map_file=bttest.map
module1.obj
module2.obj
MEMORY
{
   P_MEM :  org = 0x00000000  len = 0x00001000
   D_MEM :  org = 0x00001000  len = 0x00001000
}
SECTIONS
{
   .data:  >D_MEM
   .text:  >P_MEM
   .bss:   >D_MEM
}
```

调用链接器：

```
armcl --run_linker bttest.cmd
```

此命令会创建一个名为 bttest.out 的可执行目标文件；使用此文件作为绝对列表器的输入。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

步骤 3： 下面，调用绝对列表器：

```
armabs bttest.out
```

此命令会创建名为 module1.abs 和 module2.abs 的两个文件：

module1.abs：

```
                .nolist
    array       .setsym     000001001h
    dflag       .setsym     000001000h
    offst       .setsym     000001068h
    .data       .setsym     000001000h
    edata       .setsym     000001000h
    .text       .setsym     000000000h
    etext       .setsym     00000002ch
    .bss        .setsym     000001000h
    end         .setsym     00000106ch
                .setsect    ".text",000000000h
                .setsect    ".data",000001000h
                .setsect    ".bss",000001000h
                .list
                .text
                .copy       "module1.asm"
```

module2.abs：

```
                .nolist
    array       .setsym     000001001h
    dflag       .setsym     000001000h
    offst       .setsym     000001068h
    .data       .setsym     000001000h
    edata       .setsym     000001000h
    .text       .setsym     000000000h
    etext       .setsym     00000002ch
    .bss        .setsym     000001000h
    end         .setsym     00000106ch
                .setsect    ".text",000000020h
                .setsect    ".data",000001000h
                .setsect    ".bss",000001068h
                .list
                .text
                .copy       "module2.asm"
```

这些文件包含汇编器在步骤 4 中需要的以下信息：

- 它们包含 .setsym 指令，将值等同于全局符号。两个文件都包含 *dflag* 符号的全局等价。符号 dflag 在 globals.def 文件中定义，该文件包含在 module1.asm 和 module2.asm 中。
- 它们包含 .setsect 指令，用于定义段的绝对地址。
- 它们包含 .copy 指令，用于定义要包含的汇编语言源文件。

.setsym 和 .setsect 指令仅用于创建绝对列表，而不是正常汇编。

步骤 4： 最后，汇编由绝对列表器创建的 .abs 文件（请记住，调用汇编器时必须使用 --absolute_listing 选项）：

```
armcl --absolute_listing module1.abs
armcl --absolute_listing module2.abs
```

此命令序列会创建名为 module1.lst 和 module2.lst 的两个列表文件；不会生成对象代码。这些列表文件类似于正常的列表文件；但是，显示的地址是绝对地址。

创建的绝对列表文件是 module1.lst（参见 module1.lst）和 module2.lst（参见 module2.lst）。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

module1.lst

```
module1.abs                                                              PAGE    1
      15 00000000                    .text
      16                             .copy       "module1.asm"
 A     1 00000000                  .text
 A     2 00001000                  .bss    dflag, 1
 A     3 00001001                  .bss    array, 100
 A     4 00000000 00001000- dflag_a .word   dflag
 A     5 00000004 00001001- array_a .word   array
 A     6 00000008 00001068! offst_a .word   offst
 A     7                           .copy   globals.def
 B     1                           .global array
 B     2                           .global offst
 B     3                           .global dflag
 A     8
 A     9 0000000c E51F4010          LDR     r4, array_a
 A    10 00000010 E51F5010          LDR     r5, offst_a
 A    11 00000014 E51F301C          LDR     r3, dflag_a
 A    12 00000018 E7940005          LDR     r0, [r4, r5]
 A    13 0000001c E5830000          STR     r0, [r3]
 No Errors, No Warnings
```

module2.lst

```
module2.abs                                                              PAGE    1
      15 00000020                    .text
      16                             .copy       "module2.asm"
 A     1 00000020                  .text
 A     2 00001068                  .bss    offst, 1
 A     3 00000020 00001068- offst_a .word   offst
 A     4                           .copy   globals.def
 B     1                           .global array
 B     2                           .global offst
 B     3                           .global dflag
 A     5
 A     6 00000024 E51F400C          LDR     r4, offst_a
 A     7 00000028 E5840000          STR     r0, [r4]
 No Errors, No Warnings
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

章节 10

交叉参考列表器说明

ARM 交叉参考列表器是一个调试工具。此实用程序接受所链接的目标文件作为输入，并生成交叉参考列表作为输出。此列表显示了符号、它们的定义以及它们在所链接的源文件中的引用。

10.1 生成交叉参考列表......268

10.2 调用交叉参考列表器......269

10.3 交叉参考列表示例......270

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

10.1 生成交叉参考列表

图 10-1 展示了生成交叉参考列表所需的步骤。

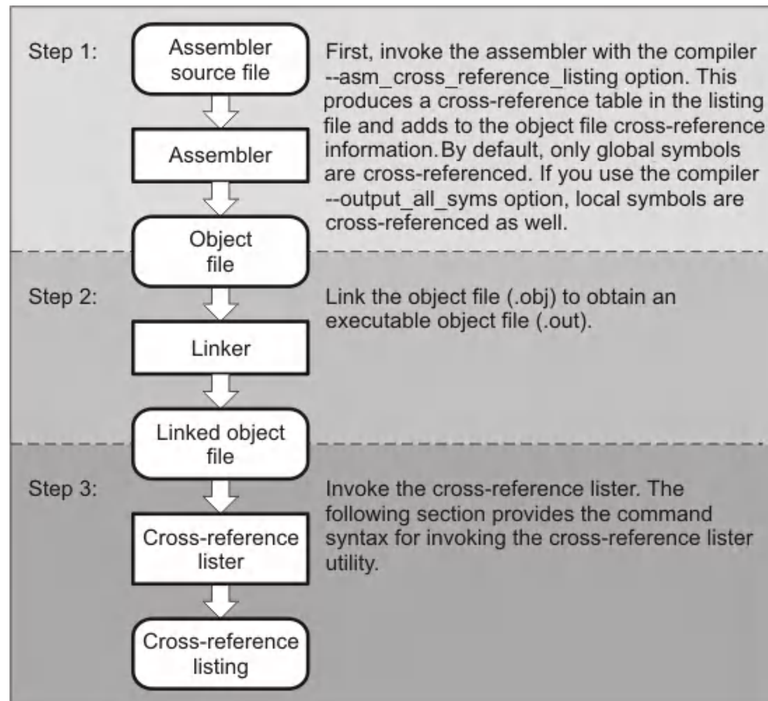


图 **10-1.** 交叉参考列表器开发流程

DRAFT TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

10.2 调用交叉参考列表器

若要使用交叉参考实用工具，必须使用正确的选项汇编文件，然后链接至可执行文件。使用 --asm_cross_reference_listing 选项对汇编语言文件进行汇编（请参阅节 4.14）。此选项会创建交叉参考列表并将交叉参考信息添加到目标文件。默认情况下，汇编器仅交叉参考全局符号，但如果使用 --output_all_syms 选项来调用汇编器，则也可以添加局部符号。链接目标文件以获取可执行文件。

若要调用交叉参考列表器，请输入以下命令：

armxref [*options*] [*input filename* [*output filename*]]

armxref 是调用交叉参考实用程序的命令。

options 标识要使用的交叉参考列表器选项。选项不区分大小写，可出现在命令行中的命令之后的任意位置。

- **-l** （小写 L）指定输出文件每页的行数。-l 选项的格式为 -l*num*，其中 num 是一个十进制常数。例如，-l30 将输出文件中的每页行数设置为 30。选项和十进制常数之间的空格是可选的。默认为每页 60 行。
- **-q** 不显示横幅和所有进度信息（静默运行）。

input filename 是已链接的目标文件。如果省略输入文件名，则该实用程序会提示输入文件名。

output filename 是交叉参考列表文件的名称。如果省略输出文件名，则默认文件名是带有 .xrf 扩展名的输入文件名。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

10.3 交叉参考列表示例

定义的这些术语出现在交叉参考列表的交叉参考列表中：

Symbol 所列符号的名称

Filename 出现符号的文件的名称

RTYP 该符号在此文件中的引用类型。可能的引用类型有：

- **STAT** 该符号在此文件中定义，未声明为全局符号。
- **EDEF** 该符号在此文件中定义，已声明为全局符号。
- **EREF** 该符号未在此文件中定义，但作为全局符号引用。
- **UNDF** 该符号未在此文件中定义，未声明为全局符号。

AsmVal 这个十六进制数是在汇编时分配给符号的值。值的前面也可以有一个描述符号属性的字符。表 10-1 列出了这些字符和名称。

LnkVal 这个十六进制数是在链接后分配给符号的值。

DefLn 定义该符号的语句编号。

RefLn 引用该符号的行编号。如果行编号后跟星号 (*)，则该引用可以修改对象的内容。此列空白表示从未使用过该符号。

表 10-1. 交叉参考列表中的符号属性

字符	含义
'	.text 段中定义的符号
"	.data 段中定义的符号
+	.sect 段中定义的符号
-	.bss 或 .usect 段中定义的符号

交叉参考列表是交叉参考列表的示例。

交叉参考列表

```
File:  bttest.out          Wed Nov 13 17:07:42 xxxxPage:   1
==============================================================================
Symbol: array
Filename      RTYP    AsmVal      LnkVal       DefLn    RefLn     RefLn     RefLn
________      ____   ________    ________      _____    _______   _______   _______
module1.asm   EDEF  -00000001    00001001        3        1A        5
==============================================================================
Symbol: array_a
Filename      RTYP    AsmVal      LnkVal       DefLn    RefLn     RefLn     RefLn
________      ____   ________    ________      _____    _______   _______   _______
module1.asm   STAT  '00000004    00000004        5        9
==============================================================================
Symbol: dflag
Filename      RTYP    AsmVal      LnkVal       DefLn    RefLn     RefLn     RefLn
________      ____   ________    ________      _____    _______   _______   _______
module1.asm   EDEF  -00000000    00001000        2        3A        4
==============================================================================
Symbol: dflag_a
Filename      RTYP    AsmVal      LnkVal       DefLn    RefLn     RefLn     RefLn
________      ____   ________    ________      _____    _______   _______   _______
module1.asm   STAT  '00000000    00000000        4       11
==============================================================================
Symbol: offst
Filename      RTYP    AsmVal      LnkVal       DefLn    RefLn     RefLn     RefLn
________      ____   ________    ________      _____    _______   _______   _______
module1.asm   EREF   00000000    00001068                 2A        6
module2.asm   EDEF  -00000000    00001068        2        2A        3
==============================================================================
Symbol: offst_a
Filename      RTYP    AsmVal      LnkVal       DefLn    RefLn     RefLn     RefLn
________      ____   ________    ________      _____    _______   _______   _______
module1.asm   STAT  '00000008    00000008        6       10
module2.asm   STAT  '00000000    00000020        3        6
==============================================================================
```

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

章节 11
目标文件实用程序

本章介绍了如何调用以下实用程序：

- **目标文件显示实用程序**以文本格式和 XML 格式打印目标文件、可执行文件和/或归档库的内容。
- **反汇编器**以目标文件和可执行文件作为输入并生成汇编列表作为输出。此列表展示了汇编指令、对应的操作码以及段程序计时器值。
- **名称使用程序**打印目标文件、可执行文件和/或归档库中定义和引用的名称列表。
- **符号去除实用程序**从目标文件和可执行文件中删除符号表和调试信息。
- **objcopy、objdump、readelf 和 size 实用程序**功能与对应的 Unix 实用程序相似。在 Microsoft Windows 上，这些实用程序的可执行名称如下所示：Unix 版本与此相同，但没有 .exe 后缀。
  - arm-none-eabi-objcopy.exe
  - arm-none-eabi-objdump.exe
  - arm-none-eabi-readelf.exe
  - arm-none-eabi-size.exe

11.1 调用目标文件显示实用程序........272
11.2 调用反汇编器........272
11.3 调用名称实用程序........274
11.4 调用符号去除实用程序........275

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

11.1 调用目标文件显示实用程序

目标文件显示实用程序 *armofd* 以文本和 XML 格式显示目标文件（.obj）、可执行文件（.out）和/或归档库（.lib）的内容。隐藏的符号列为*无名称*，而局部化的符号像任何其他局部符号一样列出。

若要调用目标文件显示实用程序，请输入以下命令：

armofd [*options*] *input filename* [*input filename*]

armofd 是调用目标文件显示实用程序的命令。

input filename 指定目标文件 (.obj)、可执行文件 (.out) 或归档库 (.lib) 源文件的名称。文件名必须包含扩展名。

options 标识要使用的目标文件显示实用程序选项。选项不区分大小写，可出现在命令行中的命令之后的任意位置。在每个选项前面加一个连字符。

- **--call_graph** 以 XML 格式显示函数栈使用信息和被调用函数信息。虽然开发人员可以访问 XML 输出，但这个选项主要是为 Code Composer Studio 等工具而设计，用来显示应用程序在最坏情况下对堆栈的使用。
- **--dwarf_display**=*attributes* 通过指定以逗号分隔的*属性* 列表来控制 DWARF 显示过滤器设置。给属性加上前缀 no disables 而不是 enables。例如：

  --dwarf_display=nodabbrev,nodline
  --dwarf_display=all,nodabbrev
  --dwarf_display=none,dinfo,types

  属性的顺序很重要（请参阅 --obj_display）。可通过调用 armofd --dwarf_display=help 获取可用显示属性列表。
- **--dynamic_info** 输出动态链接信息。
- **--dwarf** 将 DWARF 调试信息附加到程序输出中。
- **--help** 显示帮助。
- **--output**=*filename* 将程序输出发送至名为 *filename* 的文件，而不是显示在屏幕上。
- **--obj_display** *attributes* 通过指定以逗号分隔的*属性* 列表来控制目标文件显示过滤器设置。给属性加上前缀 no disables 而不是 enables。例如：

  --obj_display=rawdata,nostrings
  --obj_display=all,norawdata
  --obj_display=none,header

  属性的顺序很重要。例如，在“--obj_display=none,header”中，armofd 禁用所有输出，然后重新启用文件头信息。如果以相反的顺序指定属性 (header,none)，则启用文件头，禁用所有输出，包括文件头。因此，屏幕上不会显示给定文件的任何内容。可通过调用 armofd --obj_display=help 获取可用显示属性列表。
- **--verbose** 显示详细的文本输出。
- **--xml** 以 XML 格式显示输出。
- **--xml_indent**=*num* 设置嵌套 XML 标签的空格数。

如果将一个归档文件作为目标文件显示实用程序的输入，则就像在命令行上进行传递一样来处理归档的每个目标文件成员。按照目标文件成员在归档文件中出现的顺序来处理它们。

如果在不带任何选项的情况下调用目标文件显示实用程序，则在控制台屏幕上显示有关输入文件内容的信息。

备注

目标文件显示格式：目标文件显示实用程序默认以文本格式生成数据。此数据不打算用作进一步处理这些信息的程序的输入。应使用 XML 格式进行机械处理。

11.2 调用反汇编器

反汇编器 *armdis* 用于检查汇编器或链接器的输出。这个实用程序接受目标文件或可执行文件作为输入，并将反汇编的目标代码写入标准输出或指定文件。

若要调用反汇编器，请输入以下命令：

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

armdis [*options*] *input filename*[.] [*output filename*]

armdis 是用于调用反汇编器的命令。

options 标识要使用的名称实用程序选项。选项不区分大小写，可出现在命令行中的调用之后的任意位置。在每个选项前面加一个连字符（-）。命名实用程序选项如下所示：

armdis 是用于调用反汇编器的命令。

options 标识要使用的名称实用程序选项。选项不区分大小写，可出现在命令行中的调用之后的任意位置。在每个选项前面加一个连字符（-）。命名实用程序选项如下所示：

-a	禁止列印指令内的地址和标签名称。
-b	将数据显示为字节而非字。
-be8	在 BE-8 模式下进行反汇编。
-c	转储目标文件信息。
--copy_tables	（别名为 -y 或 -Y）显示复制表和已复制的段。首先转储表信息，然后转储每个记录及其加载和运行数据。请参阅示例 11-3。
-d	禁止显示数据段。
-e	以十六进制显示整数值。
-h	显示当前帮助屏幕。
-i	将数据段反汇编为文本。
-I	将文本反汇编为数据。
-n	转储符号表。
-q	（静默模式）不显示横幅和所有进度信息。
-qq	（超级静默模式）不显示所有头文件。
-r	使用原始寄存器 ID（R0、R1 等）。
-R	显示运行时地址（如果与加载时地址不同）。
-s	不列印地址和数据字。

input filename[.*ext*] 是输入文件的名称。如果未指定可选扩展名，则按以下顺序搜索文件：

1. *infile*
2. *infile*.out，一个可执行文件
3. *infile*.obj，一个目标文件

output filename 是将反汇编代码写入其中的可选输出文件的名称。如果未指定输出文件名，则将反汇编代码写入标准输出。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

编译示例 11-1 中的示例文件时，汇编器会生成目标文件 memcpy32.obj。

*示例 **11-1.** 目标文件 **memcpy32.asm***

```
            .global C_MEMCPY
C_MEMCPY: .asmfunc stack_usage(12)
      CMP   r2, #0                 ; CHECK FOR n == 0
      BXEQ  lr               ;
      STMFD sp!, {r0, lr}          ; SAVE RETURN VALUE AND ADDRESS
      TST   r1, #0x3         ; CHECK ADDRESS ALIGNMENT
      BNE   _unaln                 ; IF NOT WORD ALIGNED, HANDLE SPECIALLY
      TST   r0, #0x3         ;
      BNE   _saln            ;
_aln: CMP   r2, #16                ; CHECK FOR n >= 16
      BCC   _l16             ;
      STMFD sp!, {r4}        ;
      SUB   r2, r2, #16      ;
```

如示例 11-2 中所示，反汇编器可以从目标文件 memcpy32.obj 生成反汇编代码。前两行已输入命令行。

*示例 **11-2.** 从 **memcpy32.asm** 进行反汇编*

```
TEXT Section .text, 0x180 bytes at 0x0
000000:              C_MEMCPY:
000000:               .state32
000000: E3520000         CMP            R2, #0
000004: 012FFF1E         BXEQ           R14
000008: E92D4001         STMFD          R13!, {R0, R14}
00000c: E3110003         TST            R1, #3
000010: 1A00002B         BNE            0x000000C4
000014: E3100003         TST            R0, #3
000018: 1A00002F         BNE            0x000000DC
00001c: E3520010         CMP            R2, #16
000020: 3A000008         BCC            0x00000048
000024: E92D0010         STMFD          R13!, {R4}
000028: E2422010         SUB            R2, R2, #16
```

示例 11-3 举例说明了当复制记录引用不同的加载和运行段并使用了 --copy_table 选项时将如何显示。

*示例 **11-3.** 具有不同加载和运行地址的部分复制记录输出*

```
     COPY TABLE: _data2_ctbl, 0x30 at 0x5E10, 1 record(s)
     _data2_ctbl[0]: load addr=0x200158, size=0x12B, encoding=lzss
     DATA Section .data2_scn.load, 0x12B bytes at 0x200158
     200158:                $d:
     200158: 020f0000           .word 0x020f0000
     20015c: beef0003           .word 0xbeef0003
     .
     .
     _data2_ctbl[0]: run addr=0x52A0, size=0x960
     DATA Section .data1_scn, 0x960 bytes at 0x52A0
     0052a0:                data1:
     0052a0:                $d:
     0052a0:                .data1_scn:
     0052a0: 0000beef           .word 0x0000beef
     0052a4: 0000beef           .word 0x0000beef
     .
     .
```

11.3 调用名称实用程序

名称实用程序 *armnm* 列印目标文件、可执行文件或归档库中定义和引用的名称列表。它还会列印符号值和符号类型的指示。隐藏符号列为 ""。若要调用名称实用程序，请输入以下命令：

Copyright © 2023 Texas Instruments Incorporated

armnm [-*options*] [*input filenames*]

armnm 是调用名称实用程序的命令。

input filename 是目标文件 (.obj)、可执行文件 (.out) 或归档库 (.lib)。

options 标识要使用的名称实用程序选项。选项不区分大小写，可出现在命令行中的调用之后的任意位置。在每个选项前面加连字符 (-)。名称实用程序选项如下：

--all (-a)	列印所有符号。
--prep_fname (-f)	在每个符号前面加上文件名。
--global (-g)	仅列印全局符号。
--help (-h)	展示了当前帮助屏幕。
--format:long (-l)	生成符号信息的详细列表。
--sort:value (-n)	按数字而不是按字母顺序对符号进行排序。
--output (-o) *file*	输出到给定文件。
--sort:none (-p)	使名称实用程序不对任何符号进行排序。
--quiet (-q)	（静默模式）不显示横幅和所有进度信息。
--sort:reverse (-r)	按相反顺序对符号进行排序。
--dynamic (-s)	列出 ELF 目标模块的动态符号表中的符号。
--undefined (-u)	仅列印未定义的符号。

11.4 调用符号去除实用程序

符号去除实用程序 *armstrip* 可从目标文件和可执行文件中删除符号表和调试信息。若要调用符号去除实用程序，请输入以下命令：

armstrip [**-p**] *input filename* [*input filename*]

armstrip 是调用符号去除实用程序的命令。

input filename 是目标文件 (.obj) 或可执行文件 (.out)。

options 标识要使用的符号去除实用程序选项。选项不区分大小写，可出现在命令行中的调用之后的任意位置。在每个选项前面加连字符 (-)。符号去除实用程序选项如下：

--help (-h)	显示帮助信息。
--outfile (-o) *filename*	将去除符号的输出写入文件名。
--postlink (-p)	删除执行不需要的所有信息。与默认行为相比，此选项会删除更多的信息，但目标文件会处于无法链接的状态。此选项应只与可执行 (.out)文件一同使用。
--rom	去除只读段和程序段的符号。

在不带 -o 选项的情况下调用符号去除实用程序时，输入目标文件替换为已去除符号的版本。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

This page intentionally left blank.

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

章节 12
十六进制转换实用程序说明

ARM 汇编器和链接器创建的目标文件采用二进制格式，鼓励模块化编程，并提供强大而灵活的方法来管理代码段和目标系统存储器。

大多数 EPROM 编程器不接受目标文件作为输入。十六进制转换实用程序将目标文件转换为几种标准的 ASCII 十六进制格式之一，适合加载至 EPROM 编程器。该实用程序在其他需要对目标文件进行十六进制转换的应用中也很有用（例如，当使用调试器和加载程序时）。

十六进制转换实用程序可以生成以下输出文件格式：

- ASCII 十六进制，支持 16 位地址（请参阅节 12.15.1）
- 8 位格式的二进制文件（请参阅节 12.3.2）
- 扩展的 Tektronix (Tektronix)（请参阅节 12.15.4）
- Intel MCS-86 (Intel)（请参阅节 12.15.2）
- Motorola Exorciser (Motorola-S)，支持 16 位地址（请参阅节 12.15.3）
- 德州仪器 (TI) SDSMAC (TI-Tagged)，支持 16 位地址（请参阅节 12.15.5）
- 德州仪器 (TI) TI-TXT 格式，支持 16 位地址（请参阅节 12.15.6）
- C 数组

12.1 软件开发流程中十六进制转换实用程序的作用....278
12.2 调用十六进制转换实用程序....279
12.3 理解存储器宽度....282
12.4 ROMS 指令....287
12.5 SECTIONS 指令....290
12.6 加载映像格式（**--load_image** 选项）....290
12.7 排除指定段....291
12.8 分配输出文件名....292
12.9 映像模式和 **--fill** 选项....293
12.10 数组输出格式....294
12.11 为片上引导加载程序编译引导表....295
12.12 在 **TMS320F2838x** 器件上使用安全闪存引导功能....300
12.13 控制 **ROM** 器件地址....301
12.14 控制十六进制转换实用程序诊断....302
12.15 目标格式说明....303

DRAFT ONLY
TI & NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.1 软件开发流程中十六进制转换实用程序的作用

图 12-1 强调了软件开发流程中十六进制转换实用程序的作用。

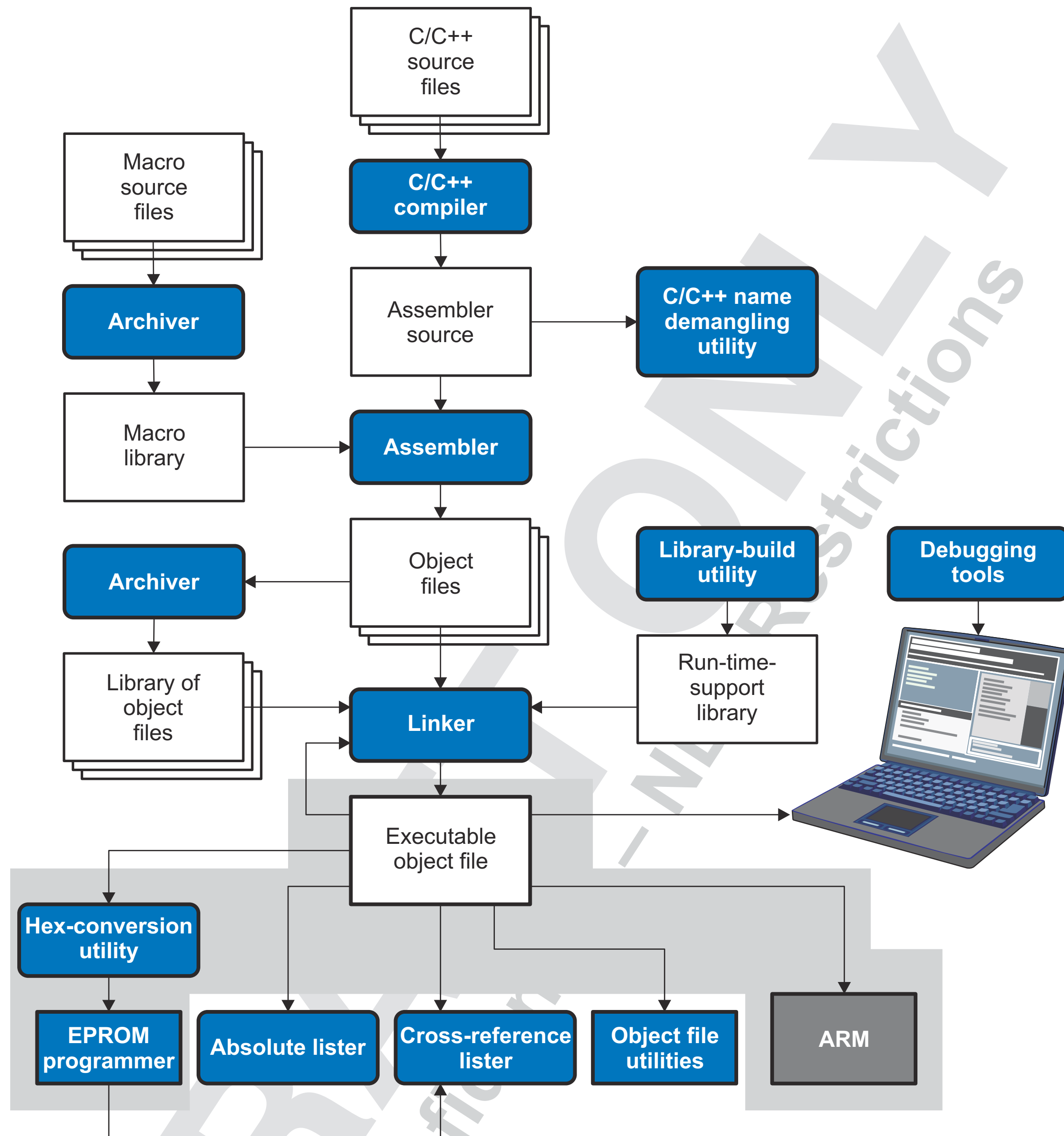


图 12-1. ARM 软件开发流程中的十六进制转换实用程序

DRAFT ONLY
TI confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.2 调用十六进制转换实用程序

调用十六进制转换实用程序有两种基本方法：

- **在命令行中指定选项和文件名。**以下示例将文件 firmware.out 转换为 TI-Tagged 格式，生成两个输出文件：firm.lsb 和 firm.msb。

```
armhex -t firmware -o firm.lsb -o firm.msb
```

- **在命令文件中指定选项和文件名。**您可以创建一个文件，该文件存储用于调用十六进制转换实用程序的命令行选项和文件名。以下示例使用名为 hexutil.cmd 的命令文件调用实用程序：

```
armhex hexutil.cmd
```

除了常规命令行信息，您可以在命令文件中使用十六进制转换实用程序 ROMS 和 SECTIONS 指令。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.2.1 从命令行调用十六进制转换实用程序

若要调用十六进制转换实用程序，请输入以下命令：

armhex [*options*] *filename*

armhex 是调用十六进制转换实用程序的命令。

options 提供用于控制十六进制转换过程的额外信息。您可以在命令行上或命令文件中使用选项。表 12-1 列出了基本选项。

- 所有选项前面加连字符，且不区分大小写。
- 有几个选项具有附加参数，这些参数必须与该选项至少用一个空格隔开。
- 具有多字符名称的选项的拼写必须与本文档中所示完全一致；不允许使用缩写。
- 选项不受其使用顺序的影响。此规则的例外情况是 --quiet 选项，它必须在所有其他选项之前使用。

filename 指定目标文件或命令文件的名称（如需更多信息，请参阅节 12.2.2）。

表 12-1. 基本十六进制转换实用程序选项

选项	别名	说明	请参阅
		通用选项	
--byte	-byte	按字节而不是按目标地址对输出位置进行编号	--
--entrypoint=*addr*	-e	指定引导加载后开始执行的入口点	节 12.11.3
--exclude={*fname*(*sname*) | *sname*}	-exclude	如果省略了文件名 (*fname*)，则将排除匹配 *sname* 的所有段。	节 12.7
--fill=*value*	-fill	使用 *值* 来填充孔洞	节 12.9.2
--help	-options, -h	显示调用实用程序的语法并列出可用选项。如果选项后跟另一个选项或词组，则会显示有关该选项或词组的详细信息。	节 12.2.2
--image	-image	选择映像模式	节 12.9.1
--linkerfill	-linkerfill	在映像中包括链接器填充段	--
--map=*filename*	-map	生成映射文件	节 12.4.2
--memwidth=*value*	-memwidth	定义系统存储器字宽度（默认为 16 位）	节 12.3.2
--outfile=*filename*	-o	指定输出文件名	节 12.8
--quiet	-q	以静默方式运行（使用时，它必须显示在其他选项*前面*）	节 12.2.2
--romwidth=*value*	-romwidth	指定 ROM 器件宽度（默认值取决于所用的格式）。对于 TI-TXT、二进制和 TI-Tagged 格式，忽略此选项。	节 12.3.3
--zero	-zero, -z	在映像模式下将地址原点重置为 0	节 12.9.3
		诊断选项	
--diag_error=*id*		将由 *id* 标识的诊断分类为错误	节 12.14
--diag_remark=*id*		将由 *id* 标识的诊断分类为备注	节 12.14
--diag_suppress=*id*		抑制由 *id* 标识的诊断。	节 12.14
--diag_warning=*id*		将由 *id* 标识的诊断分类为警告	节 12.14
--display_error_number		显示诊断的标识符及其文本	节 12.14
--issue_remarks		发出备注（非严重警告）	节 12.14
--no_warnings		抑制警告诊断（仍会发出错误）	节 12.14
--set_error_limit=*count*		将错误限制设置为 *count*。在达到此错误数量后，链接器将放弃链接。（默认为 100。）	节 12.14
		引导选项	
--cmac=file		指定包含 CMAC 密钥以与 TMS320F2838x 器件上的安全闪存引导一同使用的文件。	节 12.12
		输出选项	
--array		选择数组输出格式	节 12.10
--ascii	-a	选择 ASCII 十六进制	节 12.15.1

Copyright © 2023 Texas Instruments Incorporated

表 12-1. 基本十六进制转换实用程序选项 (continued)

选项	别名	说明	请参阅
--binary	-b	选择二进制（必须具有 8 位存储器宽度。）	--
--intel	-i	选择 Intel	节 12.15.2
--motorola=1	-m1	选择 Motorola-S1	节 12.15.3
--motorola=2	-m2	选择 Motorola-S2	节 12.15.3
--motorola=3	-m3	选择 Motorola-S3 （默认 -m 选项）	节 12.15.3
--tektronix	-x	选择 Tektronix（如果未指定输出选项，则为默认格式）	节 12.15.4
--ti_tagged	-t	选择 TI-Tagged（必须具有 16 位存储器宽度。）	节 12.15.5
--ti_txt		选择 TI-Txt（必须具有 8 位存储器宽度。）	节 12.15.6
		加载映像选项	
--load_image		输出具有加载映像对象格式的文件	节 12.6
--load_image:combine_sections =[true|false]		指定是否应合并段。默认值为 true。	节 12.6
--load_image:endian=[big|little]		指定目标文件字节序。如果省略了此选项，则使用命令行上第一个文件的字节序。	节 12.6
--load_image:file_type =[relocatable|executable]		指定目标文件以外的其他文件类型。目标文件可以相互链接，但会丢失地址。可重定位文件包含段的 sh_addr 字段中的地址。可执行文件维持地址绑定并可以直接加载。	节 12.6
--load_image:format=[coff|elf]		指定目标文件的 ABI 格式。如果省略了此选项，则通过命令行上第一个文件确定格式。	节 12.6
--load_image:globalize=*string*		不对指定符号进行本地化。可以使用 --load_image:symbol_binding 选项来设置默认值。	节 12.6
--load_image:localize=*string*		使指定符号变为局部符号。可以使用 --load_image:symbol_binding 选项来设置默认值。	节 12.6
--load_image:machine=[ARM|C2000|C6000|C7X|MSP430|PRU]		指定目标文件机器类型。如果省略了此选项，则使用命令行上第一个文件的机器类型。	节 12.6
--load_image:output_symbols =[true|false]		指定符号是否应输出到文件。默认值为 false。	节 12.6
--load_image:section_addresses =[true|false]		指定加载地址是否应写入输出文件中。仅应用至可重定位文件。默认值为 true。	节 12.6
--load_image:section_prefix =*string*		为段名指定前缀。默认值为“image_”。	节 12.6
--load_image:symbol_binding =[local|global]		指定加载映像中符号的默认绑定方式。	节 12.6

--section_name_prefix 选项已被弃用，并已替换为 --load_image:section_prefix。无文档记载的 --host_image 选项已替换为在很多情况下是相似的 --load_image 选项。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.2.2 使用命令文件调用十六进制转换实用程序

如果您打算使用相同的输入文件和选项多次调用实用程序，命令文件会很有用。如果要使用 ROMS 和 SECTIONS 十六进制转换实用程序来自定义转换过程，它也会很有用。

命令文件是包含以下一项或多项的 ASCII 文件：

- **选项和文件名。**在命令文件中使用与命令行中完全相同的方式指定它们。
- **ROMS 指令。**ROMS 指令将系统的物理存储器配置定义为地址范围参数列表。（请参阅节 12.4。）
- **SECTIONS 指令。**十六进制转换实用程序 SECTIONS 指令可指定要从目标文件中选择哪些段。（请参阅节 12.5。）
- **注释。**您可以使用 /* 和 */ 分隔符向命令文件添加注释。例如：

```
/*   This is a comment.   */
```

若要调用该实用程序并使用您在命令文件中定义的选项，请输入以下命令：

armhex *command_filename*

您还可以在命令行中指定其他选项和文件。例如，您可以使用命令文件和命令行选项调用实用程序：

```
armhex firmware.cmd --map=firmware.mxp
```

这些选项和文件名出现的顺序并不重要。在开始转换过程之前，实用程序会从命令行读取所有输入并从命令文件读取所有信息。但是，如果您使用 -q 选项，*它必须是命令行上或命令文件中的第一个选项。*

使用 **--help** 选项可显示调用编译器的语法并列出可用选项。如果 --help 选项后跟另一个选项或词组，则会显示有关该选项或词组的详细信息。例如，若要查看与生成引导表关联的选项的相关信息，请使用 --help boot。

使用 **--quiet** 选项抑制显示十六进制转换实用程序的正常横幅和进度信息。

- 假设名为 firmware.cmd 的命令文件包含这些行：

```
firmware.out          /* input file  */
--ti-tagged           /* TI-Tagged   */
--outfile=firm.lsb    /* output file */
--outfile=firm.msb    /* output file */
```

  您可以通过输入以下命令来调用十六进制转换实用程序：

```
armhex firmware.cmd
```

- 此示例展示了如何将名为 appl.out 的文件转换为八个 Intel 格式的十六进制文件。每个输出文件为一个字节宽，4K 字节长。

```
appl.out          /* input file   */
--intel                /* Intel format */
--map=appl.mxp     /* map file      */
ROMS
{
   ROW1: origin=0x00000000 len=0x4000 romwidth=8
         files={ appl.u0 appl.u1 appl.u2 appl.u3 }
   ROW2: origin=0x00004000 len=0x4000 romwidth=8
         files={ appl.u4 appl.u5 appl.u6 appl.u7 }
}
SECTIONS
{    .text, .data, .cinit, .sect1, .vectors, .const:
}
```

12.3 理解存储器宽度

十六进制转换实用程序支持指定存储器和 ROM 宽度，以使存储器架构更加灵活。如需使用十六进制转换实用程序，您必须理解该实用程序是如何处理字宽度的。在转换过程中，有三个宽度很重要：

- 目标宽度

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential – NDA Restrictions

- 存储器宽度
- ROM 宽度

目标字、存储器字和 ROM 字这些术语是指具有相应宽度的字。

图 12-2 展示了十六进制转换实用程序处理流程的不同阶段。

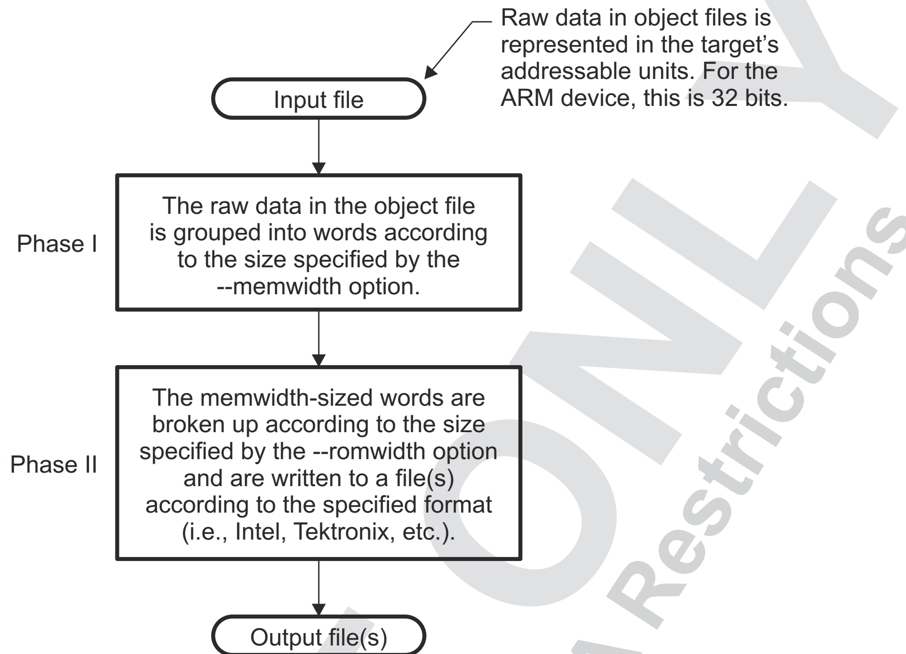


图 **12-2.** 十六进制转换实用程序处理流程

12.3.1 目标宽度

目标宽度是目标处理器字的单位大小（以“位”为单位）。每个目标的宽度是固定的，不能更改。ARM 目标的宽度为 32 位。

12.3.2 指定存储器宽度

存储器宽度是存储器系统的物理宽度（以位为单位）。存储器系统的物理宽度通常与目标处理器相同：16 位 处理器的存储器架构也是 32 位。但一些应用要求将目标字拆分为多个连续的、更窄的存储器字。

默认情况下，十六进制转换实用程序将存储器宽度设为目标宽度（即 32 位）。

您可以通过以下方式改变存储器宽度（TI-TXT、二进制和 TI-Tagged 格式除外）：

- 使用 **--memwidth** 选项。此选项会改变整个文件的存储器宽度值。
- 设置 ROMS 指令的 **memwidth** 参数。此操作会改变 ROMS 指令中指定地址范围的存储器宽度值，并覆盖该范围的 --memwidth 选项。请参阅节 12.4。

这两种方法均需使用大于或等于 8 的 2 的幂值。

您只应改变存储器宽度默认值 16：需要将单个目标字拆分为连续的、更窄的存储器字。

备注

二进制格式为 8 位宽度：二进制格式的存储器宽度无法更改。二进制十六进制格式仅支持 8 位存储器宽度。请参阅节 12.15.6，进一步了解如何使用 8 位格式的 ROMS 指令。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

备注

TI-TXT 格式为 8 位宽度：TI-TXT 格式的存储器宽度无法更改。TI-TXT 十六进制格式仅支持 8 位存储器宽度。请参阅节 12.15.6，进一步了解如何使用 TI-TXT 十六进制格式的 ROMS 指令。

图 12-3 展示了存储器宽度与目标文件数据的关系。

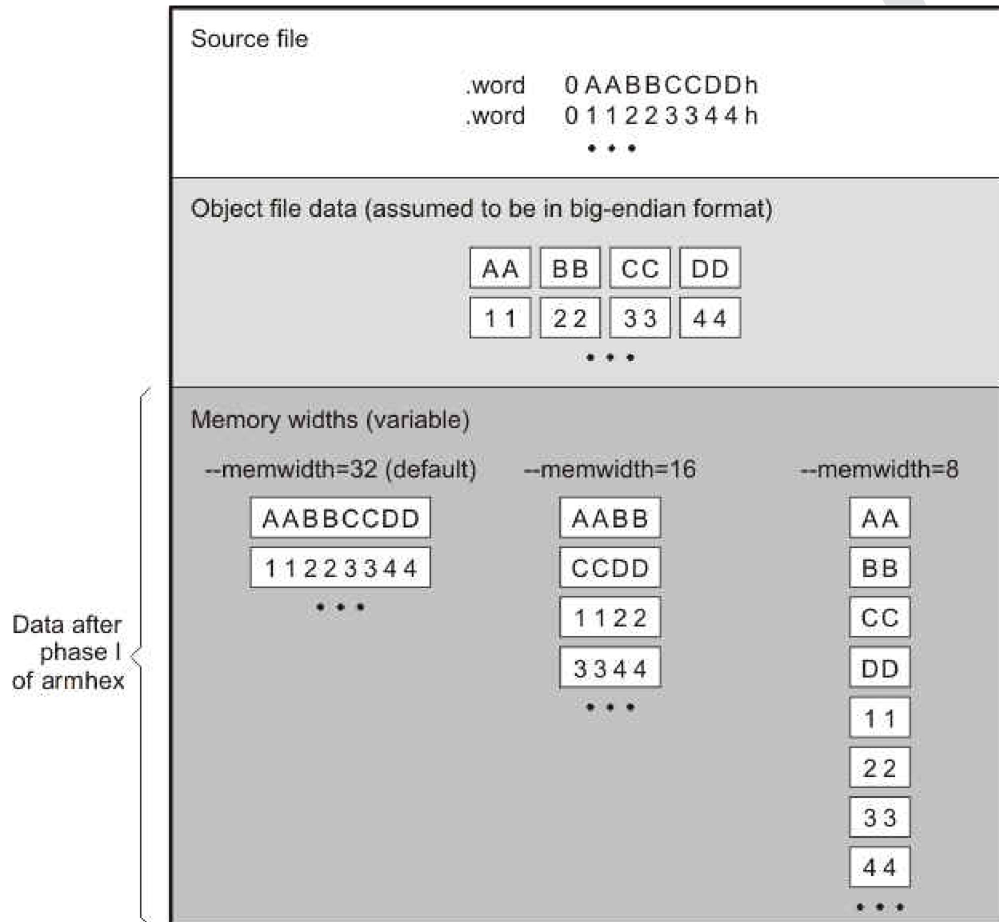


图 **12-3.** 目标文件数据与存储器宽度

DRAFT TI Confidential

Copyright © 2023 Texas Instruments Incorporated

12.3.3 将数据分入输出文件

ROM 宽度决定了十六进制转换实用程序如何将数据分入输出文件。ROM 宽度指定每个 ROM 器件的物理宽度（以位为单位）以及对应的输出文件（通常为一个字节或八位）。目标文件数据映射到存储器字后，存储器字将分解为一个或多个输出文件。输出文件的数量由以下公式决定：

- 如果存储器宽度 ≥ ROM 宽度：

  文件数量 = 存储器宽度 ÷ ROM 宽度
- 如果存储器宽度 < ROM 宽度：

  文件数量 = 1

例如，当存储器宽度为 32 时，用户可以将 ROM 宽度值指定为 32 并获得一个包含 32 位字的单一输出文件。用户也可以将 ROM 宽度值设为 16 以获得两个文件，每个文件包含每个字的 16 位。

十六进制转换实用程序使用的默认 ROM 宽度取决于输出格式：

- 除 TI-Tagged 以外的所有十六进制格式都配置为 8 位字节列表；这些格式的默认 ROM 宽度为 8 位。
- TI-Tagged 为 16 位格式；TI-Tagged 的默认 ROM 宽度为 16 位。

备注

TI-Tagged 格式为 16 位宽度：用户无法更改 TI-Tagged 格式的 ROM 宽度。TI-Tagged 格式仅支持 16 位 ROM 宽度。

备注

TI-TXT 格式为 8 位宽度：用户无法改变 TI-TXT 格式的 ROM 宽度。TI-TXT 十六进制格式仅支持 8 位 ROM 宽度。请参阅节 12.15.6，进一步了解如何使用 TI-TXT 十六进制格式的 ROMS 指令。

用户可以通过以下方式更改 ROM 宽度（TI-Tagged 和 TI-TXT 格式除外）：

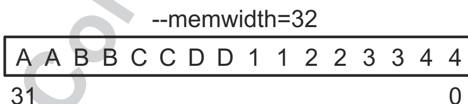
- 使用 **--romwidth** 选项。此选项会更改整个目标文件的 ROM 宽度值。
- 设置 ROMS 指令的 **romwidth** 参数。此参数会更改特定 ROM 地址范围的 ROM 宽度值，并覆盖该范围的 --romwidth 选项。请参阅节 12.4。

这两种方法均需使用大于或等于 8 的 2 的幂值。

如果用户选择的 ROM 宽度大于输出格式对应的实际大小，实用程序会向该文件中写入多字节字段。对于 TI-TXT 和 TI-Tagged 格式，--romwidth 选项会被忽略。

图 12-4 展示了目标文件数据、存储器和 ROM 宽度彼此之间的关系。

存储器宽度和 ROM 宽度仅用于对目标文件数据进行分组；它们并不代表值。因此，整个转换过程中会保持目标文件数据的字节顺序。要在存储器字中引用分区，存储器字的各个位始终要按照从右到左的顺序进行编号，如下所示：



DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

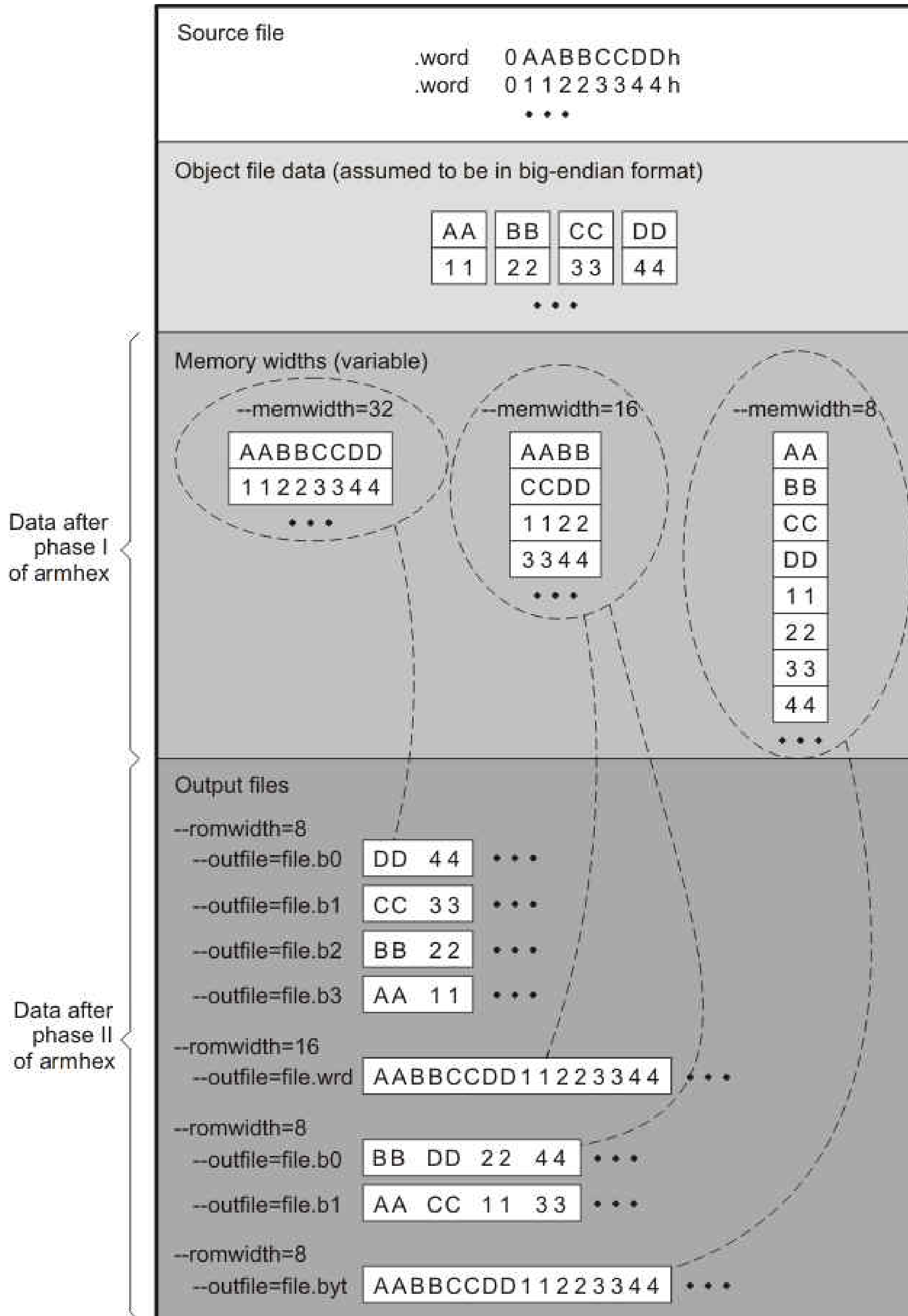


图 **12-4.** 数据、存储器和 **ROM** 宽度

Copyright © 2023 Texas Instruments Incorporated

12.4 ROMS 指令

ROMS 指令将系统的物理存储器配置指定为地址范围参数列表。每个地址范围都会生成一组文件，其中包含与该地址范围对应的十六进制转换实用程序输出数据。每个文件可用于对一个 ROM 器件进行编程。

ROMS 指令与链接器的 MEMORY 指令类似：两者都定义目标地址空间的存储器映射。ROMS 指令中的每一行条目都定义了一个具体的地址范围。通用语法为：

ROMS
{

romname **:**	**[origin=***value***,] [length=***value***,] [romwidth=***value***,]** **[memwidth=***value***,] [fill=***value***]** **[files={** *filename* 1, *filename* 2, ...**}]**
romname **:**	**[origin=***value***,] [length=***value***,] [romwidth=***value***,]** **[memwidth=***value***,] [fill=***value***]** **[files={** *filename* 1, *filename* 2, ...**}]**
...	

}

ROMS 开始指令定义。

romname 标识存储器范围。存储器范围的名称长度可为一到八个字符。名称对程序来说无关紧要；它只用于标识范围，但如果输出用于加载映像，它可表示段名。（允许存储器范围名称重复。）

origin 指定存储器范围的起始地址。它可输入为 origin（原点）、org 或 o。关联值必须为十进制、八进制或十六进制常量。如果省略原点值，则原点默认为 0。下表总结了可用于指定十进制、八进制或十六进制常量的表示法：

常量	表示法	示例
十六进制	0x 前缀或 h 后缀	0x77 或 077h
八进制	0 前缀	077
十进制	无前缀或后缀	77

length 指定存储器范围的长度为 ROM 器件的物理长度。可输入为 length（长度）、len 或 l。值必须为十进制、八进制或十六进制常量。如果省略长度，默认为整个地址空间的长度。

romwidth 指定范围的物理 ROM 宽度，以位为单位（请参阅节 12.3.3）。此处指定的任何值均会覆盖 --romwidth 选项。值必须为十进制、八进制或十六进制常量，为大于或等于 8 的 2 的幂。

memwidth 指定范围的存储器宽度，以位为单位（请参阅节 12.3.2）。您指定的任何值均可覆盖 --memwidth 选项。值必须为十进制、八进制或十六进制常量，为大于或等于 8 的 2 的幂。*使用 memwidth 参数时，必须同时在 SECTIONS 指令中指定每个段的 paddr 参数。*（请参阅节 12.5。）

fill 指定范围使用的填充值。在映像模式中，十六进制转换实用程序使用此值填充范围中任何段之间的所有空洞。空洞是两个输入段之间的区域，会形成不包含实际代码或数据的输出段。fill 值必须为十进制、八进制或十六进制常量，其宽度等于目标宽度。您指定的任何值均可覆盖 --fill 选项。使用 fill 时，必须同时使用 --image 命令行选项。（请参阅节 12.9.2。）

files 标识此范围对应的输出文件的名称。将名称列表置于大括号中，并按从*最低有效*到*最高有效*输出文件的顺序排序，期中存储器字的位从右到左进行编号。文件名的数量必须等于该范围生成的输出文件的数量。若要计算输出文件的数量，请参阅节 12.3.3。如果列出的文件名过多或过少，实用程序会发出警告。

除非使用 --image 选项，否则定义范围的所有参数都是可选的；逗号和等号也是可选的。没有原点或 长度的范围定义了整个地址空间。在映像模式中，所有范围均需要原点和长度。

范围不能互相重叠，必须按地址升序列出。

12.4.1 何时使用 ROMS 指令

如果用户不使用 ROMS 指令，实用程序会定义单一默认范围，其中包含整个地址空间。这与提供单一范围、无原点或长度的 ROMS 指令等效。

如果用户希望实现以下目标，请使用 ROMS 指令：

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY – Confidential – NDA Restrictions

- **将大量数据编程到固定大小 ROM 中**。如果指定的存储器范围与 ROM 长度对应，实用程序会自动将输出拆分为适合 ROM 的块。
- **将输出限制到特定区段**。用户还可以使用 ROMS 指令限制转换到目标地址空间的特定区段。实用程序不会转换由 ROMS 指令定义的范围之外的数据。段可跨越范围界限；实用程序会在边界处将它们拆分为多个范围。如果某个段完全位于用户定义的范围之外，实用程序不会转换该段，也不会发出消息或警告。因此，用户可以不在 SECTIONS 指令中列出这些段的名称，从而排除它们。但如果一个段有部分位于某范围内，部分位于未配置的存储器中，实用程序会发出警告，并且只转换范围内的那部分。
- **使用映像模式**。如果使用 --image 选项，则必须使用 ROMS 指令。每个范围都会全部填充，以便范围内的每个输出文件都包含整个范围的数据。段前、段后和段之间的空洞会使用 ROMS 指令用 --fill 选项指定的填充值填充，或用默认值 0 填充。

12.4.2 ROMS 指令的示例

控制由链接器生成的复制表段的放置 ROMS 指令示例中的 ROMS 指令显示了如何将 16K 字节的 16 位存储器分区为两个 8K 字节的 8 位 EPROM。图 12-5 显示了输入和输出文件。

ROMS 指令示例

```
infile.out
--image
--memwidth 16
ROMS
{
   EPROM1: org = 0x00004000, len = 0x2000, romwidth = 8
           files = { rom4000.b0, rom4000.b1}
   EPROM2: org = 0x00006000, len = 0x2000, romwidth = 8,
           fill = 0xFF00FF00,
           files = { rom6000.b0, rom6000.b1}
}
```

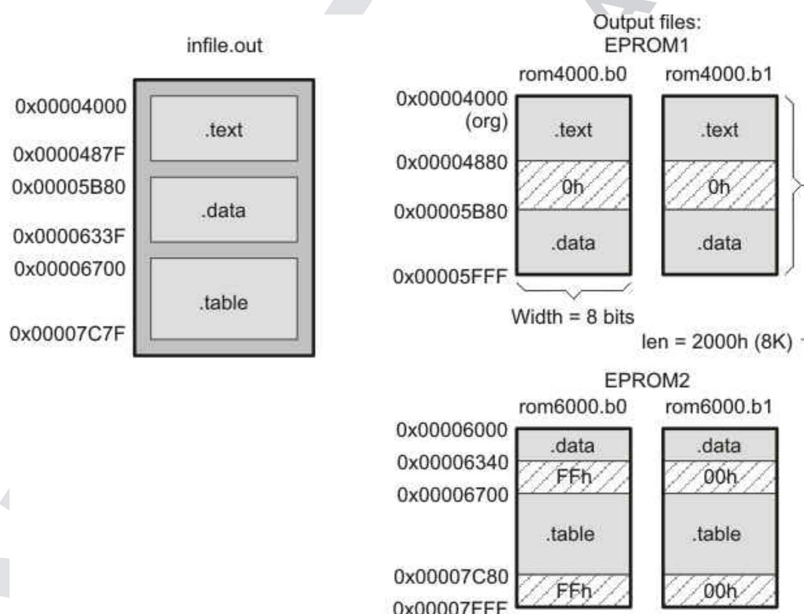


图 12-5. infile.out 文件分区为四个输出文件

当用户使用具有多个范围的 ROMS 指令时，映射文件（用 --map 选项指定）具有优势。映射文件显示每个范围、其参数、相关输出文件的名称以及按地址细分的内容列表（段名称和填充值）。显示存储器范围的中的映射文件输出是由控制由链接器生成的复制表段的放置 ROMS 指令示例中的示例生成的映射文件的一个程序段。

Copyright © 2023 Texas Instruments Incorporated

显示存储器范围的控制由链接器生成的复制表段的放置 ROMS 指令示例中的映射文件输出

```
-----------------------------------------------------
00004000..00005fff Page=0 Width=8 "EPROM1"
-----------------------------------------------------
   OUTPUT FILES:   rom4000.b0    [b0..b7]
                   rom4000.b1    [b8..b15]
   CONTENTS: 00004000..0000487f .text
             00004880..00005b7f FILL = 00000000
             00005b80..00005fff .data
-----------------------------------------------------
00006000..00007fff Page=0 Width=8 "EPROM2"
-----------------------------------------------------
   OUTPUT FILES:   rom6000.b0    [b0..b7]
                   rom6000.b1    [b8..b15]
   CONTENTS: 00006000..0000633f .data
             00006340..000066ff FILL = ff00ff00
             00006700..00007c7f .table
             00007c80..00007fff FILL = ff00ff00
```

EPROM1 定义的地址范围为从 0x00004000 到 0x00005FFF，具有以下段：

该段...	具有此范围...
.text	0x00004000 到 0x0000487F
.data	0x00005B80 到 0x00005FFF

该范围的其余部分用 0h（默认填充值）填充，转换成两个输出文件：

- rom4000.b0 包含位 0 到位 7
- rom4000.b1 包含位 8 到位 15

EPROM2 定义的地址范围为从 0x00006000 到 0x00007FFF，具有以下段：

该段...	具有此范围...
.data	0x00006000 到 0x0000633F
.table	0x00006700 到 0x00007C7F

该范围的其余部分用 0xFF00FF00（根据指定的填充值）填充。此范围中的数据转换成两个输出文件：

- rom6000.b0 包含位 0 到位 7
- rom6000.b1 包含位 8 到位 15

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.5 SECTIONS 指令

十六进制转换实用程序 SECTIONS 指令可按名称转换目标文件的具体段。您还可以指定将这些段放置在 ROM 中链接器命令文件指定*加载*地址以外的地址。如果您：

- 使用 SECTIONS 指令，实用程序只会转换指令中列出的段，而忽略目标文件中的所有其他段。
- 不使用 SECTIONS 指令，实用程序会转换所配置的存储器中所有已初始化的段。

未初始化的段*从不*转换，无论 SECTIONS 指令是否指定。

备注

由 C/C++ 编译器生成的段： ARM C/C++ 编译器会自动生成这些段：

- 已初始化的段：.text、.const、.cinit 和 .switch
- 未初始化的段： .bss、.stack 和 .sysmem

在命令文件中使用 SECTIONS 指令。（请参阅节 12.2.2。）通用语法为：

SECTIONS
{
oname(*sname*)**[:][paddr=***value***]**
oname(*sname*)**[:][paddr= boot]**
oname(*sname*)**[:][boot]**
...
}

SECTIONS	开始指令定义。
oname	标识段所在的目标文件名。如果只有一个输入文件，文件名是可选项，但在其他情况下是必备项。
sname	标识输入文件中的一个段。如果您指定了一个不存在的段，实用程序会发出警告并忽略该名称。
paddr=*value*	指定此段应放置的物理 ROM 地址。此值会覆盖由链接器分配的段加载地址。此值必须为十进制、八进制或十六进制常量。它还可以是字 **boot**（指示引导加载程序将使用的引导表段）。 *如果文件包含多个段，并且一个段使用 paddr 参数，则所有段都必须使用 paddr 参数。*
boot	配置一个段，供引导加载程序加载。它与使用 **paddr=boot** 等效。引导段的物理地址由引导表的位置决定。引导表的原始地址由 --bootorg 选项指定。

若要与链接器的 SECTIONS 指令更加相似，您可以在段名后添加冒号（取代引导键盘上的等号）。例如，以下语句是等效的：

```
SECTIONS { .text: .data: boot }
```

```
SECTIONS { .text: .data = boot }
```

在下例中，目标文件包含六个未初始化的段：.text、.data、.const、.vectors、.coeff 和 .tables。假设您只希望转换 .text 和 .data。使用 SECTIONS 指令指定：

```
SECTIONS { .text: .data: }
```

若要将这些段配置为引导加载，请添加关键字 boot：

```
SECTIONS { .text = boot .data = boot }
```

12.6 加载映像格式（--load_image 选项）

加载映像是一个目标文件，其中包含一个或多个可执行文件的加载地址和已初始化段。加载映像目标文件可用于 ROM 屏蔽，也可在后续的链接步骤中重新链接。

多个命令行选项可用于控制使用 --load_image 时所生成文件的格式。这些选项的功能如下：

Copyright © 2023 Texas Instruments Incorporated

- 利用 --load_image:file_type 选项创建可重定位或可执行的输出文件。
- 利用 -load_image:format、--load_image:machine 以及 --load_image:endian 选项分别指定 ABI、机器类型和字节序。
- 利用 --load_image:combine_sections、--load_image:section_prefix 和 --load_image:section_addresses 选项组合段，为段名添加前缀，或在输出文件中包含加载地址。
- 利用 --load_image:output_symbols 和 --load_image:symbol_binding 选项选择是否输出符号，并指定在输出中的绑定方式。
- 利用 -load_image:localize 和 --load_image:globalize 选项控制单个符号是局部符号还是全局符号。

这些命令行选项在节 12.2.1 中介绍。

12.6.1 加载映像段形成

加载映像段是通过从输入可执行文件中收集已初始化段而形成的。加载映像段的形成方式有两种：

- **使用 ROMS 指令**。ROMS 指令中给出的每个存储器范围表示一个加载映像段。*romname* 是段名。origin 和 length 参数为必填项。memwidth、romwidth 和 files 参数无效并会被忽略。

  当使用 ROMS 指令和 --load_image 选项时，需要 --image 选项。
- **默认加载映像段形成**。如果未给出 ROMS 指令，加载映像段通过组合输入可执行文件中的连续初始化段而形成。间隙小于目标字大小的段被视为连续段。

  默认段名为 image_1、image_2、...如果需要另一个前缀，可以使用 --load_image:section_prefix=*prefix* 选项。

12.6.2 加载映像特性

所有加载映像段均为已初始化的数据段。在没有 ROMS 指令的情况下，加载映像段的加载/运行地址是加载映像段中的第一个输入段的加载地址。如果使用了 SECTIONS 指令，且通过使用 paddr 参数给出了另一个加载地址，则将使用此地址。

加载映像格式始终会创建一个加载映像目标文件。加载映像目标文件的格式根据输入文件来确定。该文件没有标记为可执行，且不包含入口点。默认加载映像目标文件的名称是 ti_load_image.obj。可以使用 --outfile 选项更改此名称。在创建加载映像时，只有一个 --outfile 选项有效，系统会忽略所有其他 --outfile 选项。

一个输入映像中的段不得与另一个输入映像中的段重叠。如果段重叠，十六进制转换器会发出错误消息。

备注

关于加载映像格式：在创建加载映像时，这些选项无效：

- --memwidth
- --romwidth
- --zero
- --byte

如果使用 SECTIONS 指令或 --boot 选项创建了引导表，则必须使用 ROMS 指令。

12.7 排除指定段

--exclude *section_name* 选项可用于通知十六进制实用程序忽略指定段。如果使用 SECTIONS 指令，它将覆盖 --exclude 选项。例如，如果使用了包含段名称 *mysect* 的 SECTIONS 指令并指定了 --exclude *mysect*，则 SECTIONS 指令优先并且不会排除 *mysect*。

--exclude 选项具有有限的通配符功能。* 字符可以放在名称说明符的开头或结尾，分别表示后缀或前缀。例如，--exclude sect* 取消了所有以字符 sect 开头的段的资格。如果在命令行上使用 * 通配符来指定 --exclude 选项，请

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

在段名称和通配符两边加上引号。例如，--exclude"sect*"。使用引号可防止十六进制转换实用程序对 * 进行解释。如果 --exclude 在命令文件中，请勿使用引号。

如果给定了多个目标文件，则可按 oname(sname) 形式给出要排除的段所在的目标文件。如果未提供目标文件名，则排除与段名称匹配的所有段。通配符不能用于文件名，但可出现在括号内。

12.8 分配输出文件名

当十六进制转换实用程序将您的目标文件转换为一种数据格式时，它会将数据划分为一个或多个输出文件。当通过将存储器字拆分为 ROM 字而形成多个文件时，*文件名总是按从最低有效到最高有效的顺序分配*，其中存储器字中的位从右到左编号。无论目标顺序或字节顺序如何，都是如此。

分配输出文件名时，十六进制转换实用程序遵循以下顺序：

1. **它会查找 ROMS 指令。**如果文件与 ROMS 指令中的范围相关联，并且您在该范围内包含了文件列表 (files = {...})，实用程序将从列表中获取文件名。例如，假设目标数据是 32 位字，被转换为四个文件，每个文件为 8 位宽。若要使用 ROMS 指令命名输出文件，您可以指定：

```
ROMS
{
   RANGE1: romwidth=8, files={ xyz.b0 xyz.b1 xyz.b2 xyz.b3 }
}
```

   该实用程序会通过以下方法创建输出文件：将最低有效位写入 xyz.b0，将最高有效位写入 xyz.b3。
2. **它会查找 --outfile 选项。**您可以使用 --outfile 选项来指定输出文件的名称。如果 ROMS 指令中未列出任何文件名，并且您使用了 --outfile 选项，则该实用程序将从 --outfile 选项列表中获取文件名。以下选项与上面的 ROMS 指令具有相同的效果：

```
--outfile=xyz.b0 --outfile=xyz.b1 --outfile=xyz.b2 --outfile=xyz.b3
```

   如果同时使用了 ROMS 指令和 --outfile 选项，则 ROMS 指令会覆盖 --outfile 选项。
3. **它会分配一个默认文件名。**如果不指定文件名或指定的文件名少于输出文件，则该实用程序会分配一个默认文件名。默认文件名由输入文件的基本名称加上 2 到 3 个字符的扩展名组成。扩展名包含三个部分：
   a. 格式字符，基于输出格式（请参见节 12.15）：

a	表示 ASCII 十六进制
i	表示 Intel
m	表示 Motorola-S
t	表示 TI-Tagged
x	表示 Tektronix

   b. ROMS 指令中的范围编号。范围从 0 开始编号。如果没有 ROMS 指令，或只有一个范围，实用程序将忽略此字符。
   c. 该范围的文件集中的文件编号，从 0 开始，表示最低有效文件。

   例如，假设 a.out 用于 32 位目标处理器，并且您正在创建 Intel 格式的输出。在没有指定输出文件名的情况下，该实用程序会生成四个名为 a.i0、a.i1、a.i2、a.i3 的输出文件。

   如果您在调用十六进制转换实用程序时包含以下 ROMS 指令，您将有八个输出文件：

```
ROMS
{
   range1: o = 0x00001000 l = 0x1000
   range2: o = 0x00002000 l = 0x1000
}
```

这些输出文件...	包含这些位置的数据...
a.i00、a.i01、a.i02、a.i03	0x00001000 到 0x00001FFF

Copyright © 2023 Texas Instruments Incorporated

这些输出文件...	包含这些位置的数据...
a.i10、a.i11、a.i12、a.i13	0x00002000 到 0x00002FFF

12.9 映像模式和 --fill 选项

本节介绍了在映像模式下运行的优势，还介绍了如何生成输出文件，其具有目标存储器范围的精确、连续映像。

12.9.1 生成存储器映像

借助 --image 选项，实用程序通过完全填充 ROMS 指令中指定的所有映射范围来生成存储器映像。

目标文件由分配了存储器位置的存储器块（段）组成。通常，各段互不相邻：地址空间中的段与段之间存在一些空洞，这些空洞没有数据。在*不*使用映像模式的情况下转换此类文件时，十六进制转换实用程序会使用输出文件中的地址记录跳到下一段的开头，从而桥接这些空洞。换而言之，输出文件的地址可能存在不连续的情况。一些 EPROM 编程器不支持地址不连续。

在映像模式下，不存在不连续的情况。每个输出文件都包含与目标存储器中地址范围完全对应的连续数据流。段之前、之间或之后的任何空洞都使用用户提供的填充值来填充。

使用映像模式转换的输出文件仍然具有地址记录，因为许多十六进制格式要求每行都有一个地址。但在映像模式下，这些地址始终是连续的。

备注

定义目标存储器的范围：如果使用映像模式，还必须使用 ROMS 指令。在映像模式下，每个输出文件都直接对应于目标存储器的一个范围。用户必须定义范围。如果用户不提供目标存储器的范围，该实用程序会尝试构建整个目标处理器地址空间的存储器映像。这可能会生成大量的输出数据。为防止出现这种情况，该实用程序要求用户使用 ROMS 指令来明确限制地址空间。

12.9.2 指定填充值

--fill 选项可用于指定位于填充段之间的空洞的值。填充值必须指定为整数常量，跟在 --fill 选项之后。常量宽度假设为目标处理器上的字的宽度。例如，指定 --fill=0xFFFF 导致填充模式为 0x0000FFFF。常量值不会进行符号扩展。

如果未通过填充选项来指定值，十六进制转换实用程序将使用 0 作为默认填充值。*只有使用 --image 时 --fill 选项才是有效的*；否则将忽略它。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.9.3 使用映像模式的步骤

步骤 1： 利用 ROMS 指令定义目标存储器的范围。请参阅节 12.4。

步骤 2： 使用 --image 选项调用十六进制转换实用程序。可以选择使用 --zero 选项将每个输出文件的地址原点复位为 0。如果 ROMS 指令未指定填充值，而您又不希望使用默认值 0，请使用 --fill 选项。

12.10 数组输出格式

--array 选项会让系统以 C 数组格式生成输出。在这种格式中，包含在可执行文件的已初始化段中的数据被定义为 C 数组。输出数组可以与主机程序一同编译并用于在运行时初始化目标。

数组是通过从输入可执行文件中收集已初始化段而形成的。数组的形成方式有两种：

- **使用 ROMS 指令。**ROMS 指令中给出的每个存储器范围表示一个数组。*romname* 用作数组名称。需要 ROM 指令的 *origin* 和 *length* 参数。*memwidth*、*romwidth* 和 *files* 参数无效并会被忽略。
- **无 ROMS 指令（默认）。**如果没有给出 ROMS 指令，数组是通过组合每页内的已初始化段（从第一个已初始化段开始）形成的。数组将反映各段之间存在的所有间隙。

  为--array:name_prefix 选项可用于覆盖数组名称的默认前缀。例如，使用 --array:name_prefix=myarray 会使数组的

数组元素的数据类型是 uint8_t。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.11 为片上引导加载程序编译引导表

ARM 十六进制实用程序能够创建用于片上引导加载程序的引导表。支持的引导格式旨在用于具有 ARM 内核的 C28x 器件。引导表存储在存储器中或从器件外设加载以初始化代码或数据。

有关引导加载的一般讨论，请参阅节 3.1.2。

12.11.1 引导表说明

引导加载程序的输入是引导表。引导表包含的记录会指示片上加载程序将表中包含的数据块复制到指定目标地址。该表可以存储在存储器（例如 EPROM）中，也可以通过器件外设（例如串行或通信端口）读入。

十六进制转换实用程序会自动编译引导加载程序的引导表。使用该实用程序，您可以指定希望引导加载程序初始化的段以及表位置。十六进制转换实用程序会根据指定格式编译表的完整图像，并将其转换为十六进制格式的输出文件。然后，您可以将表刻录到 ROM 中或通过其他方式将其加载。

12.11.2 引导表格式

引导表的格式很简单。通常有一个头记录，包含一个键值，指示存储器宽度、进入点以及控制寄存器的值。后续的每个块都有一个数据头，包含该块的大小和目标地址，后跟该块的数据。可输入多个块。引导表的结尾处是一个大小为零的数据头。

12.11.3 如何构建引导表

表 12-2 概述了引导加载程序可用的十六进制转换实用程序选项。

表 12-2. 引导加载程序选项

选项	说明
--boot	将所有段都转换为可引导形式（而不是使用 SECTIONS 指令）。
--bootorg=*address*	指定引导加载程序表的源地址。
--cmac=*file*	指定包含 CMAC 密钥以与 TMS320F2838x 器件上的安全闪存引导一同使用的文件。
--divsel *value*	为 DIVSEL 寄存器指定初始值。仅当使用了 -xintf8/16 时，此选项才有效。如果未指定值，请使用 2。
--entrypoint=*value*	指定引导加载后开始执行的入口点。*值* 可以是一个地址，也可以是一个全局符号。
--gpio8	指定引导加载程序表源作为 GP I/O 端口，8 位模式。（又名 --can8。）
--gpio16	指定引导加载程序表源作为 GP I/O 端口，16 位模式。
--lospcp=*value*	为 LOSPCP 寄存器指定初始值。该值仅用于 spi8 引导表格式，所有其他格式会忽略该值。大于 0x7F 的值被截断为 0x7F。
--pllcr *value*	为 PLLCR 寄存器指定初始值。仅当使用了 --xintf8 或 --xintf16 时，此选项才有效。如果未指定值，请使用 0。
--sci8	指定引导加载程序表源作为 SCI-A 端口，8 位模式
--spi8	指定引导加载程序表源作为 SPI-A 端口，8 位模式
--spibrr=*value*	为 SPIBRR 寄存器指定初始值。该值仅用于 spi8 引导表格式，所有其他格式会忽略该值。大于 0x7F 的值被截断为 0x7F。
--xintcnf2 *value*	为 XINTCNF2 寄存器指定初始值。仅当使用了 --xintf8 或 --xintf16 时，此选项才有效。
--xintf8	指示使用并行 XINTF 流，8 位模式。
--xintf16	指示使用并行 XINTF 流，16 位模式。
--xtiming *value*	为 XTIMING*n* 寄存器指定初始值。仅当使用了 --xintf8 或 --xintf16 时，此选项才有效。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.11.3.1 构建引导表

若要构建引导表，请执行以下步骤：

步骤 1： **链接此文件。**引导表数据的每个块对应于目标文件中的一个已初始化段。十六进制转换实用程序不会转换未初始化的段（请参阅节 12.5）。

当您选择放置在引导加载程序表中的段时，十六进制转换实用程序会将该段的*加载地址*放置在引导表中块的目标地址字段中。然后将该段内容视为此块的原始数据。*十六进制转换实用程序不使用段运行地址。*链接时，您无需担心 ROM 地址或引导表的编译；十六进制转换实用程序会处理此问题。

步骤 2： **确定可引导段。**您可以使用 --boot 选项告知十六进制转换实用程序配置所有用于引导加载的段。或者，您可以使用 SECTIONS 指令来选择要配置的特定段（请参阅节 12.5）。如果您使用 SECTIONS 指令，则忽略 --boot 选项。

步骤 3： **设置引导表格式。**指定 --gpio8、--gpio16 或 --spi8 选项以设置引导表的源格式。您不需要指定 memwidth 和 romwidth，因为该实用程序会自动设置这些格式。如果在格式选项后使用 --memwidth 和 --romwidth，它们将覆盖格式的默认值。

步骤 4： **设置引导表的 ROM 地址。**使用 --bootorg 选项设置完整表的源地址。

第 5 步： **设置引导加载程序特定选项。**根据需要设置进入点和控制寄存器值。

第 6 步： **描述系统存储器配置。**请参阅节 12.3 和节 12.4。

12.11.3.2 为引导表留出空间

完整引导表类似于包含引导加载程序的所有标头记录和数据的一个段。这个段的地址是引导表的原始地址。在正常转换过程中，十六进制转换实用程序将引导表转换为十六进制格式并像任何其他段一样将它映射到输出文件。

一定要在引导表的系统存储器中留出空间，尤其是在使用 ROMS 指令时。引导表不能与其他非引导段或未配置的存储器重叠。通常来说这不是问题；系统中的一部分存储器保留用于引导表。只需将此存储器配置为 ROMS 指令中的一个或多个范围，并使用 --bootorg 选项来指定起始地址。

12.11.4 从器件外设进行引导

您可以使用 --gpio8、--gpio16 或 --spi8 引导表格式选项选择从哪个端口进行引导。

可以使用 --lospcp 选项指定 LOSPCP 寄存器的初始值。可以使用 --spibrr 选项指定 SPIBRR 寄存器的初始值。只有 --spi8 格式在引导表中使用这些控制寄存器值。

如果没有为 --spi8 格式指定寄存器值，则十六进制转换实用程序使用默认值 0x02 表示 LOSPCP，使用 0x7F 表示 SPIBRR。当指定了引导表格式选项且未指定 ROMS 指令时，ASCII 格式的十六进制实用程序输出不会产生地址记录。

12.11.5 设置引导表的进入点

完成引导加载过程后，在链接器指定的默认进入点（包含在目标文件中）开始执行。将 --entrypoint 选项与十六进制转换实用程序一同使用，可以将进入点设为另一个地址。

例如，如果您希望程序在加载后开始运行的地址为 0x0123，可在命令行或命令文件中指定 --entrypoint=0x0123。查看链接器生成的映射文件可确定 --entrypoint 地址。

备注

有效进入点

该值可以是一个常量，也可以是在汇编源代码中外部定义的符号（例如使用 .global 定义）。

12.11.6 使用 ARM 引导加载程序

本小节介绍了具有 ARM 内核的 C28x 器件如何将十六进制转换实用程序与引导加载程序结合使用。引导加载程序接受表 12-3 中列出的格式。

表 12-3. 引导表源格式

格式	选项
并行引导 GP I/O 8 位	--gpio8

Copyright © 2023 Texas Instruments Incorporated

表 12-3. 引导表源格式 (continued)

格式	选项
并行引导 GP I/O 16 位	--gpio16
8 位 SPI 引导	--spi8

具有 ARM 内核的 C28x 器件上的 ARM 可通过 8 位 SPI-A、8 位 GP I/O 或 16 位 GP I/I 接口引导。引导表的格式显示在表 12-4 中。

表 12-4. 引导表格式

说明	字节	内容
引导表头	1-2	键值（0x10AA 或 0x08AA）
	3-18	寄存器初始化值，或留作将来使用
	19-22	进入点
块标头	23-24	以字节数表示的块大小 (nl)
	25-28	块的目标地址
块数据	29-30	块的原始数据（nl 字节）
块标头	31 + nl	以字节数表示的块大小
	.	块的目标地址
块数据	.	块的原始数据
根据需要额外添加的块标头和数据	...	恰当内容
大小为 0 的块标头		0x0000；指示引导表结束。

具有 ARM 内核的 C28x 器件上的 ARM 可通过串行 8 位接口或并行接口，利用 8 位或 16 位数据引导。任意组合的格式均相同：引导表中有一个包含目标地址的字段、一个包含长度的字段，以及一个包含数据的块。您可以只引导一个段。如果您从 8 位通道引导，8 位字节将存储在表中，MSB 在最前边；十六进制转换实用程序会以正确的格式自动编译该表。使用以下选项指定引导表源：

- 若要从 SPI-A 端口引导，请在调用实用程序时指定 --spi8。不要指定 --memwidth 或 --romwidth。使用 --lospcp 设置 LOSPCP 寄存器的初始值，使用 --spibrr 设置 SPIBRR 寄存器的初始值。如果没有为 --spi8 格式指定寄存器值，十六进制转换实用程序会为 LOSPCP 使用默认值 0x02，为 SPIBRR 使用 0x7F。
- 若要从通用并行 I/O 端口加载，请在调用实用程序时使用 --gpio8 或 --gpio16。不要指定 --memwidth 或 --romwidth。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

示例 12-1 中的命令文件支持从位置 0x3FFC00 的 16 位宽 EPROM 中引导 test.out 的 .text 和 .cinit 段。另外还会生成映射文件 test.map。

示例 12-1. 从 8 位 SPI Boot 引导的示例命令文件

```
/*--------------------------------------------------------------------------*/
/* Hex converter command file.                                              */
/*--------------------------------------------------------------------------*/
test.out                 /* Input file */
--ascii                  /* Select ASCII format */
--map=test.map           /* Specify the map file */
--outfile=test_spi8.hex  /* Hex utility out file */
--boot                   /* Consider all the input sections as boot sections */
--spi8                   /* Specify the SPI 8-bit boot format */
--lospcp=0x3F            /* Set the initial value for the LOSPCP as 0x3F */
                         /* The -spibrr option is not specified to show that */
                         /* the hex utility uses the default value (0x7F)  */
--entrypoint=0x3F0000    /* Set the entry point */
```

示例 12-1 中的命令文件生成图 12-6 中的输出文件。控制寄存器值在引导表头中进行编码，并且该表头包含通过 --entrypoint 选项指定的地址。

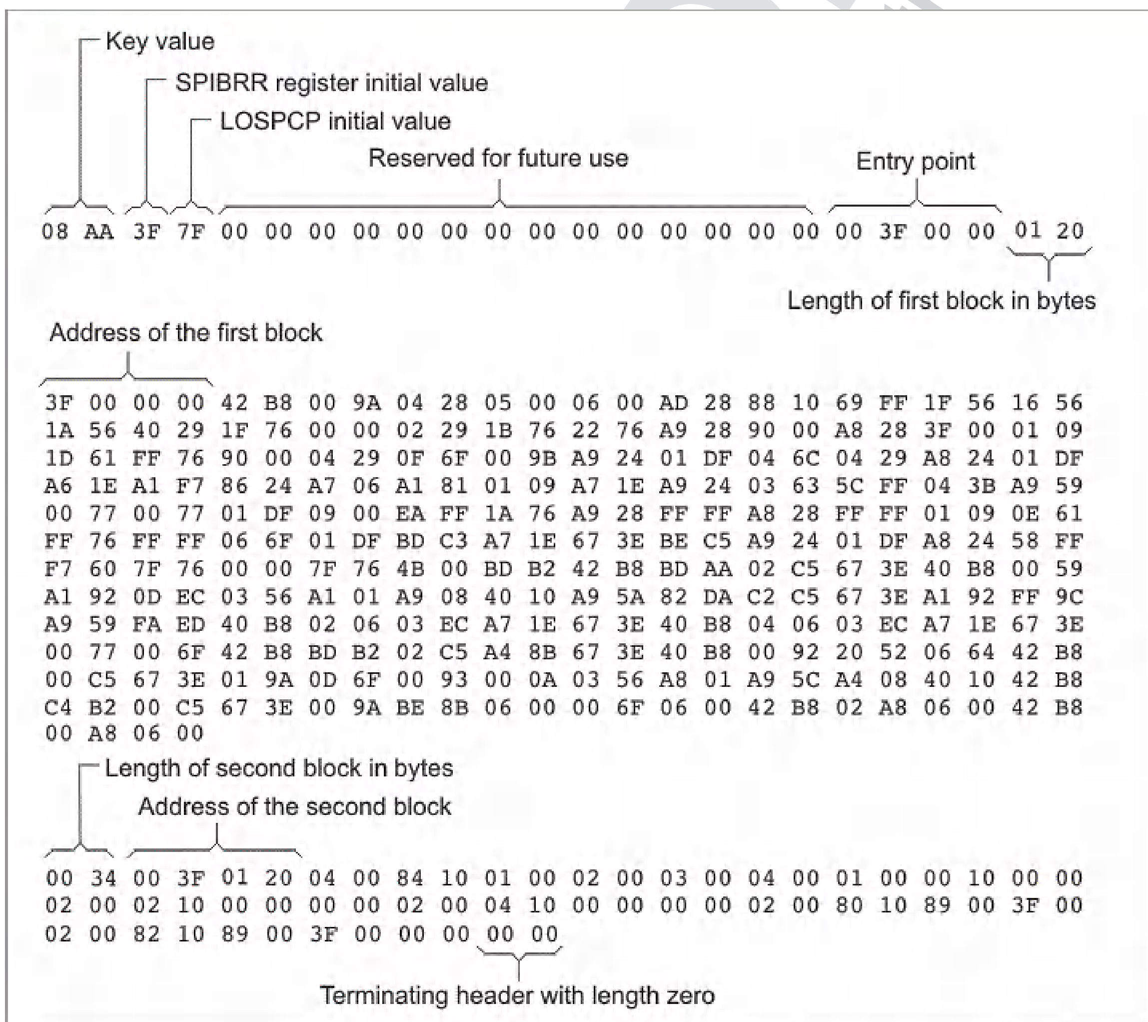


图 12-6. 从 8 位 SPI Boot 引导的示例十六进制转换器输出文件

Copyright © 2023 Texas Instruments Incorporated

示例 12-2 中的命令文件支持从 16 位并行 GP I/O 端口引导 test.out 的 .text 和 .cinit 段。另外还会生成映射文件 test.map。

示例 12-2. ARM 16 位并行引导 GP I/O 的示例命令文件

```
/*---------------------------------------------------------------------*/
/* Hex converter command file.                                         */
/*---------------------------------------------------------------------*/
test.out                   /* Input file */
--ascii                    /* Select ASCII format */
--map=test.map             /* Specify the map file */
--outfile=test_gpio16.hex  /* Hex utility out file */
--gpio16                   /* Specify the 16-bit GP I/O boot format */
SECTIONS
{
   .text: paddr=BOOT
   .cinit: paddr=BOOT
}
```

示例 12-2 中的命令文件生成图 12-7 中的输出文件。

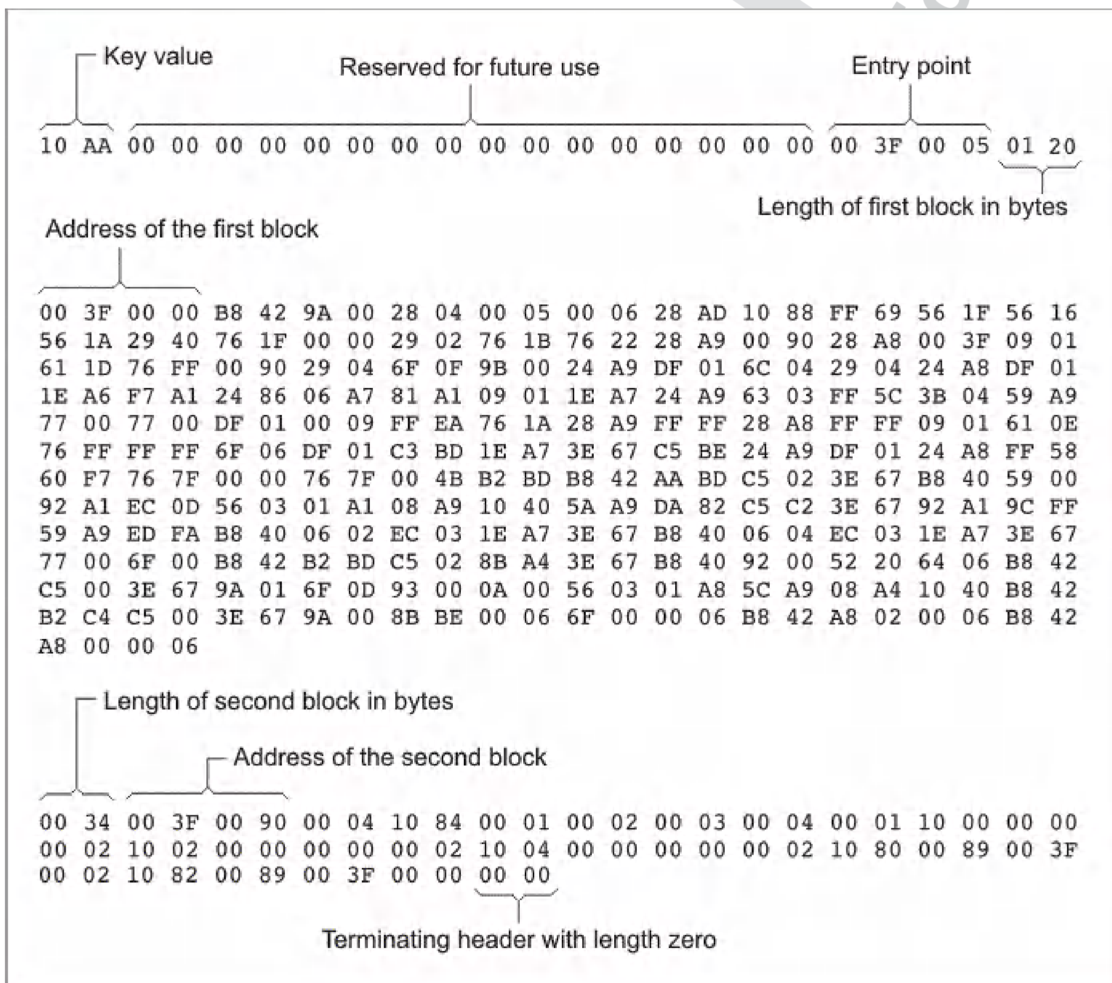


图 12-7. ARM 16 位并行引导 GP I/O 的示例十六进制转换输出文件

Copyright © 2023 Texas Instruments Incorporated

12.12 在 TMS320F2838x 器件上使用安全闪存引导功能

十六进制转换实用程序支持 TMS320F2838x 器件提供的安全闪存引导功能，该器件同时具有 C28 和 ARM 内核。安全闪存引导功能应用基于密码的消息身份验证协议 (CMAC) 算法，用于验证所分配存储器区域的 CMAC 标签。

安全闪存引导功能与常规闪存引导模式类似的方面是，引导流程会分支到闪存中配置的存储器地址。区别在于，只有在闪存存储器内容经过身份验证后才能进行分支。闪存身份验证使用 CMAC 验证 16KB 的闪存。CMAC 计算需要您定义的 128 位密钥。此外，您还必须根据 16KB 的闪存存储器范围计算黄金 CMAC 标签，并将其与应用代码一同存储于闪存中的硬编码地址。在安全闪存引导期间，计算得出的 CMAC 标签会与闪存中的黄金 CMAC 标签进行比较，以确定 CMAC 身份验证的通过/失败状态。如果通过了身份验证，引导流程会继续，并分支到闪存中以开始执行应用。请参阅 *TMS320F2838x 微控制器技术参考手册* (SPRUII0)，进一步了解有关安全闪存引导功能和 CMAC 算法的详细信息。

按如下方式使用十六进制转换实用程序，将 CMAC 算法应用于所分配存储器中的区域：

- 使用 --cmac=file 选项。文件应包含 128 位十六进 CMAC 密钥。文件中由 --cmac 命令行选项指定的 CMAC 密钥，必须使用 0xkey0key1key2key3 的格式，以访问 CMACKEY0-3 的器件寄存器。例如，以下文件内容表示 CMACKEY 寄存器包含 key0= `0x7c0b7db9`、key1= `0x811f10d0`、key2= `0x0e476c7a` 以及 key3= `0x0d92f6e0`。

```
0x7c0b7db9811f10d00e476c7a0d92f6e0
```

- 使用 --cmac 选项时应使用 --image 选项或 --load_image 选项。如果使用 --image 选项，请将 --memwidth 和 --romwidth 设为相同的值。
- 如果结合使用 --boot 选项（以及节 12.11 中介绍的其他引导表选项）和 --cmac 选项，CMAC 算法会假设使用 1 作为引导表区域之间的填充值。由于此假设，在同时使用 --boot 和 --cmac 选项时，您还应设置 --fill=0xFFFFFFFF。
- 指定 HEX 指令时用一个条目表示分配的所有闪存存储器。使用 128 位对齐长度并指定可选填充值。（默认填充值设为 0。）
- 在 C 代码中定义全局 CMAC 标签。

CMAC 功能使用四个安全闪存引导存储器区域，它们针对开始/结束/标签地址进行硬编码，还有一个灵活的 CMAC 区域。灵活区域可包含分配的整个区域作为 HEX 指令中的输入，或作为 C 代码中定义的由用户指定的开始/结束地址。

若要为 CMAC 标签符号保留空间，需要与以下代码类似的 C 代码定义。

```
struct   CMAC_TAG
{  uint8_t  tag[16];
   uint32_t start;
   uint32_t end;
};
#pragma RETAIN(cmac_sb_1)
#pragma LOCATION(cmac_sb_1, 0x00200004)
const uint8_t cmac_sb_1[16] =  { 0 };
#pragma RETAIN(cmac_sb_2)
#pragma LOCATION(cmac_sb_2, 0x00210004)
const uint8_t cmac_sb_2[16] =  { 0 };
#pragma RETAIN(cmac_sb_3)
#pragma LOCATION(cmac_sb_3, 0x00250004)
const uint8_t cmac_sb_3[16] =  { 0 };
#pragma RETAIN(cmac_sb_4)
#pragma LOCATION(cmac_sb_4, 0x0027C004)
const uint8_t cmac_sb_4[16] =  { 0 };
#pragma RETAIN(cmac_all)
#pragma LOCATION(cmac_all, 0x00204004)
const struct CMAC_TAG cmac_all =  { { 0 }, 0x0, 0x0};
```

四个安全闪存引导区域 CMAC 标签存储在 `cmac_sb_1` 到 `cmac_sb_4` 符号中。`cmac_all` 符号存储用户指定的灵活区域的 CMAC 标签。对于 `cmac_all`：

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

- 如果 `start` 和 `end` CMAC_TAG 结构成员为零，则 CMAC 算法会针对 HEX 指令中指定的整个存储器区域运行。十六进制转换实用程序会用 HEX 指令条目中的地址输入填充存储器开始和结束位置。
- 如果 `start` 和 `end` 成员不为零，则 CMAC 算法只会应用于指定地址范围。

如果在应用代码中未访问这些符号，那么 RETAIN pragma 在 C 代码中是必需的。

若要将符号放置在要求的存储器位置，LOCATION pragma 是必需的。`cmac_sb_1` 到 `cmac_sb_4` 的 LOCATION 条目位于固定地址。`cmac_all` 的 LOCATION 地址可由用户指定。但它一定不能位于任何安全闪存引导区域中，因为器件的 ROM CMAC 实现方式不支持。

CMAC 算法在十六进制转换之前应用。原始输入 ELF 可执行文件没有改动。

十六进制转换实用程序仅针对定义了全局符号的 CMAC 区域应用 CMAC 算法。如果 ELF 可执行文件只定义了 `cmac_sb_1` 和 `cmac_all` ，则只会生成这两个 CMAC 标签，并填充到生成的十六进制输出文件中。

12.13 控制 ROM 器件地址

十六进制转换实用程序输出地址字段对应于 ROM 器件地址。EPROM 编程器将数据烧录到由十六进制转换实用程序输出文件地址字段指定的位置。十六进制转换实用程序提供了一些机制来控制每段在 ROM 中的起始地址。但是，许多 EPROM 编程器都直接控制 ROM 中的数据烧录位置。

十六进制转换实用程序输出文件的地址字段由以下项（按优先级从低到高列出）控制：

1. **链接器命令文件。**默认情况下，十六进制转换实用程序输出文件的地址字段是加载地址（在链接器命令文件中给出）.
2. **SECTIONS 指令的 paddr 参数。**为段指定 paddr 参数后，十六进制转换实用程序会绕过段加载地址并将段放置在由 paddr 指定的地址中。
3. **--zero 选项。**当您使用 --zero 选项时，该实用程序会将每个输出文件的地址原点复位为 0。由于每个文件从 0 开始并向上计数，因此任何地址记录都表示从文件开头（ROM 中的地址）的偏移量，而不是数据的实际目标地址。

   您必须将 --zero 选项与 --image 选项搭配使用，以强制每个输出文件中的起始地址为零。如果您指定了不带 --image 选项的 --zero 选项，该实用程序会发出警告并忽略 --zero 选项。
4. **--byte 选项。**一些 EPROM 编程器可能要求输出文件地址字段包含字节数（而非字数）。如果您使用 -byte 选项，输出文件地址将针对每个字节递增一次。例如，如果起始地址为 0h，第一行包含 8 个字，并且您不使用 -byte 选项，则第二行将从地址 8 (8h) 开始。如果起始地址为 0h，第一行包含 8 个字，并且您使用 -byte 选项，则第二行将从地址 16 (010h) 开始。两个示例中的数据是一样的；-byte 仅会影响输出文件地址字段的计算结果，而不影响转换后数据的实际目标处理器地址。

   --byte 选项使输出文件中的地址记录引用该文件中的字节位置，无论目标处理器是否可按字节寻址。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.14 控制十六进制转换实用程序诊断

十六进制转换实用程序使用某些 C/C++ 编译器选项来控制由十六进制转换器生成的诊断。

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

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

--diag_suppress=*id* 抑制 *id* 标识的诊断。若要确定诊断消息的数字标识符，请在单独的链接中首先使用 --display_error_number 选项。然后使用 --diag_suppress=*id* 抑制诊断。只能抑制任意诊断。

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

--display_error_number 显示诊断的数字标识符及其文本。使用此选项确定需要向诊断抑制选项提供哪些参数（--diag_suppress、--diag_error、--diag_remark 和 --diag_warning）。此选项还指示诊断是否是任意的。任意诊断是指其严重性可以被忽略的诊断。任意诊断包括后缀 -D；否则，不存在后缀。有关如何理解诊断消息的更多信息，请参阅 *ARM 优化 C/C++ 编译器用户指南*。

--issue_remarks 发出默认情况下被抑制的备注（非严重警告）。

--no_warnings 抑制警告诊断（仍会发出错误）。

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

--verbose_diagnostics 提供详细的诊断，以换行方式显示原始源代码，并指示错误在源代码行中的位置

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.15 目标格式说明

十六进制转换实用程序具有标识每种格式的选项。表 12-5 指定了格式选项。这些选项将在后续部分进行介绍。

- 您只能在命令行上使用这些选项之一。如果您使用多个选项，您列出的最后一个选项会覆盖其他选项。
- 默认格式为 Tektronix（--tektronix 选项）。

表 12-5. 用于指定十六进制转换格式的选项

选项	别名	格式	地址位	默认宽度
--ascii	-a	ASCII-Hex	32	8
--intel	-i	Intel	32	8
--motorola=1	-m1	Motorola-S1	16	8
--motorola=2	-m2	Motorola-S2	24	8
--motorola=3	-m3	Motorola-S3	32	8
--ti-tagged	-t	TI-Tagged	16	16
--ti_txt		TI_TXT	8	8
--tektronix	-x	Tektronix	32	8

地址位用于确定格式支持的地址信息位数。具有 16 位地址的格式仅支持最多 64K 的地址。该实用程序会截断目标地址以适应可用位数。

默认宽度指默认的 romwidth（如果未指定）。请注意，默认宽度不一定与地址记录中输出的十六进制值的长度相同。您可以使用 --romwidth 选项或使用 ROMS 指令中的 romwidth 参数来更改默认宽度。无法更改 TI-Tagged 格式（仅限 16 位宽度）或 TI-TXT 格式（仅限 8 位宽度）的默认宽度。

12.15.1 ASCII 十六进制对象格式（--ascii 选项）

ASCII 十六进制对象格式支持 32 位地址。该格式由字节流组成，字节由空格分隔。图 12-8 演示了 ASCII 十六进制格式。

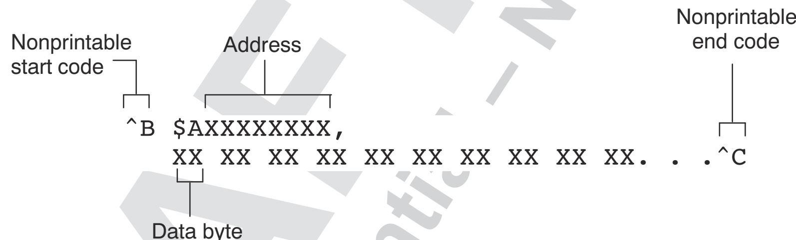


图 12-8. ASCII 十六进制对象格式

该文件以 ASCII STX 字符（ctrl-B，02h）开始，以 ASCII ETX 字符（ctrl-C，03h）结尾。地址记录用 $AXXXXXXX 表示，其中 XXXXXXXX 是一个包含至少 4 个十六进制字符的十六进制地址。地址记录仅在以下情况下出现：

- 出现不连续时
- 字节流不是从地址 0 开始时

您可以使用 --image 和 --zero 选项避免所有不连续性和任何地址记录。这将创建只包含字节值列表的输出。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.15.2 Intel MCS-86 目标格式（--intel 选项）

Intel 目标格式支持 16 位地址和 32 位扩展地址。Intel 格式包括 9 字符（4 字段）前缀（定义了记录开始、字节计数、加载地址和记录类型）、数据和 2 字符校验和后缀。

9 字符前缀表示三种记录类型：

记录类型	说明
00	数据记录
01	文件结尾记录
04	扩展线性地址记录

记录类型 00 是数据记录，以冒号 (:) 开始，后跟字节计数、第一个数据字节的地址、记录类型 (00) 和校验和。地址是 32 位地址的 16 个最低有效位；此值与最近的 04（扩展线性地址）记录中的值串联，形成完整的 32 位地址。校验和是记录中的前面字节的二进制补码（二进制格式），包括字节计数、地址和数据字节。

记录类型 01 是文件结尾记录，也以冒号 (:) 开始，后跟字节计数、地址、记录类型 (01) 和校验和。

记录类型 04 是扩展线性地址记录，指定了前 16 个地址位。它以冒号 (:) 开始，后跟字节计数、虚拟地址 0h、记录类型 (04)、地址的 16 个最高有效位和校验和。数据记录中的后续地址字段包含地址的最低有效字节。

图 12-9 显示了 Intel 十六进制目标格式。

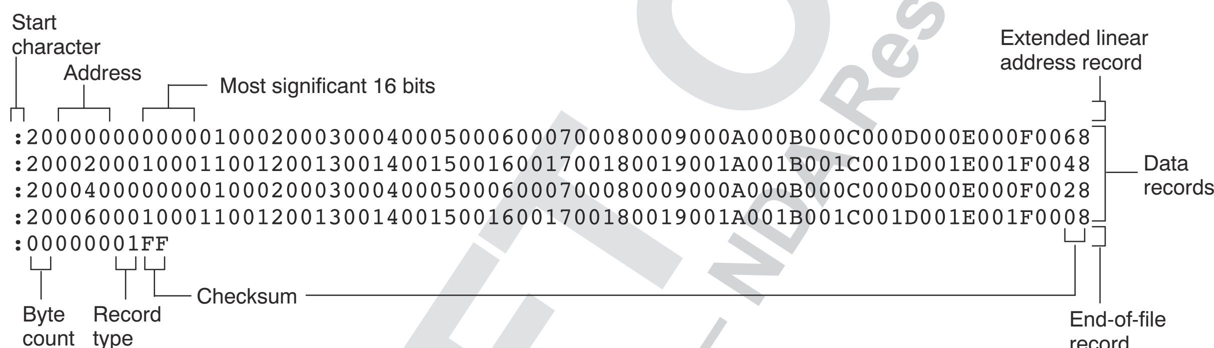


图 12-9. Intel 十六进制目标格式

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.15.3 Motorola Exorciser 对象格式（--motorola 选项）

Motorola S1、S2 和 S3 格式分别支持 16 位、24 位和 32 位地址。这些格式由文件开头（头）记录、数据记录和文件结尾（尾）记录构成。每个记录都包含五个字段：记录类型、字节计数、地址、数据和检验和。共有如下三种记录类型：

记录类型	说明
S0	头记录
S1	16 位地址的代码/数据记录（S1 格式）
S2	24 位地址的代码/数据记录（S2 格式）
S3	32 位地址的代码/数据记录（S3 格式）
S7	32 位地址的尾记录（S3 格式）
S8	24 位地址的尾记录（S2 格式）
S9	16 位地址的尾记录（S1 格式）

字节计数是记录中的字符对计数，不包括类型和字节计数本身。

检验和是几对字符所代表值之和的反码的最低有效字节，这几对字符包括字节计数、地址和代码/数据字段。

图 12-10 所示为 Motorola-S 对象格式。

图 12-10. Motorola-S 格式

English Document: SPNU118
Copyright © 2023 Texas Instruments Incorporated

12.15.4 扩展的 Tektronix 目标格式（--tektronix 选项）

Tektronix 目标格式支持 32 位地址，并有两种类型的记录：

数据记录	包含标头字段、加载地址和目标代码。
终止记录	表示模块的结束。

数据记录中的标头字段包含以下信息：

条目	ASCII 字符数	说明
%	1	数据类型为 Tektronix 格式
数据块长度	2	记录中的字符数，没有 %
数据块类型	1	6 = 数据记录 8 = 终止记录
校验和	2	除 % 和校验和本身外，记录中所有值的十六进制 2 位数总和，以 256 为模数。

数据记录中的加载地址指定目标代码的位置。第一个数字指定地址长度；始终为 8。数据记录的其余字符包含目标代码，每字节两个字符。

图 12-11 演示了 Tektronix 目标格式。

Checksum: 21h = 0+ 1+5+6+8+1+0+0+0+0+0+0+ 2+0+2+0+2+0+2+0+2+0+2+ 0

Block length 1ah = 26

Object code: 6 bytes

%15621810000000202020202020

Header character

Load address: 10000000h

Block type: 6 (data)

Length of load address

图 12-11. 扩展的 Tektronix 目标格式

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.15.5 德州仪器 (TI) SDSMAC (TI-Tagged) 目标格式（--ti_tagged 选项）

德州仪器 (TI) SDSMAC (TI-Tagged) 目标格式支持 16 位地址，包括文件开头记录、数据记录和文件结尾记录。每条数据记录均包含一系列小字段，并由一个标记字符表示：

标记字符	说明
K	后跟程序标识符
7	后跟校验和
8	后跟虚拟校验和（已忽略）
9	后跟 16 位加载地址
B	后跟数据字（四个字符）
F	标记数据记录的结束
*	后跟数据字节（两个字符）

图 12-12 展示了采用 TI-Tagged 目标格式的标记字符和字段。

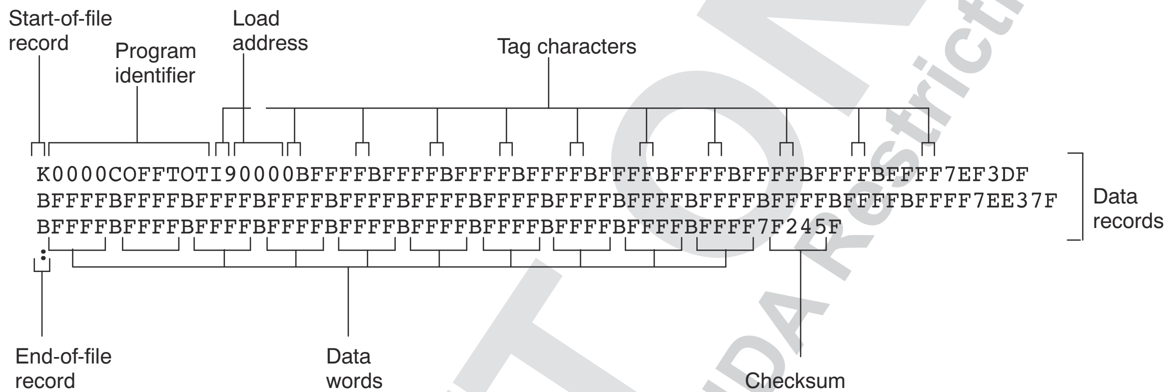


图 12-12. TI-Tagged 目标格式

如果在第一个地址之前出现任何数据字段，将为第一个字段分配地址 0000h。数据字节可能会表示出地址字段，但不是必备项。校验和字段之前的标记字符为 7，该字段是 8 位 ASCII 字符值之和的二进制补码，始于第一个标记字符，终于校验和标记字符（7 或 8）。文件结尾记录是一个冒号 (:)。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

12.15.6 TI-TXT 十六进制格式（--ti_txt 选项）

TI-TXT 十六进制格式支持 8 位十六进制数据。它包含段起始地址、数据字节和文件结束字符。以下限制条件适用：

- 段的数量是无限的。
- 每个十六进制起始地址必须是偶数。
- 每行必须有 8 个数据字节，段的最后一行除外。
- 数据字节由单个空格分隔。
- 文件结束终止标记 q 是必备项。

由于 TI-TXT 格式（以及二进制格式）仅支持一个 8 位物理存储器宽度，以及一个 8 位 ROM 宽度，如果从 16 位格式变为 8 位格式，ROMS 指令需要有双倍的原点和长度规格。如果用户收到的警告与以下内容类似，请检查 ROMS 指令。

```
warning: section file.out(.data) at 07e000000h falls in unconfigured memory
```

例如，假设某格式（例如 ASCII-Hex）使用 16 位 ROM 宽度，其 ROMS 指令使用 --romwidth=16 选项，如下所示：

```
ROMS {
   FLASH: origin=0x3f000000, length=0x1000
}
```

如果使用 8 位 ROM 宽度，则应在 ROMS 指令中将地址和长度加倍：

```
ROMS {
   FLASH: origin=0x7e000000, length=0x2000
}
```

数据记录包含以下信息：

条目	说明
@ADDR	段的十六进制起始地址
DATAn	十六进制数据字节
q	文件结束终止字符

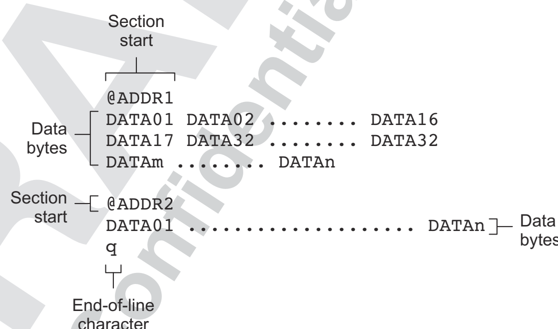


图 12-13. TI-TXT 目标格式

示例 12-3. TI-TXT 目标格式

```
@F000
31 40 00 03 B2 40 80 5A 20 01 D2 D3 22 00 D2 E3
21 00 3F 40 E8 FD 1F 83 FE 23 F9 3F
@FFFE
```


Copyright © 2023 Texas Instruments Incorporated

DRAFT ONLY
TI Confidential – NDA Restrictions

```
00 F0
Q
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

DRAFT ONLY
TI Confidential – NDA Restrictions

This page intentionally left blank.

Copyright © 2023 Texas Instruments Incorporated

章节 13

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。

13.1 .cdecls 指令概述.......... 312
13.2 C/C++ 转换注意事项.......... 313
13.3 C++ 专有转换注意事项.......... 316
13.4 汇编器特殊支持.......... 318

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

13.1 .cdecls 指令概述

.cdecls 指令允许程序员在混合汇编和 C/C++ 环境中共享 C 头文件，此头文件包含 C 和汇编代码间的声明和原型。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代码。这允许程序员在汇编代码中引用 C/C++ 结构；比如调用函数、分配空间、访问结构成员等；使用等效的汇编机制。虽然函数和变量定义会被忽略，但最常见的 C/C++ 元素被转换为汇编语言：枚举、（非函数类）宏、函数和变量原型、结构体和联合体。

有关 .cdecls 汇编器指令语法的详细信息，请参阅 .cdecls 指令说明。

.cdecls 指令可以出现在汇编源文件中的任何位置，并且可以在一个文件中多次出现。但是，一个 .cdecls 创建的 C/C++ 环境不会被后面的 .cdecls 继承；每个 .cdecls 实例的 C/C++ 环境都是全新的。

例如，以下代码会导致发出警告：

```
     .cdecls C,NOLIST
     %{
         #define ASMTEST 1
     %}
     .cdecls C,NOLIST
     %{
         #ifndef ASMTEST
             #warn "ASMTEST not defined!"  /* will be issued */
         #endif
     %}
```

因此，.cdecls 块的典型用法是在汇编源文件的开头附近单一使用，该源文件中包含所有必要的 C/C++ 头文件。

用户可以使用编译器 --include_path=*path* 选项来指定汇编中所用头文件所需的额外包含文件路径，就像编译 C 文件时那样。

由 .cdecls 代码生成的任何 C/C++ 错误或警告会像适用于 C/C++ 源代码的错误或警告那样正常发出。C/C++ 错误会导致该指令失败，并且任何转换得到的汇编代码都不会包含在内。

函数类宏或变量定义等无法转换的 C/C++ 结构会导致向转换而来的汇编文件中输出注释。例如：

```
; ASM HEADER WARNING - variable definition 'ABCD' ignored
```

每条消息的开头都会显示前缀 ASM HEADER WARNING。若要查看警告，需要指定 WARN 参数，以便在 STDERR 中显示对应消息；或者需要指定 LIST 参数，以便在列表文件中显示警告（若有）。

最后，请注意，转换而来的汇编代码不会以与原有 C/C++ 源代码相同的顺序显示，并且 C/C++ 结构可能会在转换过程中简化为标准化形式，但这应该不会影响其最终使用。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

13.2 C/C++ 转换注意事项

以下几节介绍了在与汇编源文件共享头文件时需要注意的 C 和 C++ 转换元素。

13.2.1 说明

注释完全在 C 级别使用，并且不会出现在生成的转换后汇编文件中。

13.2.2 条件编译（#if/#else/#ifdef/等）

在转换步骤中，条件编译完全在 C 级处理。在命令行上（使用编译器 --define=*name*=*value* 选项）或使用 #define 在 .cdecls 块中定义所有必要的宏。#if、#ifdef 等 C/C++ 指令不会转换为汇编 .if、.else、.elseif 和 .endif 指令。

13.2.3 Pragma

C/C++ 源代码中的 Pragma 不会被转换，因此会导致生成警告。它们对生成的汇编文件没有其他影响。请参阅 .cdecls 主题，了解关于这些警告创建位置的 WARN 和 NOWARN 参数讨论。

13.2.4 #error 和 #warning 指令

这些预处理器指令完全由编译器在转换的解析步骤期间处理。如果遇到了这些指令之一，将发出相应的错误或警告消息。这些指令不会转换为汇编输出中的 .emsg 或 .wmsg。

13.2.5 预定义符号 __ASM_HEADER__

C/C++ 宏 __ASM_HEADER__ 是在处理 .cdecls 内的代码时在编译器中定义的。这让您可以在 .cdecls 处理期间更改代码，例如不编译定义。

备注

小心处理 __ASM_HEADER__ 宏

在使用此宏时，您必须非常小心，确保不会引入任何不必要的代码更改而导致编译 C/C++ 源代码期间和转换至汇编期间所处理的代码之间存在不一致情况。

13.2.6 在 C/C++ asm( ) 语句中的用法

不允许在 C/C++ asm( ) 语句中使用 .cdecls 指令，这将导致错误。

13.2.7 #include 指令

C/C++ #include 预处理器指令由编译器在转换步骤期间透明地进行处理。#include 可根据需要像在 C/C++ 源代码中一样进行多层嵌套。汇编指令 .include 和 .copy 不需要在 .cdecls 中使用。使用命令行 --include_path 选项指定要搜索包含文件的额外路径，就像 C 编译时那样。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

13.2.8 转换 #define 宏

只有类对象的宏才会转换为汇编指令。类函数的宏没有汇编表示，因此无法转换。预定义和内置 C/C++ 宏不会转换为汇编指令（即 __FILE__、__TIME__、__TI_COMPILER_VERSION__ 等）。例如，这段代码会转换为汇编指令，因为它是一个类对象的宏：

```
#define NAME Charley
```

这段代码不会转换为汇编指令，因为它是类函数的宏：

```
#define MAX(x,y) (x>y ? x : y)
```

某些宏虽会进行转换，但在包含的汇编文件中没有任何功能用途。例如，以下结果会将汇编替换符号 FOREVER 设置为值 while(1)，但这在汇编指令中不起任何作用，因为 while(1) 不是合法的汇编代码。

```
#define FOREVER while(1)
```

宏不会按照转换的方式进行解释。例如，以下代码会将汇编器替代符号 OFFSET 设置为文字字符串值 5+12，而非值 17。发生这种情况的原因是 C/C++ 语言的语义要求宏在相关上下文中进行计算，而不是在解析时进行计算。

```
#define OFFSET 5+12
```

因为 C/C++ 中的宏是在它们的使用上下文中进行计算的，所以 C/C++ printf 转义序列（如 \n）不会在转换后的汇编宏中转换为单个字符。有关如何使用 C/C++ 宏字符串的建议，请参阅节 13.2.11。

宏使用 .define 指令（参阅节 13.4.2）进行转换，其功能类似于 .asg 汇编器指令。例外情况是 .define 不允许重新定义寄存器符号和助记符，以防止转换破坏基本汇编环境。若要从汇编范围中删除宏，可以在定义它的 .cdecls 之后使用 .undef（参阅节 13.4.3）。

#（字符串化运算符）的宏功能仅在功能宏中有用。此过程不支持功能宏，因此也不支持 #。串联运算符 ## 仅在功能上下文中起作用，但可以退化地用于串联两个字符串，因此它在该上下文中受支持。

13.2.9 #undef 指令

在 .cdecls 结束前使用 #undef 指令取消符号定义，这些符号不会转换为汇编语言。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

13.2.10 枚举

枚举成员转换为汇编语言中的 .enum 元素。例如：

```
enum state { ACTIVE=0x10, SLEEPING=0x01, INTERRUPT=0x100, POWEROFF, LAST};
```

转换为以下汇编代码：

```
state       .enum
ACTIVE      .emember 16
SLEEPING    .emember 1
NTERRUPT    .emember 256
POWEROFF    .emember 257
LAST        .emember 258
            .endenum
```

通过 .enum 指令创建的伪作用域使用这些成员。

其用法类似于访问结构成员 enum_name.member。

该伪作用域用于防止枚举成员名称破坏汇编环境中的其他符号。

13.2.11 C 字符串

由于在将诸如 \n 和 \t 之类的 C 字符串转义符用于 C/C++ 程序中的字符串常量之前，这些符号不会转换为十六进制字符 0x0A 和 0x09，因此其值为字符串的 C 宏无法在程序集替换符号中按预期表示。例如：

```
#define MSG "\tHI\n"
```

becomes, in assembly:

```
.define """\tHI\n""",MSG ; 6 quoted characters! not 5!
```

用在 C 字符串上下文中时，您希望将该语句转换为 5 个字符（制表符、H、I、换行符、NULL），但 .string 汇编器指令不知道如何执行 C 转义转换。

您可以像使用 C/C++ 一样，使用 .cstring 指令正确处理转义序列和 NULL 终止。将上述符号 MSG 与 .cstring 指令一同使用会导致分配 5 个字符的存储空间，这与在 C/C++ 强上下文中使用的结果相同。（有关 .cstring 指令语法，请参阅节 13.4.7。）

13.2.12 C/C++ 内置函数

C/C++ 内置函数（例如 sizeof( )）若用在宏中，则不会转换为与其对应的汇编表达式（如果有）。此外，它们的 C 表达式值不会插入到生成的汇编宏中，因为宏是在上下文中计算的，并且在将宏转换为汇编表达式时没有主动上下文。

合适的函数（例如 $$sizeof( )）可用在汇编表达式中。但是，int/char/float 等基本类型在汇编表达式中没有类型表示，因此无法在汇编表达式中要求 $$sizeof(int)。

13.2.13 结构体和联合体

C/C++ 结构体和联合体转换为汇编 .struct 和 .union 元素。已根据需要添加边界填充和结尾，使生成的汇编结构与 C/C++ 源代码具有相同的大小和成员偏移量。主要目的是实现对 C/C++ 结构成员的访问，并促进汇编代码的调试。对于嵌套结构，使用汇编 .tag 特性引用其他结构体/联合体。

还从 C/C++ 源代码传递了对齐特性，使汇编符号标记的对齐与 C/C++ 符号相同。（pragma 可能尝试修改结构体，有关信息请参阅节 13.2.3。）由于结构体的对齐存储在汇编符号中，$$sizeof( ) 和 $$alignof( ) 等内置汇编函数可用于生成的结构体名称符号。

Copyright © 2023 Texas Instruments Incorporated

在 typedef 中使用未命名结构体（或联合体）时，例如：

```
typedef struct { int a_member; } mystrname;
```

这是以下代码的简写：

```
struct temporary_name { int a_member; };
typedef temporary_name mystrname;
```

转换时会以相同方式处理以上语句：为结构体生成临时名称，然后使用 .define 将 typedef 从临时名称输出到用户名称。在汇编中使用 *mystrname* 的方式应与在 C/C++ 中相同，但不要混淆列表中的汇编结构定义，其中包含的是临时名称。为结构体指定名称可避免出现临时名称，例如：

```
typedef struct a_st_name { ...} mystrname;
```

如果在 C 中使用简略方法声明特定结构的变量，例如：

```
extern struct a_name { int a_member; } a_variable;
```

那么将结构体转换为汇编语言后，会生成 .tag 指令，声明外部变量的结构，例如：

```
_a_variable .tag a_st_name
```

这样可在汇编代码中引用 _a_variable.a_member。

13.2.14 函数/变量原型

非静态函数和变量原型（并非定义）将会为找到的每个符号生成一个 .global 指令。

有关 C++ 名称改编问题，请参阅节 13.3.1。

函数和变量定义将会为每个符号生成警告消息（有关创建这些警告的位置，请参阅 WARN/NOWARN 参数相关讨论），并且它们将不会在转换后的汇编语言中展现。

表示变量声明的汇编符号将不包含有关这些符号的类型信息。只会为它们发出 .global 指令。因此，您有责任确保正确使用该符号。

有关结构体/联合体类型的变量名称的信息，请参阅节 13.2.13。

13.2.15 C 常量后缀

C 常量后缀 u、l 和 f 原封不动地传递给汇编语言。如果用在汇编表达式中，汇编器将忽略这些后缀。

13.2.16 基本 C/C++ 类型

只有 C/C++ 源代码中的复合类型（结构体和联合体）才会转换为汇编语言。除了以前存在于汇编语言的任何现有 .int、.char、.float 等指令之外，int、char 或 float 等基本类型不能转换为或表示为汇编语言。因此，基本类型的 Typedef 也无法表示为转换后的汇编语言。

13.3 C++ 专有转换注意事项

以下几节介绍了在与汇编源文件共享头文件时需要注意的 C++ 专有转换元素。

13.3.1 名称改编

C++ 编译器使用*名称改编*功能来避免同名函数和变量之间的冲突。如果未使用名称改编功能，则可能会发生符号名称冲突。

您可以使用名称还原器 (armdem) 对名称进行还原并识别要在汇编语言中使用的正确符号。相关详细信息，请参阅 *ARM 优化 C/C++ 编译器用户指南* 中的“C++ 名称还原器”一章。

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY TI Confidential - NDA Restrictions

若要在 C++ 中针对不需要多态性（使用不同类型的参数调用同名函数）的符号取消名称改编，请使用以下语法：

```
extern "C" void somefunc(int arg);
```

上述格式是声明单个函数的简短方法。若要将此方法用于多个函数，还可以使用以下语法：

```
extern "C"
  {
      void somefunc(int arg);
      int  anotherfunc(int arg);
      ...
  }
```

13.3.2 衍生类

由于存在与汇编语言中不存在的 C++ 作用域相关的问题，在转换为汇编时仅部分支持衍生类。最大区别是基类成员不会自动成为衍生类的完整（顶层）成员。例如：

```
class base
{
   public:
        int b1;
};
class derived : public base
{
    public:
       int d1;
}
```

在 C++ 代码中，衍生类将包含整数 b1 和 d1。在转换后“衍生”的汇编结构中，必须使用基类的名称（例如 derived.__b_base.b1，而非预期的 derived.b1）访问基类的成员。

非虚拟、非空基类将在衍生类中的名称前加上 __b_ 以表示它是基类名称。因此，上面的示例是 derived.__b_base.b1，而不是简单的 derived.base.b1。

13.3.3 模板

不支持模板。

13.3.4 虚拟函数

虚拟函数没有汇编表示，因此不受支持。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

13.4 汇编器特殊支持

13.4.1 枚举 (.enum/.emember/.endenum)

以下指令支持枚举的伪作用域：

ENUM_NAME	**.enum**
MEMBER1	**.emember** [*value*]
MEMBER2	**.emember** [*value*]
...	
	.endenum

.enum 指令用于开始枚举定义，而 **.endenum** 用于终止枚举定义。

枚举名称 (*ENUM_NAME*) 不能用于分配空间；它的大小报告为零。

若要使用成员的值，格式应为 *ENUM_NAME.MEMBER*，类似于使用结构成员。

.emember 指令用于选择性地接受成员设置的目标值，就像在 C/C++ 中一样。如果未指定，则该成员的值比前一个成员的值大 1。与在 C/C++ 中一样，成员名称不能重复，但值可以重复。除非用 .emember 指定，否则将对第一个枚举成员赋值 0（零），就像在 C/C++ 中一样。

.endenum 指令不能与标签一同使用，而结构体 .endstruct 指令可以，因为 .endenum 指令没有像 .endstruct 那样的值（包含结构体的大小）。

条件编译指令 (.if/.else/.elseif/.endif) 是 .enum/.endenum 序列中唯一允许的其他非枚举代码。

13.4.2 .define 指令

.define 指令的运行方式与 .asg 指令相同，但 .define 不允许创建与寄存器符号或助记符同名的替代符号。它不会在汇编器中创建新的符号命名空间，而是使用现有的替代符号命名空间。此指令的语法为：

.define *substitution string* **,** *substitution symbol name*

.define 指令用于在转换 C/C++ 头文件时防止汇编环境遭损坏。

13.4.3 .undefine/.unasg 指令

.undef 指令用于删除替代符号的定义，该符号使用 .define 或 .asg 创建。此指令会将替代符号表中 .undef 所在的位置到汇编文件结尾处之间的指定符号删除。这些指令的语法为：

.undefine *substitution symbol name*

.unasg *substitution symbol name*

可用于从汇编环境中删除可导致问题的所有 C/C++ 宏。另请参阅介绍 .define 指令的节 13.4.2。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

13.4.4 $$defined( ) 内置函数

$$defined 指令用于返回 true/1 或 false/0，具体取决于名称是存在于当前替代符号表还是标准符号表中。实际上，如果汇编器范围中有任何该名称的用户符号，$$defined 返回 TRUE。它与 $$isdefed 的区别在于，$$isdefed 仅测试非替代符号。语法为：

$$defined(*substitution symbol name* **)**

“.if $$defined(macroname)”这样的语句与 C 代码“#ifdef macroname”类似。

请参阅节 13.4.2 和节 13.4.3，了解 .define 和 .undef 在汇编语言中的用法。

13.4.5 $$sizeof 内置函数

汇编内置函数 $$sizeof( ) 可用于查询汇编中某结构的大小。它是现有的 $$structsz( ) 的别名。语法为：

$$sizeof(*structure name* **)**

$$sizeof 函数的用法与 C 内置函数 sizeof( ) 类似。

无法使用汇编器的 $$sizeof( ) 内置函数查询基本 C/C++ 类型的大小，例如 $$sizeof(int)，因为这些基本类型名称不出现在汇编中。只有复合类型才会从 C/C++ 转换为汇编语言。

另请参阅节 13.2.12，那里提到如果在宏中使用 C/C++ sizeof( ) 内置函数，则不会自动进行这种转换。

13.4.6 结构体/联合体对齐和 $$alignof( )

汇编 .struct 和 .union 指令可搭配使用可选的第二个参数，用于指定可应用于符号名称的最小对齐量。转换进程使用它将具体对齐量从 C/C++ 传递到汇编。

汇编内置函数 $$alignof( ) 可用于报告这些结构体的对齐。该函数甚至还可用于汇编结构，将返回汇编器计算的最小对齐量。

13.4.7 .cstring 指令

您可以像使用 C/C++ 一样，使用 .cstring 指令正确处理转义序列和 NULL 终止。

```
    .cstring "String with C escapes.\nWill be NULL terminated.\012"
```

请参阅节 13.2.11，了解有关 .cstring 指令的更多信息。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

DRAFT ONLY
TI Confidential – NDA Restrictions

This page intentionally left blank.

Copyright © 2023 Texas Instruments Incorporated

附录 A
符号调试指令

汇编器支持 ARM C/C++ 编译器用于进行符号调试的若干指令。

这些指令并不是供汇编语言程序员使用的。它们需要的参数很难手动计算，使用时还必须符合编译器、汇编器和调试器之间预先确定的协议。本附录仅出于提供信息之目的记录这些指令。

A.1 DWARF 调试格式.....322
A.2 调试指令语法.....322

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

A.1 DWARF 调试格式

DWARF 符号调试指令的子集始终列在编译器为程序分析目的而创建的汇编语言文件中。若要列出用于完整符号调试的完整集，请使用 **--symdebug:dwarf** 选项调用编译器，如下所示：

```
armcl --symdebug:dwarf --keep_asm input_file
```

--keep_asm 选项会指示编译器保留生成的汇编文件。

若要禁止生成所有符号调试指令，请使用 **-symdebug:none** 选项调用编译器：

```
armcl --symdebug:none --keep_asm input_file
```

DWARF 调试格式包含以下指令：

- **.dwtag** 和 **.dwendtag** 指令用于在 .debug_info 段中定义调试信息条目 (DIE)。
- **.dwattr** 指令用于向现有 DIE 添加属性。
- **.dwpsn** 指令用于标识 C/C++ 语句的源位置。
- **.dwcie** 和 **.dwendentry** 指令用于在 .debug_frame 段中定义公共信息条目 (CIE)。
- **.dwfde** 和 **.dwendentry** 指令用于在 .debug_frame 段中定义帧描述条目 (FDE)。
- **.dwcfi** 指令用于定义 CIE 或 FDE 的调用帧指令。

A.2 调试指令语法

表 A-1 是按字母顺序排列的符号调试指令列表。有关 C/C++ 编译器的信息，请参阅 *ARM 优化 C/C++ 编译器用户指南*。

表 A-1. 符号调试指令

标签	指令	参数
	.dwattr	*DIE label* **,** *DIE attribute name* **(** *DIE attribute value* **)[,** *DIE attribute name* **(** *attribute value* **)** [, ...]
	.dwcfi	*call frame instruction opcode*[**,** *operand*[**,** *operand*]]
CIE 标签	**.dwcie**	*version* **,** *return address register*
	.dwendentry	
	.dwendtag	
	.dwfde	*CIE 标签*
	.dwpsn	**"** *filename* **",** *line number* **,** *column number*
DIE 标签	**.dwtag**	*DIE tag name* **,** *DIE attribute name* **(** *DIE attribute value* **)[,** *DIE attribute name* **(** *attribute value* **)** [, ...]

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

XML 链接信息文件说明

ARM 链接器支持通过 --xml_link_info *file* 选项生成 XML 链接信息文件。此选项会使链接器生成一个格式良好的 XML 文件，其中包含有关链接结果的详细信息。这个文件中的信息包括由链接器生成的映射文件中当前生成的所有信息。

随着链接器的发展，XML 链接信息文件可扩展到包含其他有用信息，以对链接器结果进行静态分析。

本附录列举了链接器在 XML 链接信息文件中生成的所有元素。

B.1 XML 信息文件元素类型....................324
B.2 文档元素....................324

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

B.1 XML 信息文件元素类型

链接器将生成以下元素类型：

- **容器元素**表示某对象包含描述该对象的其他元素。容器元素的 id 属性使其他元素可以访问它们。
- **字符串元素**包含其值的字符串表示。
- **常量元素**包含其值的 32 位 无符号长整型表示（0x 为前缀）。
- **引用元素**是包含 idref 属性的空元素，指定与另一容器元素的链接。

在节 B.2 中，在每个元素的元素说明之后，在括号中指定元素类型。例如，<link_time> 元素列出了执行链接的时间（字符串）。

B.2 文档元素

根元素或文档元素为 **<link_info>**。XML 链接信息文件中包含的所有其他元素都是 <link_info> 元素的子元素。以下各节描述了 XML 信息文件可以包含的元素。

B.2.1 标头元素

XML 链接信息文件中的第一个元素提供有关链接器和链接会话的一般信息：

- **<banner>** 元素列出了可执行文件的名称和版本信息（字符串）。
- **<copyright>** 元素列出了 TI 版权信息（字符串）。
- **<link_time>** 是链接时间的时间戳表示（无符号 32 位整数）。
- **<output_file>** 元素列出了生成的链接输出文件的名称（字符串）。
- **<entry_point>** 元素指定程序入口点，由链接器（容器）通过两个条目确定：
  - **<name>** 是入口点符号名称（字符串）（如有）。
  - **<address>** 是入口点地址（常量）。

hi.out 输出文件的标头元素

```
<banner>TMS320Cxx Linker           Version x.xx (Jan 6 2008)</banner>
<copyright>Copyright (c) 1996-2008 Texas Instruments Incorporated</copyright>
<link_time>0x43dfd8a4</link_time>
<output_file>hi.out</output_file>
<entry_point>
   <name>_c_int00</name>
   <address>0xaf80</address>
</entry_point>
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

B.2.2 输入文件列表

XML 链接信息文件的下一段是输入文件列表，用 **<input_file_list>** 容器元素分隔。<input_file_list> 可以包含任意数量的 <input_file> 元素。

每个 **<input_file>** 实例指定链接中涉及的输入文件。每个 <input_file> 都有一个 ID 属性，可以直接由其他元素（例如 <object_component>）引用。<input_file> 是包含以下元素的容器元素：

- **<path>** 元素在适用时指定目录路径的名称（字符串）。
- **<kind>** 元素指定文件类型，可以是存档或目标（字符串）。
- **<file>** 元素指定存档名称或文件名（字符串）。
- **<name>** 元素指定目标文件名或存档成员名（字符串）。

***hi.out* 输出文件的输入文件列表**

```
<input_file_list>
   <input_file id="fl-1">
      <kind>object</kind>
      <file>hi.obj</file>
      <name>hi.obj</name>
   </input_file>
   <input_file id="fl-2">
      <path>/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>boot.obj</name>
   </input_file>
   <input_file id="fl-3">
      <path>/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>exit.obj</name>
   </input_file>
   <input_file id="fl-4">
      <path>/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>printf.obj</name>
   </input_file>
...
</input_file_list>
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

B.2.3 对象组件列表

XML 链接信息文件的下一个段包含链接中涉及的所有对象组件的规格。对象组件的一个示例是输入段。通常，对象组件是链接器可以操作的最小对象元素。

<object_component_list> 是一个包含任意数量的 <object_component> 元素的容器元素。

每个 **<object_component>** 指定一个对象组件。每个 <object_component> 都有一个 ID 属性，可以由其他元素（例如 <logical_group>）直接引用。<object_component> 是包含以下元素的容器元素：

- **<name>** 元素指定对象组件（字符串）。
- **<load_address>** 元素指定对象组件的加载时地址（常量）。
- **<run_address>** 元素指定对象组件的运行时地址（常量）。
- **<size>** 元素指定对象组件的大小（常量）。
- **<input_file_ref>** 元素指定对象组件起源的源文件（引用）。

fl-4 输入文件的对象组件列表

```
<object_component id="oc-20">
   <name>.text</name>
   <load_address>0xac00</load_address>
   <run_address>0xac00</run_address>
   <size>0xc0</size>
   <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-21">
   <name>.data</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <size>0x0</size>
   <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-22">
   <name>.bss</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <size>0x0</size>
   <input_file_ref idref="fl-4"/>
</object_component>
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

B.2.4 逻辑组列表

XML 链接信息文件的 **<logical_group_list>** 段类似于由链接器生成的映射文件中的输出段。但是，XML 链接信息文件包含 GROUP 和 UNION 输出段的规范，而映射文件中没有规范。<logical_group_list> 中会出现三种类型的列表项：

- **<logical_group>** 是包含目标组件或逻辑组成员列表的段或组的规范。每个 <logical_group> 元素提供一个 id，以便可以从其他元素引用它。每个 <logical_group> 是围住以下元素的容器元素：
  - **<name>** 元素指定逻辑组的名称（字符串）。
  - **<load_address>** 元素指定逻辑组的加载时地址（常量）。
  - **<run_address>** 元素指定逻辑组的运行时地址（常量）。
  - **<size>** 元素指定逻辑组的大小（常量）。
  - **<contents>** 元素列出此逻辑组中包含的元素（容器）。这些元素引用此逻辑组中包含的每个成员对象：
    - **<object_component_ref>** 是此逻辑组中包含的目标组件（引用）。
    - **<logical_group_ref>** 是此逻辑组中包含的逻辑组（引用）。
- **<overlay>** 是特殊类型的逻辑组，表示 UNION 或共享相同存储器空间的一组对象（容器）。为每个 <overlay> 元素提供一个 id，以便可以从其他元素引用它（例如，从放置映射中的 <allocated_space> 元素引用）。每个 <overlay> 包含以下元素：
  - **<name>** 元素指定覆盖的名称（字符串）。
  - **<run_address>** 元素指定覆盖的运行时地址（常量）。
  - **<size>** 元素指定逻辑组的大小（常量）。
  - **<contents>** 容器元素列出此覆盖中包含的元素。这些元素引用此逻辑组中包含的每个成员对象：
    - **<object_component_ref>** 是此逻辑组中包含的目标组件（引用）。
    - **<logical_group_ref>** 是此逻辑组中包含的逻辑组（引用）。
- **<split_section>** 是另一个特殊类型的逻辑组，表示分拆到多个存储器区域的逻辑组集合。每个 <split_section> 元素提供一个 id，以便可以从其他元素引用它。id 由以下元素组成。
  - **<name>** 元素指定分拆段的名称（字符串）。
  - **<contents>** 容器元素列出此分拆段中包含的元素。<logical_group_ref> 元素引用此分拆段中包含的每个成员对象，引用的每个元素是此分拆段中包含的逻辑组（引用）。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

例 I-4 输入文件的逻辑组列表

```
<logical_group_list>
   ...
      <logical_group id="lg-7">
      <name>.text</name>
      <load_address>0x20</load_address>
      <run_address>0x20</run_address>
      <size>0xb240</size>
      <contents>
         <object_component_ref idref="oc-34"/>
         <object_component_ref idref="oc-108"/>
         <object_component_ref idref="oc-e2"/>
      ...
      </contents>
   </logical_group>
   ...
   <overlay id="lg-b">
      <name>UNION_1</name>
      <run_address>0xb600</run_address>
      <size>0xc0</size>
      <contents>
         <object_component_ref idref="oc-45"/>
         <logical_group_ref idref="lg-8"/>
      </contents>
   </overlay>
    ...
   <split_section id="lg-12">
      <name>.task_scn</name>
      <size>0x120</size>
      <contents>
         <logical_group_ref idref="lg-10"/>
         <logical_group_ref idref="lg-11"/>
      </contents>
   ...
</logical_group_list>
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

B.2.5 放置映射

<placement_map> 元素用于描述应用中所有指定存储器区域的存储器放置详细信息，包括特定存储器区域中已有逻辑组之间的未用空间。

<memory_area> 用于描述指定存储器区域（容器）内的放置详细信息。该描述包括以下项目：

- **<name>** 指定存储器区域的名称（字符串）。
- **<page_id>** 提供了用于定义此存储器区域的存储页面对应的 ID（常量）。
- **<origin>** 指定存储器区域的起始地址（常量）。
- **<length>** 指定存储器区域的长度（常量）。
- **<used_space>** 指定此区域中已分配的空间量（常量）。
- **<unused_space>** 指定此区域中的可用空间量（常量）。
- **<attributes>** 列出与此区域关联的 RWXI 属性（若有）（字符串）。
- **<fill_value>** 指定要在未用空间中填写的填充值，在为存储器区域指定了 fill 指令时使用（常量）。
- **<usage_details>** 会列出此存储器区域中每个已分配或可用片段的详细信息。如果片段已分配给逻辑组，则会提供 <logical_group_ref> 元素来协助访问该逻辑组的详细信息。所有片段规范均包含 <start_address> 和 <size> 元素。
  - **<allocated_space>** 元素提供此存储器区域内某个已分配片段的详细信息（容器）：
    - **<start_address>** 指定片段的地址（常量）。
    - **<size>** 指定片段的大小（常量）。
    - **<logical_group_ref>** 提供对已分配至此片段的逻辑组的引用（引用）。
  - **<available_space** 元素提供此存储器区域内某个可用片段的详细信息（容器）：
    - **<start_address>** 指定片段的地址（常量）。
    - **<size>** 指定片段的大小（常量）。

fl-4 输入文件的放置映射

```
<placement_map>
   <memory_area>
      <name>PMEM</name>
      <page_id>0x0</page_id>
      <origin>0x20</origin>
      <length>0x100000</length>
      <used_space>0xb240</used_space>
      <unused_space>0xf4dc0</unused_space>
      <attributes>RWXI</attributes>
      <usage_details>
         <allocated_space>
            <start_address>0x20</start_address>
            <size>0xb240</size>
            <logical_group_ref idref="lg-7"/>
         </allocated_space>
         <available_space>
            <start_address>0xb260</start_address>
            <size>0xf4dc0</size>
         </available_space>
      </usage_details>
   </memory_area>
   ...
</placement_map>
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

B.2.6 Far Call Trampoline 列表

<far_call_trampoline_list> 是 <far_call_trampoline> 元素的列表。链接器支持生成 far call trampoline，以帮助调用站点到达超出范围的目的地。far call trampoline 函数可以保证到达被调用函数（被调用方），因为它可以利用对被调用函数的间接调用。

<far_call_trampoline_list> 枚举由链接器为某一特定链路生成的所有 far call trampoline。<far_call_trampoline_list> 可以包含任意数量的 <far_call_trampoline> 元素。每个 **<far_call_trampoline>** 都是一个包含以下元素的容器：

- **<callee_name>** 元素命名目标函数（字符串）。
- **<callee_address>** 是被调用函数的地址（常量）。
- **<trampoline_object_component_ref>** 是对包含 trampoline 函数定义的对象组件的引用（引用）。
- **<trampoline_address>** 是 trampoline 函数的地址（常量）。
- **<caller_list>** 枚举利用此 trampoline 到达被调用函数的所有调用站点（容器）。
- **<trampoline_call_site>** 提供 trampoline 调用站点的详细信息（容器），包括以下各项：
  - **<caller_address>** 指定调用站点地址（常量）。
  - **<caller_object_component_ref>** 是调用站点所在的对象组件（引用）。

fl-4 输入文件的 Fall Call Trampoline 列表

```
<far_call_trampoline_list>
...
   <far_call_trampoline>
      <callee_name>_foo</callee_name>
      <callee_address>0x08000030</callee_address>
      <trampoline_object_component_ref idref="oc-123"/>
      <trampoline_address>0x2020</trampoline_address>
      <caller_list>
         <call_site>
            <caller_address>0x1800</caller_address>
            <caller_object_component_ref idref="oc-23"/>
         </call_site>
         <call_site>
            <caller_address>0x1810</caller_address>
            <caller_object_component_ref idref="oc-23"/>
         </call_site>
      </caller_list>
</far_call_trampoline>
...
</far_call_trampoline_list>
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

B.2.7 符号表

<symbol_table> 包含链接中所含所有全局符号的列表。该列表提供有关符号的名称和值的信息。将来，symbol_table 列表可提供类型信息、定义符号的对象组件、存储类等信息。

<symbol> 是一个容器元素，用以下元素指定符号的名称和值：

- **<name>** 元素指定符号名称（字符串）。
- **<value>** 元素指定符号值（常量）。

fl-4 输入文件的符号表

```
<symbol_table>
   <symbol>
      <name>_c_int00</name>
      <value>0xaf80</value>
   </symbol>
   <symbol>
      <name>_main</name>
      <value>0xb1e0</value>
   </symbol>
   <symbol>
      <name>_printf</name>
      <value>0xac00</value>
   </symbol>
   ...
</symbol_table>
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

This page intentionally left blank.

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

附录 C
十六进制转换实用程序示例

灵活的十六进制转换实用程序提供了许多选项和功能。了解配置 EPROM 系统的正确方式和 EPROM 编程器的要求后，您就会发现为特定应用程序转换文件非常简单。

本附录中的三种场景展示了如何使用 16-BIS（16 位指令集）代码和使用多 EPROM 系统来开发十六进制转换命令文件以避免出现空洞。这些场景均使用以下汇编代码：

```
**************************************************
* Assemble two words into section "secA"         *
**************************************************
   .sect "secA"
   .word 012345678h
   .word 0abcd1234h
**************************************************
* Assemble two words into section "secB"         *
**************************************************
   .sect "secB"
   .word 087654321h
   .word 04321dcbah
```

在使用本附录之前，请阅读章节 12，了解如何使用十六进制转换实用程序。

C.1 场景 1 -- 为单个 8 位 EPROM 构建十六进制转换命令文件.....334
C.2 场景 2 -- 为 16-BIS 代码构建十六进制转换命令文件.....338
C.3 场景 3 -- 为两个 8 位 EPROM 构建十六进制转换命令文件.....341

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

C.1 场景 1 -- 为单个 8 位 EPROM 构建十六进制转换命令文件

场景 1 展示了如何构建十六进制转换命令文件以转换图 C-1 中所示存储器系统的目标文件。此系统中有一个连接到 TMS470 目标处理器的外部 128K × 8 位 EPROM。

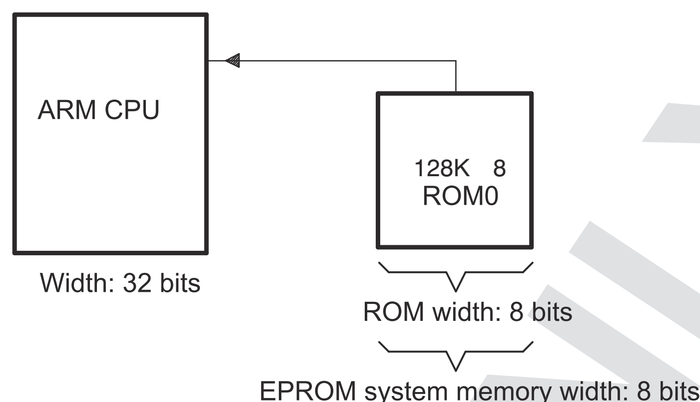


图 C-1. 场景 1 的 EPROM 存储器系统

目标文件由分配了存储器位置的存储器块（段）组成。通常，各段互不相邻：地址空间中的段与段之间存在一些空洞，这些空洞没有数据。场景 1 展示了如何使用十六进制转换实用程序的映像模式来用填充值填充段之前、之间或之后的任何空洞。

在此场景中，应用程序代码驻留在 TMS470 CPU 的程序存储器 (ROM) 中，但此代码使用的数据表驻留在片外 EPROM 中。

目标板的电路负责处理对数据的访问；0x1000 的本机 TMS470 地址访问 EPROM 上的位置 0x0。

为满足代码的地址要求，此场景需要一个链接器命令文件来分配段和存储器，如下所示：

- 必须链接程序/应用程序代码（此场景由示例 C-1 中所示的 secA 段表示），以便其地址空间驻留在 TMS470 CPU 上的程序存储器 (ROM) 中。
- 为满足数据加载到 EPROM 中的地址 0x0 处，但由地址 0x1000 处的应用程序代码引用这一条件，必须为 secB（包含此应用程序数据的段）分配一个 0x1000 的链接器加载地址，这样一来，对于该段中数据的所有引用都将相对于 TMS470 CPU 地址进行解析。在十六进制转换实用程序命令文件中，必须使用 paddr 选项来将该段数据烧录到 EPROM 地址 0x0 处。此值会覆盖由链接器分配的段加载地址。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

示例 C-1 展示了解析所述规范中所需地址的链接器命令文件。

示例 C-1. 场景 1 的链接器命令文件和链接映射

```
/****************************************************************************/
/* Scenario 1  Link Command                                                 */
/*                                                                          */
/*   Usage:  armlnk <obj files...>    -o <out file> -m <map file> lnk32.cmd */
/*           armcl  <src files...> -z -o <out file> -m <map file> lnk32.cmd */
/*                                                                          */
/*   Description: This file is a sample command file that can be used     */
/*                for linking programs built with the TMS470 C              */
/*                compiler.   Use it as a guideline; you may want to change */
/*                the allocation scheme according to the size of your      */
/*                program and the memory layout of your target system.     */
/*                                                                          */
/*   Notes: (1)   You must specify the directory in which rts32.lib is     */
/*                located.  Either add a "-i<directory>" line to this      */
/*                file, or use the system environment variable C_DIR to    */
/*                specify a search path for libraries.                     */
/*                                                                          */
/*          (2)   If the runtime-support library you are using is not      */
/*                named rts32.lib, be sure to use the correct name here.   */
/****************************************************************************/
-m example1.map
/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
    I_MEM    : org = 0x00000000   len = 0x00000020  /* INTERRUPTS          */
    D_MEM    : org = 0x00000020   len = 0x00010000  /* DATA MEMORY    (RAM) */
    P_MEM    : org = 0x00010020   len = 0x00100000  /* PROGRAM MEMORY (ROM) */
}
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
SECTIONS
{
   secA: load = P_MEM
   secB: load = 0x1000
}
```

您必须创建一个十六进制转换命令文件，以便为 EPROM 编程器生成具有正确地址和格式的十六进制输出。

在图 C-1 中所述的存储器系统中，只有应用程序数据存储在 EPROM 中；数据驻留在由链接器创建的目标文件的 secB 中。默认情况下，十六进制转换实用程序会转换目标文件中出现的所有已初始化段。为了防止转换 secA 中的应用程序代码，必须在十六进制转换命令文件中定义 SECTIONS 指令，以便明确列出要转换的部分。在本例中，secB 必须明确列为要转换的段。

在这种情况下，EPROM 编程器具有以下系统要求：

- EPROM 编程器仅加载完整的 ROM 映像。完整的 ROM 映像中存在连续地址空间（转换后的文件中没有地址空洞），并且该范围中的每个地址都包含一个已知值。创建完整的 ROM 映像需要使用 −image 选项和 ROMS 指令。
  - 使用 −image 选项会使十六进制转换实用程序创建一个输出文件，该文件在指定的存储器范围内具有连续地址，并强制实用程序填充相应地址空间，这些地址空间之前未由输入目标文件中定义的段中的原始数据填充。默认情况下，用于填充存储器范围未使用部分的值为 0。
  - 由于 -image 选项在已知的存储器地址范围内运行，因此需要使用 ROMS 指令来指定 EPROM 的存储器来源和长度。
- 要将数据段烧录到 EPROM 地址 0x0，必须使用 paddr 选项。此值会覆盖由链接器分配的段加载地址。
- 在本例中，EPROM 为 128K × 8 位。因此，EPROM 的存储器地址范围必须为 0x0 至 0x20000。
- 由于 EPROM 存储器宽度为 8 位，因此 memwidth 值必须设置为 8。
- 由于 ROM 器件的物理宽度为 8 位，因此 romwidth 值必须设置为 8。
- 必须使用 Intel 格式。

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY
TI Confidential – NDA Restrictions

由于 memwidth 和 romwidth 具有相同的值，因此只生成一个输出文件（输出文件的数量由 memwidth 与 romwidth 之比决定）。输出文件名使用 -o 选项来指定。

场景 1 的十六进制转换命令文件如示例 C-2 所示。此命令文件使用以下选项来选择系统要求：

选项	说明
-i	创建 Intel 格式
-image	生成存储器映像
-map example1.mxp	生成 example1.mxp 作为转换的映射文件
-o example1.hex	将输出文件命名为 example1.hex
-memwidth 8	将 EPROM 系统存储器宽度设置为 8
-romwidth 8	将物理 ROM 宽度设置为 8

*示例 **C-2. 场景 1** 的十六进制转换命令文件*

```
/* Hex Conversion Command file for Scenario 1             */
a.out                   /* linked object file, input */
-I                      /* Intel format */
-image
-map example1.mxp       /* Generate a map of the conversion */
-o example1.hex         /* Resulting hex output file */
-memwidth 8             /* EPROM memory system width */
-romwidth 8             /* Physical width of ROM */
ROMS
{
   EPROM: origin = 0x0, length = 0x20000
}

SECTIONS
{
   secB: paddr = 0x0    /* Select only section, secB, for conversion */
}
```

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

示例 C-3 展示了生成的映射文件 (example1.mxp) 的内容。图 C-2 展示了生成的十六进制输出文件 (example1.hex) 的内容。十六进制转换实用程序会将数据表 secB 放在地址 0 处，然后用默认填充值 0 填充地址空间的其余部分。有关 Intel MCS-86 目标格式的更多信息，请参阅图 12-9。

*示例 **C-3.** 十六进制映射文件 **example1.mxp** 的内容*

```
**********************************************************
TMS470 Hex Converter               Version x.xx
**********************************************************
Mon Sep 18 15:57:00 1995
INPUT FILE NAME: <a.out>
OUTPUT FORMAT:   Intel
PHYSICAL MEMORY PARAMETERS
   Default data width:    8
   Default memory width:  8
   Default output width:  8
OUTPUT TRANSLATION MAP
---------------------------------------------------------
00000000..0001ffff  Page=0  ROM Width=8  Memory Width=8  "EPROM"
---------------------------------------------------------
   OUTPUT FILES: example1.hex [b0..b7]
   CONTENTS: 00000000..00000007  Data Width=1  secB
             00000007..0001ffff   FILL = 00000000
```

```
:2000000087654321 4321DCBA00000000000000000000000000000000000000000000000096
:200020000000000000000000000000000000000000000000000000000000000000000000C0
:200040000000000000000000000000000000000000000000000000000000000000000000A0
.
.
.
:20FFE0000000000000000000000000000000000000000000000000000000000000000000001
:020000040001F9
:200000000000000000000000000000000000000000000000000000000000000000000000E0
:200020000000000000000000000000000000000000000000000000000000000000000000C0
.
.
.
:20FFC0000000000000000000000000000000000000000000000000000000000000000000021
:20FFE0000000000000000000000000000000000000000000000000000000000000000000001
:00000001FF
```


图 C-2. 十六进制输出文件 **example1.hex** 的内容

Copyright © 2023 Texas Instruments Incorporated

C.2 场景 2 -- 为 16-BIS 代码构建十六进制转换命令文件

场景 2 展示了如何构建十六进制转换命令文件，以便为将驻留在单个 16 位 EPROM 上的应用程序代码和数据生成正确的转换文件。图 C-3 中展示了此场景的 EPROM 存储器系统。在此场景中，TMS470 CPU 在 T 控制位被置位的情况下运行，因此处理器在 16-BIS 模式下执行指令。

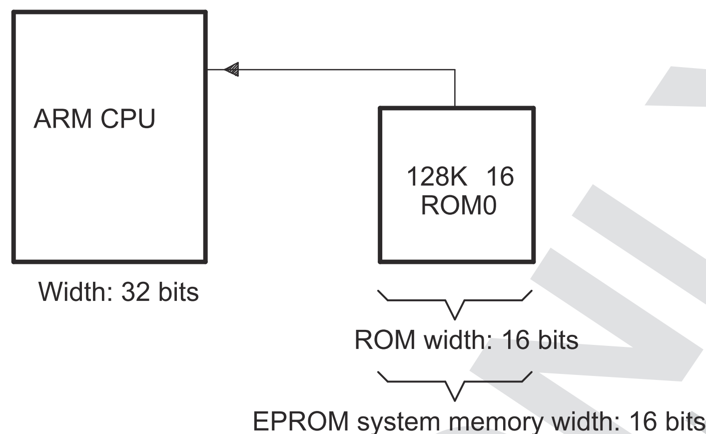


图 C-3. 场景 2 的 EPROM 存储器系统

在此场景中，应用程序代码和数据驻留在 EPROM 上：EPROM 存储器的 64K 低位字专用于应用程序代码空间，而 64K 高位字专用于数据表。应用程序代码从 EPROM 上的地址 0x0 处开始加载，但映射到 TMS470 CPU 的地址 0x3000 处。数据表从 EPROM 上的地址 0x1000 处开始加载，并映射到 TMS470 CPU 地址 0x20 处。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

示例 C-4 展示了解析 EPROM 加载和 TMS470 CPU 访问所需地址的链接器命令文件。

示例 C-4. 场景 2 的链接器命令文件

```
/****************************************************************************/
/* Scenario 2  Link Command                                                 */
/*                                                                          */
/*   Usage:  armlnk <obj files...>    -o <out file> -m <map file> lnk16.cmd */
/*           armcl  <src files...> -z -o <out file> -m <map file> lnk16.cmd */
/*                                                                          */
/*   Description: This file is a sample command file that can be used       */
/*                for linking programs built with the TMS470 C              */
/*                compiler.   Use it as a guideline; you may want to change */
/*                the allocation scheme according to the size of your       */
/*                program and the memory layout of your target system.      */
/*                                                                          */
/*   Notes: (1)   You must specify the directory in which rts16.lib is      */
/*                located.  Either add a "-i<directory>" line to this       */
/*                file, or use the system environment variable C_DIR to     */
/*                specify a search path for libraries.                      */
/*                                                                          */
/*          (2)   If the runtime-support library you are using is not       */
/*                named rts16.lib, be sure to use the correct name here.    */
/****************************************************************************/
-m example2.map
/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
    I_MEM    : org = 0x00000000   len = 0x00000020  /* INTERRUPTS           */
    D_MEM    : org = 0x00000020   len = 0x00010000  /* DATA MEMORY    (RAM) */
    P_MEM    : org = 0x00010020   len = 0x00100000  /* PROGRAM MEMORY (ROM) */
}
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
SECTIONS
{
   secA: load = 0x3000
   secB: load = 0x20
}
```

您必须创建一个十六进制转换命令文件，以便为 EPROM 编程器生成具有正确地址和格式的十六进制输出。在这种情况下，EPROM 编程器具有以下系统要求：

- 由于 EPROM 存储器宽度为 16 位，因此 memwidth 值必须设置为 16。
- 由于 ROM 器件的物理宽度为 16 位，因此 romwidth 值必须设置为 16。
- 必须使用 Intel 格式。

EPROM 编程器不需要 ROM 映像，因此输入十六进制输出文件中的地址不需要连续。

由于 memwidth 和 romwidth 具有相同的值，因此只生成一个输出文件（输出文件的数量由 memwidth 与 romwidth 之比决定）。输出文件名使用 -o 选项来指定。

这里使用 ROMS 指令，因为 paddr 选项用于重定位 secA 和 secB。

场景 2 的十六进制转换命令文件如示例 C-5 所示。此命令文件使用以下选项来选择系统要求：

选项	说明
-i	创建 Intel 格式
-map example2.mxp	生成 example2.mxp 作为转换的映射文件
-o example2.hex	将输出文件命名为 example2.hex
-memwidth 8	将 EPROM 系统存储器宽度设置为 8
-romwidth 8	将物理 ROM 宽度设置为 8

Copyright © 2023 Texas Instruments Incorporated
DRAFT ONLY
TI Confidential – NDA Restrictions

*示例 **C-5.** 场景 **2** 的十六进制转换命令文件*

```
/* Hex Conversion Command file for Scenario 2             */
a.out                   /* linked object file, input */
-I                      /* Intel format */
  /* The following two options are optional */
-map example2.mxp       /* Generate a map of the conversion */
-o example2.hex         /* Resulting Hex Output file */
  /* Specify EPROM system Memory Width and Physical ROM width */
-memwidth 16            /* EPROM memory system width */
-romwidth 16            /* Physical width of ROM */
ROMS
{
   EPROM: origin = 0x0, length = 0x20000
}
SECTIONS
{
   secA: paddr = 0x0
   secB: paddr = 0x1000
}
```

示例 C-6 展示了生成的映射文件 (example2.mxp) 的内容。图 C-4 展示了生成的十六进制输出文件 (example2.hex) 的内容。

*示例 **C-6.** 十六进制映射文件 **example2.mxp** 的内容*

```
*********************************************************
TMS470 Hex Converter               Version x.xx
**********************************************************
Mon Sep 18 19:34:47 1995
INPUT FILE NAME: <a.out>
OUTPUT FORMAT:   Intel
PHYSICAL MEMORY PARAMETERS
   Default data width:    8
   Default memory width:  16
   Default output width:  16
OUTPUT TRANSLATION MAP
---------------------------------------------------------
00000000..0001ffff  Page=0  ROM Width=16  Memory Width=16 "EPROM"
---------------------------------------------------------
   OUTPUT FILES: example2.hex [b0..b15]
   CONTENTS: 00000000..00000003  Data Width=1  secA
             00001000..00001003  Data Width=1  secB
```

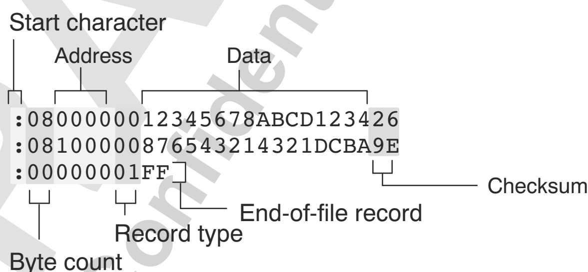


图 **C-4.** 十六进制输出文件 **example2.hex** 的内容

Copyright © 2023 Texas Instruments Incorporated

C.3 场景 3 -- 为两个 8 位 EPROM 构建十六进制转换命令文件

场景 3 展示了如何构建十六进制转换命令文件以转换图 C-5 中所示存储器系统的目标文件。此系统中有两个与 TMS470 目标处理器连接的外部 64K × 16 位 EPROM。应用程序代码和数据将从地址 0x20 开始烧录到 EPROM 上。应用程序代码将首先烧录，然后是数据表。

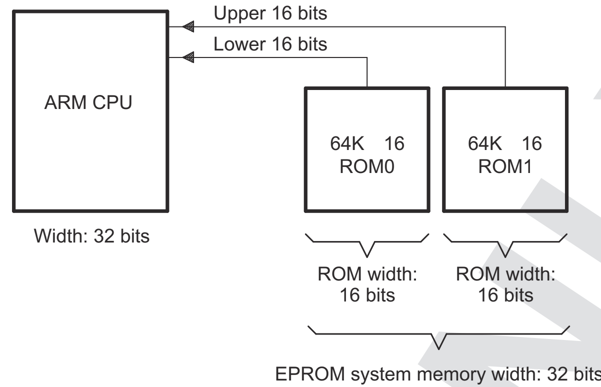


图 C-5. 场景 3 的 EPROM 存储器系统

在此场景中，应用程序代码和数据的 EPROM 加载地址也对应于用于访问代码和数据的 TMS470 CPU 地址。因此，只需要指定加载地址。

示例 C-7 展示了此场景的链接器命令文件。

示例 C-7. 场景 3 的链接器命令文件

```
/****************************************************************************/
/* Scenario 3  Link Command                                                 */
/*                                                                          */
/*   Usage:  armlnk <obj files...>    -o <out file> -m <map file> lnk32.cmd */
/*           armcl  <src files...> -z -o <out file> -m <map file> lnk32.cmd */
/*                                                                          */
/*   Description: This file is a sample command file that can be used       */
/*                for linking programs built with the TMS470 C              */
/*                compiler.   Use it as a guideline; you may want to change */
/*                the allocation scheme according to the size of your       */
/*                program and the memory layout of your target system.      */
/*                                                                          */
/*   Notes: (1)   You must specify the directory in which rts32.lib is      */
/*                located.  Either add a "-i<directory>" line to this       */
/*                file, or use the system environment variable C_DIR to     */
/*                specify a search path for libraries.                      */
/*                                                                          */
/*          (2)   If the runtime-support library you are using is not       */
/*                named rts32.lib, be sure to use the correct name here.    */
/****************************************************************************/
-m example3.map
/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
    I_MEM    : org = 0x00000000   len = 0x00000020  /* INTERRUPTS          */
    D_MEM    : org = 0x00000020   len = 0x00010000  /* DATA MEMORY    (RAM) */
    P_MEM    : org = 0x00010020   len = 0x00100000  /* PROGRAM MEMORY (ROM) */
}
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
SECTIONS
{
   secA: load = 0x20
   secB: load = D_MEM
}
```

您必须创建一个十六进制转换命令文件，以便为 EPROM 编程器生成具有正确地址和格式的十六进制输出。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

在这种情况下，EPROM 编程器具有以下系统要求：

- 在图 C-5 中所述的存储器系统中，EPROM 系统存储器宽度为 32 位，因为每个物理 ROM 都提供 32 位字的 16 位。由于 EPROM 系统存储器宽度为 32 位，因此 memwidth 值必须设置为 32。
- 由于每个物理 ROM 的宽度为 16 位，因此 romwidth 值必须设置为 16。
- 必须使用 Intel 格式。

当 memwidth 为 32 且 romwidth 为 16 时，十六进制转换实用程序会生成两个输出文件（文件数量由 memwidth 与 romwidth 之比决定）。在之前的场景下，输出文件名是使用 -o 选项指定的。指定输出文件名的另一种方法是在 ROMS 指令中使用 files 关键字。当使用 -o 或 files 关键字时，第一个输出文件名始终包含字的低位字节。

场景 3 的十六进制转换命令文件如示例 C-8 所示。此命令文件使用以下选项来选择系统要求：

选项	说明
-i	创建 Intel 格式
-map example3.mxp	生成 example3.mxp 作为转换的映射文件
-memwidth 32	将 EPROM 系统存储器宽度设置为 32
-romwidth 16	将物理 ROM 宽度设置为 16

files 关键字用于在 ROMS 指令中指定输出文件名。

*示例 **C-8.** 场景 **3** 的十六进制转换命令文件*

```
/* Hex Conversion Command file for Scenario 3             */
a.out                   /* linked object file, input */
-I                      /* Intel format */
  /* Optional Commands */
-map example3.mxp       /* Generate a map of the conversion */
  /* Specify EPROM system memory width and physical ROM width */
-memwidth 32            /* EPROM memory system width */
-romwidth 16            /* Physical width of ROM */
ROMS
{
   EPROM: org = 0x0, length = 0x20000
   files={ lower16.bit, upper16.bit }
}
```

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

示例 C-9 展示了生成的映射文件 (example3.mxp) 的内容。

示例 C-9. 十六进制映射文件 example3.mxp 的内容

```
*********************************************************
TMS470 Hex Converter                 Version x.xx
*********************************************************
Tue Sep 19 07:41:28 1995
INPUT FILE NAME: <a.out>
OUTPUT FORMAT:    Intel
PHYSICAL MEMORY PARAMETERS
   Default data width:     8
   Default memory width:  32
   Default output width:  16
OUTPUT TRANSLATION MAP
--------------------------------------------------------
00000000..0001ffff  Page=0  ROM Width=16  Memory Width=32  "EPROM"
--------------------------------------------------------
   OUTPUT FILES: lower16.bit [b0..b15]
                 upper16.bit [b16..b31]
   CONTENTS: 00000020..00000021  Data Width=1  secA
             00000028..00000029  Data Width=1  secB
```

图 C-6 和图 C-7 分别展示了输出文件 lower16.bit 和 upper16.bit 的内容。32 位输出字的低 16 位存储在 lower16.bit 文件中，而高 16 位存储在 upper16.bit 文件中。

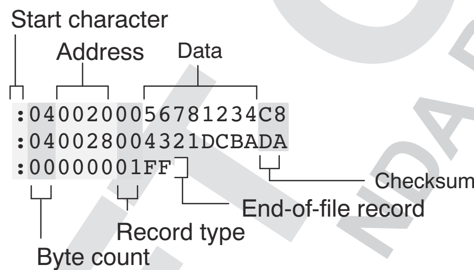


图 **C-6.** 十六进制输出文件 **lower16.bit** 的内容

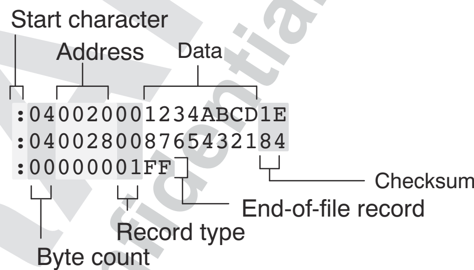


图 **C-7.** 十六进制输出文件 **upper16.bit** 的内容

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

DRAFT ONLY
TI Confidential – NDA Restrictions

This page intentionally left blank.

Copyright © 2023 Texas Instruments Incorporated

术语表

TEXAS INSTRUMENTS

DRAFT ONLY
TI Confidential – NDA Restrictions

D.1 术语

ABI 应用程序二进制接口。

绝对地址[absolute address] 为 ARM 存储器位置永久分配的地址。

绝对常量表达式[absolute constant expression] 不引用任何外部符号、寄存器或存储器的表达式。该表达式的值在汇编时必须是可知的。

绝对列表器[absolute lister] 一种调试工具，用于创建包含绝对地址的汇编器列表。

地址常量表达式[absolute constant expression] 一个符号，其值为地址加上加数，是具有整数值的绝对常量表达式。

对齐[alignment] 链接器将输出段置于 *n* 字节边界（其中 *n* 是 2 的幂）内地址的过程。您可以使用 SECTIONS 链接器指令来指定对齐。

分配[allocation] 链接器计算输出段最终存储器地址的过程。

ANSI 美国国家标准协会；这是一家建立行业自愿遵循的标准的组织。

存档库[archive library] 由归档器将单独文件组合成单个文件的集合。

归档器[archiver] 将多个单独文件集合成一个存档库文件的软件程序。借助归档器，您可以添加、删除、提取或替换存档库内的文件。

ASCII 美国信息交换标准码；一种用于表示和交换字母数字信息的计算机标准代码。

汇编器[assembler] 根据包含汇编语言指令和宏定义的源文件创建机器语言程序的软件程序。汇编器使用绝对操作码替换符号操作码，并使用绝对地址或可重定位地址替换符号地址。

汇编时常量[assembly-time constant] 利用 .set 指令来指定常量值的符号。

大端字节序[big endian] 一种寻址协议，字中的字节从左至右进行编号。字中较高的有效字节存放在低地址处。字节序视硬件而定，并在复位时确定。另请参阅*小端字节序*

绑定[binding] 为输出段或符号指定明确地址的过程。

BIS 位指令集。

块[block] 在大括号内组合在一起并被视为一个实体的一组语句。

Copyright © 2023 Texas Instruments Incorporated

.bss 段[.bss section] 默认的目标文件段之一。使用汇编器 .bss 指令在存储器映射中保留指定量的空间，用于以后存储数据。.bss 段未被初始化。

字节[byte] 根据 ANSI/ISO C，这是可容纳一个字符的最小可寻址单元。

C/C++ 编译器[C/C++ compiler] 将 C 源语句转换成汇编语言源语句的软件程序。

命令文件[command file] 包含链接器或十六进制转换实用程序选项、文件名、指令或命令的文件。

注释[comment] 用于记录或提高源文件可读性的源语句（或源语句的一部分）。不对注释进行编译、汇编或链接；不会影响对象文件。

编译器程序[compiler program] 一种实用工具，可以一步完成编辑、汇编和选择性链接操作。通过编译器（包括解析器、优化器和代码生成器）、汇编器和链接器，编译器可以运行一个或多个源代码模块。

条件处理[conditional processing] 根据对指定表达式的评估，处理一个源代码块或一个源代码替代块的方法。

配置的存储器[configured memory] 链接器指定用于分配的存储器。

常量[constant] 值不能改变的类型。

常量表达式[constant expression] 不以任何方式引用寄存器或存储器的表达式。

交叉参考列表器[cross-reference lister] 一种可生成输出文件的实用程序，其中列出了定义的符号、定义符号的文件、符号的引用类型、定义符号的行、引用符号的行以及符号汇编器和链接器的最终值。交叉参考列表器使用链接的目标文件作为输入。

交叉参考列表[cross-reference listing] 由汇编器创建的输出文件，其中列出了定义的符号、定义符号的行、引用符号的行以及符号的最终值。

.data 段[.data section] 默认的目标文件段之一。.data 段是包含已初始化数据的已初始化段。.data 指令可用于将代码汇编到 .data 段。

指令[directives] 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完全不同）。

DWARF 一种标准化调试数据格式，最初是与 ELF 共同设计的，但它可独立于目标文件格式。

EABI 一种嵌入式应用二进制接口 (ABI)，可为文件格式、数据类型等提供标准。

ELF 可执行连接格式；根据 System V 应用二进制接口规范配置的目标文件系统。

仿真器[emulator] 用于复制 ARM 运行情况的硬件开发系统。

入口点[entry point] 目标存储器中的执行起点。

环境变量[environment variable] 由用户定义并分配给字符串的系统符号。环境变量通常包含在 Windows 批处理文件或 UNIX shell 脚本（例如 .cshrc 或 .profile）中。

DRAFT ONLY TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

收尾程序[epilog]	函数中代码的一部分，用于恢复栈并返回。
可执行模块[executable module]	可在目标系统中执行的已链接目标文件。
表达式[expression]	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[external symbol]	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。
字段[field]	对于 ARM，一种可由软件配置、将长度编程为 1-32 位范围内任意值的数据类型。
全局符号[global symbol]	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。
GROUP	SECTIONS 指令的一个选项，可强制连续分配指定的输出段（作为一个组）。
十六进制转换实用程序[hex conversion utility]	一种可将目标文件转换为一种标准 ASCII 十六进制格式、适合加载到 EPROM 编程器中的实用程序。
高级语言调试[high-level language debugging]	编译器为调试工具保留符号和高级语言信息（如类型和函数定义）的能力。
空洞[hole]	两个输入段之间、会形成无代码输出段的区域。
标识符[identifier]	用作标签、寄存器和符号的名称。
立即操作数[immediate operand]	一个值必须为常量表达式的操作数。
增量链接[incremental linking]	采用多通路的链接文件。增量链接对于大型应用程序十分有用，用户可以对应用程序进行分区，分别链接各个部分，然后将所有部分链接在一起。
加载时初始化[initialization at load time]	链接 C/C++ 代码时由链接器使用的自动初始化方法。在使用 --ram_model 链接选项调用时，链接器会使用此方法。此方法在加载时而不是运行时初始化变量。
已初始化段[initialized section]	目标文件中将链接到可执行模块中的段。
输入段[input section]	目标文件中将链接到可执行模块中的段。
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，负责建立行业自愿遵循的国际标准。
标签[label]	从汇编器源语句第 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 列开始的汇编器语句。
链接器[linker]	一种软件程序，用于组合目标文件来组成可分配到系统存储器并由器件执行的目标模块。
列表文件[listing file]	由汇编器创建的输出文件，其中列出了源语句、源语句的行号以及源语句对段程序计数器 (SPC) 的影响。
字面常量[literal constant]	表示其本身的值。也被称为*字面量*或*直接值*。

DRAFT ONLY
TI Confidential - NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

小端字节序[little endian] 一种寻址协议，字中的字节从右至左进行编号。字中较高的有效字节存放在高地址处。字节序视硬件而定，并在复位时确定。另请参阅*大端字节序*

加载器[loader] 将可执行模块放入系统存储器的器件。

宏[macro] 可用作指令的用户定义例程。

宏调用[macro call] 调用一个宏的过程。

宏定义[macro definition] 定义组成宏的名称和代码的源语句块。

宏扩展[macro expansion] 将源语句插入代码以代替宏调用的过程。

宏库[macro library] 由宏组成的存档库。库中的每个文件必须包含一个宏；其名称必须与其定义的宏名称一致，且必须具有 .asm 扩展名。

映射文件[map file] 由链接器创建的输出文件，显示存储器配置、段组成、段分配、符号定义，以及为用户程序定义符号的地址。

成员[member] 结构体、联合体、存档或枚举的元素或变量。

存储器映射[memory map] 被划分为功能块的目标系统存储器空间的映射。

存储器引用操作数[memory reference operand] 使用特定目标语法引用存储器中某个位置的操作数。

助记符[mnemonic] 一个可让汇编器转换为机器代码的指令名。

模型语句[model statement] 每次调用宏时，在宏定义中进行汇编的指令或汇编器指令。

命名段[named section] 由 .sect 指令定义的已初始化段。

目标文件[object file] 包含机器语言目标代码的汇编或链接文件。

对象库[object library] 由单独的目标文件组成的存档库。

目标模块[object module] 可在目标系统上下载和执行的可执行链接目标文件。

操作数[operand] 汇编语言指令、汇编器指令或宏指令的参数，为由指令执行的操作提供信息。

优化器[optimizer] 可提高执行速度并减小 C 程序大小的软件工具。

选项[options] 在调用软件工具时使用户能够请求其他或特定函数的命令行参数。

输出模块[output module] 在目标系统上下载和执行的已链接可执行对象文件。

输出段[output section] 可执行的已链接模块中的最终分配段。

局部链接[partial linking] 采用多通路的链接文件。增量链接对于大型应用程序很有用，因为可以对应用程序分区，分别链接各个部分，然后将所有部分链接在一起。

无声运行[quiet run] 用于抑制正常横幅和进度信息的选项。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

原始数据[raw data] 输出段中的可执行代码或初始化数据。

寄存器操作数[register operand] 一个表示 CPU 寄存器的特殊预定义符号。

可重定位的常量表达式[relocatable constant expression] 至少引用一个外部符号、寄存器或存储器位置的表达式。该表达式的值直到链接时才能知道。

重定位[relocation] 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。

ROM 宽度[ROM width] 每个输出文件的宽度（以位为单位），更具体地说，是十六进制转换实用程序文件中单个数据值的宽度。ROM 宽度决定实用程序如何将数据分入输出文件。目标字映射到存储器字后，存储器字将分解为一个或多个输出文件。输出文件的数目由 ROM 宽度决定。

运行地址[run address] 段运行的地址。

运行时支持库[run-time-support library] 包含运行时支持函数源代码的库文件 rts.src。

段[section] 一个可重定址的代码块或数据块，最终将与存储器映射中的其他段接续。

段程序计数器 (SPC) [section program counter (SPC)] 可持续跟踪段中当前位置的元素；每个段都自带 SPC。

符号扩展[sign extend] 用值的符号位填充值的未使用 MSB 的过程。

模拟器[simulator] 用于仿真 ARM 运行情况的软件开发系统。

源文件[source file] 包含 C/C++ 代码或汇编语言代码的文件，这些代码经编译或汇编后可形成目标文件。

静态变量[static variable] 范围局限在一个函数或程序内的一种变量。当函数或程序退出时，静态变量的值不会被丢弃；当重新输入函数或程序时，将恢复其之前的值。

存储类[storage class] 符号表中指示如何访问符号的条目。

字符串表[string table] 存储长度超过八个字符的符号名称的表（长度为八个字符或更长的符号名称不能存储在符号表中，而是存储在字符串表中）。指向字符串表中字符串位置的符号进入点的名称部分。

结构[structure] 拥有统一名称的一个或者多个变量的集合。

子段[subsection] 一个可重定址的代码块或数据块，最终将占用存储器映射中的连续空间。子段为较大段中的较小段，子段使用户能够更严格地控制存储器映射。

符号[symbol] 表示一个地址或值的名称。

符号常量[symbolic constant] 值为绝对常量表达式的符号。

符号调试[symbolic debugging] 软件工具保留可供仿真器或模拟器等调试工具使用的符号信息的能力。

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

标签[tag]	可分配给结构体、联合体或枚举的可选*类型*名称。
目标存储器[target memory]	系统中加载可执行目标代码的物理存储器。
.text 段[.text section]	默认的目标文件段之一。.text 段被初始化并包含可执行代码。可以使用 .text 指令将代码汇编到 .text 段中。
未配置的存储器[unconfigured memory]	未定义为存储器映射的一部分，且无法加载代码或数据的存储器。
未初始化段[uninitialized section]	在存储器映射中保留空间但不包含实际内容的目标文件段。这些段由 .bss 和 .usect 指令创建。
UNION	能使链接器将同一地址分配给多个段的 SECTIONS 指令选项。
联合体[union]	可以保存不同类型和大小的对象的变量。
无符号值[unsigned value]	无论实际符号是什么都会当作非负数的值。
变量[variable]	表示可假设为任何一组值的数量的符号。
胶合代码[veneer]	在需要状态变化的情况下作为例程的备用进入点的指令序列。
定义明确的表达式[well-defined expression]	仅包含符号或汇编时常量的一个术语或一组术语，提前定义后才能出现在表达式中。
字[word]	目标存储器中的 32 位可寻址位置

DRAFT ONLY
TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

修订历史记录

Changes from MARCH 11, 2020 to MARCH 31, 2023 (from Revision Y (March 2020) to Revision Z (March 2023)) **Page**

- 更新了整个文档中的表格、图和交叉参考的编号格式。 11
- 删除了整个文档中对处理器 Wiki 的引用。 11
- 更正了 --asm_cross_reference_listing 选项的名称........ 66
- --strict_compatibility 链接器选项不再起任何作用，已从文档中删除。 178
- 记录了 --absolute_exe 和 --relocatable 选项可能无法一同使用。 180
- 更正链接器的头文件搜索路径的说明。 183
- 删除了 --symbol_map 选项示例中的引号。 195
- 阐明了链接器定义的符号的使用，包括何时以及如何使用 _symval() 运算符........ 233
- 记录了与映像加载相关的其他十六进制转换实用程序选项。 280
- 记录了与映像加载相关的其他十六进制转换实用程序选项。 290
- 说明使用十六进制转换实用程序时，来自多个输入映像的段不得重叠。 291
- 更正了 ASCII 十六进制输出格式地址大小和默认宽度........ 303

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。

添加内容的版本	章节	位置	添加/修改/删除
SPNU118Y	程序加载	节 3.3.2.3	更正了有关 RAM 和 ROM 型号使用 CINIT 进行初始化的信息。
SPNU118Y	链接器	节 8.4.27	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。
SPNU118Y	链接器	节 8.4.37	阐明了只有在使用 --rom_model 链接器选项时，才发生零初始化，使用 --ram_model 选项时则不发生。
SPNU118Y	链接器	节 8.5.4.2、节 8.5.10.7 和节 8.5.10.8	添加了使用相关存储器区域中上一个已分配字节的运行时地址来定义符号的 LAST 操作符。
SPNU118Y	十六进制转换实用程序	节 12.2.1	现已支持十六进制转换实用程序的二进制输出格式。
SPNU118Y	十六进制转换实用程序	节 12.12	现在可以将引导表与十六进制转换实用程序的安全闪存启动流程 ((--cmac) 功能搭配使用。
SPNU118Y	十六进制转换实用程序	节 12.15.6	提供示例，展示 ROMS 指令语法中 8 位存储器宽度与 16 位存储器宽度的效果对比。
SPNU118X		-- 全文 --	更改了由编译器创建的目标文件的默认文件扩展名，以防止在 C 和 C++ 文件具有相同名称时发生冲突。从 C 源文件生成的目标文件具有 .c.obj 扩展名。从 C++ 源文件生成的目标文件具有 .cpp.obj 扩展名。从汇编源文件生成的目标文件仍然具有 .obj 扩展名。
SPNU118X	十六进制转换实用程序	节 12.12	添加了对 TMS320F2838x 器件的安全闪存启动功能的支持。
SPNU118W	目标模块	节 2.6	为清楚起见，修改了有关符号类型的信息。
SPNU118W	汇编器指令	.bits 主题	修改了关于 .bits 指令的说明。

Copyright © 2023 Texas Instruments Incorporated

添加内容的版本	章节	位置	添加/修改/删除
SPNU118W	汇编器指令	.symdepend 主题、.weak 主题	拆分了 .symdepend 和 .weak 指令主题。
SPNU118W	链接器	节 8.4	添加了 --emit_references:file 链接器选项。
SPNU118V	链接器	节 8.4、节 8.4.12 和节 8.5.9	添加了 --ecc=on 链接器选项，支持生成 ECC。请注意，ECC 生成功能现在默认关闭。
SPNU118V	链接器	节 8.5.7.3	添加了链接器语法，将经过初始化的段与未经初始化的段合并。
SPNU118V	链接器	节 8.5.10.4	删除了链接器为 COFF 定义的全局符号列表，因为已不再使用 COFF。
SPNU118V	目标文件实用程序	章节 11	添加了 objcopy、objdump、readelf 和 size 实用程序。
SPNU118U	十六进制转换实用程序	节 12.2.1 和节 12.10	添加了 --array 选项，可生成阵列输出格式。
SPNU118R	链接器说明	节 8.9	提供了指向 E2E 博客文章的链接，通过示例展示如何使用由链接器生成的 CRC 表来执行循环冗余校验。
SPNU118R	链接器说明	节 8.11.2	可定义 _AEABI_PORTABILITY_LEVEL，以便在包含头文件时，全面移植目标文件。
SPNU118Q	链接器说明	节 8.5.9	记录了 ECC 指令的修改后行为。
SPNU118P	链接器说明	节 8.4	弃用、删除或重命名了几个链接器选项。链接器继续接受一些已被弃用的选项，但不建议使用它们。
SPNU118P	链接器说明	节 8.4.6	--cinit_compression 和 --copy_compression 的默认值已从 RLE 更改为 LZSS。
SPNU118O	链接器说明	节 8.5.3	增加了有关从链接器命令文件访问文件和库的信息。
SPNU118O	链接器说明	节 8.9.1.1	扩展了可用 CRC 算法的列表。
SPNU118O	目标文件实用程序	节 11.1	目标文件显示实用程序增加了 – cg 选项，能够以 XML 格式显示函数栈的使用情况和被调用函数信息。
SPNU118N	目标模块	节 2.1	不再支持 COFF 目标文件格式。ARM 代码生成工具目前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。删除或简化了本文档中提及 COFF 格式的各章节。如果希望生成 COFF 输出文件，请使用 5.2 版本的 ARM 代码生成工具，并参考 SPNU118M 文档。 弃用了 .clink 指令和 --no_sym_merge 链接器选项。
SPNU118N	目标模块、指令和链接器	节 2.6.3、.weak 主题和 节 8.6.5	可以使用汇编或链接器命令文件来声明弱符号。如果解析引用不需要弱符号，则链接器会从输出文件中将其删除。
SPNU118N	链接器	节 8.5.4.4	添加了 ALIAS 语句。
SPNU118N	链接器	节 8.4.21	为 --mapfile_contents 链接器选项添加了作为筛选器的模块。
SPNU118N	链接器	节 8.5.5.2.1	添加了在 RAM 中放置函数的示例。
SPNU118M	目标模块	节 2.4.4	添加了有关当前段以及指令如何与其交互的信息。
SPNU118M	目标模块	节 2.6 和节 2.6.4	添加了有关各种类型的符号以及符号表的信息。
SPNU118M	汇编器说明	节 4.8.6	为 Cortex-M4 添加了 __TI_ ARM_V7M4__ 预定义宏名称。
SPNU118M	汇编器说明	节 4.10.1	内置函数使用前缀 $$。
SPNU118M	链接器	节 8.4.2、节 8.5.10.7 和节 8.6	添加了有关引用链接器符号的信息。
SPNU118M	链接器	节 8.4.11	添加了链接器的预定义宏列表。
SPNU118M	链接器	节 8.5.5.1	删除了加载和填充属性的无效语法。
SPNU118M	链接器	节 8.11.5	添加了 --cinit_hold_wdt 链接器选项。

DRAFT ONLY TI Confidential – NDA Restrictions

Copyright © 2023 Texas Instruments Incorporated

重要声明和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

Copyright © 2023，德州仪器 (TI) 公司