RealView™ 编译工具

2.0 版

开发者指南

ARM®

ARM DUI 0203BSC

RealView 编译工具
开发者指南

发行版信息

已对本书做出了下列更改。

更改历史

日期	问题	更改
2002 年 8 月	A	发行版 1.2
2003 年 1 月	B	发行版 2.0

所有权声明

机密性状态

本文档是供开放访问的。对其没有传播限制。

产品状态

本文档中的信息是最终信息 （即为已开发好的产品提供的信息）。

网址

http://www.arm.com

目录
RealView 编译工具汇编程序指南

序言

本序言介绍 *RealView 编译工具 2.0 版开发者指南*。其中包含下列各部分：

- 第 vi 页的*关于本书*;
- 第 ix 页的*反馈*。

关于本书

此手册为编写以 ARM 系列处理器为目标的代码提供指导信息。

适合的读者

本手册适用于所有编写 ARM 代码的开发人员。它假设您是有经验的软件开发人员，并且熟悉 *RealView Compilation Tools v2.0 Getting Started Guide* 中描述的 ARM 开发工具。

使用本书

本书按下列各章组织：

第 1 章*简介*

阅读此章，了解 *RealView 编译工具* 的简介 (RVCT)。

第 2 章*嵌入式软件开发*

阅读此章，了解如何用 RVCT 开发嵌入式应用程序的详细信息。它描述与目标系统无关的默认 RVCT 行为，以及如何调整 C 库和映像内存映射以适应您的目标系统。

第 3 章*使用过程调用标准*

阅读此章，了解如何使用 *ARM-Thumb 过程调用标准*。使用此标准可以更方便地确保分别编译和汇编的模块可以协同工作。

第 4 章*ARM 和 Thumb 交互操作*

阅读此章，了解编写实现 Thumb ® 指令集的处理器代码时，如何在 ARM 状态和 Thumb 状态之间切换的详细信息。

第 5 章*混合使用 C、 C++ 和汇编语言*

阅读此章，了解如何编写 C、 C++ 和 ARM 汇编语言混合代码的详细信息。它还描述如何从 C 和 C++ 使用 ARM 内联和嵌入式汇编程序。

第 6 章*处理处理器异常*

阅读此章，了解如何处理 ARM 处理器支持的各种类型异常的详细信息。

第 7 章*调试通信通道*

阅读此章，了解如何使用*调试通信通道* (DCC) 的描述。

印刷约定

本书中使用了下列印刷约定：

等宽字体　表示可在键盘上输入的文本，例如命令、文件和程序名称及源代码。

等宽字体　表示命令或选项所允许的缩写词。可以输入带下划线文本来代替整个命令或选项名称。

等宽斜体

表示命令和函数的自变量，函数中的自变量将由指定的值代替。

等宽粗体

表示在示例代码外使用的语言关键字。

斜体　突出显示重要的备注，介绍特殊术语，表示内部交叉引用，以及引用。

粗体　突出显示界面元素（如菜单名称）。有时候也用于在描述性的列表中进行强调，以及表示 ARM 处理器信号名称。

更深入的读物

本节列出来自 ARM Limited 和第三方的出版物，其中提供了为 ARM 系列处理器开发代码的附加信息。

ARM 定期对其文档进行更新和修正。有关当前勘误表和增补以及 ARM 常见问题的信息，请访问 `http://www.arm.com`。

ARM 出版物

本手册包含开发 ARM 系列处理器应用程序的一般信息。有关其它组件的信息，请参阅以下 RVCT 文档集中的手册。

- *RealView 编译工具 2.0 版入门指南* (ARM DUI 0202)；
- *RealView 编译工具 2.0 版汇编程序指南* (ARM DUI 0204)；
- *RealView 编译工具 2.0 版编译程序和库指南* (ARM DUI 0205)；
- *RealView 编译工具 2.0 版链接程序和实用程序指南* (ARM DUI 0206)。

随 RealView 编译工具 提供了下列附加文档：

- *ARM FLEXlm 许可证管理指南* (ARM DUI 0209)。此指南以 DynaText 和 PDF 格式提供。
- *ARM 体系结构参考手册* (ARM DDI 0100)。该手册作为联机图书的组成部分以 DynaText 格式提供，并且以 PDF 文件 DDI0100E_ARM_ARM.pdf 形式提供，该文件位于 *install_directory*\Documentation\ARMARM\5.0\release\windows\PDF中。
- *ARM ELF 规范* (SWS ESPC 0003)。这是以 PDF 文件格式提供的，文件名为 ARMELF.pdf，位于 *install_directory*\Documentation\Specifications\1.0\release*platform*\PDF 下。
- *TIS DWARF 2 规范*。这是以 PDF 文件格式提供的，文件名为 TIS-DWARF2.pdf ，位于 *install_directory*\Documentation\Specifications\1.0\release*platform*\PDF 下。
- *ARM-Thumb 过程调用标准规范*。这是以 PDF 文件格式提供的，文件名为 ATPCS.pdf，位于 *install_directory*\Documentation\Specifications\1.0\release*platform*\PDF 下。

此外，可参考下列文档来了解与 ARM 产品相关的特定信息：

- *RealView ARMulator ISS v1.3 用户指南* (ARM DUI 0207)；
- *Multi-ICE 2.2 版用户指南* (ARM DUI 0048)，或更高版本；
- *ARM 参考外设规范* (ARM DDI 0062)；
- 您的硬件设备的 ARM 数据表或技术参考手册。

其它出版物

下面这本书提供了关于 ARM 体系结构的一般性信息：

- *ARM System-on-chip Architecture* (second edition), Furber, S., (2000). Addison Wesley. ISBN 0-201-67519-6.

反馈

ARM Limited 欢迎用户为 RealView 编译工具 及其文档提供反馈意见。

对 **RealView** 编译工具的反馈信息

如果您有关于 RealView 编译工具的任何问题，请与您的供货商联系。为了帮助它们提供快速而有用的响应，请提供：

- 您的姓名和公司；
- 产品的序列号；
- 您正在使用的版本的详细信息；
- 您正在运行的平台的详细信息，例如硬件平台、操作系统类型和版本；
- 能复现问题的一小段独立示例代码；
- 清晰地解释您期望发生的和实际发生的事件；
- 您使用的命令，包括任何命令行选项；
- 阐明问题的输出样例；
- 工具版本字符串，包括版本号和日期。

对本书的反馈

如果您注意到本书中有任何错误和遗漏，请发送电子邮件到 `errata@arm.com`，并提供：

- 文档标题；
- 文档编号；
- 您的注释所适用的页码；
- 问题的简要说明。

也欢迎对增补和改进提出一般性建议。

第 1 章
简介

本章介绍了 *RealView™ 编译工具* (RVCT)。其中包含下列各部分：

- 第 1-2 页的*关于 RVCT 开发者指南*;
- 第 1-5 页的*通用程序设计问题*;
- 第 1-10 页的*为 ARM 处理器开发代码*。

备注

本文档中给出的示例中编译程序、汇编程序和链接程序的选项使用了单短线。在 RVCT v2.0 中，可以使用双短线 -- 指定命令行关键字，比如 `--partial`。但是，编译程序中的单字母选项，如 `-E`，则没有等价的双短线选项。

为了向后兼容，仍支持以前版本的 ADS 和 RVCT 中使用的单短线命令行选项。

1.1 关于 RVCT 开发者指南

本书包括开发 ARM 系列 RISC 系列处理器代码的特定问题的帮助信息。通常，本书的各章都假设您使用 *RealView 编译工具* (RVCT) 来开发代码。

RVCT 由一套工具连同支持文档和示例组成，允许您为 ARM 系列 RISC 处理器编写和编译应用程序。可以使用 RVCT 来编译 C、C++ 或 ARM 汇编语言程序。

RVCT 工具箱包含以下主要组件：

- 命令行开发工具；
- 实用程序；
- 支持软件。

有关 RVCT 文件的列表，请参阅第 vii 页的*更深入的读物*。

1.1.1 示例代码

本书中很多示例的代码都在下面的路径中：

install_directory\RVCT\Examples\2.0\release*platform*

其中 *install_directory* 为 RVCT v2.0 的安装路径，*platform* 是 windows 或 unix。为更明了，在本文档的其它地方将这个路径称为 *Examples_directory*。

此外，包含此路径下还包含本书未介绍的示例代码。请参阅每个示例目录中的 readme.txt 文件获得更多信息。示例安装在以下几个子目录中：

asm　该目录包含了 ARM 汇编语言程序设计的几个示例。*RealView 编译工具 2.0 版汇编程序指南*中使用了这些示例。

cpp　该目录包含了几个简单的 C++ 示例。此外，rw 子目录包含了 Rogue Wave 手册和教学示例。

databort　该目录包含了标准数据中断处理程序的设计文档和示例代码。

dcc　该目录包含了说明如何使用 Debug Communications Channel 的示例代码。该示例将在第 7 章*调试通信通道*中介绍。

`cached_dhry` 该目录包含了初始化缓存和 TCM 例程的示例，它位于 Dhrystone 示例附近。

`dhry` 该目录包含了 Dhrystone 源代码。

`dhryansi` 该目录包含了 Dhrystone 的 ANSI C 版本。

`emb_sw_dev` 该目录包含了第 2 章 *嵌入式软件开发* 所参考的示例项目。包括以下子目录：

- `build`*n* 批处理和 make 文件以编译示例项目。
- `dhry` Dhrystone 基准程序的源文件 该程序为每个 `build`*n* 目录中的示例项目提供了代码基础。
- `source` 编译示例项目所需要的所有其它源文件。
- `include` 用户定义的头文件。
- `scatter` 用于构建示例项目的分散文件。

`embedded` 该目录包含了说明如何为 ROM 编写代码的源代码示例。这些示例适用于 ARM Integrator™ 电路板的应用情形。

`explasm` 该目录包含了额外的 ARM 汇编语言示例。

`fft_5te` ARM 体系结构 v5TE 的快速傅里叶变换优化源代码。

`inline` 该目录包含了说明在编译 ARM C 和 C++ 代码时如何使用内联汇编程序的示例。这些示例将在第 5 章 *混合使用 C、 C++ 和汇编语言* 中介绍。

`interwork` 该目录包含了说明如何在 ARM 代码和 Thumb 代码之间进行交互的示例。这些示例将在第 4 章 *ARM 和 Thumb 交互操作* 中介绍。

`mmugen` 该目录包含了 MMUgen 实用程序的源代码和文档。这个实用工具可从描述虚拟至物理地址所需要的转换规则文件中生成 MMU 页表数据。

`picpid` 该目录包含了说明如何编写位置无关代码的一个示例。有关详细说明，请参阅 `readme.txt`。

`sorts` 该目录包含了比较 ARM C 库中所使用的插入排序法、希尔排序法和快速排序法的示例代码。

`swi` 该目录包含了 SWI 处理程序的一个示例。

`unicode` 该项目录包含了能够检测 RVCT v2.0 支持的多字节字符的示例代码。

`vfpsupport` 该目录中包含了赋予并实现*向量浮点* (VFP) 运算的示例代码。并包含了使用 VFP 时配置调试系统的各种实用工具文件。

1.2 通用程序设计问题

ARM 系列处理器是 RISC 处理器。很多给出有效代码的程序设计策略都源于 RISC 处理器。和很多 RISC 处理器一样，ARM 系列处理器被设计用来存取对齐数据，即存储“字”的地址为四的倍数，“半字”时为二的倍数。该数据按其自然尺寸边界定位。

ARM 编译程序通常将全局变量对齐到这些自然尺寸边界上，以便通过使用 LDR 和 STR 指令有效地存取这些变量。

这与多数 CISC 体系结构不同，其使用指令直接存取未对齐的数据。因而，在从 CISC 体系结构向 ARM 处理器移植遗留代码是必须予以注意。存取未对齐数据在代码尺寸或性能都不利。

下面的各节详细地探讨了程序设计问题：

- *有效地使用 ARM-Thumb 过程调用标准*；
- *未对齐指针*；
- 第 1-6 页的*结构中的未对齐字段*；
- 第 1-8 页的*用于半字存取的非对齐 LDR*；
- 第 1-9 页的*移植代码并检测非对齐存取*。

1.2.1 有效地使用 ARM-Thumb 过程调用标准

在 *ARM-Thumb 过程调用标准*下 ARM 编译程序将头四个整数大小的函数参数传递至寄存器 r0 到 r3。其它参数传递到栈中。这意味着：

- 大量的参数通过引用的方法传递更为有效，比如 结构，
- 如有可能，函数限制使用四个或更少的整数大小的参数将更为有效。

1.2.2 未对齐指针

ARM 编译程序期望正常的 C 指针指向存储器中对齐的字，因为这可使编译程序生成更有效的代码。

比如，如果指针 `int *` 被用来读取一个字，ARM 编译程序使用生成代码中的一条 LDR 指令。当地址为四的倍数 （即在一个字的边界）即能正确读取。但是，如果该地址不是四的倍数，那么，一条 LDR 指令返回一个循环移位结果，而不是执行真正的未对齐字载入。循环移位结果取决于系统的偏移量和端序。

例如，如果代码从指针指向的地址 0x8006 载入数据，则期望从 0x8006、0x8007、0x8008 和 0x8009 载入字节的内容。但是，在 ARM 处理器上，这个存取载入了 0x8004、0x8005、0x8006 和 0x8007 字节的循环移位内容。

因而，如果想将指针定义到一个可在任何地址（即非自然对齐）的字中，那么在定义该指针时，必须使用 __packed 限定符来定义指针：

```
__packed int *pi; // pointer to unaligned int
```

然后，ARM 编译程序不使用 LDR，而是生成正确存取该值的代码，而不必考虑指针对齐状况。所生成的代码是字节存取的一个序列，或者取决于编译选项、跟变量对齐相关的移位和屏蔽。这会导致性能和代码大小的损失。

备注

千万不要用 __packed 来存取存储器映射的外围寄存器，因为 ARM 编译程序可使用多个存储器存取来重新获取数据。因而，可对附近的位置进行存取，它们可能对应于其它外部寄存器。当使用了位字段时， ARM 编译程序当前存取整个部分，而不只是所指定的字段。

1.2.3 结构中的未对齐字段

与全局变量位于其自然尺寸边界相同，结构中的字段也如此。就是说编译程序经常要在字段间插入填充字节来确保字段对齐。当编译程序插入填充字节时，使用选项 -W+s 编译产生一个警告。

有时，可能不希望这样的情况发生。可使用 __packed 限定符 来创建字段之间没有填充字节的结构，且这些结构需要非对齐存取。

如果 ARM 编译程序知道特定结构的对齐，它可以确定所存取的字段是否在一个充填结构中是对齐的。在这些情况下，编译程序尽可能地采用更有效的对齐字或半字存取。否则，编译程序使用多个对齐存储器存取（LDR、STR、LDM 和 STM）与固定移位和屏蔽相结合来存取存储器中正确的字节。

对非对齐元素的存取是通过内联还是通过调用一个函数来完成，由编译程序 -Ospace（默认，调用一个函数）和 -Otime（执行非对齐存取内联）选项来控制。

例如：

1. 创建一个文件 foo.c，它包括：

```
__packed struct mystruct {
    int aligned_i;
    short aligned_s;
    int unaligned_i;
};
struct mystruct S1;

int foo (int a, short b)
{
    S1.aligned_i=a;
    S1.aligned_s=b;
    return S1.unaligned_i;
}
```

2. 使用 armcc -c -Otime foo.c 编译。所生成的代码为：

```
MOV      r2,r0
LDR      r0,|L1.84|
MOV      r12,r2,LSR #8
STRB     r2,[r0,#0]
STRB     r12,[r0,#1]
MOV      r12,r2,LSR #16
STRB     r12,[r0,#2]
MOV      r12,r2,LSR #24
STRB     r12,[r0,#3]
MOV      r12,r1,LSR #8
STRB     r1,[r0,#4]
STRB     r12,[r0,#5]
ADD      r0,r0,#6
BIC      r3,r0,#3
AND      r0,r0,#3
LDMIA    r3,{r3,r12}
MOV      r0,r0,LSL #3
MOV      r3,r3,LSR r0
RSB      r0,r0,#0x20
ORR      r0,r3,r12,LSL r0
BX lr
```

然而，可以给编译程序更多的信息，使其能知道哪个字段是对齐的，哪个字段不是。为此，必须将未对齐字段声明为 __packed，并从 struct 本身除去 __packed 属性。这是推荐的方法，并且是保证 struct 中自然对齐成员快速存取的唯一方法。而且，哪个字段是未对齐的也更清楚，但在 struct 中增加或删除字段时需要小心。

3. 现在，修改 foo.c 中的结构定义为：

```
struct mystruct {
    int aligned_i;
    short aligned_s;
    __packed int unaligned_i;
};
struct mystruct S1;
```

4. 编译 foo.c 和下列代码，生成了更有效的代码：

```
MOV      r2,r0
LDR      r0,|L1.32|
STR      r2,[r0,#0]
STRH     r1,[r0,#4]
LDMIB    r0,{r3,r12}
MOV      r0,r3,LSR #16
ORR      r0,r0,r12,LSL #16
BX lr
```

同一原理适应于联合。使用在存储器中未对齐的联合组件的 __packed 属性。

备注

任何通过指针存取的 __packed 对象都有未知的对齐，即使是充填结构。

1.2.4 用于半字存取的非对齐 LDR

有些情况下， ARM 编译程序可特意生成非对齐 LDR 指令。特别是编译程序从存储器中载入半字时将使用该方法。这是因为，通过使用相应地址，所需的半字可以载入到寄存器顶部的一半，然后向底部移位一半。这只需要一个存储器的存取，而使用 LDRB 指令做同样的操作需要两个存储器的存取，还需要将这两个字节合并在一起的指令。在 ARM 体系结构 v3 和其早期版本中，通常使用该方法进行所有半字载入。在 ARMv4 和其以后版本中，就不经常这么做了，因为有了专门的半字载入指令，但是，非对齐仍可能会生成 LDR 指令。比如在一个充填结构中存取一个非对齐 short。

备注

如果使用 --memaccess +L41 编译程序选项使其可用，这样的非对齐 LDR 指令仅由 RVCT 编译程序生成。

1.2.5 移植代码并检测非对齐存取

其它体系结构（如 x86 CISC）所遗留的代码，可能会使用在 ARM 处理器上不起作用的指针执行对非对齐数据的存取。这是不可移植的代码，在期望对齐数据的 RISC 体系结构中，必须识别并更改此类存取才能使其发生正确存取数据。

识别非对齐存取可能会很困难，因为使用非对齐地址进行的载入或存储操作产生不正确的动作。追踪到底是哪部分的 C 源程序造成了这个问题是很困难的。

具有完整*存储器管理单元* (MMUs) 的 ARM 处理器，例如，ARM920T™，支持可选的对齐检测，处理器检测每一次的存取以确保其被正确地对齐。如果出现不正确的对齐存取， MMU 产生数据中断。

对于 ARM7TDMI® 等简单的内核，建议在 ASIC/ASSP 内部实现对齐检测。可以使用相对 ARM 内核是额外的外部硬件体来完成这一点，由其监控每次数据的存取规模和存取地址总线的最低有效位。在非对齐存取的情况下，可以通过配置 ASIC/ASSP 来产生**中断**信号。 ARM Limited 建议从其它体系结构移植代码的 ASIC/ASSP 设备中要包含这样的逻辑。

如果将系统配置为非对齐存取进行中断，则必须安装数据中断异常处理程序。出现非对齐存取时，进入数据中断处理程序，并由此识别位于 (r14-8) 的出错数据存取指令。

一旦识别出，必须通过改变 C 源程序来修复数据的存取。使用下列指令可有条件地完成修复：

```
#ifdef __arm
  #define  PACKED  __packed
#else
  #define  PACKED
#endif
:
  PACKED int *pi;
:
```

由于代码大小和性能上的开销，最好尽可能少采用存取非对齐数据。

1.3 为 ARM 处理器开发代码

本书提供了部分最常用的 ARM 程序设计任务的信息和示例代码。下面的章节总结了每章的主要内容。

- *嵌入式软件开发*;
- 第 1-11 页的*使用过程调用标准*;
- 第 1-11 页的 *ARM 与 Thumb 代码交互*;
- 第 1-12 页的*混合使用 C、 C++ 和汇编语言*;
- 第 1-12 页的*处理处理器异常*。

1.3.1 嵌入式软件开发

很多为基于 ARM 系统编写的应用程序都是嵌入式应用程序，它们存储在 ROM 中并在复位时执行。编写嵌入式操作系统，或编写在复位时执行的嵌入式应用程序 （无操作系统）时要考虑很多因素，包括：

- 地址重映射，如初始化使 ROM 的地址位于 0x0000，然后将 RAM 重映射到地址 0x0000 ；
- 初始化环境和应用程序；
- 链接一个可执行的嵌入映象将代码和数据放置在存储器的特定位置。

复位时， ARM 内核通常从地址 0x0000 开始执行指令。对于嵌入式系统，这意味着系统复位时， ROM 一定是在地址 0x0000 处。但是， ROM 一般要比 RAM 慢，且通常只有 8 或 16 位宽。由此影响了异常处理的速度。令 ROM 处在地址 0x0000 意味着异常向量不能被更改。通常的策略是在启动后将 ROM 重映射到 RAM 并将异常向量从 ROM 复制到 RAM。有关详细信息请参阅第 2-25 页的 *ROM/RAM 重新映射*。

复位后，嵌入式应用程序或操作系统必须对系统进行初始化，包括：

- 初始化执行环境，如异常向量、栈和 I/O 外设；
- 初始化应用程序，比如将非零可写数据的初始值复制到可写数据区并将 ZI 数据区清零。

有关详细信息请参阅第 2-23 页的*初始化序列*。

嵌入式系统通常采用复杂的存储器配置。例如，嵌入式系统可能会将高速 32 位 RAM 用于追求性能的代码（如中断处理程序和栈），将较慢的 16 位 RAM 用于应用程序 RW 数据，将 ROM 用于常规应用程序代码。可以使用链接程序的分散载入机制构建适合复杂系统的可执行映象。例如，分散载入描述文件可指定单独的代码和数据区的载入地址和执行地址。有关影响嵌入式应用程序（如半主机）相关主题的信息和一系列已处理过的示例，请参阅第 2 章*嵌入式软件开发*。

1.3.2 使用过程调用标准

为确保使分别编译和汇编的模块能够协同工作，必须遵循定义寄存器用法和栈约定的 *ARM-Thumb 过程调用标准* (ATPCS)。该基本标准中有很多变量。ARM 编译程序总是生成与所选 ATPCS 变量一致的代码。如果需要，链接程序会选择一个合适标准的 C 或 C++ 库来链接。

为 ARM 开发代码时，必须选择合适的 ATPCS 变量。例如，如果编写 ARM 与 Thumb 状态间交互的代码，必须选择编译程序和汇编程序中的 `--apcs /interwork` 选项。

如用 C 或 C++ 编写代码，必须确保为每一编译模块已选择了兼容的 ATPCS 选项。

如果编写自己的汇编语言例程，必须确保与相应的 ATPCS 变量一致。有关详细信息请参阅第 3 章*使用过程调用标准*。

如果采用 C 和汇编语言混合编写，请确保理解 ATPCS 的含意。

1.3.3 ARM 与 Thumb 代码交互

如果为支持 Thumb 16 位指令集的 ARM 处理器编写代码，可按需要将 ARM 和 Thumb 代码混合使用。如用 C 或 C++ 编写代码，必须使用 `--apcs /interwork` 选项进行编译。链接程序检测何时从 Thumb 状态调用 ARM 函数，或何时从 ARM 状态调用 Thumb 函数并改变调用及返回进程，或者在必要时插入交互胶合代码来改变处理器状态。

如果编写汇编语言代码，必须确保遵循交互 ATPCS 变量。有几种改变处理器状态的方法，取决于目标体系结构的版本。有关详细信息请参阅第 4 章 *ARM 和 Thumb 交互操作*。

1.3.4 混合使用 C、 C++ 和汇编语言

可以分别地将已编译和汇编的 C、 C++ 和 ARM 汇编语言模块混用在您的程序中。可在 C 或 C++ 代码中编写小的汇编语言例程。使用 ARM 编译程序的内联或嵌套的汇编程序编译这些例程。但是，如果使用内联或嵌套的汇编程序，对编写的汇编语言代码将有很多限制。这些限制描述如下：

- 第 5-7 页的*内联汇编程序和 armasm 的不同点*；
- 第 5-13 页的*嵌入式汇编语句的限制*。

此外，第 5 章*混合使用 C、 C++ 和汇编语言*给出了通用指南，以及如何在 C、C++ 和 汇编语言模块间调用的示例。

1.3.5 处理处理器异常

ARM 处理器识别出以下异常类型：

复位 在处理器复位管脚有效时发生。该异常仅在处理器上电时、或上电后的复位时才发生。可通过跳转到复位向量 (0x0000) 来完成软复位。

未定义指令 在处理器或任何协处理器均不能识别当前执行指令时发生。

软件中断 (SWI)

这是用户定义的中断指令。它使得在 User 模式下运行的程序能够请求在 Supervisor 模式下运行的特权操作，如 RTOS 函数。

预取中断 仅当处理器试图执行一个从非法地址预取的指令时才发生。非法地址是在存储器中不存在的地址，或者是存储器管理子系统决定的当前模式下处理器不可存取的地址。

数据中断 在数据传输指令试图在非法地址载入或存储数据时发生。

中断 (IRQ) 处理器外部中断请求管脚有效 (LOW) 且使能了 IRQ 中断 （CPSR 中的 I 位被清空）时发生。

快速中断 (FIQ)

处理器外部中断请求管脚有效 (LOW) 且使能了 FIQ 中断（CPSR 中的 F 位被清空）时发生。该异常特别用于要求中断反应时间保持最短的情况。

一般来说，如果编写不依靠操作系统来处理异常的应用程序 （如嵌套式应用程序），就必须为每个类型的异常编写处理程序。

在一个异常类型有多个源的情况下，比如 SWI 或 IRQ 中断，可以为每个源链结异常处理程序。有关详细信息请参阅第 6-1 页的*处理处理器异常*。

对于能够处理 Thumb 代码的处理器，处理异常时处理器转换到 ARM 状态。可以用 ARM 代码编写异常处理程序，或者用一胶合代码转换到 Thumb 状态。有关详细信息请参阅第 6-9 页的*返回地址和返回指令*。

第 2 章
嵌入式软件开发

本章介绍了在有或没有目标系统存在的情况下，如何利用 RVCT 开发嵌入式应用程序。它包含以下几个部分：

2.1 关于嵌入式软件开发

多数嵌入式应用程序最初都是在原型环境下开发的，原型环境的资源与最终产品环境是有差异的。因此，考虑将嵌入式应用程序从其所依赖的开发工具或调试环境移植到在目标硬件上独立运行的系统所涉及的过程，就是非常重要的。

使用 RVCT 开发嵌入式软件时，必须考虑以下以下方面：

- C 库如何使用硬件。
- 某些 C 库的功能通过使用调试环境资源来得以实现。如果使用了此类资源，必须利用目标硬件重新实现这些功能。
- RVCT 并不了解任何给定目标的存储器映射方面的信息。必须使映象存储器映射适合目标硬件存储器的布局。
- 嵌入式应用程序在主应用程序可以运行前必须须执行一些初始化工作。一个完整的初始化进程需要您实现的代码和 RVCT C 库初始化例程。

2.1.1 示例代码

为说明本章包括的主题，提供了相关的项目示例。本章所描述的 Dhrystone 编译代码位于 *Examples_directory*\emb_sw_dev 目录下。每个编译代码对应一个独立的目录，并提供了本章后续节内所讨论技术的示例。有关每个编译代码的特殊信息，可在 “示例”代码各内节找到。

Dhrystone 基准程序为项目示例提供了代码基础。选择 Dhrystone 是因为用它能举例说明本章描述的很多概念。

项目示例是为运行在 ARM Integrator 开发平台上而编写的。但是，示例所说明的原理适用于所有目标硬件。

备注

本章的重点并不特别放在 Dhrystone 程序上，而重点在于使其能在完全独立的系统上运行所要采取的步骤上。有关 Dhrystone 作为一种基准工具更进一步的讨论，请参阅 “应用程序注释”的第 93 条 - *以 ARMulator 为基准*。您可在 http://www.arm.com ARM 网站的 “文档”(Documentation) 区找到 ARM 应用程序注释。

在 Integrator 平台上运行 Dhrystone 编译代码

要在 Integrator 平台上运行本章介绍的 Dhrystone 编译代码，必须：

- 执行 ROM/RAM 重新映射。为此，通过接通开关 1 和 4 来运行 Boot Monitor，然后复位板卡。
- 将 `top_of_memory` 设置为 `0x40000` 或安装一个 DIMM 存储器模块。如果没有这样做，默认为 `0x80000` 的栈可能不会在有效的存储器中。

  例如，要在 RealView Debugger 中设置存储器的顶端地址，打开连接的 [Connection Properties]。在 `Advanced_Information\Default\ARM_config` 组中，将 `Top_memory` 设置成需要的值，然后保存设置。

2.2 RVCT 在目标未知情况下的缺省行为

开始编写嵌入式应用程序时，您可能不清楚目标硬件的全部技术规格。例如，您可能不知道目标外围设备、存储器映射甚至处理器本身的详细信息。

为在了解这些详细信息前能够继续软件的开发，RVCT 工具有默认的操作，使您能立即编译和调试应用程序代码。了解这一默认操作非常有用，这样您会重视从默认的编译代码移植到完全独立的应用程序所必须的步骤。

2.2.1 半主机

在 RVCT C 库中，对某些 ISO C 功能的支持由主机调试环境提供。提供该功能的机制被称为*半主机*。

半主机由一组已定义的软件中断操作来实现。执行半主机 SWI 时，调试代理程序将其识别出并暂时地停止程序的执行。在代码恢复执行之前，由调试代理程序处理半主机操作。因此，由主机本身执行的任务对该程序来说是透明的。

图 2-1 说明了一个半主机操作的示例，该示例将一字符串打印输出到调试器控制台。

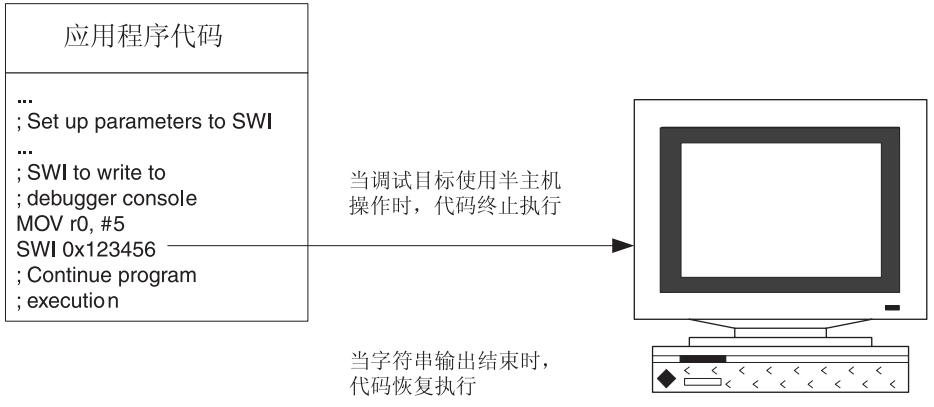

图 2-1 半主机操作示例

备注

有关半主机的详情，请参阅 *RealView 编译工具 2.0 版编译程序和库指南*中的半主机一章。

2.2.2 C 库结构

从概念上来讲，C 库可被化分成两类函数，一类为 ISO C 语言的规范部分，另一类为 ISO C 语言规范提供支持。图示请见图 2-2。

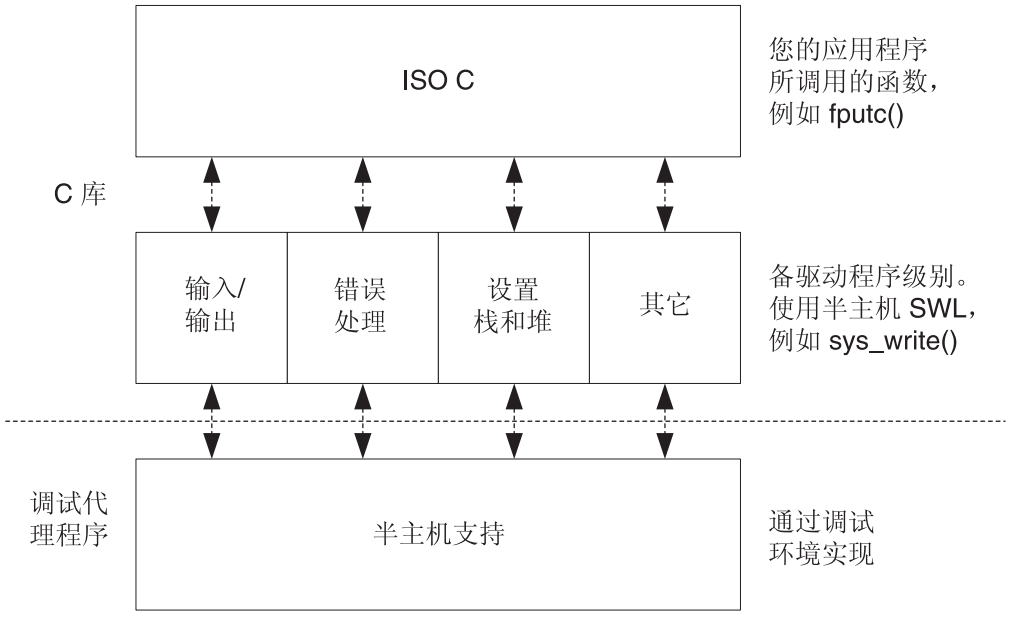

图 2-2 C 库结构

对部分 ISO C 功能的支持是由主机调试环境，在支持函数的设备驱动程序级别提供的。

例如， RVCT C 库通过写入调试器控制台窗口来实现 ISO C `printf()` 系列函数。通过调用 `__sys_write()` 来提供该功能。这是一个执行半主机 SWI 的支持函数，使字符串被写入到控制台。

2.2.3 默认存储器映射

对于没有描述存储器映射的映象，RVCT 根据默认存储器映射放置代码和数据，如图 2-3 所示。

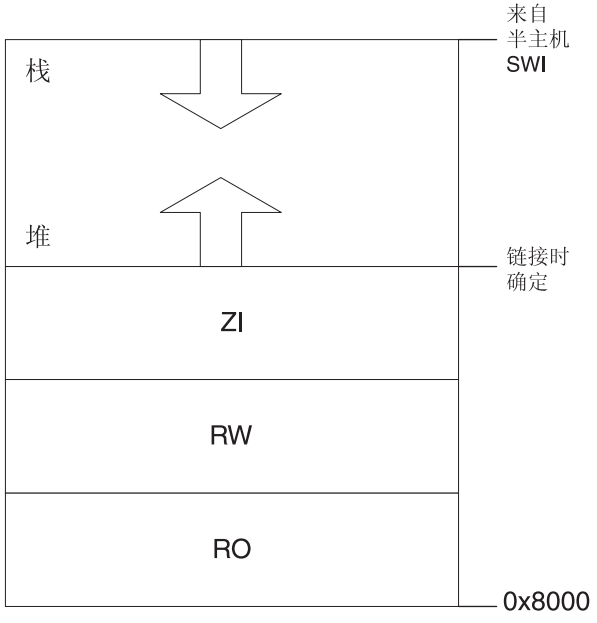

图 2-3 默认存储器映射

以下是对默认存储器映射的说明：

- 链接映象，在地址 0x8000 加载并运行。首先放置所有的 RO （只读）段，其次是 RW （读写）段，然后是 ZI （零初始化）段。
- 堆直接从 ZI 段的顶端地址算起，因此，其准确位置在链接时决定。
- 栈的基本位置在应用程序启动过程中由半主机操作提供。该半主机操作的返回值取决于调试环境：
- ARMulator 返回在配置文件 peripherals.ami 中设定的值。默认值是 0x08000000。
- Multi-ICE 返回调试器内部变量 top_of_memory 的值。默认值是 0x00080000。

2.2.4 链接程序放置规则

链接程序遵守一组规则，如图 2-4 所示，以决定代码和数据位于存储器中的什么位置。

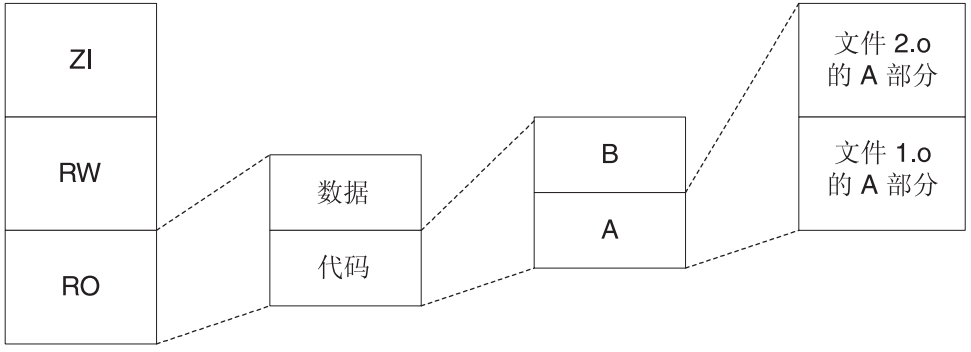

图 2-4 链接程序放置规则

映象首先按属性组织：RO 段在最低的存储器地址，其次是 RW 段，然后是 ZI 段。每一种属性中，代码在数据之前。

从那开始，链接程序按名称的字母顺序放置输入段。输入段名称和汇编程序 AREA 命令一致。

在输入段中，独立对象的代码和数据，按照对象文件在链接程序命令行中被指定的顺序放置。

要精确放置代码和数据，ARM Limited 建议不要过分依靠这些规则。相反，必须使用分散加载机制来完全控制代码和数据的放置。请参阅第 2-13 页的*调整映象存储器映射以适应目标硬件*。

备注

有关放置规则和分散加载的详细说明，请参阅 *RealView 编译工具 2.0 版链接程序和实用程序指南*。

2.2.5 应用程序启动

多数嵌入式系统中，执行主任务前，执行初始化序列来设置系统。

默认的 RVCT 初始化序列如图 2-5 所示。

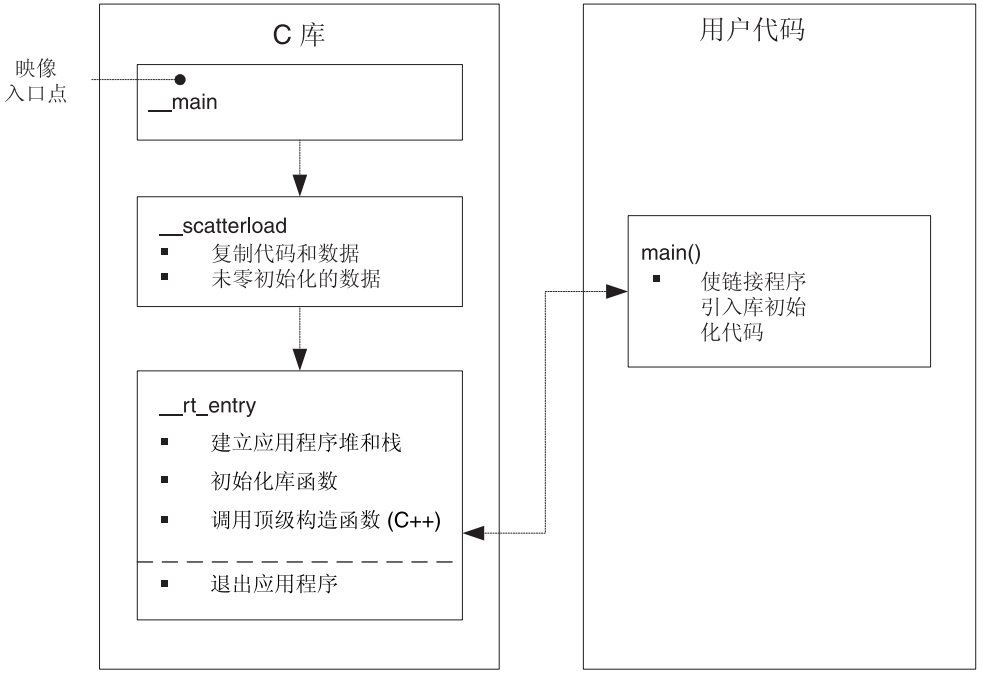

图 2-5 默认的 RVCT 初始化序列

在较高层，初始化序列可分成三个功能块。__main 直接跳转到 __scatterload。__scatterload 负责建立运行时的映像存储器映射，而 __rt_entry（运行时的入口）则负责初始化 C 库。

__scatterload 执行代码和数据复制以及 ZI 数据的清零。对于 ZI 数据的清零和未改变的 RW 数据来说，这一步总是要做的。

__scatterload 跳转到 __rt_entry。它设置应用程序的栈和堆，初始化库函数及其静态数据，并调用任何全局声明的对象的构造函数 （仅 C++）。

然后 __rt_entry 跳转到应用程序入口 main()。主应用程序的结束执行时，__rt_entry 将库关闭，然后把控制权交还给调试器。

RVCT 中，函数标签 main() 有一个特殊含意。main() 函数的存在强制链接程序链接到 __main 和 __rt_entry 中的初始化代码。没有 main() 函数，就不会链接到初始化进程，那么一些标准 C 库功能就不会得到支持。

2.2.6 编译代码 1 的示例代码

编译代码 1 是 Dhrystone Benchmark 的默认编译版本。因此，它遵从本节所介绍的 RVCT 默认操作。请参阅第 2-3 页的*在 Integrator 平台上运行 Dhrystone 编译代码*和 *Examples_directory*\emb_sw_dev\build1 中的编译代码示例文件。

2.3 调整 C 库使其适应目标硬件

默认情况下，C 库利用半主机来提供设备驱动程序级的功能，使得主机能够用作输入和输出设备。这种机制很有用，因为开发硬件经常没有最终系统的所有输入和输出设备。

2.3.1 C 库的重新目标化

您可建立属于自己的 C 库函数实现版本，它能充分利用目标硬件，并自动链接到支持您 C 库实现版本的映象。该过程称为 C 库的重新目标化，如图 2-6 所示。

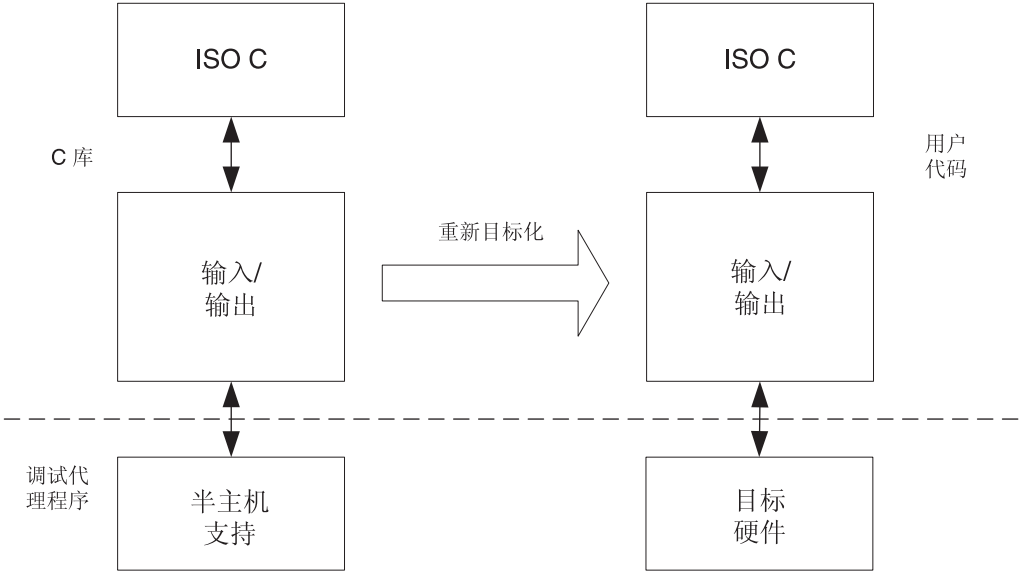

图 2-6 C 库的重新目标化

例如，fputc() 函数将结果写入调试器控制台，您可能有个外围 I/O 设备（如 UART），您想要重新实现该函数，将结果输出到 UART。由于 fputc() 的实现链接最终的映象，整个 printf() 系列函数打印输出到 UART。

fputc() 实现的一个例子如第 2-11 页的示例 2-1 所示。

该示例将 fputc() 的输入字符参数重新指向一连续输出函数 sendchar()，而这是假定在一个独立的源文件中实现的。这样，fputc() 在依目标而定的输出和 C 库标准输出函数之间充当一个抽象层。

示例 2-1 fputc() 的实现

```
extern void sendchar(char *ch);

int fputc(int ch, FILE *f)
{   /* e.g. write a character to an UART */
    char tempch = ch;
    sendchar(&tempch);
    return ch;
}
```

2.3.2 避免 C 库半主机

在一独立应用程序中，您不太可能支持半主机 SWI 操作。因此，必须确保您的应用程序中没有链接 C 库半主机函数。

为确保没有从 C 库链接使用半主机 SWI 的函数，必须引入符号 `__use_no_semihosting_swi`。可在您项目的任何 C 或汇编语言源文件中执行此操作，如下所述：

- 在 C 模块中，使用 `#pragma` 命令：

  ```
  #pragma import(__use_no_semihosting_swi)
  ```

- 在汇编语言模块中，使用 `IMPORT` 命令：

  ```
  IMPORT __use_no_semihosting_swi
  ```

如果仍然链接了使用半主机 SWI 的函数，链接程序会报告下列错误：

```
Error : L6200E: Symbol __semihosting_swi_guard multiply defined (by use_semi.o
and use_no_semi.o).
```

为找出这些函数，请使用 `-verbose` 选项链接。在输出结果中， C 库函数被标以如 `__I_use_semihosting_swi` 的标记。

```
Loading member sys_exit.o from c_a__un.l.
              definition:  _sys_exit
              reference :  __I_use_semihosting_swi
```

必须为这些函数提供您自己的实现方法。

备注

链接程序不报告您应用程序代码中的任何使用半主机 SWI 的函数。仅当从 C 库链接了使用半主机 SWI 的函数时才发生错误。

使用半主机 SWI 的 C 库函数完整清单可参见 *RealView 编译工具 2.0 版编译程序和库指南*。

2.3.3 编译代码 2 的示例代码

编译代码 2 的示例中使用了 Integrator 平台硬件处理时钟和字符串 I/O。请参见 *Examples_directory*\emb_sw_dev\build2 中的示例编译文件。

对编译代码 1 示例项目作了如下更改：

C 库的重新目标化

增加了 ISO C 函数的重新目标化层。这包括标准的输入/输出功能、时钟功能及其它一些出错信号通知以及程序退出。

依赖于目标的设备驱动程序

增加了直接与目标硬件外设交互的设备驱动程序层。

请参阅第 2-3 页的*在 Integrator 平台上运行 Dhrystone 编译代码*。

本项目中没有引入符号 __use_no_semihosting_swi。这是因为半主机 SWI 是在 C 库初始化过程中执行的，以设置应用程序堆和栈的位置。有关重新目标化栈和堆设置的详细说明，请参阅第 2-18 页的*放置栈和堆*。

备注

要查看输出结果，串行接口 A 必须连接一个终端或终端模拟程序（如超级终端）。该串行接口必须设置为 38,400 波特、无奇偶校验、一个停止位和无流控制。该终端必须配置为向引入的行末尾添加换行，并在本地显示键入的字符。

2.4 调整映象存储器映射以适应目标硬件

在最终的嵌入式系统中，没有半主机功能，您不太可能使用 RVCT 提供的默认存储器映射。目标硬件通常有几个位于不同地址范围的存储设备。为了充分利用这些设备，加载和运行时必须有分开的存储器视图。

2.4.1 分散加载

分散加载能够将加载和运行时存储器中的代码和数据描述在被称为*分散加载描述文件*的一个文本描述文件中。在命令行使用 -scatter 选项，该项文件被传递至链接程序。例如：

```
armlink -scatter scat.scf file1.o file2.o
```

分散加载描述文件根据寻址的存储器区域，为链接程序描述了加载时和运行时代码和数据应在的位置。

分散加载区域

分散加载区域分为两类：

- 加载区，包含应用程序复位和加载时的代码和数据。
- 执行区，包含应用程序执行时的代码和数据。应用程序启动过程中，从每个加载区可创建一个或多个执行区。

映象中所有的代码和数据准确地分为一个加载区和一个执行区。

启动过程中，__main 中的 C 库初始化代码实现了必要的代码和数据复制和清零，以从加载视图转为执行视图。

2.4.2 分散加载描述文件语法

分散加载描述文件语法反映了分散加载本身所提供的功能。图 2-7 说明了文件语法。

区域由头标记定义，头标记 （至少）包括一个区名和一个起始地址。可以选择性地增加最大长度和其它属性。

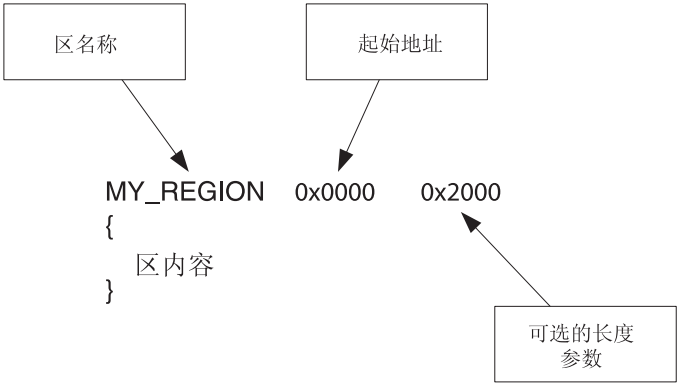

图 2-7 分散加载描述文件语法

区域中的内容取决于区域的类型：

- 加载区必须至少包含一个执行区。在实际操作中，每个加载区通常包含几个执行区。
- 执行区须至少要包含一个代码或数据段，但由 EMPTY 属性声明的区域除外（请参阅第 2-37 页的*使用分散文件的 EMPTY 属性*）。非 EMPTY 声明的区域通常包含源或库目标文件。可用通配符 (*) 语法来为分散加载描述文件中未指定的给定属性的所有段分组。

备注

有关分散加载描述文件语法的详细说明，请参阅 *RealView 编译工具 2.0 版链接程序和实用程序指南*。

2.4.3 分散加载描述文件示例

图 2-8 说明了分散加载的一个简单例子。

该示例有一个包括从地址零开始的所有代码和数据的加载区。从此加载区生成两个执行区。一个包含了所有的 RO 代码和数据，在它被加载的同一地址处执行。另一个在地址 0x10000 处，它包含了所有的 RW 和 ZI 数据。

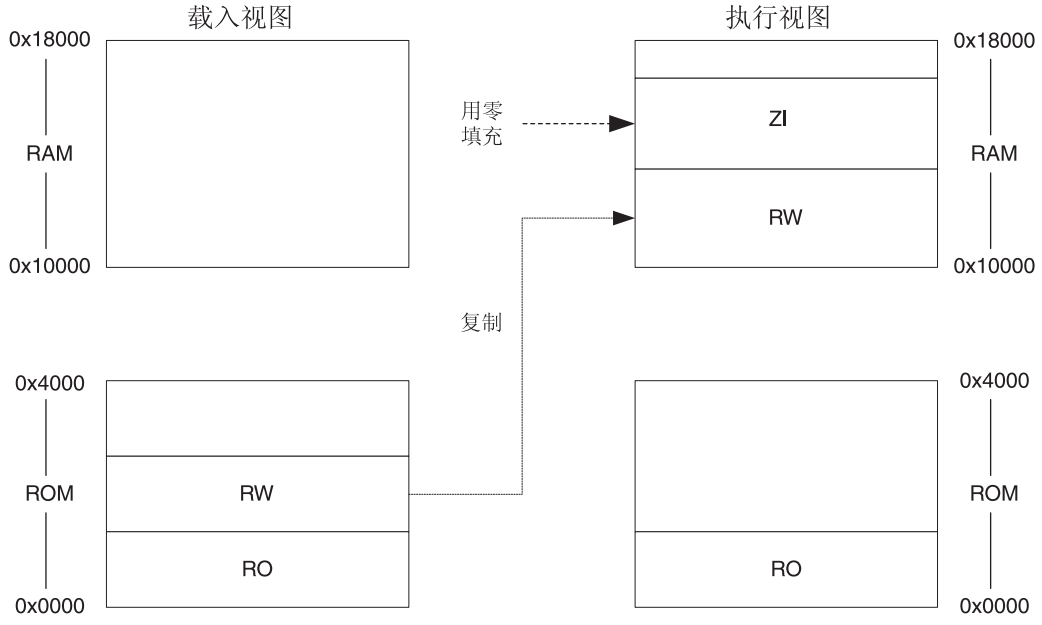

图 2-8 简单分散加载示例

示例 2-2 说明了描述上述存储器映射的描述文件。

示例 2-2 简单分散加载描述文件

```
ROM_LOAD 0x0000 0x4000
{
     ROM_EXEC 0x0000 0x4000; Root region
     {
          * (+RO); All code and constant data
     }
```

```
    RAM 0x10000 0x8000
    {
        * (+RW, +ZI); All non-constant data
    }
}
```

2.4.4 在分散加载描述文件放置对象

对多数映象来说，您可控制特定代码和数据段的放置，而不是象上例中将所有的属性组合在一起。这可通过在描述文件中直接指定单个对象来完成，而不是仅仅依靠通配符语法。

备注

描述文件中执行区对象的顺序不影响其在输出映象的顺序。第 2-7 页的*链接程序放置规则*中说明的链接程序放置规则适用于每个执行区。

要忽略标准链接程序放置规则，可使用 +FIRST 和 +LAST 分散加载命令。示例 2-3 说明了一个在执行区开始放置向量表的分散加载描述文件。在该例中，vectors.o 中的 Vect 区放置在地址 0x0000 处。有关在分散加载描述文件中放置对象的详细说明，请参阅 *RealView 编译工具 2.0 版链接程序和实用程序指南*。

示例 2-3 放置段

```
ROM_LOAD 0x0000 0x4000
{
  ROM_EXEC 0x0000 0x4000
  {
    vectors.o (Vect, +FIRST)
    * (+RO)
  }
  ; more exec regions…
}
```

2.4.5 根区

*根区*是加载地址和其执行地址相同的执行区。每一个分散加载描述文件必须至少要有一个根区。

对分散加载的一个限制就是，负责创建执行区的代码和数据不能将其本身复制到另一位置，例如代码和数据的复制和清零。因此，根区中必须包含以下段。

- `__main.o`，包含复制代码和数据的代码
- `Region$$Table` 和 `ZISection$$Table` 段，包含要复制代码和数据的地址。

由于这些段为只读属性，它们靠通配符 `* (+RO)` 语法分组。因此，如果在非根区中指定了 `* (+RO)`，必须在根区中显式地声明这些段。如示例 2-4 所示。

示例 2-4 指定根区

```
ROM_LOAD 0x0000 0x4000
{
     ROM_EXEC 0x0000 0x4000 ; root region
     {
          __main.o (+RO)       ; copying code
          * (Region$$Table)    ; RO/RW addresses to copy
          * (ZISection$$Table) ; ZI addresses to zero
     }
     RAM 0x10000 0x8000
     {
          * (+RO)              ; all other RO sections
          * (+RW, +ZI)         ; all RW and ZI sections
     }
}
```

如果根区中没有包含 `__main.o`、`Region$$Table` 和 `ZISection$$Table`，将使链接程序产生下列错误信息：

```
Error : L6202E: Section Region$$Table cannot be assigned to a non-root region.
Error: L6202E: Section ZISection$$Table cannot be assigned to a non-root
region.
```

2.4.6 放置栈和堆

分散加载为映象中的代码和静态分配数据提供了一种放置方法。本节介绍了如何放置应用程序的堆和栈。

应用程序的堆和栈是在 C 库初始化过程中建立的。可通过重新目标化负责建立堆和栈的例程来调整堆和栈的放置。在 RVCT C 库中，该例程是 `__user_initial_stackheap()`。

图 2-9 说明了带有重新目标化 `__user_initial_stackheap()` 的 C 库初始化过程。

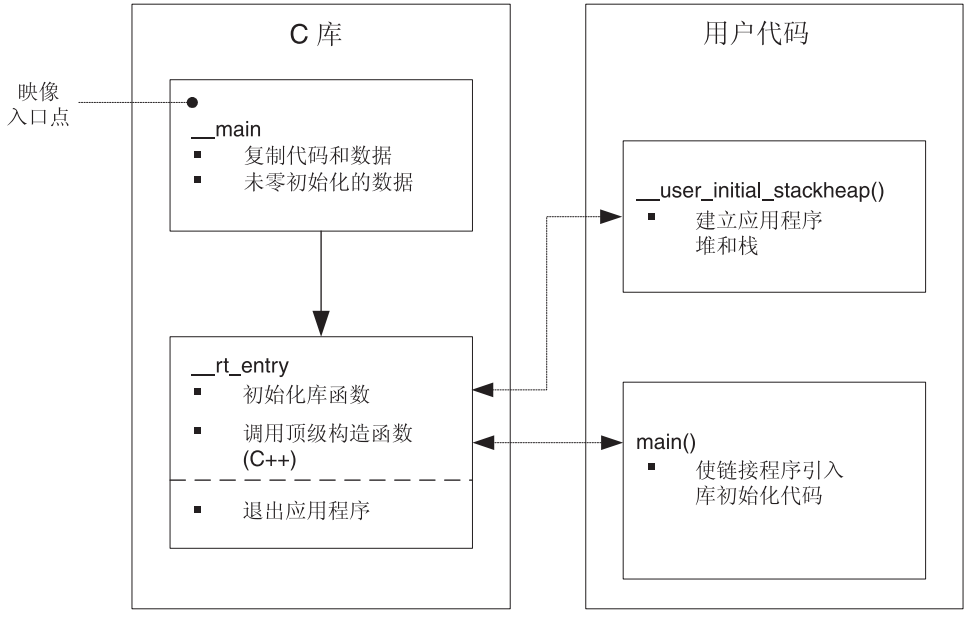

图 2-9 Retargeting __user_initial_stackheap()

`__user_initial_stackheap()` 可用 C 或 ARM 汇编语言来编写。它必须返回以下参数：

- r0 中的堆基址；
- r1 中的栈基址；
- r2 中的堆限制 （如有必要）；
- r3 中的栈限制 （如有必要）。

如果使用分散加载方法加载您的映象，必须重新实现 __user_initial_stackheap()。否则，链接程序产生如下错误：

```
Error : L6218E: Undefined symbol Image$$ZI$$Limit (referred from
sys_stackheap.o)
```

2.4.7 运行时存储器模块

RVCT 提供了两种可能用到的运行时存储器模型：

- *单区模型*为默认模块；
- 第 2-20 页的*双区模型*。

在两种运行时存储器模型中，默认情况下不检查栈的增长。可通过使用编译程序选项 --apcs /swst 编译所有的模块来启用映象中的软件栈检查。如果使用双区模型，在实现 __user_initial_stackheap() 时，还必须指定栈限制。

备注

启用软件栈检查增加了代码的实际大小和执行开销，因为必须在每一次函数调用时检查栈指针的值，看其是否在栈限制内。它也使用 r10 寄存器，因此，通常建议不要在嵌入式系统中使用。

两个例子均适用于 Integrator 系统。

单区模型

默认情况下为*单区模型*，应用程序的堆和栈在同一存储器区中互相朝向对方增长。在此情况下，分配新的堆空间时（即调用 malloc() 时），对照栈指针的值检查堆。

示例 2-5 和第 2-20 页的图 2-10 说明了 __user_initial_stackheap() 实现简单的单区模型的一个示例，其中栈从地址 0x40000 向下增长，堆从地址 0x20000 向上增长。例程将相应的值加载到寄存器 r0 和 r1，然后返回。r2 和 r3 保持不变，因为在单区模型中不使用堆限制和栈限制。

示例 2-5 单区模型例程

```
    EXPORT __user_initial_stackheap

__user_initial_stackheap
    LDR r0, =0x20000 ;HB
```

```
    LDR r1, =0x40000 ;SB
    ; r2 not used (HL)
    ; r3 not used (SL)
    MOV pc, lr
```

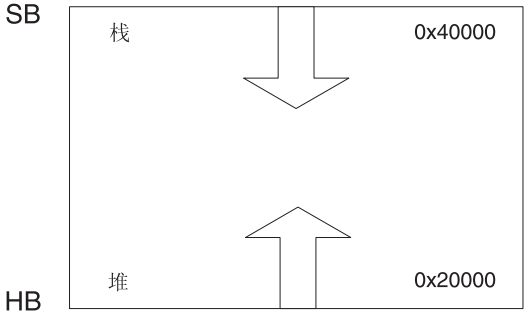

图 2-10 单区模型

双区模型

在您的系统设计中，可能会需要到将堆和栈分别放置在存储器不同的区中。例如，可能会保留一小块高速 RAM 仅供栈使用。要通知 RVCT 您想使用*双区模型*，必须使用汇编命令 IMPORT 引入符号 __use_two_region_memory。然后，对照 __user_initial_stackheap() 建立的专用堆限制来检查堆。

示例 2-6 和第 2-21 页的图 2-11 说明了一个实现双区模型的示例。栈从 0x40000 向 0x20000 的限制向下增长。为使用该栈限制，所有使用此实现的模块必须进行编译以便进行软件栈检查。堆从 0x28000000 到 0x28080000 向上增长。

示例 2-6 双区模型例程

```
    IMPORT __use_two_region_memory
    EXPORT __user_initial_stackheap

__user_initial_stackheap
    LDR r0, =0x28000000 ;HB
    LDR r1, =0x40000 ;SB
```

```
LDR r2, =0x28080000 ;HL
LDR r3, =0x20000 ;SL
MOV pc, lr
```

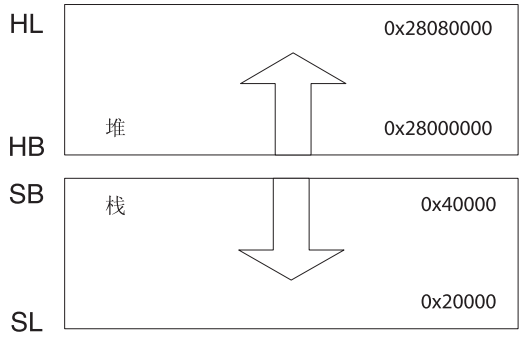

图 2-11 双区模型

2.4.8 编译代码 3 的示例代码

编译代码 3 示例实现了分散加载，并且包括一个重新目标化的 `__user_initial_stackheap()`。请参阅 *Examples_directory*\emb_sw_dev\build3 中的编译代码示例文件。

对编译代码 2 示例项目作了如下更改：

分散加载

一个简单的分散加载描述文件传递给链接程序。

重新目标化的 `__user_initial_stackheap()`

可选择单区或双区实现方法之一。默认是单区编译。可通过在编译步骤定义 `two_region` 来选择双区实现方法。

避免 C 库的半主机

该编译代码中引入了符号 `__use_no_semihosting_swi`，因为映象中不再有 C 库半主机函数存在。

请参阅第 2-3 页的*在 Integrator 平台上运行 Dhrystone 编译代码*。

备注

为避免对 `clock()` 使用半主机，在 Integrator 汇编平台上，这一功能通过读取*实时时钟* (RTC) 来重新目标化。它有一秒钟的影响，因而 Dhrystone 的结果是不精确的。该机制在编译代码 4 中进行了改进。

2.5 复位和初始化

到目前为止本章假设是从 C 库初始化例程的入口 __main 开始执行的。实际上，任何目标硬件上的嵌入式应用程序在启动时都实现了一些系统级的初始化。本节将对此予以详细说明。

2.5.1 初始化序列

图 2-12 说明了一个适用于 ARM 嵌入式系统的可能的初始化序列。

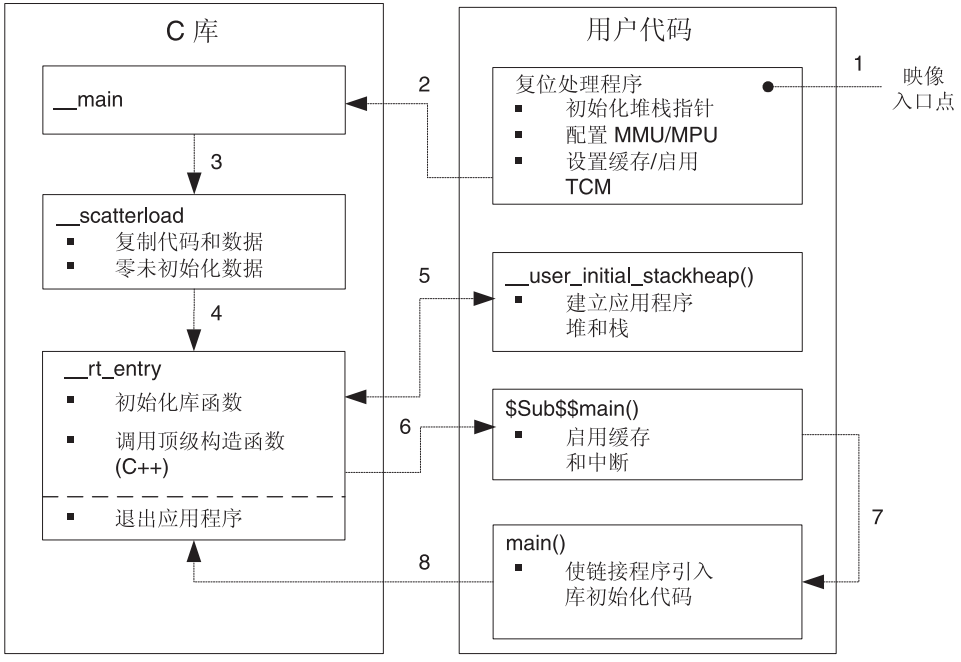

图 2-12 初始化序列

图 2-12 中的复位处理程序在系统启动时立即执行。进入主应用程序前，立即执行标有 $Sub$$main() 的代码块。

复位处理程序是用汇编语言写的短代码块，它在系统复位时执行。复位处理程序最少为应用程序运行所在的模式初始化栈指针。对于具有局部存储器的内核，如*缓存*、*紧密藕荷存储器* (TCM)、*存储器管理单元* (MMU) 和 *存储器保护单元* (MPU) 等，在初始化过程这一阶段必须完成某些配置。复位处理程序在执行之后，通常跳转到 __main 以开始 C 库的初始化序列。

系统初始化的一些组成部分，如启用中断，通常是在 C 库初始化代码执行完成后才执行。在主应用程序开始执行前，标有 $Sub$$main() 的代码块立即执行这些任务。

有关初始化序列各个组成部分的详细说明，请参阅*向量表*。

2.5.2 向量表

所有的 ARM 系统都有一个*向量表*。向量表不是初始化序列的一部分，但是，对每个要处理的异常，它必须存在。

示例 2-7 中的代码引入了各种异常处理程序，可能可在其它模块中进行代码移植。向量表是转到各种异常处理程序的跳转指令列表。

FIQ 处理程序被直接放置在地址 0x1c。它省下了到 FIQ 处理程序的一次跳转，因而优化了 FIQ 的反应时间。

示例 2-7 引入异常处理程序

```
    AREA Vectors, CODE, READONLY
    IMPORT Reset_Handler
; import other exception handlers
    ; …
                ENTRY
    B   Reset_Handler
    B   Undefined_Handler
    B   SWI_Handler
    B   Prefetch_Handler
    B   Data_Handler
    NOP ; Reserved vector
    B   IRQ_Handler
        ; FIQ_Handler will follow directly
        END
```

备注

向量表标有 ENTRY 标签。该标签通知链接程序该代码是一个可能的入口点，因而在链接时，不能从映象中将其清除。必须使用 -entry 链接程序选项在可能的映象入口中选一个作为应用程序的真正入口点。有关详细说明，请参阅 *RealView 编译工具 2.0 版编译程序和实用程序指南*。

2.5.3 ROM/RAM 重新映射

必须考虑您的系统在执行的第一条指令的地址 0x0000 处是什么样的存储器。

备注

本节假设 ARM 内核开始在 0x0000 地址取指令，这是以 ARM 为内核的系统的标准。但是，有些 ARM 内核可通过配置从 0xFFFF0000 地址开始取指令。

启动时，0x0000 处必须要有一条有效指令，因此，复位时 0x0000 地址必须为非易失性存储器。

一种实现方法就是将 ROM 定位在 0x0000。但是，这样配置有几个缺点。对 ROM 的存取速度通常较 RAM 要慢，当跳转到异常处理程序时，系统性能可能会大受影响。并且，将向量表放于 ROM 中，使您不能在运行时修改它。

另一个解决方法如第 2-26 页的图 2-13 所示。 ROM 位于地址 0x10000，但是该存储器在复位时被存储控制器还分配了第二个地址：0。复位后，复位处理程序的代码跳转到 ROM 的实际地址。然后存储器控制器清除 ROM 的临时第二地址设置，因而在地址 0x0000 显示 RAM。在 __main 中，向量表被复制到 0x0000 地址的 RAM 中，从而异常得以处理。

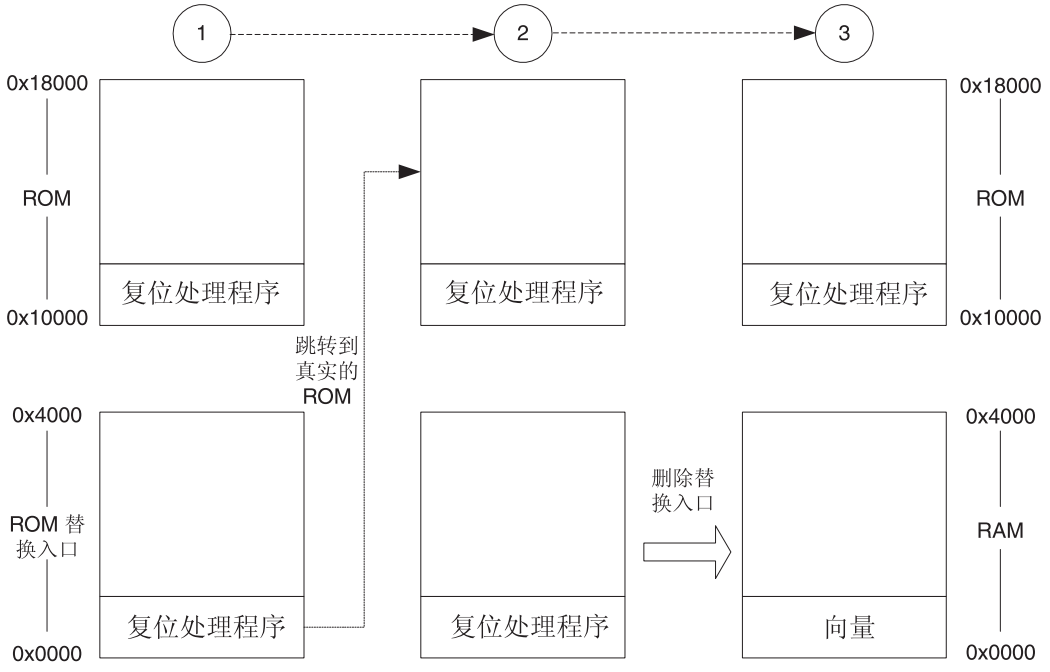

图 2-13 ROM/RAM 重新映射

第 2-27 页的示例 2-8 说明了如何在 ARM 汇编模块中实现 ROM/RAM 的重新映射。此处所显示的常量是特定于 Integrator 平台的，但是，同样的方法适用于用类似方法实现 ROM/RAM 重映射的任何平台。

第一条指令是从 ROM 临时地址 （0 地址）到实际 ROM 的一条转移指令。可以这样做是因为标签 `Instruct_2` 位于实际 ROM 地址处。

之后，通过转换综合内核模块 (Integrator Core Module) 控制寄存器的重映射位，清除 ROM 的临时地址设置。

该代码通常在系统复位后立即执行。重新映射必须在执行 C 库初始化代码前完成。

备注

在具有 MMU 的系统中，可通过在系统启动时配置 MMU 来实现重映射。

示例 2-8 ROM/RAM 重新映射

```
; --- Integrator CM control reg
CM_ctl_reg      EQU  0x1000000C     ; Address of CM Control Register
Remap_bit       EQU  0x04           ; Bit 2 is remap bit of CM_ctl

        ENTRY

; Code execution starts here on reset
; On reset, an alias of ROM is at 0x0, so jump to 'real' ROM.
        LDR     pc, =Instruct_2

Instruct_2
; Remap by setting Remap bit of the CM_ctl register
        LDR     r1, =CM_ctl_reg
        LDR     r0, [r1]
        ORR     r0, r0, #Remap_bit
        STR     r0,[r1]

; RAM is now at 0x0.
; The exception vectors must be copied from ROM to RAM (in __main)

; Reset_Handler follows on from here
```

2.5.4 局部存储器设置有关的考虑事项

许多 ARM 处理器内核具有芯片级的存储器系统，如 MMU 或 MPU。这些设备通常是在系统启动过程中进行设置并启用的。因此，带有局部存储器系统的内核的初始化序列需要特别地考虑。

如本章所述，__main 中 C 库初始化代码负责建立映象在执行时的存储器映射。因而，在跳转到 __main 前，必须建立处理器内核的运行时存储器视图。这就是说，在复位处理程序中必须设置并启用 MMU 或 MPU。

在跳转到 __main 前（通常在 MMU/MPU 设置前），必须启用 TCM，因为在通常情况下都是采用分散加载方法将代码和数据装入 TCM。必须注意，不必存取启用 TCM 后屏蔽的存储器。

在跳转到 __main 前，如果启用了高速缓存，可能还会遇到高速缓存一致性的问题。__main 中的代码从其加载地址向其执行地址复制代码区，基本上是将指令作为数据进行处理。结果，一些指令可放入数据高速缓存中，在此情况下，对指令路径来说，它们是不可见的。

为避免这些一致性问题，在 C 库初始化序列执行完成后启用高速缓存。

2.5.5 分散加载和存储器设置

在一个内核的复位时存储器视图会改变的系统中，不论是通过 ROM/RAM 重映射还是通过 MMU 配置，分散加载文件必须描述重映射发生以后的存储器映射关系。

示例 2-9 中的描述文件和重映射后第 2-25 页的 *ROM/RAM 重新映射* 中的示例相关。

示例 2-9

```
ROM_LOAD 0x10000 0x8000
{
     ROM_EXEC 0x10000 0x8000
     {
          reset_handler.o (+RO, +FIRST)   ; executed on hard reset
          …
     }

     RAM 0x0000 0x4000
     {
            vectors.o (+RO, +FIRST)  ; vector table copied from ROM to RAM at
zero
          …
     }
}
```

加载区 ROM_LOAD 放置在 0x10000，因为它指示了重映射发生后代码和数据的加载地址。

2.5.6 栈指针初始化

复位处理程序最少必须为应用程序所使用的任何执行模式的栈指针分配初始值。

在示例 2-10 中，栈位于 stack_base 地址。该符号可以是个硬编码地址，或者可以是在独立的汇编源文件中定义并由分散加载描述文件定位。有关如何完成的详细信息，请参阅第 2-34 页的*在分散加载描述文件中放置堆和栈*。

该示例中为 FIQ 和 IRQ 模式分配了 256 字节的栈，也可对其它执行模式这么做。为了设置栈指针，进入每种模式 （中断禁用）并为栈指针分配适合的值。要利用软件栈检查，也必须在此设置栈限制。

复位处理程序中设置的栈指针和栈限制值由 C 库初始化代码作为参数自动传递给 __user_initial_stackheap()。因此，不允许 __user_initial_stackheap() 更改这些值。

示例 2-10 初始化栈指针

```
; --- Amount of memory (in bytes) allocated for stacks
Len_FIQ_Stack    EQU     256
Len_IRQ_Stack    EQU     256
…

Offset_FIQ_Stack         EQU     0
Offset_IRQ_Stack         EQU     Offset_FIQ_Stack + Len_FIQ_Stack
…

Reset_Handler

; stack_base could be defined above, or located in a description file
        LDR     r0, stack_base ;

; Enter each mode in turn and set up the stack pointer
        MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit
        SUB     sp, r0, #Offset_FIQ_Stack

        MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit
        SUB     sp, r0, #Offset_IRQ_Stack
        …
; Set up stack limit if needed
        LDR     r10, stack_limit
```

示例 2-11 说明了 __user_initial_stackheap() 的一种实现，可以与第 2-29 页的示例 2-10 中所示的栈指针设置配合使用。

示例 2-11

```
    IMPORT heap_base
    EXPORT __user_initial_stackheap()

__user_initial_stackheap()

; heap base could be hard coded, or placed by description file
    LDR r0,=heap_base
    ; r1 contains SB value
    MOV pc,lr
```

2.5.7 硬件初始化

一般来说，将所有系统初始化的代码和主应用程序分开是非常好的一种做法。但是，部分系统初始化过程，如启用高速缓存和中断，必须在 C 库初始化代码执行完成后才能发生。

可利用 $Sub 和 $Super 函数包装符来有效地插入一个例程，该例程在进入主应用程序前立即执行。这一机制使您能在不改变源代码的情况下扩展函数的功能。

示例 2-12 说明了如何以这种方式使用 $Sub 和 $Super。链接程序通过调用 $Sub$$main() 取代了对 main() 函数的调用。从该处可调用启用高速缓存的例程，也可调用启用中断的另一例程。

通过调用 $Sub$$main()，代码跳转到实际的 main()。

备注

有关 $Sub 和 $Super 的更多说明，请参阅 *RealView 编译工具 2.0 版链接程序和实用程序指南*。

示例 2-12 使用 $Sub 和 $Super

```
extern void $Super$$main(void);

void $Sub$$main(void)
{
  cache_enable();           // enables caches
```



```
  int_enable();      // enables interrupts
  $Super$$main();            // calls original main()
}
```

2.5.8 执行模式需考虑的事项

必须考虑主应用程序要运行的模式。您的选择会影响如何实现系统的初始化。

许多想在启动时在复位处理程序和 `$Sub$$main` 中实现的功能，仅能够在特权模式下执行方可完成。例如，芯片级存储器操作及启用中断。

如果您想在特权模式下（如超级用户 (Supervisor) 模式）运行应用程序，这不成问题。请确保在退出复位处理程序前切换到适当的模式。

如果想在用户 (User) 模式下运行应用程序，在优先模式下完成必要的任务后，切换到用户模式即可。这在 `Sub$$main()` 中最可能发生。

备注

`__user_initial_stackheap()` 必须设置应用程序模式栈。所以，必须退出系统模式下的，使用用户模式寄存器的复位处理程序。`__user_initial_stackheap()` 然后在系统模式下执行，因而，在进入用户模式时，应用程序的堆和栈仍然存在。

2.5.9 编译代码 4 示例代码

编译代码 4 示例可在 Integrator 平台上独立运行。请参阅 *Examples_directory*\emb_sw_dev\build4 中的编译代码示例文件。

对编译代码 3 示例项目作了如下更改：

向量表

对项目增加了一个向量表，由分散加载描述文件放置。

复位处理程序

init.s 中增加了复位处理程序。 ARM926EJ-S 编译代码中包含了分别负责设置 TCM 和 MMU 的两个独立模块。运行在带任何内核的 Integrator 平台上的 ARM7/TDMI 编译代码不包括这些模块。复位后， ROM/RAM 重映射立即发生。

`$Sub$$main()`

对于 ARM926EJ-S 编译代码，进入主应用程序前，在 `$Sub$$main()` 中启用高速缓存。

嵌入式描述文件

使用了嵌入式描述文件，它反映了重映射后的存储器视图。

这些编译代码的编译文件生成了一个适合下载至地址 0x24000000 处的 Integrator AP 应用程序闪存的二进制文件。

使用 AP 主板上的定时器实现精确的定时器。这产生一个 IRQ，随安装一个处理程序每隔 1/100 秒增加计数器的计数。

2.6 进一步的存储器映射考虑事项

本章的上一节介绍了分散加载描述文件中代码和数据的放置，但是，其目标硬件外设的位置和堆和栈限制被假设为源文件或头文件中的硬编码。在描述文件中定位所有与目标的存储器映射有关的信息是大有好处的，因此从源代码中删除所有引用绝对地址的代码。

2.6.1 在分散加载描述文件中定位目标外设

按惯例，外设寄存器的地址在项目源文件或头文件中是硬编码的。也可声明映射到外设寄存器的结构，并将这些结构放置在描述文件中。

例如，目标可能会有一个带双存储器映射 32 位寄存器的定时器外设。示例 2-13 说明了映射到这些寄存器的一个 C 结构。

示例 2-13 映射到外设寄存器

```
struct {
            volatile unsigned ctrl; /* timer control */
            volatile unsigned tmr;  /* timer value   */
       } timer_regs;
```

要把该结构放在存储器映射的特定地址，需创建一个新的执行区来装入该结构。

示例 2-14 说明的描述文件将 `timer_regs` 结构定位在 `0x40000000` 地址处。

应用程序启动过程中不将这些寄存器的内容初始化为零是非常重要的，因为这很可能改变系统的状态。将执行区标记上 `UNINIT` 属性可避免该区中的 ZI 数据初始化为零。

示例 2-14 放置映射结构

```
ROM_LOAD 0x24000000 0x04000000
{
 ; …
 TIMER 0x40000000 UNINIT
 {
     timer_regs.o (+ZI)


 }
 ; …
}
```

2.6.2 在分散加载描述文件中放置堆和栈

多数情况下，在描述文件中指定堆和栈的位置是更可取的做法。它有两个主要优点：

- 有关存储器映射的所有信息保存在一个文件中。
- 改变堆和栈只要求重新链接，而不需要重新编译。

本节介绍了实现该功能的如下几个方法：

- *显式放置符号*，两种方法中最简单的方法；
- 第 2-35 页的*使用链接程序生成符号*；
- 第 2-37 页的*使用分散文件的 EMPTY 属性*。

显式放置符号

第 2-29 页的*栈指针初始化*作为可放置在描述文件中的参照符号引用符号 stack_base 和 heap_base。为此，需在名为 stackheap.s 的汇编模块中创建标有 stack_base 和 heap_base 的符号。对于双区存储器模型的堆和栈限制也可执行同样的操作。

可在描述文件的各自执行区中定位每个符号，如示例 2-15 所示。

示例 2-15 在 stackheap.s 中显式地放置符号

```
        AREA   stacks, DATA, NOINIT
        EXPORT stack_base

stack_base        SPACE   1

        AREA   heap, DATA, NOINIT
        EXPORT heap_base

heap_base         SPACE   1
        END
```

第 2-35 页的示例 2-16 和第 2-35 页的图 2-14 说明了如何在 0x20000 放置堆基址，如何在 0x40000 放置栈基址。堆基址和栈基址的位置可通过分别编辑其执行区予以改变。

该方法的缺点是在该栈区的上部占用 SPACE (stack_base) 的一个字。

示例 2-16 在分散文件中显式地放置符号

```
LOAD_FLASH 0x24000000 0x04000000
{
 ; …
    HEAP 0x20000 UNINIT
    {
        stackheap.o (heap)
    }

    STACKS 0x40000 UNINIT
    {
        stackheap.o (stacks)
    }
 ; …
}
```

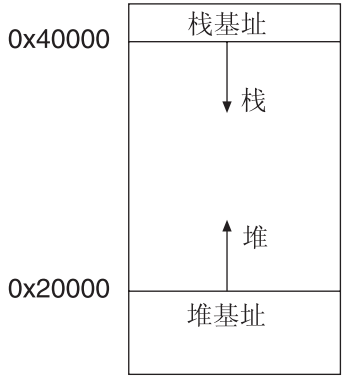

图 2-14 显式放置符号

使用链接程序生成符号

该方法需要在目标文件中指定堆和栈的大小。

首先，在一个汇编源文件中为堆和栈定义一个适当大小的区域，例如，stackheap.s，如第 2-36 页的示例 2-17 所示。使用 SPACE 命令保留一个清零的存储器块。设置 NOINIT 区域属性避免这一清零过程。开发过程中，可能会选择将栈零初始化以最大化地使用栈。此源文件中不需要标签。

示例 2-17 放置用于栈和堆的段

```
        AREA stack, DATA, NOINIT
    SPACE   0x3000 ; Reserve stack space

        AREA heap, DATA, NOINIT
    SPACE   0x3000 ; Reserve heap space

        END
```

然后就可在分散加载描述文件的各自执行区中放置这些段，如示例 2-18 所示。

示例 2-18 放置用于栈和堆的段

```
LOAD_FLASH 0x24000000 0x04000000
{
    :
    STACK 0x1000 UNINIT   ; length = 0x3000
    {
        stackheap.o (stack)  ; stack = 0x4000 to 0x1000
    }

    HEAP 0x15000 UNINIT   ; length = 0x3000
    {
        stackheap.o (heap)  ; heap = 0x15000 to 0x18000
    }
}
```

链接程序生成了指向每个执行区基址和限制的符号，可将其引入重新目标化目标代码中供 __user_initial_stackheap() 使用：

```
Image$$STACK$$ZI$$Limit = 0x4000
Image$$STACK$$ZI$$Base  = 0x1000
Image$$HEAP$$ZI$$Base   = 0x15000
Image$$HEAP$$ZI$$Limit  = 0x18000
```

通过使用 DCD 命令赋予这些值更有意义的名称可使该代码可读性更高，如第 2-37 页的示例 2-19 所示。

示例 2-19 使用 DCD 命令

```
        IMPORT      ||Image$$STACK$$ZI$$Base||
        IMPORT      ||Image$$STACK$$ZI$$Limit||
        IMPORT      ||Image$$HEAP$$ZI$$Base||
        IMPORT      ||Image$$HEAP$$ZI$$Limit||

    stack_base  DCD     ||Image$$STACK$$ZI$$Limit||       ; = 0x4000
    stack_limit DCD     ||Image$$STACK$$ZI$$Base||        ; = 0x1000

    heap_base   DCD     ||Image$$HEAP$$ZI$$Base||         ; = 0x15000
    heap_limit  DCD     ||Image$$HEAP$$ZI$$Limit||        ; = 0x18000
```

可使用这些示例将堆基址放置在 0x15000，将栈基址放置在 0x1000。然后，堆和栈的基址位置通过分别编辑其执行区可以很容易地改变。

使用分散文件的 EMPTY 属性

该方法使用了链接程序的分散文件 EMPTY 属性。这将使得要定义的区域不包括目标代码或数据。这是定义堆和栈的一个方便方法。区域的长度在 EMPTY 属性后指定。对于存储器中向上增长的堆，其区域的长度为正。对于栈，其长度被标为负数，指示它在存储器中是向下增长的。示例 2-20 说明了如何使用 EMPTY 属性。

该方法的优点是堆和栈的大小和位置是在一个地方定义的，即在分散文件中。您不必创建 stackheap.s 文件。

示例 2-20 使用 EMPTY 放置栈区和堆区

```
ROM_LOAD 0x24000000 0x04000000
{
:
     HEAP 0x30000 EMPTY 0x3000
     {
     }

     STACKS 0x40000 EMPTY -0x3000
     {
     }
:
}
```

链接时，链接程序生成代表这些 EMPTY 区的符号。

```
Image$$HEAP$$ZI$$Base      = 0x30000
Image$$HEAP$$ZI$$Limit     = 0x33000
Image$$STACKS$$ZI$$Base    = 0x3D000
Image$$STACKS$$ZI$$Limit   = 0x40000
```

然后，应用程序代码就可处理这些符号，如示例 2-21 所示。

示例 2-21 链接程序生成代表 EMPTY 区的符号

```
                IMPORT      ||Image$$HEAP$$ZI$$Base||
                IMPORT      ||Image$$HEAP$$ZI$$Limit||

heap_base       DCD         ||Image$$HEAP$$ZI$$Base||
heap_limit      DCD         ||Image$$HEAP$$ZI$$Limit||

                IMPORT      ||Image$$STACKS$$ZI$$Base||
                IMPORT      ||Image$$STACKS$$ZI$$Limit||

stack_base      DCD         ||Image$$STACKS$$ZI$$Limit||
stack_limit     DCD         ||Image$$STACKS$$ZI$$Base||
```

2.6.3 编译代码 5 示例代码

编译代码 5 示例与编译代码 4 等价，但是，所有的目标存储器映射信息在分散加载描述文件中定位，如第 2-34 页的*显式放置符号*所示。

分散加载描述文件符号

定位栈、堆和外设的符号在汇编模块中声明

更新了分散加载描述文件

对编译代码 4 的嵌入式描述文件进行了更新，以定位栈、堆、数据 TCM 和外设。

请参阅 *Examples_directory*\emb_sw_dev\build5 中的编译代码示例文件。

使用链接程序符号定位堆和栈，请参阅第 2-35 页的*使用链接程序生成符号*。

第 3 章
使用过程调用标准

本章描述如何使用 *ARM-Thumb 过程调用标准* (ATPCS)。遵循 ATPCS 可以确保分别编译和汇编的模块可以协同工作。本章包含以下几个部分：

3.1 关于 ARM-Thumb 过程调用标准

遵循 *ARM-Thumb 过程调用标准*(ATPCS) 可以确保分别编译或汇编的子程序能够协同工作。本章描述如何使用 ATPCS。

ATPCS 有多个变体。本章提供的信息使您能够选择要使用的变体。

不论使用什么变体，此标准的许多细节都是相同的。请参阅：

- 第 3-4 页的*寄存器角色和名称*;
- 第 3-6 页的*栈*;
- 第 3-9 页的*参数传递*。

3.1.1 ATPCS 变体

变体包含基本标准，可以独立选择一些选项来修改它。符合基本标准的代码比符合其它变体的代码运行速度快，并且占用内存少。然而，符合基本标准的代码不提供：

- ARM 状态和 Thumb 状态之间的交互操作；
- 数据或代码位置独立；
- 使用独立数据的可重入调用；
- 栈检查。

编译程序或汇编程序设置 ELF 目标文件中的*属性*，记录您选择的变体。一般来说，您必须选择一个变体，然后将它用于必须协同工作的所有子程序。此规则的异常在正文中描述。

选项在以下标题中描述：

- 第 3-11 页的*栈限制检查*;
- 第 3-14 页的*只读位置无关*;
- 第 3-15 页的*读写位置无关*;
- 第 3-16 页的 *ARM 状态和 Thumb 状态之间的交互操作*;
- 第 3-17 页的*浮点选项*。

3.1.2 ARM C 库

有多个 ARM C 库变体（请参阅 *RealView Compilation Tools v2.0 编译程序和库指南*）。链接程序选择连接目标文件所用的变体。它选择符合目标文件中记录的 ATPCS 选项的最佳变体。请参阅 *RealView Compilation Tools v2.0 链接程序和实用程序指南*中的链接程序一章。

3.1.3 符合 ATPCS

使用 ARM 编译程序编译的 extern 函数符合选定的 ATPCS 变体。

您负责确保用汇编语言编写的函数符合选定的 ATPCS 变体。

要符合 ATPCS，汇编语言函数必须：

- 在公开可见的接口中遵循所有标准细节；
- 在任何时候都遵循 ATPCS 的栈使用规则；
- 选择适当的 --apcs 选项进行汇编。

3.1.4 进程和内存模型

ATPCS 应用单*执行线程*或*进程*。进程的*内存状态*由处理器寄存器内容和它可以寻址的内存内容定义。

进程可以寻址以下一些或全部内存类型：

- 只读存储器。
- 静态分配读写存储器。
- 动态分配读写存储器。这称为*堆*内存。
- 栈内存。请参阅第 3-6 页的*栈*。

一个进程不得改变另一个进程的内存状态，除非特别指定两个进程协同工作。

3.2 寄存器角色和名称

ATPCS 指定用于特定用途的寄存器。

3.2.1 寄存器角色

除非另外声明，以下寄存器用法适用于所有 ATPCS 变体。要符合 ATPCS，必须遵循这些规则：

- 使用寄存器 r0-r3 将参数值传送到函数，并将结果值传出。可以用 a1-a4 来引用 r0-r3，以使此用法透明。请参阅第 3-9 页的*参数传递*。在子程序调用之间，可以将 r0-r3 用于任何用途。被调用函数在返回之前不必恢复 r0-r3。如果调用函数需要再次使用 r0-r3 的内容，则它必须保留这些内容。
- 使用寄存器 r4-r11 存放函数的局部变量。可以用 v1-v8 来引用这些寄存器，以使此用法透明。在 Thumb 状态下，在大多数指令中只有寄存器 r4-r7 可以用于局部变量。

  如果被调用函数使用了这些寄存器，它在返回之前必须恢复这些寄存器的值。
- 寄存器 r12 是内部调用暂时寄存器 ip。它在过程链接胶合代码（例如，交互操作胶合代码）中用于此角色。在过程调用之间，可以将它用于任何用途。被调用函数在返回之前不必恢复 r12。
- 寄存器 r13 是栈指针 sp。它不能用于任何其它用途。sp 中存放的值在退出被调用函数时必须与进入时的值相同。
- 寄存器 r14 是链接寄存器 lr。如果您保存了返回地址，则可以在调用之间将 r14 用于其它用途。
- 寄存器 r15 是程序计数器 PC。它不能用于任何其它用途。

3.2.2 寄存器名称

表 3-1 列出处理器寄存器的定义角色和相关名称。这些名称及其同义词预先定义在汇编程序中。生成汇编程序语言时，编译程序使用特定名称和基本寄存器名。

表 3-1 ATPCS 中的寄存器角色和名称

寄存器	同义词	特定名称	过程调用标准中的角色
r15	-	PC	程序计数器。
r14	-	lr	链接寄存器。
r13	-	sp	栈指针。
r12	-	ip	内部过程调用暂时寄存器。
r11	v8	-	ARM 状态变量寄存器 8。
r10	v7	sl	ARM 状态变量寄存器 7。栈检查变体中的栈限制指针。
r9	v6	sb	ARM 状态变量寄存器 6。RWPI 变体中的静态基址。
r8	v5	-	ARM 状态变量寄存器 5。
r7	v4	-	变量寄存器 4。
r6	v3	-	变量寄存器 3。
r5	v2	-	变量寄存器 2。
r4	v1	-	变量寄存器 1。
r3	a4	-	自变量 / 结果 / 暂时寄存器 4。
r2	a3	-	自变量 / 结果 / 暂时寄存器 3。
r1	a2	-	自变量 / 结果 / 暂时寄存器 2。
r0	a1	-	自变量 / 结果 / 暂时寄存器 1。

另外，s0-s31、d0-d15 和 f0-f31 是浮点协处理器中寄存器的预定义名称。有关详细信息请参阅第 3-18 页的 *VFP 体系结构*和第 3-20 页的 *FPA 体系结构*。

3.3 栈

此部分描述在基本标准中如何使用栈。另请参阅第 3-11 页的*栈限制检查*。

ATPCS 指定：

- 完整的降序栈；
- 所有外部接口的 8 字节栈对齐。

3.3.1 栈术语

ATPCS 中使用以下栈相关术语：

栈指针 写入（推入）栈的最后一个值的地址。

栈基址 栈顶的地址，栈从此处向下增长。栈实际使用的最高位置是栈基址*下面*的第一个字。

栈限制 允许当前进程使用的栈的最低地址。

已使用栈 栈基址和栈指针之间的内存区。它包括栈指针，但不包括栈基址。

未使用栈 栈指针和栈限制之间的内存区。它包括栈限制，但不包括栈指针。

栈框架 函数分配的栈内存区，用于保存寄存器和存放局部变量。

进程可能有（或可能没有）访问栈基址和栈限制当前值的权限。

中断处理程序可以使用被它中断的进程的栈。在此情况下，程序员负责确保不超出栈限制。

第 3-7 页的图 3-1 显示栈内存布局。

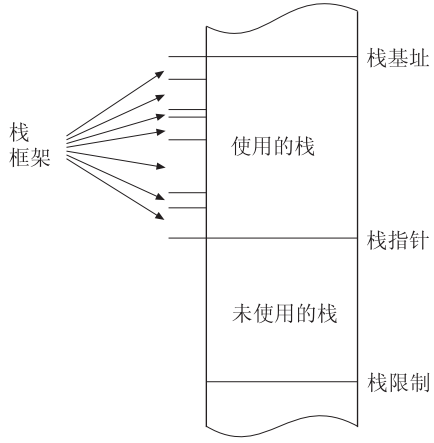

图 3-1 栈内存布局

3.3.2 栈展开

编译的目标文件总是包含 DWARF2 调试框架信息。在调试期间，调试程序使用此信息在必要时展开栈。这使您可以在调试程序中回溯栈

在汇编语言中，您负责使用 FRAME 指令描述栈框架。当使用 -g 汇编时，汇编程序使用这些描述生成 DWARF2 调试框架信息。请参阅 *编写 ARM 和 Thumb 汇编语言*和 *RealView 编译工具 2.0 版汇编程序指南* 中的*指令参考*一章。

3.3.3 8 字节对齐

对于一些系统中的多个传输， 8 字节对齐地址可以改进内存访问速度。对于 ARMv5TE 和其后的新处理器中的 LDRD 和 STRD 指令，需要 8 字节对齐。

编译程序生成的目标文件在所有外部接口保留 8 字节对齐栈。编译程序设置编译属性，向链接程序指示此特征。

要在汇编语言中符合 ATPCS，除非目标文件不包含外部调用，否则必须：

- 确保在所有外部接口保留了 8 字节对齐栈。（在代码入口和代码中的任何外部调用之间，栈指针必须总是移动偶数个字。）
- 使用 PRESERVE8 指令通知链接程序，已保留了 8 字节对齐（请参阅 *RealView 编译工具 2.0 版汇编程序指南*中的*指令参考*一章）。

3.4 参数传递

有可变个数自变量的函数是 *variadic*。有固定个数自变量的函数是 *nonvariadic*。向 variadic 和 nonvariadic 函数传递参数有不同的规则。

此部分描述基本标准。有关浮点选项的更多信息，请参阅第 3-17 页的*浮点选项*。

3.4.1 Nonvariadic 函数

参数值以下列方式传递到 nonvariadic 函数：

1. 前面的整型自变量按顺序分配给 r0-r3 （请参阅*长整数分配*）。
2. 其余参数按顺序分配给栈 （请参阅*长整数分配*）。

警告

栈访问时将显著增加代码大小并降低执行速度。尽可能将参数个数限制为少于 5 个。

长整数分配

超过 32 位的整型参数 （例如， `long long` 型）有 8 字节对齐。传递 `long long` 型参数时，将它分配给寄存器 r2 和 r3，或者分配给栈。

浮点数分配

如果系统有浮点硬件，则 FP 参数如下列方式分配给 FP 寄存器：

1. 按次序检查每个 FP 参数。
2. 对于每个参数，检查可用的 FP 寄存器组。
3. 如果有一个可用，则将编号最低、尺寸适合于参数的相邻 FP 寄存器组分配给参数。

3.4.2 Variadic 函数

参数值在整型寄存器 a1-a4 中、必要时在栈中传递到 variadic 函数（a1-a4 是 r0-r3 的同义词）。

使用的字顺序如同参数值存储在连续内存字中，然后传输到：

1. a1-a4，首先是 a1。
2. 栈，首先是最低地址。（这表示它们以相反的顺序推入栈中。）

备注

因此，浮点值可能在整型寄存器、栈中传递，或者分在整型寄存器和栈中传递。

3.4.3 结果返回

函数可以：

- 在 a1 中返回单字整型值。
- 在 a1-a2、 a1-a3 或 a1-a4 中返回双字或四字整型值。
- 在 f0、 d0 或 s0 中返回浮点值。
- 在 f0-f*N* 或 d0-d*N* 中返回复合浮点值 （如 `complex`）。 *N* 的最大值取决于所选的浮点结构 （请参阅第 3-17 页的*浮点选项*）。
- 较长的值必须在内存中间接返回。

3.5 栈限制检查

除非可以在设计阶段精确地计算出整个程序所需的最大栈内存量，否则请选择*软件栈限制检查*选项 (-apcs /swst)。

仅当可以在设计阶段精确地计算出整个程序所需的最大栈内存量时，才选择*不要软件栈限制检查*选项 (-apcs /swst)。这是默认选项。

可以编写汇编代码，使栈限制检查不相关。文件中的代码可能不需要栈限制检查，但仍然与其它用 -apcs /swst 或默认的 -apcs /noswst 汇编的代码兼容。在此情况下，请使用*软件栈限制检查不适用*选项 (-apcs /swstna)。

3.5.1 栈限制检查代码规则

在 ATPCS 的栈限制检查变体中：

- *栈限制指针* (sl) 必须指向栈中最低可用地址之上至少 256 字节处。

  备注

  如果中断处理程序可以使用 “用户” 模式栈，则必须在 s1 和栈中最低可用地址之间，在 256 字节之外为它分配足够的空间。

- 不能用选择了栈限制检查进行编译或汇编的代码改变 s1。s1 由运行时支持的代码改变。
- 存放在*栈指针* (sp) 中的值必须总是大于或等于 sl 中的值。

3.5.2 栈限制检查的寄存器用法

不得在选择了栈检查选项进行汇编或编译的函数中改变 r10 或恢复它。寄存器 r10 是栈限制指针 sl。

在所有其它方面，寄存器的用法相同，不论使用或不使用栈限制检查 （请参阅第 3-4 页的*寄存器角色和名称*）。

3.5.3 C 和 C++ 中的栈检查

如果选择软件栈限制检查 (-apcs /swst) 选项，则编译程序生成执行栈检查的对象代码。

3.5.4 汇编语言中的栈检查

如果您对汇编代码选择了软件栈检查 (/swst) 选项，则您负责编写执行栈检查的代码。

*叶函数*是不调用任何其它子程序的函数。

必须考虑以下情况：

- *使用少于 256 字节栈的叶函数*；
- *使用少于 256 字节栈的非叶函数*；
- 第 3-13 页的*使用超过 256 字节栈的函数*。

因此，叶函数包含的函数中每个调用都是尾调用。

使用少于 256 字节栈的叶函数

使用少于 256 字节栈的叶函数不需要检查栈限制。这是上述规则的结果 （请参阅第 3-11 页的*栈限制检查代码规则*）。

因此，叶函数可以是总计栈使用量少于 256 字节的多个函数的组合。

使用少于 256 字节栈的非叶函数

凋用都是尾调用

使用少于 256 字节栈的非叶函数可以使用如下的限界检查序列：

```
SUB     sp, sp, #size          ; ARM code version
CMP     sp, sl
BLLO    __ARM_stack_overflow
```

或用 Thumb 代码：

```
ADD     sp, #-size             ; Thumb code version
CMP     sp, sl
BLLO    __Thumb_stack_overflow
```

——— 备注 ———

名称 __ARM_stack_overflow 和 __Thumb_stack_overflow 是说明性的，不对应于任何实际的实现。

使用超过 256 字节栈的函数

在此情况下，必须使用如下的序列向限界检查代码*提供*一个新 sp 值。

```
SUB     ip, sp, #size           ; ARM code version
CMP     ip, sl
BLLO    __ARM_stack_overflow
```

或用 Thumb 代码：

```
LDR     r7, #-size              ; Thumb code version
ADD     r7, sp
CMP     r7, sl
BLLO    __Thumb_stack_overflow
```

这是必需的，以确保 sp 不小于栈中的最低可用地址。

备注

名称 `__ARM_stack_overflow` 和 `__Thumb_stack_overflow` 是说明性的，不对应于任何实际的实现。

3.6 只读位置无关

如果程序的所有只读段都是位置无关的，则该程序是*只读位置无关的* (ROPI)。

ROPI 段通常是*位置无关代码* (PIC)，但也可能是只读数据或 PIC 和只读数据的组合。

选择 ROPI 选项可以避免必须将代码载入特定的内存位置。这对以下函数尤其有用：

- 根据运行时事件载入的函数；
- 与不同环境中的其它函数的不同组合一起载入内存的函数；
- 在执行期间映射到不同地址的函数。

3.6.1 ROPI 的寄存器用法

不论使用或未使用 ROPI，寄存器的用法相同 （请参阅第 3-4 页的*寄存器角色和名称*）。

3.6.2 编写 ROPI 代码

当编写 ROPI 代码时：

- ROPI 段中的代码对同一个 ROPI 段的符号的每个引用必须是相对于 PC 的。 ATPCS 不为只读段定义任何其它基本寄存器。一个 ROPI 段中的项的地址不能分配给不同 ROPI 段中的项。
- ROPI 段中的代码对不同 ROPI 段的符号的每个引用必须是相对于 PC 的。两个段必须是互相固定相对的。
- 来自 ROPI 段的其它引用必须指向：
  - 绝对地址；
  - 对可写数据的 sb 相对引用 （请参阅第 3-15 页的*读写位置无关*）。
- 引用 ROPI 段的符号的可读写字在 ROPI 段移动时必须进行调整。

3.7 读写位置无关

如果程序的所有读写段都是位置无关的，则该程序是*读写位置无关的* (RWPI)。

RWPI 段通常是*位置无关数据* (PID)。

选择 RWPI 选项可以避免将数据固定在内存中特定的位置。这对于必须为可重入函数而多次实例化的数据尤其有用。

3.7.1 可重入函数

可重入函数可以同时成为多个进程的*线程*。每个进程有其各自的该函数读写段副本。每个副本用不同的静态基本寄存器编址。

3.7.2 RWPI 的寄存器用法

寄存器 r9 是静态基址 sb。当调用任何外部可见函数时，它必须指向适当的静态数据段基址。

在不使用 sb 的函数中，可以将 r9 用于其它用途。如果这样做，您必须在进入函数时保存 sb 的内容，并且在退出之前恢复它。在调用任何外部函数之前，也必须恢复它。

在所有其它方面，不论使用或未使用 RWPI，寄存器的用法相同（请参阅第 3-4 页的*寄存器角色和名称*）。

3.7.3 位置无关数据编址

RWPI 段可以在首次使用之前重新定位。 RWPI 段中符号的地址如下计算：

1. 链接程序从段中一个固定位置计算只读偏移。按照惯例，该固定位置是程序的最低地址 RWPI 段的第一个字节。
2. 运行时，它作为偏移添加到静态基本寄存器 sb 的内容上。

3.7.4 编写汇编语言 RWPI

通过 sb 值加上一个固定 （只读）偏移，构建从只读段到 RWPI 段的引用 （请参阅 *RealView 编译工具 2.0 版汇编程序指南*中*指令参考*一章里的 DCD0）。

3.8 ARM 状态和 Thumb 状态之间的交互操作

当编译或汇编代码时，可以选择 /interwork 选项来满足以下需求：

- ARM 函数能够返回到 Thumb 状态调用者；
- Thumb 函数能够返回到 ARM 状态调用者；
- 当从 ARM 调用 Thumb 或从 Thumb 调用 ARM 时，链接程序提供用于改变状态的代码。

当编译或汇编代码时，在以下情况下，应选择 /nointerwork 选项：

- 系统未使用 Thumb ；
- 您提供用于处理所有状态改变的汇编程序代码。

默认设置是：

- /interwork，如果编译或汇编 ARM v5T 处理器代码；
- /nointerwork，用于其它情况。

如果选择交互操作选项，则可以调用不同模块中的函数，无需考虑它所使用的指令。如果有必要，链接程序将插入一个交互操作调用胶合代码，或者修补调用点。这适用于已编译或已汇编的代码。

有关详细信息，请参阅第 4 章 *ARM 和 Thumb 交互操作*。

3.8.1 交互操作的寄存器用法

不论使用或未使用交互操作，寄存器的用法相同 （请参阅第 3-4 页的*寄存器角色和名称*）。

3.9 浮点选项

ATPCS 支持以下浮点硬件体系结构和指令集：

- VFP 体系结构 （请参阅第 3-18 页的 *VFP 体系结构*）。
- FPA 体系结构 （请参阅第 3-20 页的 *FPA 体系结构*）。这只是为了向后兼容。

一个体系结构的代码不能用于其它体系结构。

ARM 编译程序和汇编程序有以下浮点选项：

- `-fpu vfp`；
- `-fpu vfpv1`；
- `-fpu vfpv2`；
- `-fpu fpa`；
- `-fpu softvfp`；
- `-fpu softvfp+vfp`；
- `-fpu softvfp+vfpv2`；
- `-fpu softfpa`；
- `-fpu none`。

如果目标系统有浮点硬件，请选择 `vfp`、`softvfp+vfp` 或 `fpa`。

如果系统有浮点硬件，并且想要从 Thumb 代码使用浮点库函数，请使用 `softvfp+vfp` 或 `softvfp+vfpv2`。

如果目标系统没有浮点硬件，则：

- 如果需要与 FPA 系统或 SDT 下生成的对象兼容，请选择 `softfpa`。
- 如果编译或汇编的模块未使用浮点算法，并且需要与 FPA 和 VFP 系统兼容，请选择 `none`。
- 否则，请选择 `softvfp`。这是默认选项。

另请参阅第 3-21 页的*没有浮点硬件*。

3.9.1 VFP 体系结构

VFP 体系结构有十六个双精度寄存器 d0-d15。每个双精度寄存器可以用作两个单精度寄存器。作为单精度寄存器，它们称为 s0-s31。例如， d5 相当于 s10 和 s11。

VFP 体系结构不支持扩展精度。

向量和标量模式

VFP 体系结构有以下操作模式：

- 标量模式；
- 向量模式。

ATPCS 仅适用于标量模式操作。进入和退出符合 ATPCS 的公开可见函数时，向量长度和向量跨度都必须设为 1。

VFP 的寄存器用法

前 8 个双精度寄存器 d0-d7 可以用于：

- 将浮点值传递到函数；
- 将浮点值传出函数；
- 函数内的暂时寄存器。

每个双精度寄存器可以存放一个双精度值或两个单精度值。通过将每个值依次分配给下一个适当类型的空闲寄存器，将浮点自变量值分配给浮点寄存器。

图 3-2 显示参数值分配给寄存器，其中传递值 1.0 （双精度）、2.0 （双精度）、3.0 （单精度）、4.0 （双精度）、5.0 （单精度）和 6.0 （单精度）。

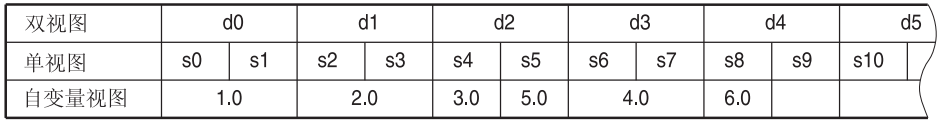

图 3-2 将参数值分配给寄存器

要符合 ATPCS，如果在函数中使用寄存器 d8-d15，则必须在进入时保存它们的值，并且在退出之前恢复它们。可以使用单个 FSTMX 指令保存它们，使用单个 FLDMX 指令恢复它们。它们以位模式保存和恢复，无需解释为单精度或双精度数。保存的 *N* 个单精度值占用 *N*+1 个字。

VFP 值格式

单精度和双精度值遵循 IEEE 754 标准格式。双精度值视为真 64 位值：

- 在小端模式中，包含指数的双字双精度值的更有效字有较高地址；
- 在大端模式中，更有效的字有较低地址。

备注

VFP 中的小端双精度值是纯小端的。这与 FPA 体系结构不同。

VFP 和 FPA 体系结构中的大端双精度值是相同的，都是纯大端的。

IEEE 舍入模式和异常启用标志

在进入或退出符合标准的函数时， ATPCS 不对这些状态指定任何约束。

3.9.2 FPA 体系结构

FPA 体系结构有 8 个浮点寄存器 f0-f7。每个寄存器可以存放单精度、双精度或扩展精度值。

FPA 的寄存器用法

前四个浮点寄存器 f0-f3 可以用于：

- 将浮点值传递到函数；
- 将浮点结果传出函数；
- 函数内的暂存寄存器。

要符合 ATPCS，如果在函数中使用浮点寄存器 f4-f7，则必须在进入时保存它们的值，并且在退出之前恢复它们。可以使用单个 `SFM` 指令保存它们，使用单个 `LFM` 指令恢复它们。每个保存的值占用三个字。

FPA 值格式

单精度和双精度值遵循 IEEE 754 标准格式。包含指数的浮点值的最高有效字有最低内存地址。无论字中的字节顺序是大端或小端，都是一样的。

备注

小端双精度值既不是纯小端的，也不是纯大端的。

IEEE 舍入模式和异常启用标志

在进入或退出符合标准的函数时， ATPCS 不对这些状态指定任何约束。

3.9.3 没有浮点硬件

softvfp 和 softfpa 之间的唯一不同点是小端模式中双精度值的字顺序 （请参阅第 3-18 页的 *VFP 体系结构* 和第 3-20 页的 *FPA 体系结构*）。

如果指定 -fpu none，则不能使用浮点值。

softvfp 和 softfpa 的寄存器用法

每个浮点自变量转换为一个或两个整型字的位模式，如同保存在内存中一样。如第 3-9 页的*参数传递*中所述传递结果整型值。

单精度浮点结果以位模式在 r0 中返回。

双精度浮点结果在 r0 和 r1 中返回。 r0 包含的字对应于内存中表示的值的低地址字。

3.9.4 softvfp+vfp 和 softvfp+vfpv2

Thumb 代码不能在浮点寄存器中传递浮点值，因为 Thumb 没有协处理器指令。

如果有 VFP 协处理器，并且想要从 Thumb 代码使用浮点函数，请选择 -fpu softvfp+vfp 或 -fpu softvfp+vfpv2 选项。

这指示编译程序使用与 -fpu softvfp 相同的参数传递规则生成代码。C 库浮点函数使用 ARM 状态的 VFP 指令。

第 4 章
ARM 和 Thumb 交互操作

本章介绍为实现 Thumb 指令集的处理器编写代码时如何在 ARM 状态和 Thumb 状态之间变化。它包含以下几个部分：

- 第 4-2 页的*关于交互操作*;
- 第 4-5 页的*汇编语言交互操作*;
- 第 4-10 页的 *C 和 C++ 交互操作和胶合代码*;
- 第 4-14 页的*使用胶合代码的汇编语言交互操作*。

4.1 关于交互操作

根据需要可将 ARM 和 Thumb 代码混合，其条件是代码符合 *ARM-Thumb 过程调用标准* (ATPCS) 的要求。ARM 编译程序总是生成符合此标准的代码。编写 ARM 汇编语言模块时，必须确保代码的一致性。

ARM 链接程序检测何时从 Thumb 状态下调用 ARM 函数，或者何时从 ARM 状态调用 Thumb 函数。ARM 链接程序通过改变调用和返回指令或插入名为*胶合代码*的小代码段，按照需要更改处理器的状态。

ARM 体系结构 5T 版提供更改处理器状态而不使用任何额外指令的方法。通常在 ARM 体系结构 5T 版处理器上交互操作无需任何成本。

如果将几个源文件链接在一起，所有的文件必须使用兼容的 ATPCS 选项如果检测到不兼容的选项，链接程序会产生一条错误信息。

4.1.1 何时使用交互操作

在为能够支持 Thumb 的 ARM 处理器编写代码时，可能多数的程序需要在 Thumb 状态下运行。这会使代码密度最佳。使用 8 位或 16 位宽存储器，它也使其性能最佳。但是，可能需要部分在 ARM 状态下运行的应用程序，其原因如下:

速度 在应用程序的某些部分，速度可能是关键的。这些段在 ARM 状态下运行时的效率可能比在 Thumb 状态下运行时更高。在某些情况下，单条 ARM 指令可以比等价的 Thumb 指令执行更多操作

有些系统包括少量的快速 32 位存储器。 ARM 代码可在其中运行，而无需从 8 位或 16 位存储器存取每一条指令的开销。

功能

Thumb 指令不如其等价的 ARM 指令灵活。在 Thumb 状态下有些操作是不可能执行的。例如，不能启用或禁用中断、或者访问协处理器。需要更改状态，才能执行这些操作。

异常处理

发生处理器异常时处理器自动进入 ARM 状态。这意味着异常处理程序的第一部分必须用 ARM 指令进行编码，即使它重新进入 Thumb 状态执行异常的主处理。在此类处理的末尾，处理器必须返回 ARM 状态，以便从处理程序返回主应用程序。

独立的 Thumb 程序

能够支持 Thumb 的 ARM 处理器总是以 ARM 状态启动。要在调试程序下运行简单的 Thumb 汇编语言程序，请添加一个 ARM 头文件，执行对 Thumb 状态的更改，然后调用主 Thumb 例程。请参阅第 4-6 页的 *ARM 头文件示例*的一个示例。

4.1.2 使用 /interwork 选项

ARM 编译程序和汇编程序中提供 `--apcs /interwork` 选项。如果设置此选项：

- 编译程序或汇编程序将交互操作属性记录在目标文件中。
- 链接程序为子例程入口提供交互操作胶合代码。
- 在汇编语言中，必须编写返回调用程序指令集状态的函数退出代码，例如 `BX lr`。
- 在 C 或 C++ 中，编译程序创建返回调用程序指令集状态的函数退出代码。
- 在 C 或 C++ 中，编译程序将 `BX` 指令用于间接或虚拟调用。

如果目标文件包括以下内容，请使用 `/interwork` 选项：

- 可能需要返回到 ARM 代码的 Thumb 子例程；
- 可能需要返回到 Thumb 代码的 ARM 子例程；
- 可能间接或虚拟调用 ARM 代码的 Thumb 子例程；
- 可能间接或虚拟调用 Thumb 代码的 ARM 子例程。

备注

如果某个模块包含带有 `#pragma arm` 或 `#pragma thumb` 标记的函数，通常必须使用 `--apcs /interwork` 对该模块进行编译。这确保了可以从其它状态 (ARM 或 Thumb) 成功调用该函数。

否则，不必使用 /interwork 选项。例如，目标文件中可能包括以下不需要 /interwork 选项的内容：

- 可由异常中断的 Thumb 代码。异常强制处理器进入 ARM 状态，因而不需要胶合代码。
- 可处理从 Thumb 代码发生的异常的处理代码。其返回不需要胶合代码。
- 调用其它文件 ARM 子例程的 Thumb 代码（其交互操作返回序列属于被调用程序，而不是调用程序）。
- 调用其它文件 Thumb 子例程的 ARM 代码（其交互操作返回序列属于被调用程序，而不是调用程序）。

4.1.3 检测交互操作调用

在被调用例程不是为了交互操作而编译的情况下，如果检测到直接的 ARM/Thumb 交互操作调用，则链接程序生成一个错误。为了交互操作，必须重新编译被调用的例程。

例如，示例 4-1 说明了当第 4-11 页的示例 4-3 中的 ARM 例程在没有 --apcs /interwork 选项的情况下被编译和链接时产生的错误。

示例 4-1

```
Error: L6239E: Cannot call ARM symbol 'arm_function' in non-interworking object
armsub.o from THUMB code in thumbmain.o(.text)
```

这些类型错误表示，在从目标模块目标到例程符号的过程中，ARM 到 Thumb 或 Thumb 到 ARM 的交互操作调用被检测出来，但是，被调用的例程尚未为交互操作而被编译。必须重新编译包含该符号的模块，并指定 --apcs /interwork 选项。

4.2 汇编语言交互操作

在汇编语言源文件中，可以有多个区域（对应于 ELF 段）每个区域都可包括 ARM 指令、 Thumb 指令或两者兼而有之。

可使用链接程序来修复对通过调用程序使用不同指令集的例程的调用和来自该例程的返回。要执行此操作，请使用 BL 来调用例程（参阅第 4-14 页的*使用胶合代码的汇编语言交互操作*）。

如果您愿意，可以编写代码，显式地更改指令集。在某些情况下，通过执行此操作可编写更小更快的代码。

以下指令执行处理器状态的更改：

- BX，请参阅*跳转和交换指令*；
- BLX、LDR、LDM 和 POP（仅限于 ARM 体系结构 5 版及更高版本），请参阅第 4-8 页的 *ARM 体系结构 5T 版*。

以下指令指示汇编程序从相应的指令集汇编指令（参阅第 4-6 页的*更改汇编程序模式*）：

- CODE16；
- CODE32。

4.2.1 跳转和交换指令

BX 指令跳转到指定寄存器包含的地址。跳转地址 0 位的值确定是否继续在 ARM 状态或 Thumb 状态下执行。有关 ARM 体系结构 5 版提供的附加指令，请参阅第 4-8 页的 *ARM 体系结构 5T 版*。

可以按照这种方式使用地址的 0 位，原因是：

- 所有的 ARM 指令都是字对齐的，所以任何 ARM 指令地址的 0 位和 1 位都未被使用；
- 所有的 Thumb 指令都是半字对齐的，所以任何 Thumb 指令地址的 0 位都未被使用。

语法

BX 的语法为以下之一：

Thumb BX R*n*
ARM BX{*cond*} R*n*

其中：

Rn 是 r0 到 r15 范围中的一个寄存器，包含所要跳转的目标地址。此寄存器中 0 位的值确定处理器的状态：

- 如果 0 位已设置，则在 Thumb 状态下执行跳转地址处的指令；
- 如果 0 位已清除，则在 ARM 状态下执行跳转地址处的指令。

cond 是一个可选的条件码。仅 BX 的 ARM 版能被有条件地执行。

4.2.2 更改汇编程序模式

ARM 汇编程序可汇编 Thumb 代码和 ARM 代码。在默认情况下，它汇编 ARM 代码，除非用 -16 选项调用。

因为所有能够支持 Thumb 的 ARM 处理器均以 ARM 状态启动，所以，必须使用 BX 指令跳转和交换到 Thumb 状态，然后使用 CODE16 指令指示汇编程序汇编 Thumb 指令。使用相应的 CODE32 指令指示汇编程序返回到正在汇编的 ARM 指令。

有关这些指令的更多信息，请参阅 *RealView Compilation Tools 2.0 版汇编程序指南*。

4.2.3 ARM 头文件示例

第 4-7 页的示例 4-2 包括四段代码。第一段实现将处理器更改到 Thumb 状态所需的 ARM 代码小头文件段。

头文件代码使用：

- 装载跳转地址并设置最低有效位的 ADR 伪指令。该 ADR 伪指令通过用 pc+offset+1 装载 r0 来生成地址。有关 ADR 伪指令的更多信息，请参阅 *RealView Compilation Tools 2.0 版汇编程序指南*。
- 跳转到 Thumb 代码并更改处理器状态的 BX 指令。

该模块的第二段 （标记为 ThumbProg），由 CODE16 指令加上前缀，而该指令指示汇编程序将以下代码视为为 Thumb 代码。Thumb 代码将两个寄存器的内容合在一起。

处理器被更改回 ARM 状态。代码再次使用 ADR 指令来获取标签的地址，但是，这次最低有效位没有被使用。BX 指令更改其状态。

第三段代码将两个寄存器的内容合在一起。

最后一段标记为 stop 的代码使用半主机的 SWI 来报告应用程序的正常退出。有关半主机的更多信息，请参阅 *RealView Compilation Tools 2.0 版编译程序和库指南*。

备注

该 Thumb 半主机 SWI 的编号不同于 ARM 半主机 SWI （是 0xAB 而不是 0x123456）。

示例 4-2

```
     AREA     AddReg,CODE,READONLY         ; Name this block of code.
     ENTRY                      ; Mark first instruction to call.
main
     ADR r0, ThumbProg + 1      ; Generate branch target address
                                ; and set bit 0, hence arrive
                                ; at target in Thumb state.
     BX r0                      ; Branch exchange to ThumbProg.

     CODE16                     ; Subsequent instructions are Thumb code.
ThumbProg
     MOV r2, #2                 ; Load r2 with value 2.
     MOV r3, #3                 ; Load r3 with value 3.
     ADD r2, r2, r3             ; r2 = r2 + r3
     ADR r0, ARMProg
     BX r0
     CODE32                     ; Subsequent instructions are ARM code.
ARMProg
     MOV r4, #4
     MOV r5, #5
     ADD r4, r4, r5

stop MOV r0, #0x18              ; angel_SWIreason_ReportException
     LDR r1, =0x20026           ; ADP_Stopped_ApplicationExit
     SWI 0x123456               ; ARM semihosting SWI

     END                        ; Mark end of this file.
```

编译示例

编译并执行示例：

1. 使用任何文本编辑器输入代码并将文件保存为 addreg.s。
2. 在命令提示行键入 armasm -g addreg.s 来汇编源文件。
3. 键入 armlink addreg.o -o addreg 链接该文件。
4. 将 ELF/DWARF2 兼容调试器与相应调试目标配合使用，运行映象。如果每次一条指令地执行程序，会看到处理器进入 Thumb 状态。参阅所使用的调试器用户文档，找出指示此变更的方法。

4.2.4 ARM 体系结构 5T 版

在 ARM 体系结构 5T 版及更高版本中：

- 以下额外的交互操作指令可用：

  BLX *address*

  处理器执行一个与 PC 相关的跳转，转到含有链接的 *address* 并更改状态。*address* 必须在 ARM 代码下 PC 的 32MB 之内，在 Thumb 代码下 PC 的 4MB 之内。

  BLX *register*

  处理器执行含有链接的跳转，转到指定寄存器包含的地址。[0] 位的值确定新的处理器状态。

  在每种情况下，lr 的 [0] 位均设置成 CPSR 中 Thumb 位的当前值。这意味着返回指令可以自动返回正确的处理器状态。

- 如果 LDR、LDM 或 POP 载入 PC，它们将 CPSR 中的 Thumb 位设置成载入 PC 的值的 [0] 位。可以使用此项来更改指令集。对于从子例程返回，这尤其有用。相同的返回指令可以返回到 ARM 调用程序，也可以返回到 Thumb 调用程序。

有关更多信息，请参阅 *RealView Compilation Tools 2.0 版编译程序指南*和 *ARM 体系结构参考手册*。

4.2.5 Thumb 代码中的标签

链接程序区别引用以下内容的标签：

- ARM 指令；
- Thumb 指令；
- 数据。

当链接程序重定位一个引用 Thumb 指令的标签的值时，它将最低有效位设置成该重定位值。这意味着到标签的跳转可自动选择相应的指令集。如果将以下任何指令用于跳转，这都可实现：

- ARM 体系结构 4T 版中的 BX；
- 体系结构 5T 版及更高版本中的 BX、BLX 或 LDR。

在早于 1.2 和 SDT 的 ADS 版本中，将 Thumb 代码中的数据标记上 DATA 指令是必要的。现在，这不再必要了。

4.3 C 和 C++ 交互操作和胶合代码

可以自由地混合为 ARM 和 Thumb 而编译的 C 和 C++ 代码，但是，在 ARM 体系结构 4T 版中，ARM 和 Thumb 代码之间需要一小段称为*胶合代码*的代码，才能执行状态更改。ARM 链接程序在检测到交互操作调用时生成这些交互操作胶合代码。

4.3.1 为交互操作编译代码

--apcs /interwork 编译程序选项使 ARM 编译程序能够编译包括例程的 C 和 C++ 模块，而该例程可以由其它处理器状态编译的例程进行调用：

```
armcc --thumb --apcs /interwork
armcc --arm --apcs /interwork
armcc --cpp --thumb --apcs /interwork
armcc --cpp --arm --apcs /interwork
```

备注

--arm 为默认选项。

为 ARM 体系结构 4T 版上交互操作而编译的模块生成稍大的代码，通常对于 Thumb 大 2%，对于 ARM 大 1% 以下。对于 ARM 体系结构 5 版没有区别。

在叶函数 （其函数体不包含函数的调用）中，由编译程序生成的代码的唯一更改是用 BX lr 替换 MOV pc,lr。MOV 指令不造成必要的状态更改。

在 Thumb 模式下为 ARM 体系结构 4T 版而编译的非叶函数中，编译程序必须替换单个指令 （举例说明）：

```
POP  {r4,r5,pc}
```

成为序列：

```
POP  {r4,r5}
POP  {r3}
BX   r3
```

这对性能有一点影响。为交互操作编译所有源模块，除非能够确保它们永远不会用于交互操作。

--apcs /interwork 选项还为编译模块的目标代码区域而设置交互操作属性。链接程序检测此属性并插入相应的胶合代码。

备注

为交互操作而编译的 ARM 代码只能用于 ARM 体系结构 4T 版及更高版本，因为早期的处理器不实现 BX 指令。

使用 armlink -info veneers 选项查找胶合代码所占空间的大小。

C 交互操作示例

示例 4-3 说明执行 ARM 子例程交互操作调用的一个 Thumb 例程。ARM 子例程的调用对 Thumb 库中的 printf() 进行交互操作调用。这两个模块作为 thumbmain.c 和 armsub.c，在 *Examples_directory*\interwork 中提供。

示例 4-3

```
    /**********************
    *       thumbmain.c  *
    **********************/
    #include <stdio.h>
    extern void arm_function(void);
int main(void)
    {
         printf("Hello from Thumb World\n");
         arm_function();
         printf("And goodbye from Thumb World\n");
         return (0);
    }

    /**********************
    *        armsub.c    *
    **********************/
    #include <stdio.h>
    void arm_function(void)
    {
         printf("Hello and Goodbye from ARM world\n");
    }
```

编译并链接这些模块：

1. 在系统提示行键入 armcc --thumb -c --apcs /interwork -o thumbmain.o thumbmain.c，为交互操作编译 Thumb 代码。
2. 在系统提示行键入 armcc -c --apcs /interwork -o armsub.o armsub.c，为交互操作编译 ARM 代码。

3. 键入 armlink thumbmain.o armsub.o -o thumbtoarm.axf 链接目标文件。
   或者，查看交互操作胶合代码（示例 4-4）的大小：
   armlink armsub.o thumbmain.o -o thumbtoarm.axf -info veneers

示例 4-4

```
Adding Veneers to the image

    Adding TA veneer (4 bytes, Inline) for call to 'arm_function' from thumbmain.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '_printf' from armsub.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_ll_cmpge' from __vfpntf.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_ll_neg' from __vfpntf.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_fp_display_gate' from __vfpntf.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_ll_ushift_r' from __vfpntf.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_ll_cmpu' from __vfpntf.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_ll_udiv10' from __vfpntf.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_udiv10' from __vfpntf.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '__rt_lib_init' from kernel.o(.text).
    Adding AT veneer (12 bytes, Long) for call to '__rt_lib_shutdown' from kernel.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_memclr_w' from stdio.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_raise' from stdio.o(.text).
    Adding TA veneer (8 bytes, Short) for call to '__rt_exit' from exit.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__user_libspace' from free.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_fp_init' from lib_init.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__heap_extend' from malloc.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '__raise' from rt_raise.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_errno_addr' from ftell.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '_no_fp_display' from printf2.o(x$fpl$printf2).

20 Veneer(s) (total 108 bytes) added to the image.
```

4.3.2 交互操作的基本规则

下列规则应用于应用程序之内的交互操作：

- 必须使用 --apcs /interwork 命令行选项，编译任何包含可能返回其它指令集的 C 或 C++ 模块。
- 必须使用 --apcs /interwork 命令行选项，编译任何包含可能间接或虚拟调用其它指令集函数的 C 或 C++ 模块。
- 永远不要从其它状态的代码中间接调用非交互操作代码，如使用函数指针的调用。
- 如果输入对象包含 Thumb 代码，则链接程序选择 Thumb 运行时的库。这些为交互操作而编译。

  如果在链接程序命令行中显式地指定自己的库中的一个库，必须确保它是一个合适的交互操作库。

4.3.3 使用同一函数的两个副本

可以有两个名字相同的函数，一个编译成 ARM，另一个编译成 Thumb，但是，不建议这样做。在绝大多数情况下，这并不能明显地增加性能。

备注

两个版本的函数必须用 --apcs /interwork 选项进行编译。这是因为不能保证只从 Thumb 状态调用 Thumb 版本的函数和只从 ARM 状态调用 ARM 版本的函数。

链接程序启用重复定义，其条件是一个定义定义 Thumb 例程，另一个则定义 ARM 例程。

4.4 使用胶合代码的汇编语言交互操作

第 4-5 页的*汇编语言交互操作*中介绍的汇编语言 ARM/Thumb 交互操作方法，执行了所有必要的中间处理。对于链接程序插入交互操作胶合代码，没有任何要求。

本节介绍如何利用胶合代码进行：

- 汇编语言模块之间的交互操作；
- 汇编语言和 C 或 C++ 模块之间的交互操作。

4.4.1 使用胶合代码的汇编间交互调用

可编写汇编语言 ARM/Thumb 交互操作代码，使用链接程序生成的交互操作胶合代码。要执行此操作，需编写：

- 一个象任何非交互操作例程的调用例程，使用 `BL` 指令来进行调用。调用例程可通过 `--apcs /interwork` 或 `--apcs /nointerwork` 来进行汇编。
- 使用 `BX` 指令返回的被调用例程。被调用例程必须用 `--apcs /interwork` 来进行汇编。

通常只有在 ARM 体系结构 4T 版中，或调用程序和被调用程序分布广泛或在不同的区域中时，这才是必要的。在 ARM 体系结构 5T 版中，如果调用程序和被调用程序距离足够近，则无需胶合代码。

使用胶合代码的汇编语言交互操作的示例

示例 4-5 说明了将寄存器 r0 到 r2 分别设置为 1、2 和 3 的代码。寄存器 r0 和 r2 由 ARM 代码设置。r1 由 Thumb 代码设置。请注意：

- 该代码必须用 --apcs /interwork 选项来进行汇编；
- 使用 BX lr 指令从子例程中返回，而不是通常的 MOV pc,lr。

示例 4-5

```
    ; *****
    ; arm.s
    ; *****
    AREA     Arm,CODE,READONLY   ; Name this block of code.
    IMPORT     ThumbProg
    ENTRY                        ; Mark 1st instruction to call.
ARMProg
    MOV  r0,#1                   ; Set r0 to show in ARM code.
    BL   ThumbProg               ; Call Thumb subroutine.
    MOV  r2,#3                   ; Set r2 to show returned to ARM.
                                 ; Terminate execution.
    MOV  r0, #0x18               ; angel_SWIreason_ReportException
    LDR  r1, =0x20026            ; ADP_Stopped_ApplicationExit
    SWI  0x123456                ; ARM semihosting SWI
    END

    ; *******
    ; thumb.s
    ; *******
    AREA  Thumb,CODE,READONLY    ; Name this block of code.
    CODE16                       ; Subsequent instructions are Thumb.
    EXPORT ThumbProg
ThumbProg
    MOV  r1, #2                  ; Set r1 to show reached Thumb code.
    BX   lr                      ; Return to ARM subroutine.
    END                          ; Mark end of this file.
```

按照这些步骤编译并链接模块，然后检查交互操作胶合代码：

1. 键入 armasm arm.s 来汇编 ARM 代码。
2. 键入 armasm -16 --apcs /interwork thumb.s 来汇编 Thumb 代码。
3. 键入 armlink arm.o thumb.o -o count 来链接两个目标文件。

4. 将 ELF/DWARF2 兼容调试器与相应调试目标配合使用，运行映象。

4.4.2 使用胶合代码的 C、 C++ 和汇编语言交互操作

编译后在某一状态下运行的 C 和 C++ 代码，可以调用设计在其它状态下运行的汇编语言代码，反之亦然。要执行此操作，请编写调用例程作为非交互操作例程，并且在通过汇编语言进行调用时，使用 BL 指令来进行调用（参阅示例 4-6）。然后：

- 如果被调用例程是用 C 语言编写的，请用 --apcs /interwork 来进行汇编；
- 如果被调用例程是用汇编语言编写的，请用 --apcs /interwork 选项来进行汇编，用 BX lr 来返回。

备注

使用此方式的任何汇编语言或用户库代码，必须在相应处与 ATPCS 保持一致。

示例 4-6

```
    /**********************
    *       thumb.c      *
    **********************/
    #include <stdio.h>
    extern int arm_function(int);
     int main(void)
    {
         int i = 1;
         printf("i = %d\n", i);
         printf("And now i = %d\n", arm_function(i));
         return (0);
    }

    ; *****
    ; arm.s
    ; *****
    AREA  Arm,CODE,READONLY ; Name this block of code.
    EXPORT arm_function
arm_function
    ADD   r0,r0,#4          ; Add 4 to first parameter.
    BX    lr                ; Return
    END
```

按照这些步骤编译并链接模块：

1. 键入 `armcc --thumb -c --apcs /interwork thumb.c` 来汇编 Thumb 代码。
2. 键入 `armasm --apcs /interwork arm.s` 来汇编 ARM 代码。
3. 键入 `armlink arm.o thumb.o -o add` 来链接两个目标文件。
4. 将 ELF/DWARF2 兼容调试器与相应调试目标配合使用，运行映象。

第 5 章
混合使用 C、 C++ 和汇编语言

本章描述如何编写 C、C++ 和 ARM 汇编语言混合代码。还描述如何从 C 和 C++ 使用 ARM 内联和嵌入式汇编程序。它包含以下几个部分：

5.1 使用内联汇编程序

ARM 编译程序内建的内联汇编程序使您可以使用不能从 C 直接访问的目标处理器功能。例如：

- 饱和算法（请参阅 *RealView 编译工具 2.0 版汇编程序指南*）；
- 定制协处理器；
- PSR。

内联汇编程序支持与 C 和 C++ 非常灵活的交互。任何寄存器操作数都可以是任意的 C 或 C++ 表达式。内联汇编程序还扩展了复杂指令，并优化了汇编语言代码。

备注

如果默认情况下，或者用 `-O1` 或 `-O2` 编译程序选项启用了优化，则内联汇编语言将由编译程序进行优化。

ARM 代码内联汇编程序实现了 ARM 指令集中的大多数指令，包括一般协处理器指令、半字指令和长乘法。

备注

在 RVCT v2.0 中不支持 Thumb 代码内联汇编程序。

有关限制的信息，请参阅第 5-7 页的*内联汇编程序和 armasm 的不同点*。

内联汇编程序是一个高级汇编程序。它生成的代码与您所生成的代码并不总是完全一致。不能用它来生成比编译程序生成代码更有效的代码。应当使用 ARM 汇编程序 `armasm` 来实现此目的。

不支持 ARM 汇编程序 `armasm` 可以使用的某些低级功能，如不能通过写入 PC 进行跳转。

有关内联汇编程序的详细信息，请参阅 *RealView 编译工具 2.0 版编译程序和库指南*中关于内联和嵌入式汇编程序一章。

5.1.1 调用内联汇编程序

如何调用内联汇编程序取决于编译的是 C 还是 C++：

- 编译 C 时， ARM 编译程序通过 __asm 说明符支持内联汇编语言。
- 编译 C++ 时，ARM 编译程序通过 __asm 说明符和 asm 关键字支持内联汇编语言。

有关内联汇编程序的详细信息，请参阅 *RealView 编译工具 2.0 版编译程序和库指南*中关于内联和嵌入式汇编程序一章。

带有 __asm 说明符的内联汇编程序

编译 C 或 C++ 时，可以通过 __asm 汇编程序说明符调用内联汇编程序。说明符后面跟随有一列包含在大括号中的汇编程序指令。例如：

```
__asm
{
    instruction [; instruction]
    ...
    [instruction]
}
```

如果两条指令在同一行中，必须用分号将其分隔。如果一条指令占用多行，必须用反斜线符号（\）指定续行。可在内联汇编语言块内的任意位置处使用 C 或 C++ 注释。

可在任何可以使用 C 或 C++ 语句的地方使用 __asm 语句。

带有 asm 关键字的内联汇编程序

编译 C++ 时，ARM 编译程序支持 ISO C++ 标准中建议的 asm 语法，其限制是字符串文字必须是单个字符串。例如：

```
asm("instruction[;instruction]");
```

asm 语句必须在 C++ 函数内。字符串文字中不能包括注释。可在任何可以使用 C 或 C++ 语句的地方使用 asm 语句。

用内联汇编代码实现加法示例

示例 5-1 显示使用内联汇编代码将两个值相加的示例。将它与第 5-13 页的示例 5-4 中的嵌入式汇编函数进行比较。

示例 5-1 用内联汇编代码实现加法

```
#include <stdio.h>

void main() {
    int r0=12345;
    int r1=67890;
    int res;

    __asm {
        ADD res, r0, r1    // r0 and r1 are C/C++ variables (virtual
registers),
                           // not the physical registers
    }

    printf("12345 + 67890 = %d\n", res);
}
```

5.1.2 ARM 指令集

ARM 指令集在 *ARM 体系结构参考手册* 中描述。所有指令操作码和寄存器说明符都可以用小写或大写字母表达。

操作数表达式

任何寄存器或常数操作数可以是任意 C 或 C++ 表达式，因此可以读写变量。表达式必须是整型可分配的，即 char、short、int 或 long 型。对 char 和 short 型不执行符号扩展。您必须对这些类型明确执行符号扩展。编译程序可能会添加代码，以计算这些表达式并将它们分配给寄存器。

当操作数用作目标时，表达式必须是可赋值的。编写同时使用物理寄存器和表达式的代码时必须小心，不要使用需要计算很多寄存器的复杂表达式。如果编译程序在分配寄存器时检测到冲突，将发出错误信息。

虚拟和物理寄存器

内联汇编程序提供对 ARM 处理器物理寄存器的非直接访问。如果使用 ARM 寄存器名作为内联汇编程序指令的操作数，则它总是表示虚拟寄存器，而不是物理 ARM 整型寄存器。

在优化和代码生成过程中，ARM 编译程序给每个虚拟寄存器分配相应的物理寄存器。然而，汇编代码中使用的物理寄存器可能与在指令中指定的不同。可将这些虚拟寄存器显式定义为标准 C 或 C++ 变量。如果这些虚拟寄存器未定义，编译程序会为它们提供隐式定义。

不能为 PC (r15)、lr (r14) 和 sp (r13) 寄存器创建虚拟寄存器，而且不能在内联汇编代码中读取或直接修改它们。任何试图使用这些寄存器的操作都会导致出现错误消息。

不存在虚拟*处理器状态寄存器* (PSR)。任何对 PSR 的引用总是指向物理 PSR。

备注

每个虚拟寄存器中的初值是不可预测的。必须在读取之前将初值写入虚拟寄存器。如果在写入之前试图读虚拟寄存器，编译程序会给予警告。

常数

常数表达式说明符 # 是可选的。如果使用了它，其后的表达式必须是常数。

指令扩展

带有常数操作的指令中的常数不仅仅局限于该指令所允许的值。这样的指令能

数用作目标时

够被解释为具有同等效果的指令序列。例如：

```
ADD r0, r0, #1023
```

可能解释为：

```
ADD r0, r0, #1024
SUB r0, r0, #1
```

除了协处理器指令之外，所有带有常数操作数的 ARM 指令都支持指令扩展。此外，当 MUL 指令的第三个操作数是常数时，该指令可以扩展为一系列加法和移位指令。

用扩展指令更新 CPSR 的效果是：

- 算法指令正确设置 NZCV 标志。
- 逻辑指令：
  - 正确设置 NZ 标志；
  - 不改变 V 标志；
  - 破坏 C 标志。

标号

在内联汇编程序语句中可以使用 C 和 C++ 标号。只能用以下格式通过跳转指令跳转到 C 和 C++ 标号：

```
B{cond} label
```

存储器声明

所有存储器都可以用 C 或 C++ 进行声明，并使用变量传递给内联汇编程序。因此，未实现由 armasm 支持的存储器声明。

SWI 和 BL 指令

利用内联汇编程序的 SWI 和 BL 指令，可在常规指令字段后指定 3 个可选寄存器列表。寄存器列表指定：

- 作为输入参数的寄存器；
- 返回后作为输出参数的寄存器；
- 被所调用函数破坏的寄存器。

这些指令的语法是：

```
SWI{cond} swi_num, {input_param_list}, {output_value_list}, {corrupt_reg_list}
BL{cond} function, {input_param_list}, {output_value_list}, {corrupt_reg_list}
```

例如：

```
BL foo {r0=expression1, r1=expression2, r2}, {result=r0, r1}
```

省略的列表将视为空列表，而 BL 总是破坏 ip 和 lr。BL 的默认破坏列表是 r0-r3。

如果未指定任何列表，则：

- r0-r3 用作输入参数；
- r0 用于输出值；
- r12 和 r14 被破坏。

寄存器列表的语法与 LDM 和 STM 寄存器列表相同。如果修改了 NZCV 标志，则必须在破坏寄存器列表中指定 PSR。

5.1.3 内联汇编程序和 armasm 的不同点

内联汇编程序接受的汇编语言和 ARM 汇编程序接受的汇编语言之间有大量不同之处和限制。对于内联汇编程序：

- 不能用点表示法 (.) 或 {PC} 获取当前指令的地址。
- 不支持 LDR Rn, =*表达式* 伪指令。请使用 MOV Rn, *表达式* 代替 （它可以从文字池装载）。
- 不支持标号表达式。
- 不支持 ADR 和 ADRL 伪指令。
- 不能使用 & 操作符表示十六进制常数。应使用 0x 前缀来代替。例如：

  __asm {AND x, y, 0xF00}
- 用于指定 8 位常数实际循环的表示法在内联汇编语言中不能使用。这意味着在使用 8 位移位常数时，如果更新了 NZCV 标志，C 标志必须视为已破坏的。
- 必须小心使用寄存器（如 r0-r3、ip、lr）和 CPSR 中的 NZCV 标志。如果使用 C 或 C++ 表达式，这些寄存器可能被用作临时寄存器，并且 NZCV 标志可能在计算表达式时被编译程序破坏。请参阅第 5-5 页的*虚拟和物理寄存器*。
- 没有为 PC (r15)、lr (r14) 和 sp (r13) 寄存器创建虚拟寄存器，而且不能在内联汇编代码中读取或直接修改它们。有关虚拟寄存器的详细信息，请参阅第 5-5 页的*虚拟和物理寄存器*。
- 您不能写入 PC。不支持 BX、 BXJ、 BLX 和 *BKPT* 指令。
- 不得修改栈。这是没有必要的，因为编译程序根据需要自动将任何工作寄存器压栈和恢复。不允许明确压栈和恢复工作寄存器。

- 可以更改处理器模式、改变 ATPCS 寄存器 fp、 sl 和 sb，或者改变协处理器状态，但编译程序不会意识到所做的更改。如果更改处理器模式，则直到改回到原模式时才能使用 C 或 C++ 表达式，因为编译程序将破坏用于更改后处理器模式的寄存器。

  类似地，如果通过执行浮点指令更改了浮点协处理器状态，则直到恢复原状态后，才能使用浮点表达式。

5.1.4 用法

以下几点适用于内联汇编语言：

- 在汇编语言中，逗号用作分隔符，因此使用逗号操作符的 C 表达式必须包含在括号中以区分它们：

  ```
  __asm {ADD x, y, (f(), z)}
  ```

- 在内联汇编程序中，寄存器名视为 C 或 C++ 变量。它们不一定与相同名称的物理寄存器相关（请参阅第 5-5 页的*虚拟和物理寄存器*）。如果未将寄存器声明为 C 或 C++ 变量，则编译程序会发出警告，提醒您将它声明为变量。

- 在内联汇编程序中不要保存和恢复寄存器。编译程序为您完成此操作。此外，内联汇编程序不提供对物理寄存器的直接访问（请参阅第 5-5 页的*虚拟和物理寄存器*）。对除 CPSR 和 SPSR 之外的寄存器，如果在读取其之前没有写入，将发出错误信息。例如：

  ```
  int f(int x)
  {
      __asm
      {
          STMFD sp!, {r0}    // save r0 - illegal: read before write
          ADD r0, x, 1
          EOR x, r0, x
          LDMFD sp!, {r0}    // restore r0 - not needed.
      }
      return x;
  }
  ```

  该函数必须写为下列形式：

  ```
  int f(int x)
  {
      int r0;
      __asm
      {
          ADD r0, x, 1
  ```



```
            EOR x, r0, x
        }
        return x;
    }
```

5.1.5 示例

示例 5-2 和第 5-10 页的示例 5-3 中示范了一些可以有效使用内联汇编语言的方法。

启用和禁用中断

通过读取 CPSR 标志并更新第 7 位，可以启用或禁用中断。示例 5-2 显示如何使用可以内联的小函数完成此操作。

在 *Examples_directory*\inline\irqs.c. 中也有此代码。

这些函数仅在特权模式下起作用，因为在 “用户” 模式下不能更改 CPSR 和 SPSR 的控制位。

示例 5-2 中断

```
__inline void enable_IRQ(void)
{
    int tmp;
    __asm
    {
        MRS tmp, CPSR
        BIC tmp, tmp, #0x80
        MSR CPSR_c, tmp
    }
}

__inline void disable_IRQ(void)
{
    int tmp;
    __asm
    {
        MRS tmp, CPSR
        ORR tmp, tmp, #0x80
        MSR CPSR_c, tmp
    }
}

int main(void)
{
```

```
    disable_IRQ();
    enable_IRQ();
}
```

标量积

示例 5-3 计算两个整型数组的标量积。它说明对于内联汇编程序不直接支持的 C 或 C++ 表达式和数据类型，内联汇编语言如何与其交互。内联函数 `mlal()` 被优化为单个 `SMLAL` 指令。使用 `-S -fs` 编译程序选项可以查看编译程序生成的汇编语言代码。

`long long` 与 `__int64` 相同。

在 *Examples_directory*\inline\dotprod.c 中也有此代码。

示例 5-3 标量积

```
#include <stdio.h>
/* change word order if big-endian */
#define lo64(a) (((unsigned*) &a)[0])    /* low 32 bits of a long long */
#define hi64(a) (((int*) &a)[1])          /* high 32 bits of a long long */

__inline __int64 mlal(__int64 sum, int a, int b)
{
#if !defined(__thumb) && defined(__TARGET_FEATURE_MULTIPLY)
    __asm
    {
    SMLAL lo64(sum), hi64(sum), a, b
    }
#else
    sum += (__int64) a * (__int64) b;
#endif
    return sum;
}

__int64 dotprod(int *a, int *b, unsigned n)
{
    __int64 sum = 0;
    do
        sum = mlal(sum, *a++, *b++);
    while (--n != 0);
    return sum;
}
int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int b[10] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
```



```
int main(void)
{
    printf("Dotproduct %lld (should be %d)\n", dotprod(a, b, 10), 220);
    return 0;
}
```

5.2 使用嵌入式汇编程序

ARM 编译程序中内建的嵌入式汇编程序使您可以在一个或多个 C 或 C++ 函数定义中包含外联的汇编代码。嵌入式汇编程序提供对目标处理器不受限制的低级别访问，利用它可以使用 C 和 C++ 预处理器命令并易于访问结构成员偏移。

嵌入式汇编程序允许您使用全部 ARM 汇编程序指令集，包括汇编程序命令。嵌入式汇编代码独立于 C 或 C++ 代码进行汇编。然后，所生成的编译对象与 C 或 C++ 源代码编译对象相结合。

ARM 和 Thumb 代码都支持嵌入式汇编程序。有关 ARM 指令集的详细信息，请参阅 *RealView 编译工具 2.0 版汇编程序指南*。

有关嵌入式汇编程序的详细信息，请参阅 *RealView 编译工具 2.0 版编译程序和库指南*中关于内联和嵌入式汇编程序一章。

5.2.1 调用嵌入式汇编程序

嵌入式汇编函数的定义用 `__asm` 函数限定符标记出来。用 `__asm` 声明的函数可以有自变量并返回一个类型。它们从 C 和 C++ 中调用的方式与普通 C 和 C++ 函数调用方式相同。嵌入式汇编函数语法是：

```
__asm return-type function-name(parameter-list)
{
  // ARM/Thumb assembler code

  指令
  ...
  [instruction]
}
```

备注

自变量名允许用在参数列表中，但不能用在嵌入式汇编函数体内。例如，以下函数在函数体内使用整数 `i`，但在汇编中无效：

```
__asm int f(int i) {
    ADD i, i, #1 // error
}
```

例如，可以使用 `r0` 代替 `i`。

用嵌入式汇编函数实现加法示例

示例 5-4 显示使用嵌入式汇编函数将两个值相加的示例。将它与第 5-4 页的示例 5-1 中的内联汇编代码进行比较。

示例 5-4 用嵌入式汇编函数实现加法

```
#include <stdio.h>

__asm int add(int i, int j) {
    ADD  r0,r0,r1        // Value of i in r0 and j in r1, result in r0
        MOV     pc,lr
}

void main() {
    printf("12345 + 67890 = %d\n", add(12345, 67890));
}
```

5.2.2 嵌入式汇编语句的限制

以下约束适用于嵌入式汇编函数：

- 预处理后，__asm 函数只能包含汇编代码，以下标识符除外：

  ```
  __cpp(expr)
  __offsetof_base(D, B)
  __mcall_is_virtual(D, f)
  __mcall_is_in_vbase(D, f)
  __mcall_this_offset(D, f)
  __vcall_offsetof_vfunc(D, f)
  ```

- 编译程序不为 __asm 函数生成返回指令。如果要从 __asm 函数返回，必须将用汇编代码编写的返回指令包含到函数体内。

  备注

  因为嵌入式汇编程序保证按照已定义的顺序发出 __asm 函数，所以这使得转到下一个函数成为可能。然而，内联和模板函数的表现有所不同。

- __asm 函数不更改应用的 ATPCS 规则。这意味着，即使 __asm 函数可用的汇编代码（例如，更改状态）中没有限制，在 __asm 函数和普通 C 或 C++ 函数之间的所有调用也必须紧随 ATPCS。

有关详细信息，请参阅 *RealView 编译工具 2.0 版编译程序和库指南* 中关于内联和嵌入式汇编程序一章。

5.2.3 嵌入式汇编程序和 C 或 C++ 的不同点

要清楚嵌入式汇编程序表达式和 C 或 C++ 表达式之间的以下差异：

- 汇编程序表达式总是无符号的。相同的表达式在汇编程序和 C 或 C++ 中有不同值。例如：

```
MOV r0, #(-33554432 / 2)         // result is 0x7f000000
MOV r0, #__cpp(-33554432 / 2)    // result is 0xff000000
```

- 含有前导零的汇编程序编号仍是十进制的。例如：

```
MOV r0, #0700               // decimal 700
MOV r0, #__cpp(0700)        // octal 0700 == decimal 448
```

- 汇编程序运算符优先顺序与 C 和 C++ 不同。例如：

```
MOV r0, #(0x23 :AND: 0xf + 1)     // ((0x23 & 0xf) + 1) => 4
MOV r0, #__cpp(0x23 & 0xf + 1)    // (0x23 & (0xf + 1)) => 0
```

- 汇编程序字符串不是空终止的：

```
DCB "no trailing null"                  // 16 bytes
DCB __cpp("I have a trailing null!!")   // 25 bytes
```

备注

汇编程序规则应用于 __cpp 之外，而 C 或 C++ 规则应用于 __cpp 之内。有关 __cpp 关键字的详细信息，请参阅 *RealView 编译工具 2.0 版编译程序和库指南*中关于内联和嵌入式汇编程序一章。

5.3 内联汇编代码与嵌入式汇编代码之间的差异

内联和嵌入式汇编的编译方法有所不同：

- 内联汇编代码使用高级处理器分离，并在代码生成过程中与 C 和 C++ 代码集成。因此，编译程序将 C 和 C++ 代码与汇编代码一起进行优化。
- 与内联汇编代码不同，嵌入式汇编代码从 C 和 C++ 代码中分离出来单独进行汇编，产生与 C 和 C++ 源代码编译对象相结合的编译对象。
- 可通过编译程序来内联内联汇编代码，但无论是显式还是隐式，都无法内联嵌入式汇编代码。

表 5-1 汇总了内联汇编程序与嵌入式汇编程序之间的主要差异。

表 5-1 内联汇编程序与嵌入式汇编程序之间的差异

功能	嵌入式汇编程序	内联汇编程序
指令集	ARM 和 Thumb	仅限 ARM
ARM 汇编程序命令	全部支持	不支持
C/C++ 表达式	仅限常量表达式	完整 C/C++ 表达式
优化汇编代码	不优化	全部优化
内联	否	可能
寄存器访问	使用指定的物理寄存器。还可以使用 PC、LR 和 SP。	使用虚拟寄存器（参阅第 5-5 页的*虚拟和物理寄存器*）
返回指令	必须将其添加到代码中。	自动生成

备注

嵌入式汇编程序和 C 或 C++ 之间的差异列表位于 第 5-14 页的*嵌入式汇编程序和 C 或 C++ 的不同点*。

5.4 从汇编代码访问 C 全局变量

只能通过地址间接访问全局变量。要访问全局变量，请使用 IMPORT 命令输入全局变量，然后将地址载入寄存器。可以根据变量的类型使用载入和存储指令访问该变量。

对于**无符号**变量，使用：

- LDRB/STRB 用于 char 型；
- LDRH/STRH 用于 short 型 （对于 ARM 体系结构 v3，使用两个 LDRB/STRB 指令）；
- LDR/STR 用于 int 型。

对于**有符号**变量，请使用等效的有符号指令，如 LDRSB 和 LDRSH。

可以用 LDM 和 STM 指令作为整体访问少于 8 个字的小结构。可以用适当类型的载入和存储指令访问结构的单个成员。为了访问成员，必须了解该成员从结构开始处的偏移量。

示例 5-5 将整型全局变量 globvar 的地址载入 r1、将该地址中包含的值载入 r0、将它与 2 相加，然后将新值存回 globvar 中。

示例 5-5 全局地址

```
    AREA     globals,CODE,READONLY

    EXPORT    asmsubroutine
    IMPORT    globvar

asmsubroutine
    LDR  r1, =globvar   ; read address of globvar into
                        ; r1 from literal pool
    LDR  r0, [r1]
    ADD  r0, r0, #2
    STR  r0,[r1]
    MOV  pc, lr
    END
```

5.5 在 C++ 中使用 C 头文件

本节描述如何在 C++ 代码中使用 C 头文件。从 C++ 调用 C 头文件之前，C 头文件必须包含在 extern "C" 命令中。

5.5.1 包括系统 C 头文件

要包括标准的系统 C 头文件，如 stdio.h，不必进行特殊操作。标准 C 头文件已经包含适当的 extern "C" 命令。例如：

```
#include <stdio.h>
int main()
{
    ...       // C++ code
    return 0;
}
```

如果使用此语法包含头文件，则所有库名都放在全局命名空间中。

C++ 标准指定可通过特定的 C++ 头文件获取 C 头文件的功能。这些文件与标准 C 头文件一起安装在 *install_directory*\RVCT\Data\2.0*build_num**platform*\include 中，可以用常规方法进行引用。例如：

```
#include <cstdio>
int main()
{
    ...       // C++ code
    return 0;
}
```

在 ARM C++ 中，这些头文件中包含 (#include) C 头文件。如果使用此语法包含头文件，则所有 C++ 标准库名都在命名空间 std 中定义，包括 C 库名。这意味着必须使用下列方法之一来限定所有的库名称。

- 指定标准命名空间，例如：
  ```
  std::printf("example\n");
  ```
- 使用 C++ 关键字 using 向全局命名空间输入一个名称：
  ```
  using namespace std;
  printf("example\n");
  ```
- 使用编译程序选项 --using_std。

5.5.2 包含您自己的 C 头文件

要包含自己的 C 头文件，您必须将 #include 命令包在 extern "C" 语句中。可以用以下方法完成此操作：

- 当文件已经被包含 (#include) 时。如示例 5-6 所示。
- 将 extern "C" 语句添加到头文件。如示例 5-7 所示。

示例 5-6 包含文件之前使用命令

```
// C++ code

extern "C" {
#include "my-header1.h"
#include "my-header2.h"
}

int main()
{
    // ...
    return 0;
}
```

示例 5-7 头文件中使用命令

```
/* C header file */

#ifdef __cplusplus    /* Insert start of extern C construct */
extern "C" {
#endif

/* Body of header file */

#ifdef __cplusplus  /* Insert end of extern C construct */
}                   /* The C header file can now be */
#endif              /* included in either C or C++ code. */
```

5.6 C、 C++ 和 ARM 汇编语言之间的调用

本节提供一些示例，帮助您从 C++ 调用 C 和汇编语言代码，以及从 C 和汇编语言调用 C++ 代码。还描述了调用约定和数据类型。

只要遵循正确的过程 ATPCS 调用标准，就可以混合调用 C、 C++ 和汇编语言例程。有关 ATPCS 的更多信息，请参阅第 3 章 *使用过程调用标准*。

备注

本节中的信息与具体的实现情况相关，可能在将来版本的工具箱中会有所改变。

5.6.1 语言交叉调用的一般规则

以下一般规则适用于 C、 C++ 和汇编语言之间的调用。有关的详细信息，请参阅 *RealView 编译工具 2.0 版编译程序和库指南*。

嵌入式汇编程序以及与 ARM 嵌入式应用程序二进制接口 (EABI) 的兼容使得混合语言编程更易于实现。它们可提供以下帮助：

- 使用 `__cpp` 关键字进行名称延伸；
- 传递隐含 `this` 参数的方式；
- 调用虚函数的方式；
- 引用的表示；
- 具有基类或虚成员函数的 C++ 类的类型布局；
- 非 *plain old data* (POD) 结构的类对象传递。

以下一般规则适用于混合语言编程：

- 使用 C 调用约定。
- 在 C++ 中，非成员函数可以声明为 `extern "C"`，以指定它们有 C 链接。在此版本 RVCT 中，带有 C 链接意味着定义函数的符号未延伸。C 链接可以用于以一种语言实现函数，然后用另一种语言调用它。

  备注

  不能超载用 `extern "C"` 声明的函数。

- 汇编语言模块所必须符合的 ATPCS 调用标准，应当适合于应用程序所使用的存储器模型。

以下规则适用于从 C 和汇编语言调用 C++ 函数：

- 要调用全局 （非成员） C++ 函数，应将它声明为 `extern "C"`，以提供 C 链接。
- 成员函数 （静态和非静态）总是有已延伸的名称。使用嵌入式汇编程序的 `__cpp` 关键字，您可以不必手工寻找已延伸的名称。
- 不能从 C 调用 C++ 内联函数，除非确保 C++ 编译程序生成了函数的外联副本。例如，取得函数地址将导致生成外联副本。
- 非静态成员函数接受隐含 `this` 参数作为 r0 中的第一个自变量，或作为 r1 中第二个自变量 （如果函数返回非 `int` 类结构）。静态成员函数不接受隐含 `this` 参数。

5.6.2 C++ 的特定信息

以下内容专门适用于 C++。

C++ 调用约定

ARM C++ 使用与 ARM C 相同的调用约定，但有以下异常：

- 调用非静态成员函数时，隐含的 `this` 参数是第一个自变量，或者是第二个自变量 （如果被调用函数返回非 `int` 类的 `struct`）。这可能在将来实现时有所变化。

C++ 数据类型

ARM C++ 使用与 ARM C 相同的数据类型，但有以下异常：

- 如果 `struct` 或 `class` 类型的 C++ 对象没有基类或虚函数，则它们的布局与 ARM C 相同。如果这样的 `struct` 没有用户定义的复制赋值运算符或用户定义的析构函数，则它是 POD 结构。
- 引用表示为指针。
- C 函数指针和 C++ （非成员）函数指针没有区别。

符号名称延伸

链接程序将取消信息中符号名称的延伸。

在 C++ 程序中， C 名称必须声明为 extern "C"。已经为 ARM ISO C 头文件完成此操作。有关详细信息请参阅第 5-17 页的*在 C++ 中使用 C 头文件*。

5.6.3 示例

下列章节中包含有代码示例，说明：

- *从 C 调用汇编语言*;
- 第 5-23 页的*从汇编语言调用 C* ;
- 第 5-23 页的*从 C++ 调用 C* ;
- 第 5-24 页的*从 C++ 调用汇编语言*;
- 第 5-25 页的*从 C 调用 C++* ;
- 第 5-26 页的*从汇编语言调用 C++* ;
- 第 5-28 页的*从 C 或汇编语言调用 C++* ;
- 第 5-27 页的*在 C 和 C++ 之间传递引用*。

这些示例假设默认为使用非软件栈检查的 ATPCS 变体，因为它们执行栈操作时不检查栈溢出。

从 C 调用汇编语言

示例 5-8 和第 5-22 页的示例 5-9 显示的 C 程序调用了汇编语言子程序，其中将一个字符串复制到另一个字符串中。

示例 5-8 从 C 调用汇编语言

```
#include <stdio.h>
extern void strcopy(char *d, const char *s);
int main()
{   const char *srcstr = "First string - source ";
    char dststr[] = "Second string - destination ";
/* dststr is an array since we珍e going to change it */
    printf("Before copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);
    strcopy(dststr,srcstr);
    printf("After copying:\n");
    printf("  %s\n  %s\n",srcstr,dststr);
    return (0);
}
```

示例 5-9 汇编语言字符串复制子程序

```
    AREA    SCopy, CODE, READONLY
    EXPORT strcopy
strcopy               ; r0 points to destination string.
                      ; r1 points to source string.
    LDRB r2, [r1],#1  ; Load byte and update address.
    STRB r2, [r0],#1  ; Store byte and update address.
    CMP r2, #0        ; Check for zero terminator.
    BNE strcopy       ; Keep going if not.
    MOV pc,lr         ; Return.
    END
```

第 5-21 页的示例 5-8 位于 *Examples_directory*\asm 中，文件名为 strtest.c 和 scopy.s。按以下步骤从命令行编译该示例：

1. 键入 armasm -g scopy.s 编译汇编语言源代码。
2. 键入 armcc -c -g strtest.c 编译 C 源代码。
3. 键入 armlink strtest.o scopy.o -o strtest 链接目标文件。
4. 将 ELF/DWARF2 兼容调试器与相应调试目标配合使用，运行映象。

从汇编语言调用 C

示例 5-10 和示例 5-11 显示如何从汇编语言调用 C。

示例 5-10 定义 C 语言函数

```
int g(int a, int b, int c, int d, int e)
{
        return a + b + c + d + e;
}
```

示例 5-11 汇编语言调用

```
    ; int f(int i) { return g(i, 2*i, 3*i, 4*i, 5*i); }

    EXPORT f
    AREA f, CODE, READONLY
    IMPORT g            ; i is in r0
    STR lr, [sp, #-4]! ; preserve lr
    ADD r1, r0, r0      ; compute 2*i (2nd param)
    ADD r2, r1, r0      ; compute 3*i (3rd param)
    ADD r3, r1, r2      ; compute 5*i
    STR r3, [sp, #-4]! ; 5th param on stack
    ADD r3, r1, r1      ; compute 4*i (4th param)
    BL g                ; branch to C function
    ADD sp, sp, #4      ; remove 5th param
    LDR pc, [sp], #4    ; return
    END
```

从 C++ 调用 C

示例 5-12 和第 5-24 页的示例 5-13 显示如何从 C++ 调用 C。

示例 5-12 从 C++ 调用 C 函数

```
struct S {              // has no base classes
                        // or virtual functions
    S(int s) : i(s) { }
    int i;
};
extern "C" void cfunc(S *);
// declare the C function to be called from C++
```

```
int f(){
    S s(2);           // initialize 's'
    cfunc(&s);        // call 'cfunc' so it can change 's'
   return s.i * 3;
}
```

示例 5-13 定义 C 语言函数

```
struct S {
int i;
};
void cfunc(struct S *p) {
/* the definition of the C function to be called from C++ */
    p->i += 5;
}
```

从 C++ 调用汇编语言

示例 5-14 和第 5-25 页的示例 5-15 显示如何从 C++ 调用汇编语言。

示例 5-14 从 C++ 调用汇编语言

```
struct S {        // has no base classes
                  // or virtual functions
    S(int s) : i(s) { }
int i;
};

extern "C" void asmfunc(S *);   // declare the Asm function
                                // to be called
int f() {
    S s(2);                    // initialize 's'
    asmfunc(&s);               // call 'asmfunc' so it
                               // can change 's'
    return s.i * 3;
}
```

示例 5-15 定义汇编语言函数

```
    AREA Asm, CODE
    EXPORT asmfunc
asmfunc                 ; the definition of the Asm
    LDR r1, [r0]        ; function to be called from C++
    ADD r1, r1, #5
    STR r1, [r0]
    MOV      pc, lr
    END
```

从 C 调用 C++

示例 5-16 和示例 5-17 显示如何从 C 调用 C++。

示例 5-16 定义被调用的 C++ 函数

```
struct S {        // has no base classes or virtual functions
    S(int s) : i(s) { }
    int i;
};

extern "C" void cppfunc(S *p) {
// Definition of the C++ function to be called from C.
// The function is written in C++, only the linkage is C
    p->i += 5;                //
}
```

示例 5-17 在 C 中声明并调用该函数

```
struct S {
    int i;
};

extern void cppfunc(struct S *p);
/* Declaration of the C++ function to be called from C */

int f(void) {
    struct S s;
    s.i = 2;               /* initialize 's' */
    cppfunc(&s);           /* call 'cppfunc' so it */
```

```
                                /* can change 's' */
    return s.i * 3;
}
```

从汇编语言调用 C++

示例 5-18 和示例 5-19 显示如何从汇编语言调用 C++。

示例 5-18 定义被调用的 C++ 函数

```
struct S {           // has no base classes or virtual functions
    S(int s) : i(s) { }
    int i;
};
extern "C" void cppfunc(S * p) {
// Definition of the C++ function to be called from ASM.
// The body is C++, only the linkage is C
    p->i += 5;
}
```

在 ARM 汇编语言中，输入 C++ 函数名并使用带有链接指令的“跳转”调用它：

示例 5-19 定义汇编语言函数

```
    AREA Asm, CODE
    IMPORT cppfunc    ; import the name of the C++
                      ; function to be called from Asm

    EXPORT   f
f
    STMFD  sp!,{lr}
    MOV    r0,#2
    STR    r0,[sp,#-4]! ; initialize struct
    MOV    r0,sp        ; argument is pointer to struct
    BL     cppfunc      ; call 'cppfunc' so it can change
                        ; the struct
    LDR    r0, [sp], #4
    ADD    r0, r0, r0,LSL #1
    LDMFD  sp!,{pc}
    END
```

在 C 和 C++ 之间传递引用

示例 5-20 和示例 5-21 显示如何在 C 和 C++ 之间传递引用。

示例 5-20 C++ 函数

```
extern "C" int cfunc(const int&);
// Declaration of the C function to be called from C++

extern "C" int cppfunc(const int& r) {
// Definition of the C++ to be called from C.
    return 7 * r;
}

int f() {
    int i = 3;
    return cfunc(i);    // passes a pointer to 'i'
}
```

示例 5-21 定义 C 函数

```
extern int cppfunc(const int*);
/* declaration of the C++ to be called from C */

int cfunc(const int* p) {
/* definition of the C function to be called from C++ */
    int k = *p + 4;
    return cppfunc(&k);
}
```

从 C 或汇编语言调用 C++

示例 5-22、示例 5-23 和第 5-29 页的示例 5-24 中的代码说明如何从 C 或汇编语言调用非静态、非虚的 C++ 成员函数。可以使用编译程序的汇编程序输出来查找已延伸的函数名。

示例 5-22 调用 C++ 成员函数

```
struct T {
    T(int i) : t(i) { }
    int t;
    int f(int i);
};

int T::f(int i) { return i + t; }
// Definition of the C++ function to be called from C.

extern "C" int cfunc(T*);
// declaration of the C function to be called from C++

int f() {
    T t(5);                    // create an object of type T
    return cfunc(&t);
}
```

示例 5-23 定义 C 函数

```
struct T;

extern int _ZN1T1fEi(struct T*, int);
    /* the mangled name of the C++ */
    /* function to be called */

int cfunc(struct T* t) {
/* Definition of the C function to be called from C++. */
    return 3 * _ZN1T1fEi(t, 2);    /* like '3 * t->f(2)' */
}
```

示例 5-24 在汇编语言中执行该函数

```
   EXPORT cfunc
   AREA cfunc, CODE
   IMPORT  _ZN1T1fEi
   STMFD   sp!,{lr}  ; r0 already contains the object pointer
   MOV r1, #2
   BL _ZN1T1fEi
   ADD r0, r0, r0, LSL #1   ; multiply by 3
   LDMFD sp!,{pc}
   END
```

也可以使用嵌入式汇编实现第 5-28 页的示例 5-22 和示例 5-24，如示例 5-25 中所示。在此例中，使用 __cpp 关键字引用该函数。因此，您不必了解已延伸的函数名。

示例 5-25 在嵌入式汇编中执行该函数

```
struct T {
   T(int i) : t(i) { }
   int t;
   int f(int i);
};
int T::f(int i) { return i + t; }

// Definition of asm function called from C++
__asm int asm_func(T*) {
   STMFD sp!, {lr}
   MOV r1, #2;
   BL __cpp(T::f);
   ADD r0, r0, r0, LSL #1 ; multiply by 3
   LDMFD sp!, {pc}
}

int f() {
   T t(5); // create an object of type T
   return asm_func(&t);
}
```


第 6 章
处理处理器异常

本章介绍了如何处理 ARM 处理器支持的各种类型的异常。其中包含下列各部分：

6.1 关于处理器异常

程序正常执行流程中，程序计数器在其地址范围内连续增加，还可以跳转至附近程序标号或跳转并链接到子例程。

当该常规执行流程被转向到启用处理器处理内部或外部资源产生的事件时，即发生了处理器异常。此类事件的示例有：

- 外部产生的中断；
- 处理器试图执行一个未定义的指令；
- 访问有特权的操作系统函数。

处理此类异常时，有必要保护处理器先前的状态，以保证在完成相应的异常处理例程后能够恢复产生异常时程序运行的状态，使其继续执行。

表 6-1 展示了 ARM 处理器识别的不同类型的异常。

表 6-1 异常类型

异常	说明
复位	在处理器复位管脚有效时发生。该异常仅在处理器上电时、或上电后的复位时才发生。可通过跳转到复位向量 (0x0000) 来完成软复位。
未定义指令	在处理器或任何协处理器均不能识别当前执行指令时发生。
软件中断 (SWI)	这是一个用户定义的同步中断指令。它使得在 User 模式下运行的程序能够请求在 Supervisor 模式下运行的特权操作，如 RTOS 函数。
预取中断	在处理器试图执行一个未取指令时发生，因为地址非法[a]。
数据中断	在数据传送指令试图在非法地址 [a] 载入或存储数据时发生。
IRQ	在处理器外部中断申请管脚有效 (低电平) 且 CPSR 中的 I 位为清除时发生。
FIQ	在处理器外部中断申请管脚有效 (低电平) 且 CPSR 中的 F 位为清除时发生。

a. 非法虚拟地址是与当前物理存储器地址不符的地址，或者是存储器管理子系统决定在当前模式下处理器不可读取的地址。

6.1.1 向量表

处理器异常的处理由一个*向量表控制*。向量表为一个 32 字节的保留区，通常在存储器映射的底部。它为每一类型的异常分配了一个字的空间，当前有一个保留字。

这不是包含处理程序全部代码所需的足够空间，因此在每个异常类型的向量入口通常包含一个跳转指令或载入 PC 指令来继续执行相应的处理程序。

6.1.2 异常使用的模式和寄存器

通常，应用程序运行在*用户模式*下，但是，为异常提供服务需要有特权的（即非“用户”模式）操作。异常改变了处理器模式，这反过来意味着每个异常处理程序可以访问编组寄存器的某些子集：

- 其自身的 r13 或*栈指针* (sp_*mode*)；
- 其自身的 r14 或*链接寄存器* (lr_*mode*)；
- 其自身的*程序状态存储寄存器* (spsr_ *mode*)。

在 FIQ 情况下，每个异常处理程序占用多达五个通用寄存器（r8_FIQ 到 r12_FIQ）。

每个异常处理程序必须保证在退出时将其它寄存器恢复到其初始内容。可通过将该处理程序必须使用的任意寄存器的内容存储在其栈中，并在返回时恢复这些寄存器的内容即可实现一点目的。如果你正在使用 Angel™ 或 ARMulator®，将自动设置所需要的栈。否则，需自行设置。

备注

提供的汇编程序*没有*预先声明 *register_mode* 格式的符号寄存器名称。要使用该格式，必须用 RN 汇编程序命令声明其相应的符号名称。例如，lr_FIQ RN r14 声明了 r14 寄存器的符号名称为 lr_FIQ。有关 RN 命令的更多信息，请参阅 *RealView 编译工具 2.0 版汇编程序指南*有关命令的章节。

6.1.3 异常优先级

当几个异常同时发生时，它们以固定的优先级顺序处理。用户程序继续执行前，依次处理每个异常。所有异常均同时产生是不可能的。例如，“未定义指令”和 SWI 异常是相互排斥的，因为两者均是通过执行一个指令而触发的。

表 6-2 展示了异常情况、其对应的处理器模式和处理优先级。

由于 “数据中断”异常比 FIQ 异常具有更高的优先级，“数据中断”实际上在处理 FIQ 之前即被寄存。进入了 “数据中断”处理程序，但是，控制权立即交给了 FIQ 处理程序。处理完 FIQ 后，控制权返回 “数据中断”处理程序。这意味着如果先处理 FIQ，数据传输也不会发生漏过数据传送错误检测。

表 6-2 异常优先级

向量地址	异常类型	异常模式	优先级 （**1=** 高， **6=** 低）
0x0	复位	超级用户 (SVC)	1
0x4	未定义指令	未定义	6
0x8	软件中断 (SWI)	超级用户 (SVC)	6
0xC	预取中断	中断	5
0x10	数据中断	Abort	2
0x14	*保留*	*不可用*	*不可用*
0x18	中断 (IRQ)	中断 (IRQ)	4
0x1C	快速中断 (FIQ)	快速中断 (FIQ)	3

6.2 确定处理器状态

产生异常时，异常处理程序可能要确定处理器是在 ARM 状态还是在 Thumb 状态。特别是 SWI 处理程序，更需要读取处理器状态。通过检测 SPSR 的 T 位完成确定处理器状态。该位在 Thumb 状态时为设置，在 ARM 状态时为清除。

ARM 和 Thumb 指令集均有 SWI 指令。在 Thumb 状态下调用 SWI 时，必须考虑：

- 指令的地址在 (lr - 2)，而不在 (lr - 4)；
- 该指令本身为 16 位，因而需要半字节载入（请参阅图 6-1）；
- 在 AMR 状态下， SWI 号码以 8 位存储，而不是 24 位。

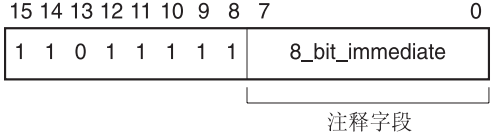

图 6-1 Thumb SWI 指令

示例 6-1 说明了在两种资源中处理一个 SWI 的 ARM 代码。考虑如下要点：

- 如果需要改进编码的密度，每个 do_swi_x 例程均可实行转换到 Thumb 状态并且能转换回来。
- 可通过调用一个包含 switch() 语句的 C 函数来执行 SWI 以替换跳转表。
- 根据调用状态不同， SWI 号码的处理也可能不同。
- Thumb 状态下可获取的 SWI 号码范围可通过动态调用 SWI 来增加（如第 6-18 页的 *SWI 处理程序*说明）。

示例 6-1

```
T_bit   EQU    0x20                    ; Thumb bit of CPSR/SPSR, that is, bit 5.
        :
        :
SWIHandler
        STMFD   sp!, {r0-r3,r12,lr}    ; Store registers.
        MRS     r0, spsr               ; Move SPSR into general purpose register.
        TST     r0, #T_bit             ; Occurred in Thumb state?
        LDRNEH  r0,[lr,#-2]            ; Yes: load halfword and...
```

```
        BICNE   r0,r0,#0xFF00           ; ...extract comment field.
        LDREQ   r0,[lr,#-4]             ; No: load word and...
        BICEQ   r0,r0,#0xFF000000       ; ...extract comment field.

            ; r0 now contains SWI number

        CMP     r0, #MaxSWI             ; Rangecheck
        LDRLS   pc, [pc, r0, LSL#2]     ; Jump to the appropriate routine.
        B       SWIOutOfRange
switable
        DCD     do_swi_1
        DCD     do_swi_2
        :
        :
do_swi_1
        ; Handle the SWI.
        LDMFD   sp!, {r0-r3,r12,pc}^    ; Restore the registers and return.
do_swi_2
        :
```

6.3 进入和离开异常

本章说明了处理器对异常的反应，处理完异常后如何返回发生异常之处。根据异常类型的不同，返回的方式也不同。

能够处理 Thumb 代码的处理器与不能处理 Thumb 代码的处理器使用相同的基本异常处理机理。异常造成下一条指令要从相应的向量表入口读取。

Thumb 状态和 ARM 状态使用同一向量表。*处理器对异常的反应*所描述的异常处理过程中增加了转至 ARM 的初始步骤。

当编写适合于能处理 Thumb 代码的处理器的异常处理程序时，必须考虑额外的因素，这些已被明显标出。

6.3.1 处理器对异常的反应

产生异常时，处理器采取如下动作：

1. 将要处理的异常模式的*当前程序状态寄存器* (CPSR) 复制到*程序状态存储寄存器* (SPSR)。因而保存了当前模式、中断屏蔽和条件标志。
2. *仅限于能处理 Thumb 代码的处理器，*转至 ARM 状态。
3. 请改变相应的 CPRS 模式位以便：
   - 改变相应的模式及该模式下相应编组寄存器中的映射。
   - 禁用中断。发生中断时，禁用 IRQ。发生 FIQ 时和在复位时，禁用 FIQ。
4. 设置 lr_*mode* 到返回地址，如第 6-9 页的*返回地址和返回指令*中所定义。
5. 将程序计数器设置至异常向量地址。

   *对于不能处理 Thumb 代码的 ARM 处理器，*这将强行跳转至相应的异常处理程序。

   *对于能处理 Thumb 代码的处理器，*步骤 2 中由 Thumb 状态转至 ARM 状态保证了安装在该向量地址 （跳转或与 PC 相关的载入）的 ARM 指令能够正确读取、解码和执行。这将迫使跳转至高层胶合代码，该胶合代码必须用 ARM 代码编写。

6.3.2 从异常处理程序的返回

从异常返回的方法取决于该异常处理程序使用或不使用堆操作。在两种情况下，要返回到异常发生处后继续执行，异常处理程序必须：

- 从 spsr_*mode* 恢复 CPSR ；
- 使用存储在 lr_*mode* 中的返回地址恢复程序计数器。

对于不需要将目的模式寄存器从堆中恢复的简单返回，通过执行如下数据处理指令，异常处理程序完成这些操作，它们是：

- 设置 S 标志；
- 用目的寄存器恢复程序计数器。

所需的返回指令取决于异常的类型。有关如何从每种类型的异常返回的说明，请参阅第 6-9 页的*返回地址和返回指令*。

备注

不必从复位处理程序返回，因为复位处理程序直接执行主代码。

处理异常时，如果异常处理程序入口代码使用了堆来存储必须保留的寄存器，可通过使用装载多个带 ^ 限定符的指令来返回。异常处理程序可使用一条指令返回，例如使用：

```
LDMFD sp!,{r0-r12,pc}^
```

如果它将如下内容存储在堆中：

- 调用处理程序时将使用所有的工作寄存器；
- 为产生与下面说明的数据处理指令相同的效果而修改链接寄存器。

^ 限定符指定从 SPSR 中恢复 CPSR。仅在特权模式下使用。有关 LDM 和 STM 的堆执行，请在 *RealView 编译工具 2.0 版汇编程序指南*中参阅介绍堆操作的更多一般信息。

6.3.3 返回地址和返回指令

产生异常时由程序计数器指向的实际位置取决于异常的类型。返回地址可能不必是程序计数器指向的下一条指令。

如果异常发生在 ARM 状态，处理器则将 (PC - 4) 存储在 lr_ *mode*。但是，对于在 Thumb 状态下发生的异常，处理器为每个异常类型自动存储一个不同的值。这样的调整是必需的，因为 Thumb 指令仅占用半个字，而不像 ARM 指令占用整个字。

如果处理器不对此进行修正，处理程序必须确定处理器原始状态并使用不同指令返回到 Thumb 代码，而不是返回 ARM 代码。然而，通过这样的调整，处理器能使处理程序使用一条指令完成正确返回，而不用考虑异常发生时处理器的状态 （是 ARM 或 Thumb）。

下面的章节较详细地说明了每一种异常类型如何从处理代码正确返回的指令。

从 SWI 和 未定义指令处理程序返回

SWI 和未定义指令异常是由指令本身造成的，因此，处理异常时，程序计数器保持不变。处理器将 (PC - 4) 存储在 lr_ *mode* 中。因而使 lr_*mode* 指向下一条要执行的指令。要从 lr 中恢复程序计数器，则使用：

```
MOVS        pc, lr
```

从处理程序返回控制权。

将返回地址推入堆中并在返回时将其弹出的处理程序入口和出口代码为：

```
STMFD       sp!,{reglist,lr}
;...
LDMFD       sp!,{reglist,pc}^
```

对于在 Thumb 状态下发生的异常，处理程序返回指令 (`MOVS pc,lr`) 将程序计数器指向下一条要执行指令的地址。这是在 (PC - 2)，因此处理器存储在 lr_*mode* 中的值为 (PC - 2)。

从 FIQ 和 IRQ 处理程序返回

执行完每一条指令后，处理器检测中断管脚是否为 LOW （电平），以及 CPSR 中断禁用位是否为清除。结果，仅在程序计数器被更新后才发生 IRQ 或 FIQ 异常。处理器将 (PC - 4) 存储在 lr_*mode* 中。使 lr_*mode* 指向发生异常时尚未完成的一条指令。处理程序完成后，必须从 lr_*mode* 指向的上一条指令处继续运行。

该继续执行地址较 lr_*mode* 中的地址少一个字（四个字节），因此，其返回指令为：

```
SUBS        pc, lr, #4
```

将返回地址推入堆中并在返回时将其弹出的处理程序入口和出口代码为：

```
SUB         lr,lr,#4
STMFD       sp!,{reglist,lr}
;...
LDMFD         sp!,{reglist,pc}^
```

对于发生在 Thumb 状态下的异常，处理程序返回指令 (SUBS pc,lr,#4) 改变了程序计数器，将其指向下一条要执行指令的地址。由于程序计数器的更新是在处理异常之前完成，下一条指令应在 (PC - 4) 。因此，处理器存储在 lr_*mode* 的值为 PC 。

从预取中断处理程序返回

如果处理器试图在非法地址取指令，则该指令被标志为无效。继续执行已经在流水线中的指令至遇到产生 “预取中断”处的无效指令为止。

如有将虚拟存储器位置映射到该物理存储器的指令，异常处理程序将无映射关系的指令装入物理存储器并使用 MMU。然后，处理程序必须返回，再次使着运行产生异常的指令。现在装入并执行指令。

因为发出预取中断时程序计数器还没有被更新，所以 lr_ABT 指向产生异常的下一条指令。处理程序必须返回至 lr_ABT - 4 的指令，请使用下列指令：

```
SUBS        pc,lr, #4
```

将返回地址推入堆中并在返回时将其弹出的处理程序入口和出口代码为：

```
SUB         lr,lr,#4
STMFD       sp!,{reglist,lr}
;...
LDMFD       sp!,{reglist,pc}^
```

对于发生在 Thumb 状态下的异常，处理程序返回指令 (SUBS pc,lr,#4) 改变了程序计数器，将其指向产生中断指令的地址。由于程序计数器的更新不是在处理异常之前完成，中断指令应在 (PC - 4)。因此，处理器存储在 lr_*mode* 的值为 PC。

从数据中断处理程序返回

当装入或存储指令试图访问存储器时，程序计数器被更新。(PC - 4) 的存储值指向产生异常地址处的第二条指令。MMU（如果有）将相应地址映射至物理存储器，处理程序必须返回到原来中断的指令，以便进行第二次执行尝试。因此，返回地址较 lr_ABT 中少两个字 （八个字节），使用如下返回指令：

```
SUBS        pc, lr, #8
```

将返回地址推入堆中并在返回时将其弹出的处理程序入口和出口代码为：

```
SUB         lr,lr,#8
STMFD       sp!,{reglist,lr}
;...
LDMFD       sp!,{reglist,pc}^
```

对于发生在 Thumb 状态下的异常，处理程序返回指令 (SUBS pc,lr,#8) 改变了程序计数器，将其指向产生中断指令的地址。由于程序计数器的更新是在处理异常之前，中断指令应在 (PC - 6)。因此，处理器存储在 lr_*mode* 的值为 (PC + 2)。

6.4 处理异常

顶层胶合代码必须将处理器的状态和任何所需的寄存器存储在堆中。然后可以按下面的选项编写异常处理程序。

- 完全以 ARM 代码编写异常处理程序。
- 执行 BX （跳转和互换）转到处理异常的 Thumb 代码例程。为从异常中返回，该例程必须返回到 ARM 胶合代码，因为 Thumb 指令集没有要求从 spsr 中恢复 cpsr 的指令。

第二个策略如图 6-2 所示。有关如何将 ARM 和 Thumb 代码以这种方式结合的详细说明，请参阅第 4 章 *ARM 和 Thumb 交互操作*。

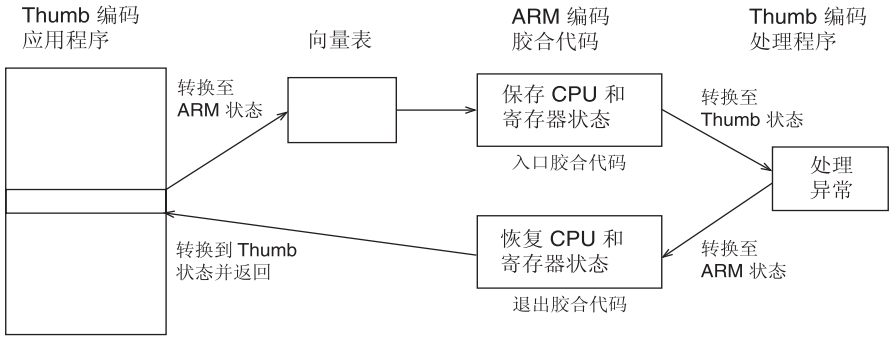

图 6-2 在 ARM 或 Thumb 状态下处理异常

6.5 安装异常处理程序

必须将任何新的异常处理程序安装在向量表中。安装完成后，不论什么时候产生相应的异常，都会执行新的处理程序。

可以按以下方法安装异常处理程序：

跳转指令

这是转到异常处理程序最简单的方法。向量表中的每个入口都包括一个能够转到处理例程的跳转。但是，这种方法确实有局限性。因跳转指令相对于 PC 仅有 32MB 的范围，在某些存储器组织情况下，该跳转可能无法转到处理程序。

装入 PC 指令

使用该方法，强制程序计数器直接指向处理程序的方法有：

1. 将处理程序的绝对地址存储在合适的存储器位置 （在向量地址的 4BK 范围内）。
2. 将一个指令存放在向量中，该向量装入包含所选存储器位置的程序计数器。

6.5.1 在复位时安装处理程序

如果应用程序不是靠调试器或调试监督程序启动程序的执行，可以使用汇编语言复位 （或启动）代码直接装入向量表 。

如果 ROM 是在存储器的 0x0 位置，则可在代码的起始处为每个向量分配一个跳转语句。如果 FIQ 处理程序是直接从 0x1c 运行的，则也可将 FIQ 处理程序包含在其中 （请参阅第 6-27 页的*中断处理程序*）。

示例 6-2 展示了如果它们位于 ROM 的零地址时建立向量的代码。可替换装入的跳转语句。

示例 6-2

```
Vector_Init_Block
                LDR    pc, Reset_Addr
                LDR    pc, Undefined_Addr
                LDR    pc, SWI_Addr
                LDR    pc, Prefetch_Addr
                LDR    pc, Abort_Addr
                NOP                        ;Reserved vector
```

```
                LDR     pc, IRQ_Addr
                LDR     pc, FIQ_Addr

Reset_Addr      DCD     Start_Boot
Undefined_Addr  DCD     Undefined_Handler
SWI_Addr        DCD     SWI_Handler
Prefetch_Addr   DCD     Prefetch_Handler
Abort_Addr      DCD     Abort_Handler
                DCD     0               ;Reserved vector
IRQ_Addr        DCD     IRQ_Handler
FIQ_Addr        DCD     FIQ_Handler
```

复位状态下，必须使 ROM 处在 0x0 位置。复位代码可将 RAM 映射到 0x0 位置 。这样做之前，它需从 ROM 的某个区域将该向量（根据需要还有 FIQ 处理程序）复制到 RAM 中。

在此情况下，必须用一条 LDR pc 指令为复位处理程序确定地址，以便使复位向量代码能独立定位。

示例 6-3 将第 6-13 页的示例 6-2 给出的向量复制到了 RAM 中的向量表。

示例 6-3

```
    MOV         r8, #0
    ADR         r9, Vector_Init_Block
    LDMIA       r9!,{r0-r7}           ;Copy the vectors (8 words)
    STMIA       r8!,{r0-r7}
    LDMIA       r9!,{r0-r7}           ;Copy the DCD'ed addresses
    STMIA       r8!,{r0-r7}           ;(8 words again)
```

此外，还可使用分散载入机制来定义向量表的载入和执行地址。在此情况下，C 库将为您复制向量表 （请参阅第 2 章 *嵌入式软件开发*）。

6.5.2 从 C 安装处理程序

开发过程中有时必需从主应用程序直接将异常处理程序安装在向量中。结果，所需的指令编码必须被写到相应的向量地址中。这可由跳转和载入 PC 两种方法来转到该处理程序。

跳转方法

所需的指令可按如下方法构成：

1. 获取异常处理程序的地址。
2. 减去相应向量的地址。
3. 减去 0x8 以便预取。
4. 将结果右移两位给出一个字的偏移，而不是一个字节的偏移。
5. 测试其高八位为清除，确保结果仅为 24 位长 （因为跳转的偏移被限制为此长度）。
6. 把它与 0xEA000000 （跳转指令操作码）进行逻辑 “或”运算，生成要放在向量中的值。

示例 6-4 展示了实现该算法的 C 函数。它占用了以下几个自变量：

- 处理程序的地址；
- 将处理程序安装在其中的向量地址。

该函数可以安装处理程序并返回该向量的初始内容。这个结果可以用来为一个特定的异常创建一系列的处理程序。详细信息请参阅第 6-41 页的*链结异常处理程序*。

示例 6-4

```
unsigned Install_Handler (unsigned routine, unsigned *vector)
/* Updates contents of 'vector' to contain branch instruction */
/* to reach 'routine' from 'vector'. Function return value is */
/* original contents of 'vector'.*/
/* NB: 'Routine' must be within range of 32MB from 'vector'.*/

{   unsigned vec, oldvec;
    vec = ((routine - (unsigned)vector - 0x8)>>2);
    if ((vec & 0xFF000000))
    {
```

```
        /* diagnose the fault */
        prinf ("Installation of Handler failed");
        exit (1);
    }
    vec = 0xEA000000 | vec;
    oldvec = *vector;
    *vector = vec;
    return (oldvec);
}
```

以下代码调用它来安装 IRQ 处理程序：

```
unsigned *irqvec = (unsigned *)0x18;
Install_Handler ((unsigned)IRQHandler, irqvec);
```

在此情况下，将丢弃返回的初始 IRQ 向量内容。

装入 PC 方法

所需的指令可按如下方法构成：

1. 获取包含异常处理程序地址那个字的地址。
2. 减去相应向量的地址。
3. 减去 0x8 以便预取。
4. 检查其结果能否用 12 位表示。
5. 把它与 0xe59FF000（LDR pc, [pc,#offset] 的操作码）进行逻辑“或”运算，生成要放在向量中的值。
6. 将异常处理程序的地址放入该存储位置。

示例 6-5 展示了实现了该方法的 C 例程。

示例 6-5

```
unsigned Install_Handler (unsigned location, unsigned *vector)

/* Updates contents of 'vector' to contain LDR pc, [pc, #offset] */
/* instruction to cause long branch to address in 'location'. */
/* Function return value is original contents of 'vector'. */

{   unsigned vec, oldvec;
    vec = ((unsigned)location - (unsigned)vector - 0x8) | 0xe59ff000;
    oldvec = *vector;
```



```
    *vector = vec;
    return (oldvec);
}
```

以下代码调用它来安装 IRQ 处理程序：

```
unsigned *irqvec = (unsigned *)0x18;
static unsigned pIRQ_Handler = (unsigned)IRQ_handler
Install_Handler (&pIRQHandler, irqvec);
```

在此示例中，返回的初始 IRQ 向量内容被再次被丢弃，但可用它来创建一系列处理程序。有关详细信息请参阅第 6-41 页的*链结异常处理程序*。

备注

如果正在使用带单独指令和数据缓存的处理器，例如 StrongARM® 或 ARM940T，必须保证缓存一致性问题不会阻碍使用新的向量内容 。

为确保新的向量内容写入主存储器中，必须清除数据缓存 （或至少其入口含有修改过的向量）。然后必须将指令缓存刷新，确保可从主存储器中读取新的向量内容。

有关清除和刷新缓存操作的详细说明，请参阅目标处理器的技术参考手册。

6.6 SWI 处理程序

进入 SWI 处理程序时，必须设定哪个 SWI 将被调用。该信息可存储在指令本身的 0-23 位，如图 6-3 所示，或将其传递到一个整数寄存器，通常为 r0-r3 中的一个。

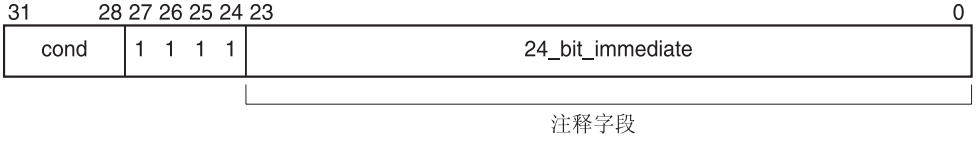

图 6-3 ARM SWI 指令

最高层的 SWI 处理程序可载入与链接寄存器关联的 SWI 指令 (LDR swi, [lr, #-4]) 。请用汇编语言、 C/C++ 内联或嵌入汇编语言编写。

处理程序必须首先将导致异常的 SWI 指令装入寄存器。在这一点上， lr_SVC 存储着 SWI 指令的下一个指令的地址，因此 SWI 使用下面的指令被载入到了寄存器（此例中为 r0）：

```
LDR r0, [lr,#-4]
```

然后，处理程序可检查字段位，以决定所需的操作。通过清除操作码的高八位来提取 SWI 编号。

```
BIC r0, r0, #0xFF000000
```

示例 6-6 展示了如何将这些指令结合形成一个最高层的 SWI 处理程序。

有关处理 ARM 状态和 Thumb 状态的 SWI 指令的处理程序示例，请参阅第 6-5 页的*确定处理器状态*。

示例 6-6

```
    AREA TopLevelSwi, CODE, READONLY  ; Name this block of code.
    EXPORT     SWI_Handler
SWI_Handler
    STMFD      sp!,{r0-r12,lr}        ; Store registers.
    LDR        r0,[lr,#-4]            ; Calculate address of SWI instruction and load it into r0.
    BIC        r0,r0,#0xff000000      ; Mask off top 8 bits of instruction to give SWI number.
    ;
    ; Use value in r0 to determine which SWI routine to execute.
```

```
;
LDMFD        sp!, {r0-r12,pc}^    ; Restore registers and return.
END                               ; Mark end of this file.
```

6.6.1 汇编语言编写的 SWI 处理程序

调用所需 SWI 编号的处理程序，最简单的方法是使用跳转表。如果 r0 中有 SWI 编号，示例 6-7 中的代码可被插在第 6-18 页的示例 6-6 给出的最高层处理程序中，插入位置在 BIC 指令之后。

示例 6-7 SWI 跳转表

```
    CMP    r0,#MaxSWI          ; Range check
    LDRLS  pc, [pc,r0,LSL #2]
    B      SWIOutOfRange
SWIJumpTable
    DCD    SWInum0
    DCD    SWInum1
                   ; DCD for each of other SWI routines
SWInum0            ; SWI number 0 code
    B    EndofSWI
SWInum1            ; SWI number 1 code
    B    EndofSWI
                   ; Rest of SWI handling code
                   ;
EndofSWI
                   ; Return execution to top level
                   ; SWI handler so as to restore
                   ; registers and return to program.
```

6.6.2 C 语言和汇编语言编写的 SWI 处理程序

虽然最高层的处理程序总是用 ARM 汇编语言编写，但处理每一 SWI 的例程却是既可用汇编语言编写，又可用 C 语言来编写。有关限制情况的说明，请参阅第 6-21 页的*在超级用户模式下使用 SWI*。

最高层的处理程序通过使用一个 BL （链接跳转）指令跳转到相应的 C 函数。由于 SWI 编号被汇编语言例程装入了 r0，并将其作为第一个参数（与 “ARM 过程调用标准”一致）传递至 C 函数。该函数可以使用这个值，例如，一个 switch() 语句。

可在第 6-18 页的示例 6-6 的 SWI_Handler 例程中加入下列行：

```
    BL    C_SWI_Handler     ; Call C routine to handle the SWI
```

示例 6-8 展示了如何实现 C 函数。

示例 6-8

```
void C_SWI_handler (unsigned number)
{ switch (number)
    {case 0 :                /* SWI number 0 code */
        break;
    case 1 :                 /* SWI number 1 code */
        break;
    :
    :
    default :                /* Unknown SWI - report error */
    }
}
```

超级用户堆的空间可能是有限的，因此要避免使用需要大量堆空间的函数。

可将值传入和传出用 C 语言编写的 SWI 处理程序中，前提条件是最高层处理程序将堆指针值作为第二参数 （r1 中）传递到 C 函数中：

```
    MOV     r1, sp        ; Second parameter to C routine...
                          ; ...is pointer to register values.
    BL    C_SWI_Handler   ; Call C routine to handle the SWI
```

然后更新 C 函数以访问它：

```
void C_SWI_handler(unsigned number, unsigned *reg)
```

现在，C 函数在主应用程序代码（请参阅第 6-21 页的图 6-4）中遇到 SWI 指令时即可存取存储在寄存器中的值了。它可从其中读取：

```
    value_in_reg_0 = reg [0];
    value_in_reg_1 = reg [1];
    value_in_reg_2 = reg [2];
    value_in_reg_3 = reg [3];
```

并可写入其中：

```
    reg [0] = updated_value_0;
    reg [1] = updated_value_1;
    reg [2] = updated_value_2;
    reg [3] = updated_value_3;
```


这使已被更新的值写入了堆的相应位置，然后通过最高层处理程序将其恢复到寄存器中。

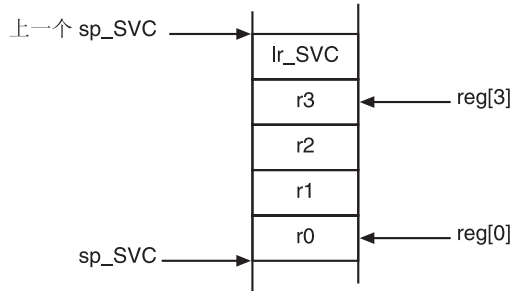

图 6-4 存取超级用户堆

6.6.3 在超级用户模式下使用 SWI

当执行 SWI 指令时：

1. 处理器进入超级用户模式
2. CPSR 被存储在 spsr_SVC 中。
3. 返回地址被存储在 lr_SVC 中 （请参阅第 6-7 页的*处理器对异常的反应*）。

如处理器已经处在超级用户模式下， lr_SVC 和 spsr_SVC 被破坏。

在超级用户模式下，如果调用 SWI，则需存储 lr_SVC 和 spsr_SVC，以确保链接寄存器和 SPSR 中的初始值不被丢失。例如，一个特定 SWI 编号的处理程序例程调用了另一个 SWI，必须确保该该处理程序例程将 lr_SVC 和 spsr_SVC 存储在堆中。它确保了处理程序的每一次调用都保存了返回到调用 SWI 指令所需要的信息。示例 6-9 展示了如何实现的方法。

示例 6-9 SWI 处理程序

```
STMFD    sp!,{r0-r3,r12,lr}   ; Store registers.
MOV      r1, sp               ; Set pointer to parameters.
MRS      r0, spsr             ; Get spsr.
STMFD    sp!, {r0}            ; Store spsr onto stack. This is only really needed in case of
                              ; nested SWIs.

    ; the next two instructions only work for SWI calls from ARM state.
```

```
    ; See Example 6-18 on page 6-33 for a version that works for calls from either ARM or Thumb.

LDR      r0,[lr,#-4]          ; Calculate address of SWI instruction and load it into r0.
BIC      r0,r0,#0xFF000000    ; Mask off top 8 bits of instruction to give SWI number.

    ; r0 now contains SWI number
    ; r1 now contains pointer to stacked registers

BL       C_SWI_Handler        ; Call C routine to handle the SWI.
LDMFD    sp!, {r0}            ; Get spsr from stack.
MSR      spsr_cf, r0          ; Restore spsr.
LDMFD    sp!, {r0-r3,r12,pc}^ ; Restore registers and return.
```

C 和 C++ 编写的嵌套 SWI

可用 C or C++ 汇编语言编写嵌套的 SWI。由 ARM 编译程序生成的代码按需要存储和载入 lr_SVC。

6.6.4 从应用程序调用 SWI

可从汇编语言或 C/C++ 调用 SWI。

在汇编语言中，设定所有必须的值并发出相关的 SWI。例如：

```
MOV    r0, #65    ; load r0 with the value 65
SWI    0x0        ; Call SWI 0x0 with parameter value in r0
```

几乎像所有的 ARM 指令那样，SWI 指令可被有条件地执行。

从 C/C++ 中将 SWI 声明为一个 __SWI 函数并调用它。例如：

```
__swi(0) void my_swi(int);
.
.
.
my_swi(65);
```

它确保了 SWI 以内联方式进行编译，勿需额外的调用开销，其条件是：

- 任何自变量只能被传递到 r0-r3 ；
- 任何结果只能被返回到 r0-r3。

其参数被传递到了 SWI，好像该 SWI 是一个真正的函数调用。但是，如果有二到四个返回值，则必须告诉编译程序返回值是以结构形式返回的，并使用 __value_in_regs 命令。这是因为基于**结构**值的函数通常被处理为一个 void （空）函数，其第一个自变量必须为存放结果结构的地址。

示例 6-10 和示例 6-11 展示了提供 SWI 编号 0x0、0x1、0x2 和 0x3 的 SWI 处理程序。SWI 0x0 和 0x1 每一个都占用二个整型参数并返回一个单一结果。SWI 0x2 占用四个参数并返回一个单一结果。SWI 0x3 占用四个参数并返回四个结果。该示例位于 *Examples_directory*\SWI\main.c 和 *Examples_directory*\SWI\swi.h。

示例 6-10 main.c

```
#include <stdio.h>
#include "swi.h"

unsigned *swi_vec = (unsigned *)0x08;
extern void SWI_Handler(void);

int main(void)
{
    int result1, result2;
    struct four_results res_3;
    Install_Handler( (unsigned) SWI_Handler, swi_vec );
    printf("result1 = multiply_two(2,4) = %d\n", result1 = multiply_two(2,4));
    printf("result2 = multiply_two(3,6) = %d\n", result2 = multiply_two(3,6));
    printf("add_two( result1, result2 ) = %d\n", add_two( result1, result2 ));
    printf("add_multiply_two(2,4,3,6) = %d\n", add_multiply_two(2,4,3,6));
    res_3 = many_operations( 12, 4, 3, 1 );
    printf("res_3.a = %d\n", res_3.a );
    printf("res_3.b = %d\n", res_3.b );
    printf("res_3.c = %d\n", res_3.c );
    printf("res_3.d = %d\n", res_3.d );
    return 0;
}
```

示例 6-11 swi.h

```
__swi(0) int multiply_two(int, int);
__swi(1) int add_two(int, int);
__swi(2) int add_multiply_two(int, int, int, int);

struct four_results
```

```
{
    int a;
    int b;
    int c;
    int d;
};

__swi(3) __value_in_regs struct four_results
    many_operations(int, int, int, int);
```

6.6.5 从应用程序动态调用 SWI

在某些情形下，需要调用直到运行时才会知道其编号的 SWI。例如，当有很多相关操作可在同一目标上执行，并且每一个操作都有其自己的 SWI 时，就会发生这种情况。在此情况下，上述方法不适用。

有几种解决方法，例如，可以通过：

- 从 SWI 编号构建 `SWI` 指令，将它存储在某处，然后执行其。
- 使用通用的 SWI（作为一个额外的自变量）获取一个代码，以便对其自变量执行实际操作。该通用 SWI 对操作进行解码并予以执行。

通过传递寄存器（通常为 r0 或 r12）中所需要的操作数，用汇编语言实现第二种机制。这样就可重新编写 SWI 处理程序，对相应寄存器中的值进行处理。因为有些值必须以注释字段传递到 SWI，所以有可能将这两种方法结合起来使用。

例如，操作系统可能会使用单一的一条 `SWI` 指令并用寄存器来传递所需运算的编号。这使得其它 SWI 空间可用于特定应用程序的 SWI。在一个特定的应用程序中，如果从指令中提取 SWI 编号的开销太大，就可使用这个方法。ARM (`0x123456`) 和 Thumb (`0xAB`) 半主机方式的 SWI 就是这样实现的。

第 6-25 页的示例 6-12 展示了如何使用 `__swi` 将 C 函数调用映射到半主机方式的 SWI。它是从 *Examples_directory*\embedded\embed\retarget.c 中派生出来的。

示例 6-12 映射 C 函数到半主机 SWI

```
#ifdef __thumb
/* Thumb Semihosting SWI */
#define SemiSWI 0xAB
#else
/* ARM Semihosting SWI */
#define SemiSWI 0x123456
#endif

/* Semihosting SWI to write a character */
__swi(SemiSWI) void Semihosting(unsigned op, char *c);
#define WriteC(c) Semihosting (0x3,c)

void write_a_character(int ch)
{
    char tempch = ch;
    WriteC( &tempch );
}
```

编译程序含有一个机制，用以支持使用 r12 来传递所需运算的值。在 “ARM 过程调用标准”下，r12 为 ip 寄存器，并且专用于函数调用。其它时间内可将其用作暂存寄存器。通用 SWI 的自变量被传递到 r0-r3 寄存器中，如前面所述，其值被有选择地返回到 r0-r3 中。传递到 r12 中的操作数可以（但不要求）是通用 SWI 调用的 SWI 编号。

示例 6-13 展示了使用通用或*间接* SWI 的 C 语言程序段。

示例 6-13

```
__swi_indirect(0x80)
    unsigned SWI_ManipulateObject(unsigned operationNumber,
                                  unsigned object,unsigned parameter);

unsigned DoSelectedManipulation(unsigned object,
                                unsigned parameter, unsigned operation)
{ return SWI_ManipulateObject(operation, object, parameter);
}
```

它生成了以下代码：

```
DoSelectedManipulation PROC
        STMFD    sp!,{r3,lr}
        MOV      r12,r2
        SWI      0x80
        LDMFD    sp!,{r3,pc}
        ENDP
```

还可利用 __swi 机制从 C 中将 SWI 编号传递到 r0 中。例如，如果将 SWI 0x0 用作通用 SWI，且操作 0 为字符读，操作 1 为字符写，那么，就可如下建立：

```
__swi (0) char __ReadCharacter (unsigned op);
__swi (0) void __WriteCharacter (unsigned op, char c);
```

可通过如下定义使其具有更好的可读性风格：

```
#define ReadCharacter () __ReadCharacter (0);
#define WriteCharacter (c) __WriteCharacter (1, c);
```

但是，如果以这种方式使用的 r0，那么，仅有三个寄存器可用于将参数传递到 SWI。通常，在不得不将除 r0-r3 之外的更多参数传递给子例程时，可通过使用堆来完成 但是， SWI 处理程序不容易存取堆参数，因为通常它们是存在用户模式的堆里而不是在 SWI 处理程序使用的超级用户模式的堆里。

作为另一种选择，其中一个寄存器 （通常是 r1）可用来指向存储其它参数的存储器块。

6.7 中断处理程序

ARM 处理器有 FIQ 和 IRQ 两级外部中断，均是由低 (LOW) 电平信号激活进入内核程序。为响应一个中断， CPSR 中相应的中断使能位必须清空。

在以下方式中， FIQ 较 IRQ 有更高的优先级：

- 当发生多个中断时，首先处理 FIQ。
- 处理 FIQ 造成禁用 IRQ，直至 FIQ 处理程序再次将其置为可用状态前禁止使用。这通常是通过在处理程序结束时将 CPSR 从 SPSR 中恢复来完成的。

FIQ 向量是向量表 （在地址 0x1C 处）最后的入口，使得 FIQ 处理程序被直接放置在该向量位置并从该地址顺序执行。它避免了跳转以及相关的延迟，并意味着如果系统有缓存，向量表和 FIQ 处理程序必须被锁定在其所在的块内。这一点非常重要，因为 FIQ 是为尽快处理中断而设计的。五个额外的 FIQ 模式的编组寄存器使得调用与处理程序之间的状态得以保存，从而又加快了其执行速度。

备注

中断处理程序必须包括清除中断来源的代码。

6.7.1 C 语言编写的简单中断处理程序

可通过使用 __irq 函数声明关键字来编写简单的 C 语言中断处理程序。可用 __irq 关键字来编写简单单级中断处理程序，以及调用子例程的中断处理程序。但是，不能用 __irq 关键字来编写 *reentrant*（重入）中断处理程序，因它对 SPSR 不进行存储或恢复。本文中，“重入”的含义是指处理程序能再次设置中断并能将其自身中断。有关详细信息请参阅第 6-29 页的 *可重入中断处理程序*。

__irq 关键字：

- 保护所有 ATPCS 易损坏的寄存器；
- 保护所有被函数使用的其它寄存器 （不包括浮点寄存器）；
- 通过将程序计数器设置到 (lr - 4) 并恢复 CPSR 的初始数据值来退出函数。

如果该函数调用了一个子例程，__irq 除保护其它易损坏寄存器外，还为中断模式保护链接寄存器。有关详细信息请参阅*从中断处理程序调用子例程*。

备注

在使用汇编 Thumb C 代码时， C 语言中断处理程序不能以这种方式生成。在 Thumb 模式下编译时，__irq 关键字有问题，产生中断或任何其它异常时，处理器总是转到 ARM 状态。

但是，由 __irq 函数调用的子例程可以为 Thumb 进行编译（当启用交互时）。有关交互的详细信息，请参阅第 4 章 *ARM 和 Thumb 交互操作*。

从中断处理程序调用子例程

如果从高级中断处理程序调用子例程，__irq 关键字也将从堆中恢复 lr_IRQ 的值，以便它能被 SUBS 指令使用，使得在处理完该中断后返回到正确的地址。

示例 6-14 说明了这一实现过程。高层中断处理程序在 0x80000000 基地址读取存储器映射中断控制器的值。如果该地址的值为 1，高层处理程序跳转到一个用 C 语言编写的处理程序。

示例 6-14

```
__irq void IRQHandler (void)
{
    volatile unsigned int *base = (unsigned int *) 0x80000000;

    if (*base == 1)          // which interrupt was it?
    {
        C_int_handler();     // process the interrupt
    }
    *(base+1) = 0;           // clear the interrupt
}
```

用 armcc 编译，示例 6-14 生成以下代码：

```
IRQHandler PROC
        STMFD    sp!,{r0-r4,r12,lr}
        MOV      r4,#0x80000000
        LDR      r0,[r4,#0]
        SUB      sp,sp,#4
        CMP      r0,#1
```

```
        BLEQ    C_int_handler
        MOV     r0,#0
        STR     r0,[r4,#4]
        ADD     sp,sp,#4
        LDMFD   sp!,{r0-r4,r12,lr}
        SUBS    pc,lr,#4
        ENDP
```

将其与没有使用 __irq 关键字时的结果比较：

```
IRQHandler PROC
        STMFD   sp!,{r4,lr}
        MOV     r4,#0x80000000
        LDR     r0,[r4,#0]
        CMP     r0,#1
        BLEQ    C_int_handler
        MOV     r0,#0
        STR     r0,[r4,#4]
        LDMFD   sp!,{r4,pc}
        ENDP
```

6.7.2 可重入中断处理程序

如果中断处理程序再次激活了中断，然后调用一个子例程，并且产生了另个一中断，则在第二个 IRQ 被占用时，子例程的返回地址（存储在 lr_IRQ 中）将被破坏。在 C 语言中使用 __irq 关键字不会造成重入中断处理程序所要求的对 SPSR 进行存储和恢复，因而要使用汇编语言编写高层中断处理程序。

在跳转到嵌套子例程或 C 函数前，重入中断处理程序须保存 IRQ 状态、切换处理器模式并为新的处理器模式保存该状态。

ARM 版本 4 或以后版本可切换至系统模式。系统模式使用了用户模式寄存器，并启用您的异常处理程序所要求的特权存取方式。有关详细信息请参阅第 6-43 页的*系统模式*。相反，在早于 v4 版本的 ARM 体系结构中，必须切换到超级用户模式。

备注

该方法适用于 IRQ 和 FIQ 中断。但是，因 FIQ 中断要求尽快得到服务，通常仅有一个中断源，所以，可能不必为其提供重入特性。

IRQ 处理程序安全地激活中断所需的步骤是：

1. 构造返回地址并保存在 IRQ 堆中。
2. 保存工作寄存器和 spsr_IRQ。

3. 清除中断源。
4. 切换到系统模式并再次激活中断。
5. 保存用户模式链接寄存器和非被调用函数存储的寄存器。
6. 调用 C 中断处理程序函数。
7. C 中断处理程序返回时，恢复用户模式寄存器并禁用中断。
8. 切换至 IRQ 模式，禁用中断。
9. 恢复工作寄存器和 spsr_IRQ。
10. 从 IRQ 返回。

示例 6-15 展示了它在系统模式下是如何进行的。在 lr_IRQ 被推到栈顶后，寄存器 r12 和 r14 被用作临时工作寄存器。

示例 6-15

```
    AREA INTERRUPT, CODE, READONLY
    IMPORT C_irq_handler
IRQ
    SUB     lr, lr, #4        ; construct the return address
    STMFD   sp!, {lr}         ; and push the adjusted lr_IRQ
    MRS     r14, SPSR         ; copy spsr_IRQ to r14
    STMFD   sp!, {r12, r14}   ; save work regs and spsr_IRQ

    ; Add instructions to clear the interrupt here
    ; then re-enable interrupts.

    MSR     CPSR_c, #0x1F     ; switch to SYS mode, FIQ and IRQ
                              ; enabled. USR mode registers
                              ; are now current.
    STMFD  sp!, {r0-r3, lr}   ; save lr_USR and non-callee
                              ; saved registers
    BL      C_irq_handler     ; branch to C IRQ handler.
    LDMFD   sp!, {r0-r3, lr}  ; restore registers
    MSR     CPSR_c, #0x92     ; switch to IRQ mode and disable
                              ; IRQs. FIQ is still enabled.

    LDMFD   sp!, {r12, r14}   ; restore work regs and spsr_IRQ
    MSR     SPSR_cf, r14
    LDMFD   sp!, {pc}^        ; return from IRQ.
    END
```

本例假设 FIQ 为永久启用。

6.7.3 汇编语言编写的中断处理程序示例

为确保快速执行，通常采用汇编语言来编写中断处理程序。以下章节给出了几个示例：

- *单通道 DMA 传送*；
- 第 6-32 页的*双通道 DMA 传送*；
- 第 6-33 页的*中断的优先化*；
- 第 6-35 页的*上下文切换*。

单通道 DMA 传送

示例 6-16 展示了一个中断处理程序，其执行从 I/O 到存储器传送（软 DMA）中断驱动。该代码是一个 FIQ 处理程序。它使用 FIQ 编组寄存器来维护中断间的状态。该代码最适合位于 0x1C。

在示例代码中：

r8 指向数据读取的 I/O 设备的基本地址。

IOData 是读取基地址到 32 位数据寄存器的偏移量。读取此寄存器清除该中断。

r9 指向被传送数据所在的存储器位置。

r10 指向传送至目的地的最后一个地址。

处理正常传送的全部序列为四个指令。位于有条件返回语句后的代码被用作传送结束信号。

示例 6-16

```
LDR     r11, [r8, #IOData]     ; Load port data from the IO
                               ; device.
STR     r11, [r9], #4          ; Store it to memory: update
                               ; the pointer.
CMP     r9, r10                ; Reached the end ?
SUBLSS  pc, lr, #4             ; No, so return.
                               ; Insert transfer complete
                               ; code here.
```

可通过使用载入字节指令替换载入指令来实现字节传送。从存储器到一个 I/O 设备的传送是通过交换载入和存储指令的选址方式来完成的。

双通道 DMA 传送

除处理两个通道外，示例 6-17 类似于第 6-31 页的示例 6-16。该代码是一个 FIQ 处理程序。它使用 FIQ 编组寄存器来维护中断间的状态。该代码最适合位于 0x1c。

在示例代码中：

r8 指向读取数据的 I/O 设备的基本地址。

IOStat 是从基本地址到寄存器的偏移量，指示两个端口中的哪个造成了中断。

IOPort1Active 是一个位屏蔽，指示是否是第一个端口造成了中断（否则就假设第二个接口造成了中断）。

IOPort1, IOPort2 是要读取的寄存器的偏移量。读取一个数据寄存器清除了相应端口的中断。

r9 指向从第一个端口传送数据到该存储器的位置。

r10 指向从第二个端口传送数据到该存储器的位置。

r11, r12 指向传送目的地的最后一个地址（r11 用于第一个端口，r12 用于第二个端口）。

处理正常传送的全部序列为九个指令。位于有条件返回语句后的代码被用作传送结束信号。

示例 6-17

```
LDR     r13, [r8, #IOStat]      ; Load status register to find which port
                                ; caused the interrupt.
TST     r13, #IOPort1Active
LDREQ   r13, [r8, #IOPort1]     ; Load port 1 data.
LDRNE   r13, [r8, #IOPort2]     ; Load port 2 data.
STREQ   r13, [r9], #4           ; Store to buffer 1.
STRNE   r13, [r10], #4          ; Store to buffer 2.
CMP     r9, r11                 ; Reached the end?
CMPLE   r10, r12                ; On either channel?
SUBNES  pc, lr, #4              ; Return
                        ; Insert transfer complete code here.
```

可通过使用载入字节指令替换载入指令来实现字节传送。从存储器到 I/O 设备的传送是通过交换条件载入和条件存储指令的选址方式来完成的。

中断的优先化

示例 6-18 为高达 32 个的中断源分派了其相应的处理程序例程。由于它是为使用正常中断向量 (IRQ) 而设计的，所以从位置 0x18 跳转。

由外设模块来对中断进行优先级处理，并在 I/O 寄存器中提供当前的有效高优先级中断。

在示例代码中：

IntBase 存储中断控制器的基本地址。

IntLevel 存储包含现行中断高度优先级的寄存器的偏移量。

r13 假设指向一个小而完全递减的堆。

十条指令后（包括跳转到本代码的那条指令）重新开启了中断。

再经过两条指令进入了每个中断的特定处理程序（所有寄存器均存储在堆中）。

此外，每个处理程序的最后三条指令在再次关闭中断的情况下执行，因而可以保证从堆中安全地恢复 SPSR。

备注

Application Note 30：*软件中断的优先化*介绍了使用软件进行多个中断源的优先化，与此处所述的用硬件进行的优先化相反。

示例 6-18

```
; first save the critical state
SUB     lr, lr, #4              ; Adjust the return address
                                ; before we save it.
STMFD   sp!, {lr}               ; Stack return address
MRS     r14, SPSR               ; get the SPSR ...
STMFD   sp!, {r12, r14}         ; ... and stack that plus a
                                ; working register too.
```

```
                                    ; Now get the priority level of the
                                    ; highest priority active interrupt.
    MOV     r12, #IntBase           ; Get the interrupt controller's
                                    ; base address.
    LDR     r12, [r12, #IntLevel]   ; Get the interrupt level (0 to 31).

    ; Now read-modify-write the CPSR to enable interrupts.

    MRS     r14, CPSR               ; Read the status register.
    BIC     r14, r14, #0x80         ; Clear the I bit
                                    ; (use 0x40 for the F bit).
    MSR     CPSR_c, r14             ; Write it back to re-enable
                                    ; interrupts and
    LDR     pc, [pc, r12, LSL #2]   ; jump to the correct handler.
                                    ; PC base address points to this
                                    ; instruction + 8
    NOP                             ; pad so the PC indexes this table.

                                    ; Table of handler start addresses
    DCD     Priority0Handler
    DCD     Priority1Handler
    DCD     Priority2Handler
; ...
    Priority0Handler
    STMFD   sp!, {r0 - r11}         ; Save other working registers.
                                    ; Insert handler code here.
; ...
    LDMFD   sp!, {r0 - r11}         ; Restore working registers (not r12).

    ; Now read-modify-write the CPSR to disable interrupts.
    MRS     r12, CPSR               ; Read the status register.
    ORR     r12, r12, #0x80         ; Set the I bit
                                    ; (use 0x40 for the F bit).
    MSR     CPSR_c, r12             ; Write it back to disable interrupts.

    ; Now that interrupt disabled, can safely restore SPSR then return.
    LDMFD   sp!, {r12, r14}         ; Restore r12 and get SPSR.
    MSR     SPSR_csxf, r14          ; Restore status register from r14.
    LDMFD   sp!, {pc}^              ; Return from handler.
Priority1Handler
; ...
```

上下文切换

第 6-35 页的示例 6-19 实现了在用户模式下的环境切换。其代码是基于指向它要运行的过程的*过程控制块* (PCBs) 指针列表。

图 6-5 展示了示例所期望的 PCB 布局。

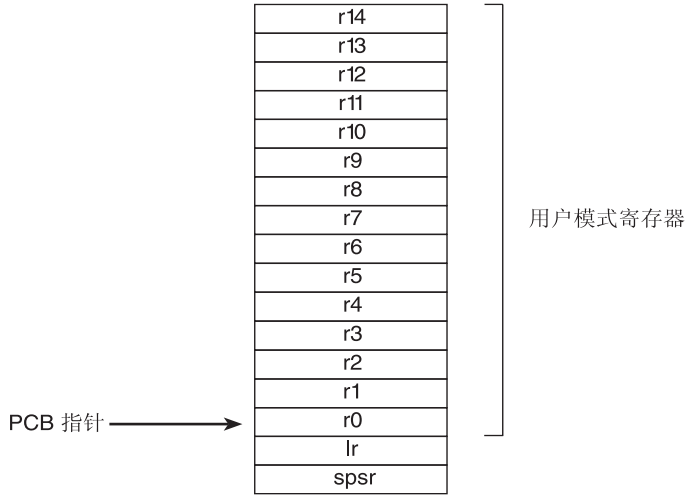

图 6-5 PCB 布局

指向下一个要运行的过程的 PCB 指针是由 r12 完成的，且该项列表末尾有一零指针。r13 寄存器是指向 PCB 的指针，并在一小段时间间隔被保存，因而，进入时它指向现行运行过程的 PCB。

示例 6-19

```
STMIA   r13, {r0 - r14}^        ; Dump user registers above r13.
MRS     r0, SPSR                ; Pick up the user status
STMDB   r13, {r0, lr}           ; and dump with return address below.
LDR     r13, [r12], #4          ; Load next process info pointer.
CMP     r13, #0                 ; If it is zero, it is invalid
LDMNEDB r13, {r0, lr}           ; Pick up status and return address.
MSRNE   SPSR_cxsf, r0           ; Restore the status.
LDMNEIA r13, {r0 - r14}^        ; Get the rest of the registers
NOP
SUBNES  pc, lr, #4              ; and return and restore CPSR.
                ; Insert "no next process code" here.
```

6.8 复位处理程序

复位处理程序实施的操作取决于所开发软件的运行系统。例如，可以是：

- 设置异常向量。详细信息请参阅第 6-13 页的*安装异常处理程序*。
- 初始化堆和寄存器。
- 如果使用 MMU，初始化存储器系统。
- 初始化任何关键 I/O 设备。
- 激活中断。
- 改变处理器模式和 / 或状态。
- 初始化 C 语言和调用主应用程序所需的变量。

有关详细信息请参阅第 2 章*嵌入式软件开发*。

6.9 未定义的指令处理程序

提供处理器不能识别的指令给系统所配的协处理器。如果该指令仍不能被识别，就产生了未定义指令异常。可能的情形是，该指令为协处理器所建，但相关的协处理器 （如浮点加速器）没有配到系统上。但是，可能会有一个软件模拟程序来替代该协处理器。

这样的模拟程序必须：

1. 将其分配到未定义指令向量并存储旧的内容。
2. 检查该未定义指令确定其是否必须进行模拟。这与 SWI 处理程序提取 SWI 编号类似，但模拟程序不是提取底部的 24 位，而是提取 27-24 位。

   这些位按如下方式决定指令是否为协处理器操作：

   - 如果 27 到 24 位 = b1110 或 b110x，该指令即为协处理器指令。
   - 如果 8-11 位显示该协处理器模拟程序须处理该项指令，则模拟程序必须处理该指令并返回用户程序。
3. 否则，模拟程序必须通过使用模拟程序安装时存储的向量，将该异常传递到其原始处理程序 （或链表中的下一个模拟程序）。

链表上的模拟程序用完时，不再处理指令，因而，未定义指令处理程序必须报告出错并退出。有关详细信息请参阅第 6-41 页的*链结异常处理程序*。

备注

Thumb 指令集没有协处理器指令，因此不要求未定义指令处理程序模拟这类指令。

6.10 预取中断处理程序

如果系统中没有 MMU，预取中断处理程序可报告出错并退出。否则，造成中断的地址必须存储在物理存储器中。lr_ABT 指向造成中断的下一条指令的地址，因此，该地址被存储在 lr_ABT - 4。虚拟存储器报告该地址出错就会得到处理，并再次进行取指令操作。处理程序返回同一个指令而不是它下面的那个指令，例如：

```
SUBS    pc,lr,#4
```

6.11 数据中断处理程序

如果没有 MMU，数据中断处理程序必须报告出错并退出。如果有 MMU，处理程序必须处理虚拟存储器错误。

造成中断的指令是在 lr_ABT - 8，因为 lr_ABT 指向造成中断指令的下面两个指令。

以下类型的指令均可造成该中断：

单一寄存器载入或存储 （LDR 或 STR）

反应取决于处理器类型：

- 如果中断发生在以 ARM6 为基础的处理器：
  - 如果处理器是在早期中断模式并且要求回写，则不刷新地址寄存器 。
  - 如果处理器是在晚期中断模式并且要求回写，则刷新地址寄存器 其刷新一定是未完成。
- 如果中断产生在以 ARM7 为基础的处理器，包括 ARM7TDMI，则刷新地址寄存器，并刷新一定是未完成。
- 如果中断是发生在基于 ARM9™、ARM10™ 或 StrongARM 的处理器，该地址被处理器恢复到该指令开始前的值。不必进行进一步的恢复操作。

交换 (SWP) 该指令不涉及地址寄存器刷新。

载入多个或存储多个 （LDM 或 STM）

反应取决于处理器类型：

- 如果中断产生在基于 ARM6 或 ARM7 的处理器，启用回写，刷新基址寄存器，就好像产生了完整的传送。
  在寄存器列表中存在含有基址寄存器的 LDM 情况下，处理器以修改了的基值来替换该覆盖值，以便能够恢复。可以使用所涉及到的寄存器编号重新计算初始基值。
- 如果中断产生在基于 ARM9 或 ARM10 的处理器且启用回写，该基址寄存器则被恢复到该指令开始前它所保存的值。

上述三个例子中，MMU 均可将所需的虚拟存储器装入物理存储器。MMU *故障地址寄存器* (FAR) 包含造成该中断的地址。完成后，处理程序可返回并尝试再次执行该指令。

可在 *Examples_directory*\databort 找到数据中断处理程序代码的示例。

6.12 链结异常处理程序

某些情况下，一个特定的异常可能会有几个不同的源。例如：

- Angel 使用一条未定义指令来执行断点。但是，当该协处理器不存在时执行协处理器指令也会发生未定义指令异常。
- Angel 使用一个多用途的 SWI，例如从用户模式进入超级用户模式，并支持开发过程中的半主机申请。但是， RTOS 或应用程序也可能会执行一些 SWI。

在此情况下，可采用下列方法扩展该异常处理代码。

- *单层扩展处理程序*;
- *几个链结的处理程序*。

6.12.1 单层扩展处理程序

某些情况下，将异常处理程序的代码加以扩展来确定产生异常的源是可能的，然后直接调用相应代码。这种情况下，需要为异常处理程序修改代码。

Angel 是为简化该方法而编写。 Angel 将 SWI 和未定义指令进行解码，并且 Angel 异常处理程序可进行扩展来处理非 Angel SWI 和未定义指令。

但是，该方法仅在编写单层异常处理程序时所有的异常源均为已知的情况下才是有用的。

6.12.2 几个链结的处理程序

某些情况下不仅仅要求单层处理程序。在标准 Angel 调试器执行的情况下，下载了某个想支持其它 SWI 的独立用户应用程序（或 RTOS）。新载入的应用程序可能会有其完全独立的异常处理程序要求进行安装，但它不能取代 Angel 处理程序。

在此情况下，当新处理程序发现异常源不是它能处理的源时，则必须标记原有的处理程序的地址、以便该新处理程序能够调用原有的处理程序。例如，在发现某个 SWI 不是 RTOS SWI 时， RTOS SWI 处理程序调用 Angel SWI 处理程序，使得该 Angel SWI 处理程序有机会处理它。

该方法可扩展到任何多个级，以建立一个处理程序链。虽然采用该方法的代码能使每一处理程序完全独立，但是，它比采用单层处理程序的效率要低，或至少它在处理程序链上走得越深，其效率就会越低。

第 6-15 页的*从 C 安装处理程序*中给出的两个例程都返回原有的向量内容。可将其解码以便将其赋予：

跳转指令的偏移量

可利用其计算初始处理程序的位置，使其激活一个新的跳转指令，构建该指令并将其存储在存储器中的合适地点。如果替换的处理程序无法处理该异常，它可跳转到所构建的转移指令然后再跳转到原先的处理程序。

用来存储原先的处理程序地址的位置

如果应用程序的处理程序无法处理该异常，必须从该位置载入程序计数器。

多数情况下，不必进行这样的计算，因为调试监督程序或 RTOS 处理程序的信息可供您使用。如果是这样，要求链入下一个处理程序的指令可被硬性编码到该应用程序中。处理程序的最后一部分必须检查所处理的异常产生原因。如检查到，处理程序可返回到该应用程序。如没有检查到，则必须调用链上的下一个处理程序。

备注

在调试监督处理程序之前链结一个处理程序，从系统中清除该监督程序时必须清除该链，然后直接安装应用程序的处理程序。

6.13 系统模式

ARM 体系结构定义了一个用户模式，该模式有 15 个通用寄存器、一个 PC 和一个 CPSR。除该模式外，还有五个处理器特权模式，每个模式都有一个 SPSR 和许多寄存器，以便替换用户模式下 15 个通用寄存器中的部分寄存器 。

备注

本节内容仅适用于 ARM 体系结构的版本 v4、 v4T 和更新的版本。

发生处理器异常时，当前程序计数器被复制到异常模式的链寄存器中，CPSR 被复制到该异常模式的 SPSR 中。然后， CPSR 以与异常相关的方式被改变，程序计数器被设置到一个异常定义的地址，开始执行异常处理程序。

改变程序计算器前，ARM 子例程调用指令 (`BL`) 将返回地址复制到 r14 中，因而，子例程返回指令将 r14 移动到 PC (`MOV pc,lr`)。

所有这些动作说明，处理异常的 ARM 模式必须确保如果其调用子例程，不会产生另一个同类型的异常，因为该子例程的返回地址被异常的返回地址所覆盖。

（在早期版本的 ARM 体系结构中，解决这个问题的方法是尽量避免在异常模式下进行子例程调用或避免从特权模式改为用户模式。第一个解决方法限制太大，第二个则说明可能不允许任务具有其正确运行所要的特权存取）。

ARM 体系结构版本 v4 和更新版本提供了一个被称为 *system*（系统）模式的处理器模式来克服这个问题。系统模式是一个有特权的处理器模式，该模式共享用户模式的寄存器。可在该模式下运行特权模式任务，且异常不再覆盖链接寄存器。

备注

异常不能进入系统模式。异常处理程序修改 CPSR 以便进入系统模式。请参阅第 6-29 页的*可重入中断处理程序*的一个示例。

第 7 章
调试通信通道

本章解释如何使用*调试通信通道* (DCC)。其中包含下列各部分：

7.1 关于调试通信通道

ARM 内核（如 ARM7TDMI 和 ARM9TDMI®）中的 EmbeddedICE 逻辑单元包含调试通信通道。它允许使用 JTAG 端口和协议转换器（如 Multi-ICE®）在目标和主机调试器之间传递数据，无需停止程序流程或进入调试状态。本章描述目标上运行的程序和主机调试器如何访问调试通信通道。

RVCT 通过 Multi-ICE 半主机提供对调试通信通道的访问。

7.2 目标数据传送

目标使用 ARM 指令 MCR 和 MRC 将调试通信通道作为 ARM 内核上的协处理器 14 进行访问。

提供了两个寄存器用于传送数据：

通信数据读取寄存器

一个 32 位宽寄存器，用于接收来自调试器的数据。以下指令在 Rd 中返回读取寄存器的值：

```
MRC p14, 0, Rd, c1, c0
```

通信数据写入寄存器

一个 32 位宽寄存器，用于向调试器发送数据。以下指令将 Rn 值写到写入寄存器：

```
MCR p14, 0, Rn, c1, c0
```

备注

有关访问 ARM 10 内核 DCC 寄存器的信息，请参阅 *ARM10 Technical Reference Manual*。在 ARM9 之后的处理器中，所用指令、状态位位置以及状态位解释都有所不同。

7.3 轮询调试通信

在通信数据读取和写入寄存器之外，调试通信通道提供了通信数据控制寄存器。

以下指令在 Rd 中返回控制寄存器值：

```
MRC p14, 0, Rd, c0, c0
```

控制寄存器中的两个位提供目标和主机调试器之间的同步握手：

位 1 （W 位） 表示通信数据写入寄存器是否空闲 （从目标的角度）：
- **W = 0** 目标应用程序可以写入新数据。
- **W = 1** 主机调试器可以从写入寄存器中扫描出新数据。

位 0 （R 位） 表示通信数据读取寄存器中是否有新数据（从目标的角度）：
- **R = 1** 有新数据，目标应用程序可以读取。
- **R = 0** 主机调试器可以将新数据扫描到读取寄存器中。

备注

调试器不能用协处理器 14 直接访问调试通信通道，因为这对调试器无意义。但是，调试器可以使用扫描链读写调试通信通道寄存器。调试通信通道数据和控制寄存器映射到 EmbeddedICE 逻辑单元中的地址，请参阅*查看 EmbeddedICE 逻辑寄存器*。

7.3.1 查看 EmbeddedICE 逻辑寄存器

要查看 EmbeddedICE 逻辑寄存器，请参阅 *Multi-ICE 2.2 版用户指南*或更新版本。

7.3.2 目标到调试器的通信

这是用于 ARM 内核上运行的应用程序与主机上运行的调试器进行通信的一系列事件：

1. 目标应用程序检查调试通信通道写入寄存器是否空闲可用。 为此，它使用 MRC 指令读取调试通信通道控制寄存器，以检查 W 位是否已清除。

2. 如果 W 位已清除，则调试通信写入寄存器是已清除的，而应用程序对协处理器 14 使用 MCR 指令将字写入寄存器。写入寄存器操作自动设置 W 位。如果已设置 W 位，则调试器还没有清空调试通信写入寄存器。如果应用程序需要发送另一个字，它必须轮询 W 位，直到它已清除。

3. 调试器通过扫描链 2 轮询调试通信控制寄存器。如果调试器看到 W 位已设置，则它可以读调试通信通道数据寄存器，以读取应用程序发送的消息。读取数据进程自动清除调试通信控制寄存器中的 W 位。

示例 7-1 说明了这一实现过程。示例代码可以在 *Examples_directory*\dcc\outchan.s 中得到。

示例 7-1

```
     AREA  OutChannel, CODE, READONLY
     ENTRY
     MOV   r1,#3          ; Number of words to send
     ADR   r2, outdata    ; Address of data to send
pollout
     MRC   p14,0,r0,c0,c0 ; Read control register
     TST   r0, #2
     BNE   pollout        ; if W set, register still full
write
     LDR   r3,[r2],#4     ; Read word from outdata
                          ; into r3 and update the pointer
     MCR   p14,0,r3,c1,c0 ; Write word from r3
     SUBS  r1,r1,#1       ; Update counter
     BNE   pollout        ; Loop if more words to be written
     MOV   r0, #0x18      ; Angel_SWIreason_ReportException
     LDR   r1, =0x20026   ; ADP_Stopped_ApplicationExit
     SWI   0x123456       ; ARM semihosting SWI
outdata
     DCB "Hello there!"
     END
```

要执行此示例：

1. 汇编 outchan.s：

   ```
   armasm -g outchan.s
   ```

2. 链接输出对象：

   ```
   armlink outchan.o -o outchan.axf
   ```

   链接步骤创建可执行文件 outchan.axf

3. 载入并执行该映像。详细信息请参阅所用调试程序的文档。

7.3.3 调试器到目标的通信

这是用于从主机上运行的调试器向内核上运行的应用程序进行消息传递的一系列事件：

1. 调试器轮询调试通信控制寄存器的 R 位。如果 R 位已清除，则调试通信读取寄存器已清除，可以在此写入数据供目标应用程序读取。

2. 调试器通过扫描链 2 将数据扫描到调试通信读取寄存器中。此操作自动设置调试通信控制寄存器中的 R 位。

3. 目标应用程序轮询调试通信控制寄存器中的 R 位。如果它已设置，则调试通信读取寄存器中有数据，应用程序可以使用 MRC 指令从协处理器 14 读，以读取该数据。作为读指令的一部分，R 位被清除。

   位于 *Examples_directory*\dcc\inchan.s 中的以下目标应用程序代码片段显示此操作如何工作：

```
     AREA  InChannel, CODE, READONLY
     ENTRY
     MOV   r1,#3          ; Number of words to read
     LDR   r2, =indata    ; Address to store data read
pollin
     MRC   p14,0,r0,c0,c0 ; Read control register
     TST   r0, #1
     BEQ   pollin         ; If R bit clear then loop
read
     MRC   p14,0,r3,c1,c0 ; read word into r3
     STR   r3,[r2],#4     ; Store to memory and
                          ; update pointer
     SUBS  r1,r1,#1       ; Update counter
     BNE   pollin         ; Loop if more words to read
     MOV   r0, #0x18      ; Angel_SWIreason_ReportException
     LDR   r1, =0x20026   ; ADP_Stopped_ApplicationExit
     SWI   0x123456       ; ARM semihosting SWI
```



```
    AREA  Storage, DATA, READWRITE
indata
    DCB   "Duffmessage#"
    END
```

4. 在主机上创建一个输入文件，其中包含 （例如） And goodbye!。

5. 用以下命令汇编并链接此代码：

```
armasm -g inchan.s
armlink inchan.o -o inchan.axf
```

在称为 inchan 的文件中创建了一个可执行映像。要载入并执行该映像，请参阅您的调试器文档。

7.4 中断驱动调试通信

第 7-4 页的*轮询调试通信*中提供的示例演示如何轮询 DCC。通过将 Embedded ICE 逻辑单元中的 COMMRX 和 COMMTX 信号链接到中断控制程序，可以将这些示例转换成中断驱动示例。

然后，以上提供的读写代码可以移到中断处理程序中。

有关编写中断处理程序的信息，请参阅第 6-27 页的*中断处理程序*。

7.5 从 Thumb 状态访问

因为 Thumb 指令集不包含协处理器指令，所以当内核处于 Thumb 状态时，不能使用调试通信通道。

有三种可能的解决方法：

- 可以在 SWI 处理程序中编写每个轮询例程，然后可以在 ARM 或 Thumb 状态下执行该处理程序。进入 SWI 处理程序立即将内核转到 ARM 状态，在该状态下可以使用协处理器指令。有关 SWI 的详细信息，请参阅第 6 章*处理处理器异常*。
- Thumb 代码可以交互调用 ARM 子例程，由该子例程实现轮询。有关混合使用 ARM 和 Thumb 代码的详细信息，请参阅第 4 章 *ARM 和 Thumb 交互操作*。
- 使用中断驱动通信，而不使用轮询通信。中断处理程序用 ARM 指令编写，因此可以直接访问协处理器指令。

7.6 半主机

如果使用 `$semihosting_enabled=2` 的 Multi-ICE，可以将调试通信通道用于半主机。有关的更多信息，请参阅 *Multi-ICE v2.2 用户指南*。

词汇表

Angel Angel 是一个在目标系统上运行的调试监控程序，使您能开发和调试在基于 ARM 的硬件上运行的应用程序。 Angel 可调试运行在 ARM 状态或 Thumb 状态下的应用程序。

ANSI *参阅*美国国家标准协会。

ARMulator RealView ARMulator ISS 是一个指令集模拟器。它是一个可模拟各种 ARM 处理器指令集和体系结构的模块集合。

ARM-Thumb 过程调用标准 (ATPCS)

*ARM-Thumb 过程调用标准*定义在子程序调用过程当中如何使用寄存器和堆栈。

ATPCS *参阅* ARM-Thumb 过程调用标准。

半主机 (semihosting)

一种通信机制，目标将应用程序请求的 I/O 操作传送给主机系统处理，而不是试图自身支持 I/O。

半字 (half-word) 一个 16 位的信息单元。除非另外声明，否则将数字当作无符号整数。

CFA *请参阅*规范框架地址。

程序映像 请参阅映像。

重新进入 子例程具有多个活动代码实例的能力。子例程调用的每个实例具有各自所需的所有静态数据副本。

重新确定 将为一个运行环境设计的代码移到新运行环境中的过程。

重新映射 应用程序开始执行之后，更改物理内存或设备的地址。这样做通常是为了在初始化结束时启用 RAM 替代 ROM。

存储器管理单元 (MMU)

控制高速缓存和存储器块访问权限，并将虚拟地址转换为物理地址的硬件。

DWARF 一种特殊的调试格式。

大端 (Big-Endian) 将一个字的最低位字节存储在比最高位字节更高的地址上的存储结构。

读写位置无关 (RWPI) 读 / 写数据地址可在运行时动态改变。

段 映像中的一段代码或数据。

*另请参阅*输入段。

堆 可用于创建新变量的计算机内存部分。

EC++ 一个用于嵌入式应用程序的 C++ 变体。

ELF 可执行、可链接格式。

分散加载 单独分配地址和对代码和数据段进行分组，而不使用单个大块。

规范框架地址 (CFA) 在 DWARF 2 中，这是指栈上的一个地址，其指定一个中断函数的调用框架的位置。

国际标准组织 (ISO) 制定计算机软件和其它行业标准的一个组织。

ICE 线路中仿真器 (In Circuit Emulator)。

ISO *参阅*国际标准组织。

胶合代码 (Veneer) 是指当需要改变处理器状态或跳转目的在当前处理状态所能到达的范围之外时，与程序调用或跳转配合使用的一小块代码。

交互操作 (Interworking)

产生使用 ARM 和 Thumb 两种指令的应用程序。

紧偶合存储器 (TCM) 启用后，紧偶合存储器将替代块芯片外存储器区域。

局部 只能由创建它的子程序访问的对象。

库 (Library) 由汇编程序或编译程序产生的目标对象组合形成的一种集合。

链接程序 (Linker) 由一个或多个源代码汇编程序或编译程序产生的对象生成一个映像的软件。

Multi-ICE 用于嵌入式系统的基于 JTAG 的多处理器调试工具。Multi-ICE 是一个 ARM 注册商标。

MMU *请参阅*存储器管理单元。

美国国家标准协会 (ANSI)

制定计算机软件和其它行业标准的一个组织。

目标 (Target) 正在运行目标应用程序的实际目标处理器 （真实的或模拟的）。

内联 不使用公共子程序，而是在每次使用时以代码重复执行的功能。放置在 C 或 C++ 程序中的汇编程序代码。

*另请参阅*输出段和嵌入式。

PCS 过程调用标准 (Procedure Call Standard)。

另请参阅 ATPCS。

PIC 位置无关代码。

另请参阅 ROPI。

PID 位置无关的数据*或* ARM 平台无关开发 (PID) 卡。

另请参阅 RWPI

剖析 执行调试程序期间收集统计信息，以度量性能或确定代码的关键区域。

嵌入式 作为固件开发的应用程序。汇编程序函数在 C 或 C++ 程序中外联使用。

*另请参阅*内联。

区 在映像中，区是指一至三个 （RO、 RW 和 ZI）输出段的相连序列。

RealView 编译工具 (RVCT)

RealView 编译工具是一整套工具以及支持文档和范例，使用户能编写和编译用于 ARM 系列 RISC 处理器的应用程序。

ROPI *另请参阅*只读位置无关。

RTOS 实时操作系统。

RVCT *请参阅* RealView 编译工具。

RWPI *请参阅*读写位置无关。

软件中断 (SWI) 使处理器调用程序员指定的子程序的一个指令。 ARM 在处理半主机时候用到。

SWI *参阅*软件中断。

闪存 (Flash memory)

常用于保存应用程序代码的永久内存。

输出段 具有相同 RO、RW 或 ZI 属性的输入段的相连序列。段在更大的、称为区的片段中组合在一起。多个区组合在一起，成为最终的可执行映像。

*另请参阅*区。

输入段 输入段包含代码或初始化数据，或者描述应用程序启动前必须设为 0 的存储器片段。

*另请参阅*输出段。

双字 (Double-word) 一个 64 位的信息单元。除非另外声明，否则将数字当作无符号整数。

TCM *请参阅*紧偶合存储器。

线程 (Thread) 处理器上的一种执行环境。线程总是与处理器相关，可能 （或可能不）与映像相关。

协处理器 (Coprocessor)

用于特定运算的附加处理器。通常用于浮点数学计算、信号处理或内存管理。

映像 已加载到处理器上可以运行的可执行文件。

加载到处理器上并拥有了执行线程的二进制执行文件。一个映像可能有多个线程。一个映像与运行其默认线程的处理器相关。

载入视图 在映像载入内存但尚未开始执行时，区域和段的地址。

只读位置无关 (ROPI) 代码和只读数据地址可在运行时动态改变。

执行视图 在映像载入内存并开始执行之后区域和段的地址。

字 (Word) 一个 32 位的信息单元。除非另外声明，否则将数字当作无符号整数。

索引

本索引中的条目按字母顺序列出，符号和数字显示在末尾。所给出的参考是按页码提供的。

A

B

C

D

E

F

G

H

J

K

L

M

N

P

Q

R

S

T

W

X

Y

Z

数字

符号