RealView® 编译工具

3.1 版

开发指南

ARM®

ARM DUI 00203HC

RealView 编译工具
开发指南

版本信息

本手册进行了以下更改。

更改历史记录

日期	发行号	保密性	更改
2002 年 8 月	A	非保密	1.2 版
2003 年 1 月	B	非保密	2.0 版
2003 年 9 月	C	非保密	RVDS v2.0 的 2.0.1 版
2004 年 1 月	D	非保密	RVDS v2.1 的 2.1 版
2004 年 12 月	E	非保密	RVDS v2.2 的 2.2 版
2005 年 5 月	F	非保密	RVDS v2.2 SP1 的 2.2 版
2006 年 3 月	G	非保密	RVDS 3.0 的 3.0 版
2007 年 3 月	H	非保密	RVDS 3.1 的 3.1 版

所有权声明

保密状态

本文档的内容是非保密的。根据 ARM 与 ARM 将本文档交予的参与方的协议条款，使用、复制和公开本文档内容的权利可能会受到许可限制的制约。

产品状态

本文档的信息是开发的产品的最新信息。

网址

`http://www.arm.com`

目录
RealView 编译工具
开发指南

附录 A **半主机**

前言

本前言介绍 *RealView 编译工具开发指南*。它包含以下几节：

关于本手册

本手册包含开发 ARM® *系列精简指令集计算* (RISC) 系列处理器代码的特定问题的帮助信息。本手册中的各章节和所用示例都假设您开发代码时使用的是最新版本的 *RealView® 编译工具* (RVCT)。

适用对象

本手册是为所有基于 ARM 体系结构的处理器编写代码的开发者编写的。本手册假定您是一位有经验的软件开发人员，并且熟悉 *RealView 编译工具要点指南* 中所介绍的开发工具。

使用本手册

本手册由以下章节组成：

第 1 章 *简介*

本章简要介绍了 RVCT。

第 2 章 *嵌入式软件开发*

本章提供了有关如何用 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缺省 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 *编写与位置无关的代码和数据*

本章提供了如何编写与位置无关的代码和数据，使之符合 *ARM 体系结构的过程调用标准* (AAPCS) 的相关信息。

第 4 章 *交互操作 ARM 和 Thumb*

本章提供了有关编写执行 Thumb 指令集的处理器代码时，如何在 ARM 状态和 Thumb 状态之间切换的信息。

第 5 章 *混合使用 C、C++ 和汇编语言*

本章提供了有关如何编写 C、C++ 和 ARM 汇编语言混合代码的信息。本章还介绍如何从 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*

本章提供了有关如何处理 ARM 处理器（而不是 Cortex-M1 或 Cortex-M3 处理器）支持的各种异常类型的信息。

第 7 章 ***处理 Cortex-M3 处理器异常***

本章提供了有关如何处理 Cortex-M3 处理器支持的各种异常类型的信息。

第 8 章 ***调试通信通道***

本章提供了有关如何使用 *调试通信通道* (DCC) 的说明。

第 9 章 ***半主机***

本附录提供了有关半主机机制的信息，该机制可使运行在 ARM 目标上的代码使用运行 ARM 调试器的主机上的 I/O 功能。

本手册假定 ARM 软件安装在缺省位置。例如，在 Windows 上，这可能是 *volume*:\Program Files\ARM。引用路径名时，假定安装位置为 *install_directory*。例如，*install_directory*\Documentation\...。如果将 ARM 软件安装在其他位置，则可能需要更改此位置。

印刷约定

本手册使用了以下印刷约定:

monospace　表示可以从键盘输入的文本，如命令、文件和程序名以及源代码。

mono**space**　表示允许的命令或选项缩写。可只输入下划线标记的文本，无需输入命令或选项的全名。

monospace italic

表示此处的命令和函数的自变量可用特定值代替。

等宽粗体

表示在示例代码以外使用的语言关键字。

斜体　突出显示重要注释、介绍特殊术语以及表示内部交叉引用和引文。

粗体　突出显示界面元素，如菜单名称。有时候也用在描述性列表中以示强调，以及表示 ARM 处理器信号名称。

更多参考出版物

本部分列出了 ARM 公司和第三方发布的、可提供有关 ARM 系列处理器开发代码的附加信息的出版物。

ARM 将定期对其文档进行更新和更正。有关最新勘误表、附录以及 ARM 常见问题 (FAQ)，请访问 http://www.arm.com。

ARM 公司出版物

本手册包含开发 ARM 系列处理器应用程序的一般信息。有关其他组件的信息，请参阅 RVCT 文档集中的以下手册：

- *《RVCT 要点指南》* (ARM DUI 0202)
- *《RVCT 编译器用户指南》* (ARM DUI 0205)
- *《RVCT 编译器参考指南》* (ARM DUI 0348)
- *《RVCT 库和浮点支持指南》* (ARM DUI 0349)
- *《RVCT 链接器和实用程序指南》* (ARM DUI 0206)。
- *《RVCT 汇编程序指南》* (ARM DUI 0204)
- *《RealView Development Suite 术语表》* (ARM DUI 0324)。

有关基本标准、软件接口以及 ARM 支持的标准的完整信息，请参阅 *install_directory*\Documentation\Specifications\...。

此外，有关与 ARM 产品相关的特定信息，请参阅下列文档：

- *《ARM6-M 体系结构参考手册》* (ARM DDI 0419)
- *《ARM7-M 体系结构参考手册》* (ARM DDI 0403)
- *《ARM 体系结构参考手册》，ARMv7-A 和 ARMv7-R 版* (ARM DDI 0406)
- *《ARM 体系结构参考手册 Thumb® 2 补充》* (ARM DDI 0308)
- *《ARM 体系结构参考手册安全扩展补充》* (ARM DDI 0309)
- *《ARM 体系结构参考手册 Thumb-2 执行环境补充》* (ARM DDI 0376)
- *《ARM 体系结构参考手册高级 SIMD 扩展和 VFPv3 补充》* (ARM DDI 0268)
- *《ARM 参考外围设备规范》* (ARM DDI 0062)
- 您的硬件设备的 ARM 数据表或技术参考手册。

其他出版物

有关对 ARM 体系结构的介绍，请参阅 Andrew N. Sloss、Dominic Symes 和 Chris Wright 合著的*《ARM 系统开发人员指南：设计和优化系统软件》*(2004)。Morgan Kaufmann，ISBN 1-558-60874-5。

有关使用 ARM 处理器内核的片上系统设计人员和使用 ARM 的工程师的要点手册，请参阅 Steve Furber 编著的*《ARM 芯片系统体系结构》*（第二版，2000）。Addison Wesley，ISBN 0-201-67519-6。

反馈

ARM 公司欢迎用户为 RealView 编译工具及其文档提供反馈意见。

对 RealView 编译工具的反馈

如果您有关于 RVCT 的任何问题，请与您的供应商联系。为便于他们快速提供有用的答复，请提供：

- 您的姓名和公司
- 产品序列号
- 您所用版本的详细信息
- 您运行的平台的详细信息，如硬件平台、操作系统类型和版本
- 能重现问题的一小段独立的程序
- 您预期发生和实际发生的情况的详细说明
- 您使用的命令，包括所有命令行选项
- 能说明问题的例程输出
- 工具的版本字符串，包括版本号和日期。

关于本手册的反馈

如果您发现本手册有任何错误或遗漏之处，请发送电子邮件到 errata@arm.com，并提供：

- 文档标题
- 文档编号
- 您有疑问的页码
- 问题的简要说明。

我们还欢迎您对需要增加和改进之处提出建议。

第 1 章
简介

本章简要介绍本手册，同时开始讲述如何使用 *RealView® 编译工具* (RVCT) 来开发代码。它包含以下几节：

- 第 1-2 页的*关于 RVCT*
- 第 1-3 页的*为 ARM 处理器开发代码*

1.1 关于 RVCT

RVCT 由一套应用程序以及支持文档和示例组成，用于编写适合 ARM® 系列 RISC 处理器的应用程序。您可以使用 RVCT 来编译 C、C++ 和 ARM 汇编语言程序。

本手册包含的信息可帮助您解决在开发基于 ARM 系统的代码时可能会遇到的问题。本手册中的各章节和所用示例都假设您开发代码时使用的是最新版本的 RVCT。

如果要将 RVCT 的旧版本升级，请务必阅读 *RealView 编译工具要点指南*，以获得此版本的新功能以及改进提高的相关信息。

如果您不熟悉 RVCT，请阅读 *RealView 编译工具要点指南*，以对 ARM 工具及其在开发工程项目中的使用方法有一个大致了解。

有关 RVCT 旧版本的相关信息，请参阅 *RealView 编译工具要点指南* 中的附录 A。

请参阅第 x 页的*ARM 公司出版物*，以获取有关 RVCT 文档套件中的其他手册的列表，它们提供有关 ARM 汇编程序、编译器和支持软件的相关信息。

1.1.1 使用示例

本手册引用了 RealView Development Suite 随附的示例，这些示例位于主示例目录 *install_directory*\RVDS\Examples 中。有关所提供示例的汇总，请参阅 *RealView 编译工具要点指南*。

1.2 为 ARM 处理器开发代码

本手册不但提供最为常见的一些 ARM 编程任务的相关信息和示例代码，同时还向使用 ARM 体系结构的程序开发人员提供必要的信息。

1.2.1 嵌入式软件开发

许多针对基于 ARM 体系结构的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，并在重置时执行。编写嵌入式操作系统，或编写在重置时执行的无操作系统嵌入式应用程序时必须考虑多项因素，如下所列：

- 地址重映射，例如利用 ROM 地址 0x0000，然后将 RAM 重映射到地址 0x0000
- 初始化环境和应用程序
- 链接可执行的嵌入映像，以将代码和数据放置到内存中的特定位置。

重置时，ARM 内核通常从地址 0x0000 开始执行指令。对于嵌入式系统，这意味着系统重置时地址 0x0000 处必须为 ROM。但 ROM 一般要比 RAM 慢，且通常只有 8 或 16 位宽。这会影响异常处理的速度。地址 0x0000 为 ROM 意味着无法更改异常向量。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 2-26 页的*ROM/RAM 重映射*。

重置后，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、堆栈和 I/O 外设
- 初始化应用程序，例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 2-24 页的*初始化序列*。

嵌入式系统的内存配置通常都比较复杂。例如，嵌入式系统可能使用高速 32 位 RAM 来存储性能要求高的代码，如中断处理程序和堆栈，而用较慢的 16 位 RAM 来存储应用程序读写数据，用 ROM 存储常规应用程序代码。您可利用链接器的分散加载机制来构建适用于复杂系统的可执行映像。例如，分散加载描述文件可指定各个代码和数据区的加载地址和执行地址。有关一系列已处理过的示例以及影响嵌入式应用程序的其他问题（如半主机）的信息，请参阅第 2 章*嵌入式软件开发*。

1.2.2 ARM 与 Thumb 代码交互

如果要编写支持 Thumb 16 位指令集的 ARM 处理器代码，可按需要将 ARM 和 Thumb 代码混合。如果使用的编程语言是 C 或 C++，则必须使用 `--apcs /interwork` 编译选项。链接器会检测从 Thumb 状态调用 ARM 函数或从 ARM 状态调用 Thumb 函数的情况，并会改变调用及返回序列，或者在必要时插入交互胶合代码来更改处理器状态。

注意

如果要对 Thumb 函数使用绝对地址，请参阅第4-15 页的*Thumb 状态下函数的指针*。

如果编写的是汇编语言代码，则必须确保遵循*《ARM 体系结构的过程调用标准》*(AAPCS) 的交互变量。有多种更改处理器状态的方法，取决于目标体系结构的版本。有关详细信息，请参阅第 4 章 *交互操作 ARM 和 Thumb*。

1.2.3 混合使用 C、C++ 和汇编语言

您可在程序中将单独编译汇编完毕的 C、C++ 和 ARM 汇编语言模块混用使用。您还可在 C 或 C++ 代码中编写小的汇编语言例程。这些例程可使用 ARM 编译程序的内联汇编程序或嵌入式汇编程序来进行编译。如果要使用内联汇编程序或嵌入式汇编程序，则所编写的汇编语言代码将要受到一些限制。有关这些限制的说明，请参阅*编译器用户指南*中的第 6 章 *使用嵌入式汇编程序和嵌入式汇编*。

此外，第 5 章 *混合使用 C、C++ 和汇编语言*还给出了如何在 C、C++ 和汇编语言模块之间进行调用的通用指南和示例。

1.2.4 处理处理器异常

ARM 处理器可识别以下类型的异常:

重置 在处理器重置引脚生效时发生。仅当出现处理器电源接通信号时或假定电源已接通而重置时才发生此异常。跳转到重置向量 (0x0000) 可完成软重置。

未定义指令 在处理器或任何附加的协处理器均不能识别当前执行指令时发生。

超级用户调用 (SVC-SWI 的前身)

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

预取中止 在处理器试图执行一个从非法地址预取的指令时发生。非法地址是指内存中不存在的地址，或者是内存管理子系统所确定的在当前模式下处理器无法访问的地址。

数据中止 当数据传输指令试图向非法地址加载或存储数据时发生。

中断 (IRQ) 在处理器外部中断请求引脚生效（低电平），且 CPSR 的 I 位被清除时发生。

快速中断 (FIQ)

在处理器外部快速中断请求引脚生效（低电平），且 CPSR 的 F 位被清除时发生。该异常通常用于要求中断等待时间最短的情况。

Cortex-M1 处理器或 Cortex-M3 处理器无法实现快速中断，所有中断都拥有可编程的优先级。有关详细信息，请参阅第 7 章 *处理 Cortex-M3 处理器异常*。

一般来说，如果所编写的应用程序不依靠操作系统来处理异常，例如嵌入式应用程序，则必须为每种类型的异常编写处理程序。

如果一种异常类型有多个源，例如 SVC 或 IRQ 中断，则可为每个源链接异常处理程序。

对于支持 Thumb 指令的大多数处理器，在进行异常处理时，处理器会切换为 ARM 状态。您既可用 ARM 代码编写异常处理程序，也可使用胶合代码切换到 Thumb 状态。有关详细信息，请参阅第6-10 页的*返回地址和返回指令*。

Cortex-M1 处理器和 Cortex-M3 处理器均不支持 ARM 指令，因此处理异常时，这些处理器仍保留 Thumb 状态。

其他支持 Thumb-2 指令的处理器可切换到 Thumb 状态，并执行全部由 Thumb-2 代码编写的异常处理程序。进入异常处理时选择 ARM 状态还是 Thumb 状态是通过 CP15 寄存器 1 的 TE 位实现的，通过将 **TEINIT** 输入设为内核高电平，可在重置时设置该位。有关详细信息，请参阅您所用处理器的技术参考手册。

1.2.5 使用 AAPCS

《ARM 体系结构的过程调用标准》 (AAPCS) 定义了必须遵守的寄存器用法和堆栈约定，以确保单独编译和汇编的模块能够协同工作。在基本标准上有很多变体。ARM 编译器所生成的代码始终满足所选 AAPCS 变体的要求。如果需要，链接器会选择适当的标准 C 或 C++ 库来进行链接。

开发用于 ARM 处理器的代码时，必须选择适当的 AAPCS 变体，例如：

- 如果要编写可在 ARM 状态和 Thumb 状态间进行交互的代码，必须在编译器和汇编程序中选择 `--apcs /interwork` 选项
- 如果要编写 C 或 C++ 代码，必须确保为每个编译的模块选择 AAPCS 兼容选项。
- 如果要编写自己的汇编语言例程，必须确保满足相应的 AAPCS 变体要求。

有关详细信息，请参阅 *《ARM 体系结构的过程调用标准》*规范文件 `aapcs.pdf`，它位于 *`install_directory`*`\Documentation\Specifications\...` 下。

注意

如果采用 C 语言和汇编语言混合编写，请确保明白 AAPCS 的相关规定。

1.2.6 与旧对象和库的兼容性

如果要将旧版本的 RVCT 升级，请务必阅读 *RealView 编译工具要点指南* 中的*差异*一章，以获取有关 RVCT 的新版本和以前版本之间的兼容性信息。

第 2 章
嵌入式软件开发

本章介绍了在有或没有目标系统的情况下，如何使用 *RealView® 编译工具* (RVCT) 开发嵌入式应用程序。它包含以下几节:

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，是非常重要的。

使用 RVCT 开发嵌入式软件时，必须考虑以下几个方面：

- C 库如何使用硬件。
- 某些 C 库的功能是通过使用调试环境资源来实现的。如果使用了此类资源，必须重新实现这些功能才能利用目标硬件。
- RVCT 事先并不了解任何给定目标的内存映射方面的信息。必须使映像内存映射适合目标硬件内存的布局。
- 嵌入式应用程序事先必须执行一些初始化工作，然后主应用程序才能运行。一个完整的初始化序列除了需要 RVCT C 库初始化例程，还需要您实现的代码。

2.1.1 示例代码

为说明本章涉及的主题，提供了相关的示例工程项目。本章所介绍的 Dhrystone 编译代码位于主示例目录的 `...\emb_sw_dev` 下。每个编译对应一个独立的目录，并提供了本章后续各节所述技术的示例。有关每个编译的特定信息，可在以下位置找到：

- 第2-10 页的*编译代码 1 的示例代码*
- 第2-13 页的*编译代码 2 的示例代码*
- 第2-21 页的*编译代码 3 的示例代码*
- 第2-32 页的*编译代码 4 的示例代码*
- 第2-34 页的*编译代码 5 的示例代码*。

Dhrystone 基准程序为示例工程提供了代码基础。选择 Dhrystone 是因为它能说明本章介绍的很多概念。

示例工程是针对 ARM® Integrator™ 开发平台而编写的。但是，这些示例所说明的原理适用于所有目标硬件。

注意

本章的重点并不特别放在 Dhrystone 程序上，而在于使其能在完全独立的系统上运行所要采取的步骤上。有关使用 Dhrystone 作为基准工具的详细信息，请参阅“应用程序注释”的第 93 条 - *以 ARMulator® 为基准*。您可在 ARM 网站 `http://www.arm.com` 的“文档”区域找到 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 重映射。为此，请开启开关 1 和 4 来运行 Boot Monitor，然后重置该面板。
- 将 `top_of_memory` 设置为 `0x40000`，或安装一个 DIMM 内存模块。如果不执行此操作，则缺省设置为 `0x80000` 的堆栈可能不在有效的内存范围中。

2.2 目标系统未知情况下的缺省编译工具行为

当您开始编写嵌入式应用程序软件时，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、内存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，编译工具的缺省行为可使您立即开始编译和调试应用程序代码。了解这种缺省行为非常有用，这样您会重视从缺省编译转为完全独立的应用程序的必要步骤。

2.2.1 半主机

在 ARM C 库中，主机调试环境提供对某些 ISO C 功能的支持。提供此功能的机制被称为*半主机*。

在 ARMv7 系统中，半主机是通过 BKPT 指令实现的。在较早的 ARM 系统中，半主机是通过 SVC 指令由一组已定义的“超级用户调用”操作实现的。

执行半主机时，调试代理识别半主机，然后暂时停止程序的执行。在代码恢复执行之前，由调试代理处理半主机操作。因此，由主机本身执行的任务对程序来说是透明的。

图2-1 显示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。

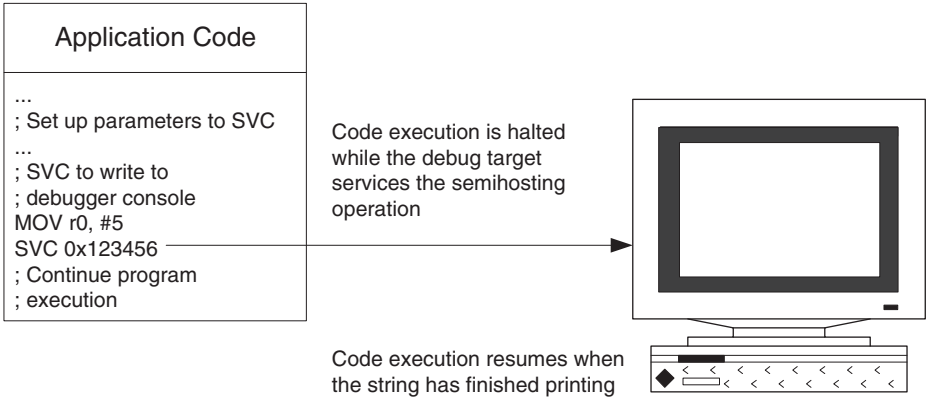


图2-1 半主机操作示例

注意

有关详细信息，请参阅第 9 章 *半主机*。

2.2.2 C 库结构

从概念上来讲，C 库可被划分成两类函数，一类为 ISO C 语言规范部分，另一类为 ISO C 语言规范提供支持。如图2-2 所示。

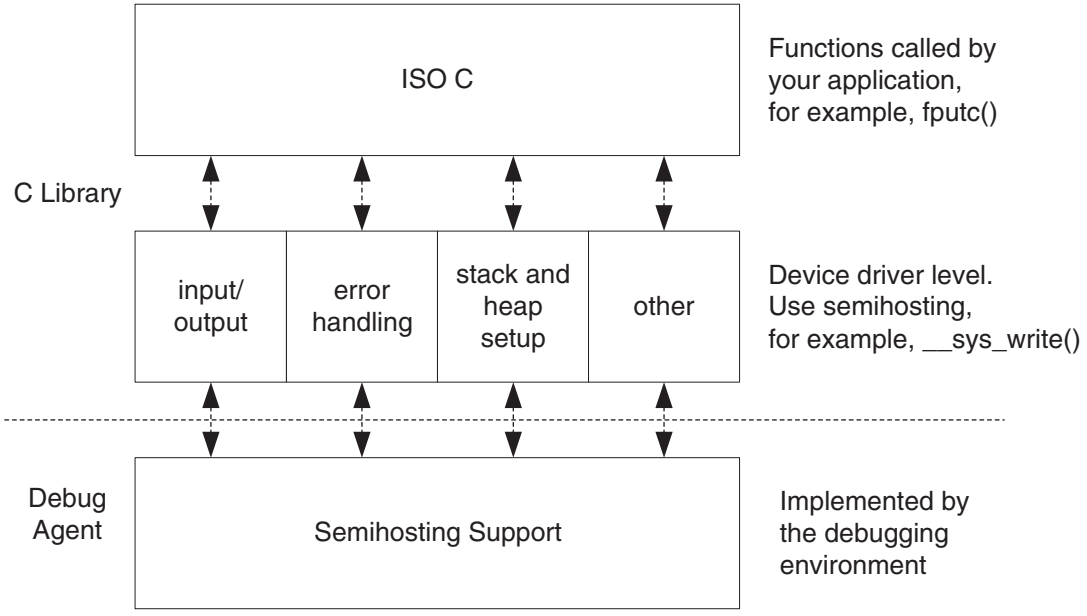


图2-2 C 库结构

对部分 ISO C 功能的支持是由主机调试环境在设备驱动程序级别提供的。

例如，RVCT C 库通过写调试器控制台窗口来实现 ISO C `printf()` 系列函数。通过调用 `__sys_write()` 来提供此功能。这是一个执行半主机调用的支持函数，将字符串写入控制台。

2.2.3 缺省内存映射

对于没有介绍内存映射的映像，链接器根据缺省内存映射放置代码和数据，如图2-3 所示。

该链接器可将此缺省内存映射用于 Cortex-M1 和 Cortex-M3 模型，即使 Cortex-M1 和 Cortex-M3 具有与缺省映射不同的固定内存映射。使用缺省内存映射的这些处理器的模型可以起作用，但是您无法编译这类系统。有关 Cortex-M1 和 Cortex-M3 固定内存映射的详细信息，请参阅 第2-22 页的*ARMv6-M 和 ARMv7-M 内存映射*。

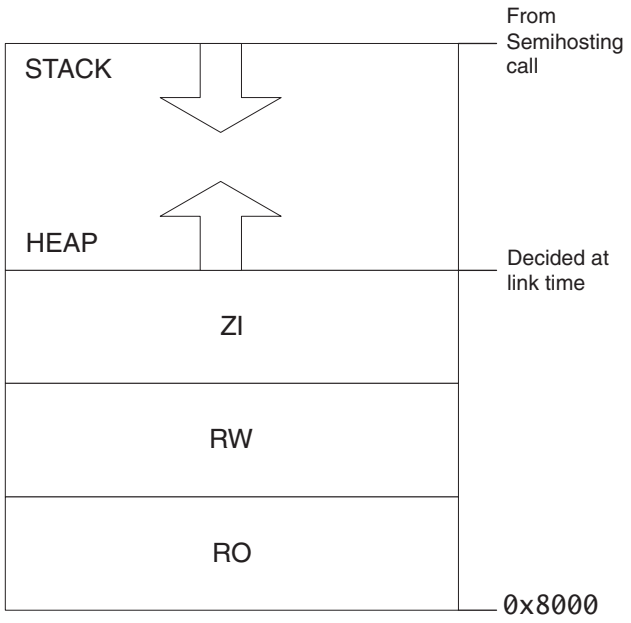


图2-3 缺省内存映射

以下是对缺省内存映射的说明：

- 链接映像，以便在地址 0x8000 加载并运行。所有的*只读* (RO) 节放在最前面，其次是*读- 写* (RW) 节，然后是*零初始化* (ZI) 节。
- 堆直接从 ZI 节的顶端地址算起，因此，其准确位置在链接时决定。

- 栈的基址位置在应用程序启动过程中由半主机操作提供。此半主机操作的返回值取决于调试环境:
  - RealView ARMulator ISS (RVISS) 返回在配置文件 `peripherals.ami` 中设定的值。缺省为 `0x08000000`。
  - RealView ICE 返回调试器内部变量 `top_of_memory` 的值。缺省为 `0x00080000`。

2.2.4 链接器放置规则

链接器在决定代码和数据位于内存中的什么位置时要遵守一组规则，如 图2-4 所示。

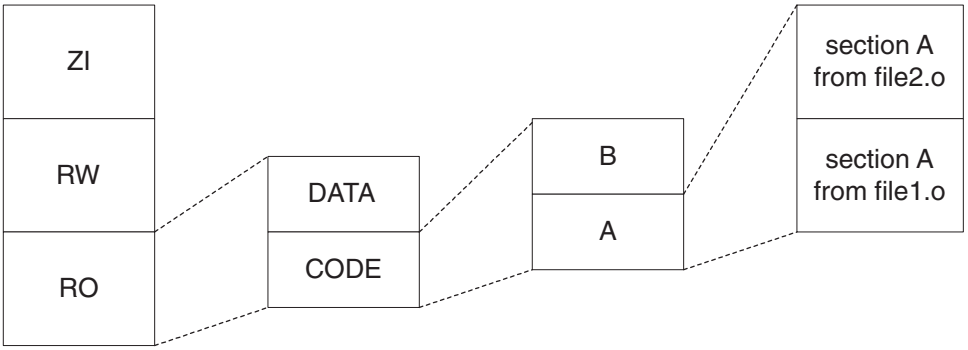


图2-4 链接器放置规则

映像首先按属性进行组织，RO 节在最低的内存地址，其次是 RW 节，然后是 ZI 节。在每种属性中，代码都在数据之前。

从此处开始，链接器按名称的字母顺序放置输入节。输入节的名称与汇编程序 AREA 指令一致。

在输入节中，各个对象的代码和数据根据链接器命令行中相应目标文件的指定顺序放置。

为了能够精确放置代码和数据，ARM 建议不要依赖这些规则。而是必须使用分散加载机制，这样才能完全控制代码和数据的放置。请参阅 第2-14 页的*调整映像内存映射以使其适应目标硬件*。

注意

请参阅*链接器和实用程序指南* 中的 第5-2 页的*关于分散加载*。

2.2.5 应用程序启动

在多数嵌入式系统中，通过在执行主任务前执行初始化序列来设置系统。

图2-5 显示了缺省的初始化序列。

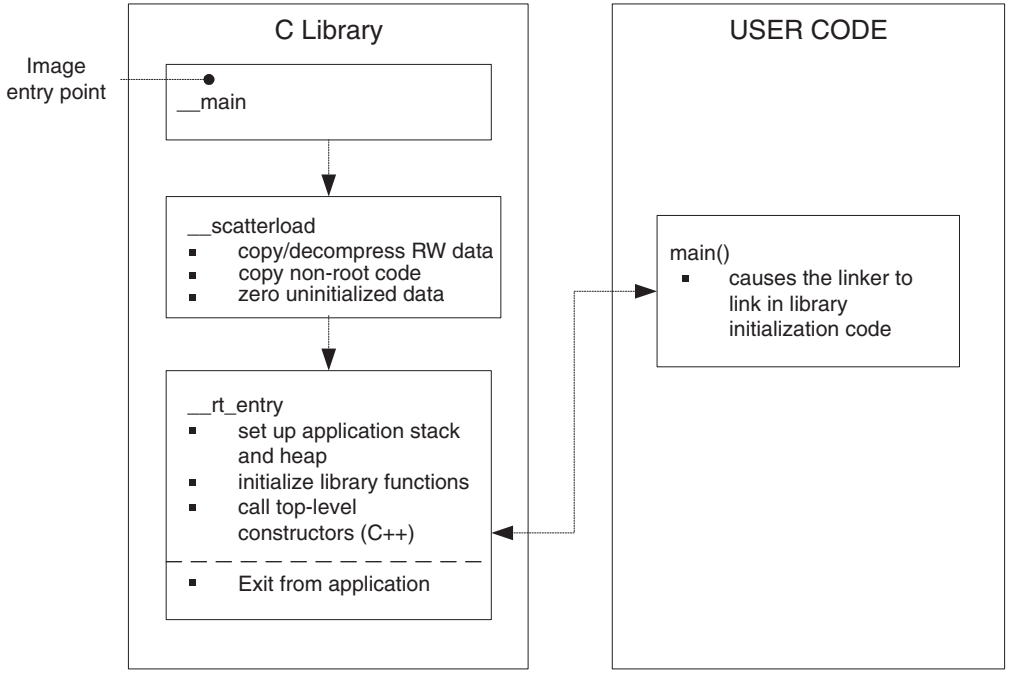


图2-5 缺省的初始化序列

在较高层，初始化序列可分为三个功能块。__main 直接跳转到 __scatterload。__scatterload 负责设置运行时映像内存映射，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，如有必要，还可执行 RW 数据的解压缩。

__scatterload 跳转到 __rt_entry。它可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++）。

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

函数标签 main() 具有特殊含义。main() 函数的存在强制链接器链接到 __main 和 __rt_entry 中的初始化代码。如果没有标记为 main() 的函数，就不会链接到初始化序列，则一些标准 C 库功能就不会得到支持。有关与不同于 __main 的启动符一起使用的备选 C 库的详细信息，请参阅 *链接器和实用程序指南* 的 第2-15 页的 *控制映像内容* 中的 --[no_]startup *symbol*。

2.2.6 编译代码 1 的示例代码

编译代码 1 是 Dhrystone 基准程序的缺省编译版本。因此，它遵从本节所介绍的缺省 RVCT 行为。请参阅 第2-3 页的 *在 Integrator 平台上运行 Dhrystone 编译代码* 和主示例目录的 ...\emb_sw_dev\build1 中的示例编译文件。

2.3 调整 C 库以使其适应目标硬件

缺省情况下，C 库利用半主机来提供设备驱动程序级功能，使主机能够用作输入和输出设备。这种机制很有用，因为开发时使用的硬件通常没有最终系统的所有输入和输出设备。

2.3.1 重定向 C 库的目标

您可提供自己的利用目标硬件的 C 库函数实现，并自动链接到支持 C 库实现的映像。此过程称为重定向 C 库目标，如 图2-6 所示。

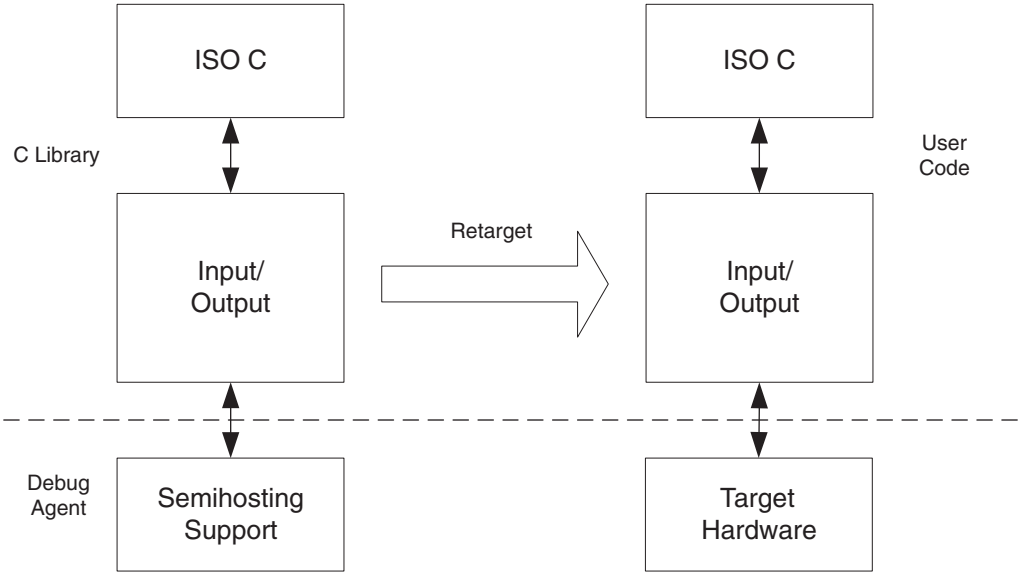


图2-6 重定向 C 库的目标

例如，您可能有一个外围 I/O 设备（如 UART），并且想重写 `fputc()` 的库实现，即从原来的写入调试器控制台变为输出到 UART。因为 `fputc()` 的实现与最终映像链接，所以整个 `printf()` 系列函数打印输出到 UART。

示例 2-1 显示了 `fputc()` 的实现示例。该示例将 `fputc()` 的输入字符参数重定向到串行输出函数 `sendchar()`，这是假定在一个单独的源文件中实现的。依靠这种方式，`fputc()` 充当与目标相关的输出函数和 C 库标准输出函数之间的抽象层。

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 避免使用 C 库半主机

在独立应用程序中，您不太可能支持半主机操作。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 库链接使用半主机的函数，必须导入符号 __use_no_semihosting。可在您工程的任何 C 或汇编语言源文件中执行此操作，如下所示：

- 在 C 模块中，使用 #pragma 指令：

  ```
  #pragma import(__use_no_semihosting)
  ```

- 在汇编语言模块中，使用 IMPORT 指令：

  ```
  IMPORT __use_no_semihosting
  ```

如果仍然链接了使用半主机的函数，则链接器会报告错误。

若要识别这些函数，请使用 ---verbose 选项进行链接。在结果输出中，C 库函数被加上了 __I_use_semihosting 标记，例如：

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

您必须为这些函数提供自己的实现，如此例中的 _sys_exit。

注意

链接器不报告应用程序代码中任何使用半主机的函数。仅当从 C 库链接了这种类型的函数时才会发生错误。

有关使用半主机的 C 库函数的完整列表，请参阅 第 9 章 *半主机*。

2.3.3 编译代码 2 的示例代码

Dhrystone 基准程序的编译代码 2 将 Integrator 平台硬件用于计时和字符串 I/O。请参阅主示例目录的 ...\emb_sw_dev\build2 中的示例编译文件。

对示例工程的编译代码 1 作了如下更改:

重定向 C 库的目标

添加了 ISO C 函数的重定向目标层。这包括标准 I/O 函数和时钟功能、一些附加错误信号和程序退出。

与目标相关的设备驱动程序

添加了一个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*。

本工程中未导入符号 __use_no_semihosting。这是因为半主机调用是在 C 库初始化过程中执行的，以设置应用程序栈和堆的位置。在 第2-19 页的*放置栈和堆* 中详细介绍了重定向栈和堆目标的设置。

注意

若要查看输出结果，串行接口 A 必须连接一个终端或终端仿真器。该串行接口必须设置为 38,400 波特、无奇偶校验、一个停止位以及无流控制。该终端必须配置为向输入行的末尾添加换行，并在本地回显键入的字符。

2.4 调整映像内存映射以使其适应目标硬件

在最终的嵌入式系统中，如果没有半主机功能，您不太可能使用缺省内存映射。目标硬件通常有几个位于不同地址范围的内存设备。为了充分利用这些设备，加载和运行时必须有不同的内存视图。

2.4.1 分散加载

使用分散加载，您可在被称为*分散加载描述文件*的文本描述文件中介绍内存中代码和数据的加载时和运行时位置。在命令行使用 `--scatter` 选项可将该文件传递给链接器。例如：

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

分散加载描述文件根据寻址内存区域，向链接器描述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：

- 载入区包含应用程序重置和加载时的代码和数据。
- 执行区包含执行应用程序时的代码和数据。应用程序启动过程中，可从每个载入区创建一个或多个执行区。

映像中所有的代码和数据都正好在一个载入区和一个执行区中。

启动过程中，`__main` 中的 C 库初始化代码执行必要的代码和数据的复制和清零，以从映像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 显示了该文件语法。

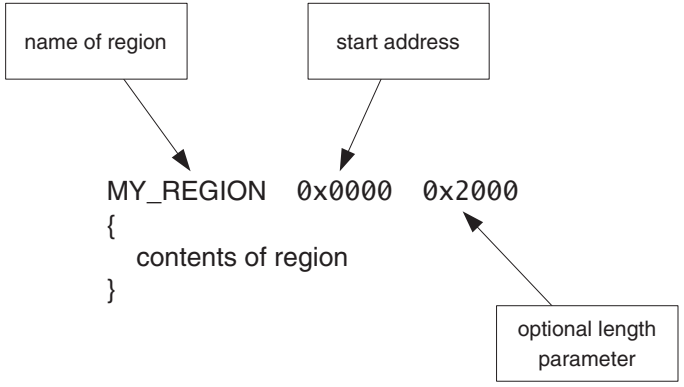


图2-7 分散加载描述文件语法

区域由至少包括一个区域名和一个起始地址的头标记定义。也可添加最大长度和各种属性。

区域中的内容取决于区域的类型：

- 载入区必须包含至少一个执行区。在实际操作中，每个载入区通常包含几个执行区。
- 执行区必须至少包括一个代码或数据节，但由 EMPTY 属性声明的区域除外。非 EMPTY 区域通常包含源或库目标文件。可使用通配符 (*) 语法来为分散加载描述文件中具有未指定的给定属性的所有节分组。

注意

有关分散加载描述文件语法的详细信息，请参阅 *链接器和实用程序指南* 中的 第 5 章 *使用分散加载描述文件*。

2.4.3 分散加载描述文件示例

图2-8 显示了分散加载的一个简单示例。

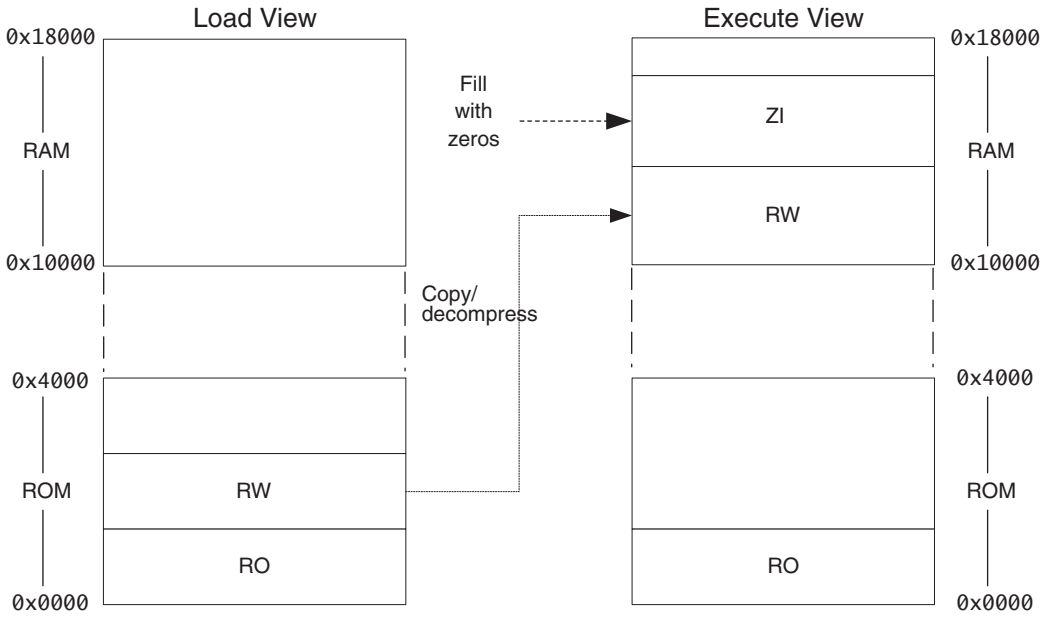


图2-8 简单的分散加载示例

该示例有一个包含所有代码和数据，从地址 0x0000 开始的载入区。从此载入区可创建两个执行区。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，它包含所有的 RW 和 ZI 数据。

示例 2-2 显示了描述 图2-8 中给定的内存映射的描述文件。

示例 2-2 简单的分散加载描述文件

```
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中放置对象

对多数映像来说，您可控制特定代码和数据节的放置，而不是如 第2-16 页的示例 2-2 中将所有的属性合在一起。这可通过在描述文件中直接指定各个对象来完成，而不是仅依靠通配符语法。

注意

在描述文件执行区中，对象的顺序不影响其在输出映像中的顺序。第2-8 页的*链接器放置规则*中介绍的链接器放置规则适用于每个执行区。

若要覆盖标准链接器放置规则，可以使用 +FIRST 和 +LAST 分散加载指令。示例 2-3 显示了一个在执行区的开头放置向量表的分散加载描述文件。在此示例中，vectors.o 中的 Vect 区域放置在地址 0x0000 处。

示例 2-3 放置节

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

有关在分散加载描述文件中放置对象的详细信息，请参阅 *链接器和实用程序指南* 中的 第 5 章 *使用分散加载描述文件*。

2.4.5 根区

根区 是一个加载地址与其执行地址相同的执行区。每个分散加载描述文件必须至少有一个根区。

对分散加载的一个放置限制是负责创建执行区的代码和数据不能将其自身复制到另一位置，例如代码和数据的复制和清零。因此，根区中必须包含以下节：

- 包含复制代码和数据的代码的 __main.o 和 __scatter*.o
- 执行压缩的 __dc*.o
- 包含要复制或压缩的代码和数据地址的 Region$$Table 节。

但是，可以使用 InRoot$$Sections 描述这些节。

由于这些节被定义为只读，使用了 * (+RO) 通配符语法对其进行分组。因此，如果在非根区指定了 * (+RO)，则必须在根区显式地声明这些节。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
  {
    vectors.o (Vect, +FIRST)  ; Vector table
    * (InRoot$$Sections)      ; All library sections that must be in a
                              ; root region, for example, __main.o,
                              ; __scatter*.o, __dc*.o, and * Region$$Table
  }
  RAM 0x10000 0x8000
  {
    * (+RO, +RW, +ZI)         ; all other sections
  }
}
```

如果根区中不包括 __main.o, __scatter.o、__dc*.o 和 Region$$Table，则会导致链接器生成错误消息。

2.4.6 放置栈和堆

分散加载提供了一种指定在映像中放置代码和静态分配数据的方法。应用程序的栈和堆是在 C 库初始化过程中设置的。通过重新实现 __user_initial_stackheap() 函数可调整栈和堆的放置。另外，还可使用特别命名的 ARM_LIB_HEAP 和 ARM_LIB_STACK 执行区。有关详细信息，请参阅 链接器和实用程序指南 中的 第5-3 页的*使用分散加载描述文件指定堆栈和堆*。

2.4.7 运行时内存模型

有两种运行时内存模型。在两种运行时内存模型中，都不对栈的增长进行检查。

注意

两个示例均适用于 Integrator 系统。

单区模型

缺省情况下为*单区模型*，应用程序的栈和堆在同一内存区中互相朝向对方增长。在此情况下，分配新的堆空间时，根据栈指针的值对堆进行检查，例如，调用 malloc() 时。

第2-19 页的图2-9 和 第2-20 页的示例 2-5 显示了 __user_initial_stackheap() 实现一个简单的单区模型的示例，其中栈从地址 0x40000 向下增长，堆从地址 0x20000 向上增长。

例程将适当的值加载到寄存器 r0 和 r1 中，然后返回。寄存器 r2 保持不变，这是因为在单区模型中不使用堆限制。未使用寄存器 r3。

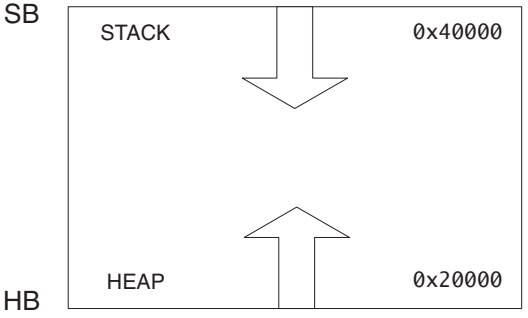


图2-9 单区模型

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
    MOV pc, lr
```

双区模型

在您的系统设计中，可能需要到将栈和堆放在不同的内存区域中。

例如，可能要保留一小块高速 RAM 仅供栈使用。为了通知链接器您要使用*双区模型*，必须使用汇编程序 IMPORT 指令导入符号 __use_two_region_memory。然后，根据由 __user_initial_stackheap() 设置的专用堆限制来对堆进行检查。

第2-20 页的图2-10 和 第2-20 页的示例 2-6 显示了实现双区模型的示例。

在此示例中，堆从 0x28000000 向 0x28080000 增长，栈从 0x40000 向下增长。

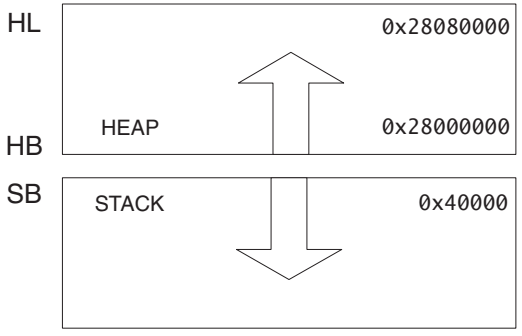


图2-10 双区模型

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
; r3 not used
MOV pc, lr
```

2.4.8 编译代码 3 的示例代码

示例的编译代码 3 实现了分散加载，而且包含一个重新实现的 `__user_initial_stackheap()`。请参阅主示例目录的 `...\emb_sw_dev\build3` 中的示例编译文件。

对示例工程的编译代码 2 作了如下修改：

分散加载

将一个简单的分散加载描述文件传递给链接器。

重定向目标的 __user_initial_stackheap()

可以选择单区或双区实现中的一个。缺省编译使用单区。汇编时定义 `TWO_REGION_MODEL` 可选择双区实现。

避免使用 C 库半主机

在编译代码 3 中导入了符号 `__use_no_semihosting`，这是因为映像中不再存在任何 C 库半主机函数。

注意

为了避免将半主机用于 `clock()`，此函数已重定向目标为在 Integrator AP 上读取 *实时时钟*(RTC)。它的误差范围是一秒，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，请参阅 第2-32 页的*编译代码 4 的示例代码*。

若要在 Integrator AP 上运行此编译代码，必须执行 ROM/RAM 重映射。为此，请接通开关 1 和 4，然后运行 Boot Monitor。

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

注意

如果要使用基于 ARM7 内核的目标，必须禁用所有的向量捕获和半主机。否则，调试器会将地址 `0x0` 和 `0x1C` 之间的指令执行解释为异常，并在对话框中报告。有关如何禁用向量捕获和半主机的信息，请参阅调试器文档。

2.4.9 ARMv6-M 和 ARMv7-M 内存映射

与多数以前的 ARM 内核不同，基于 ARMv6-M 和 ARMv7-M 处理器设备的内存映射的整体布局是固定的。这使得在基于这些处理器的不同系统之间移植系统更加轻松。

表2-1 显示了这些内存映射的主要分区。

表2-1

内存区域	说明	访问方式	地址范围
代码	通常为闪存 SRAM 或 ROM	ICode 和 Dcode 总线	0x00000000-0x1FFFFFFF
SRAM	具有位处理操作功能的片上 SRAM	系统总线	0x20000000-0x3FFFFFFF
外设	具有位处理操作功能的标准外设	系统总线	0x40000000-0x5FFFFFFF
外部 RAM	外部存储器	系统总线	0x60000000-0x9FFFFFFF
外部设备	外围设备或共享存储器	系统总线	0xA0000000-0xDFFFFFFF
专用外设总线	系统设备，请参阅 第2-23 页的表2-3	系统总线	0xE0000000-0xE00FFFFF
供应商特定	-	-	0xE0100000-0xfFFFFFFF

SRAM 和外设区各有一个*位处理操作*功能。您可以在其他地址（称为位处理操作的*别名*）单独访问位处理操作区的每个位。例如，若要访问地址为 0x20000001 的字的位[13]，可以使用地址 0x2200002D。

表2-2 显示了 SRAM 和外设存储器区的位处理操作区及别名。

表2-2

内存区域	说明	地址范围
SRAM	位处理操作区	0x20000000-0x200FFFFF
	位处理操作别名	0x22000000-0x23FFFFFF
外设	位处理操作区	0x40000000-0x400FFFFF
	位处理操作别名	0x42000000-0x43FFFFFF

表2-3 显示了专用外设总线存储器区的细分。

表2-3

内存区域	说明	地址范围
专用外设总线，外部	ITM	0xE0000000-0xE0000FFF
	DWT	0xE0001000-0xE0001FFF
	FPB	0xE0002000-0xE0002FFF
	保留	0xE0003000-0xE000DFFF
	系统控制空间	0xE000E000-0xE000EFFF
	保留	0xE000F000-0xE003FFFF
专用外设总线，内部	TPIU	0xE0040000-0xE0040FFF
	ETM	0xE0041000-0xE0041FFF
	外部 PPB	0xE0042000-0xE00FEFFF
	ROM 表	0xE00FF000-0xE00FFFFF

2.5 重置和初始化

到目前为止，本章假设是从 C 库初始化例程的入口点 __main 开始执行的。事实上，任何目标硬件上的嵌入式应用程序在启动时都执行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2-11 显示了一个基于 ARM 体系结构的嵌入式系统的可能的初始化序列。

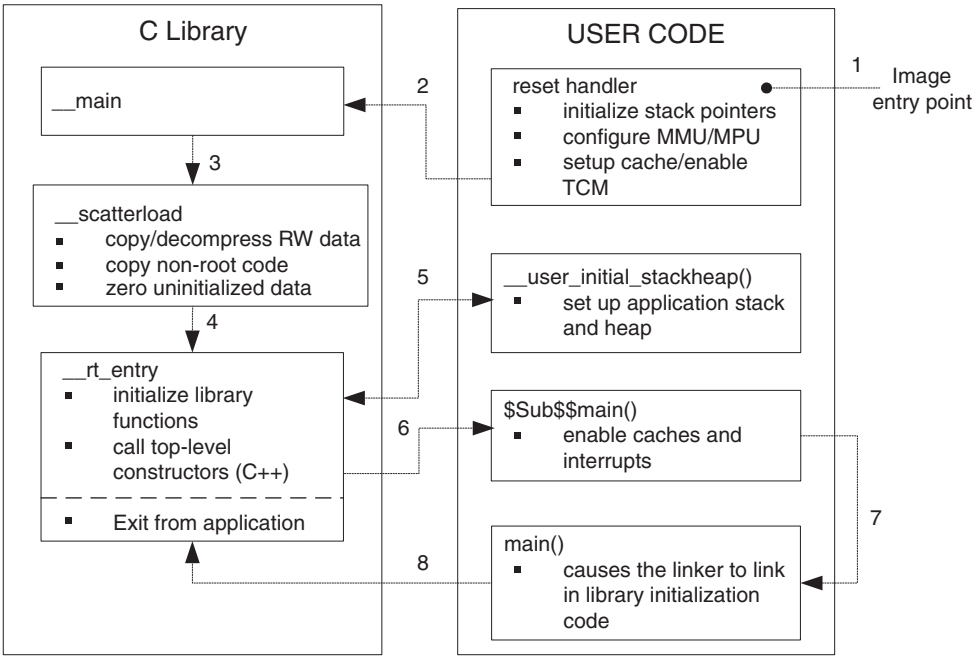


图2-11 初始化序列

重置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。

重置处理程序是用汇编语言编写的短模块，它在系统重置时执行。重置处理程序最少要为应用程序所运行的模式初始化栈指针。对于具有局部内存系统的内核，如缓存、*紧密耦合内存* (TCM)、*内存管理单元* (MMU) 和 *内存保护单元* (MPU)，在初始化过程的这一阶段必须完成某些配置。重置处理程序在执行之后，通常跳转到 __main 以开始 C 库初始化序列。

系统初始化的一些组件，如启用中断，通常是在 C 库初始化代码执行完成后才执行。在开始执行主应用程序前，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信息，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。

——注意——

本节内容不适用于 ARMv6-M 或 ARMv7-M 处理器。有关这些处理器的向量表的详细信息，请参阅 第7-5 页的*向量表*。

示例 2-7 中的代码导入了可移植到其他模块中的各种异常处理程序。向量表是转到各异常处理程序的跳转指令列表。

FIQ 处理程序直接放置在地址 0x1C 处。这样可避免执行跳到 FIQ 处理程序的跳转，从而优化了 FIQ 的响应时间。

示例 2-7 向量表代码

```
        PRESERVE8

    AREA Vectors, CODE, READONLY
    IMPORT Reset_Handler
; import other exception handlers
    ; ...
    ENTRY
    B   Reset_Handler
    B   Undefined_Handler
    B   SVC_Handler
    B   Prefetch_Handler
    B   Abort_Handler
    NOP                 ; Reserved vector
    B   IRQ_Handler
    B   FIQ_Handler
        END
```

注意

向量表标有 ENTRY 标签。此标签通知链接器该代码是一个可能的入口点，因而在链接时不能从映像中将其清除。必须使用 --entry 链接器选项在可能的映像入口点中选一个作为应用程序的真正入口点。请参阅 *链接器和实用程序指南* 中的第2-15 页的*控制映像内容*。

2.5.3 ROM/RAM 重映射

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

注意

本节假设 ARM 内核在 0x0000 处开始获取指令。这是基于 ARM 内核的系统标准。但是，有些 ARM 内核可配置为从 0xFFFF0000 地址开始取指令。

启动时，0x0000 处必须有一条有效指令，因此在重置时，0x0000 处必须是非易失性的存储器。

一种实现方法是在 0x0000 处放置 ROM。但是，这样配置有几个缺点。对 ROM 的存取速度通常慢于 RAM，当跳转到异常处理程序时，系统性能可能会大受影响。并且，将向量表放入 ROM 中，您就不能在运行时修改它。

另一个解决方案如 第2-27 页的图2-12 所示。ROM 位于地址 0x10000 处，但是此存储位置在重置时被存储控制器分配了别名: 0。重置后，重置处理程序中的代码跳转到 ROM 的实际地址。然后，存储控制器清除 ROM 的别名，使地址 0x0000 处显示 RAM。在 __main 中，向量表被复制到 0x0000 处的 RAM 中，从而可对异常进行处理。

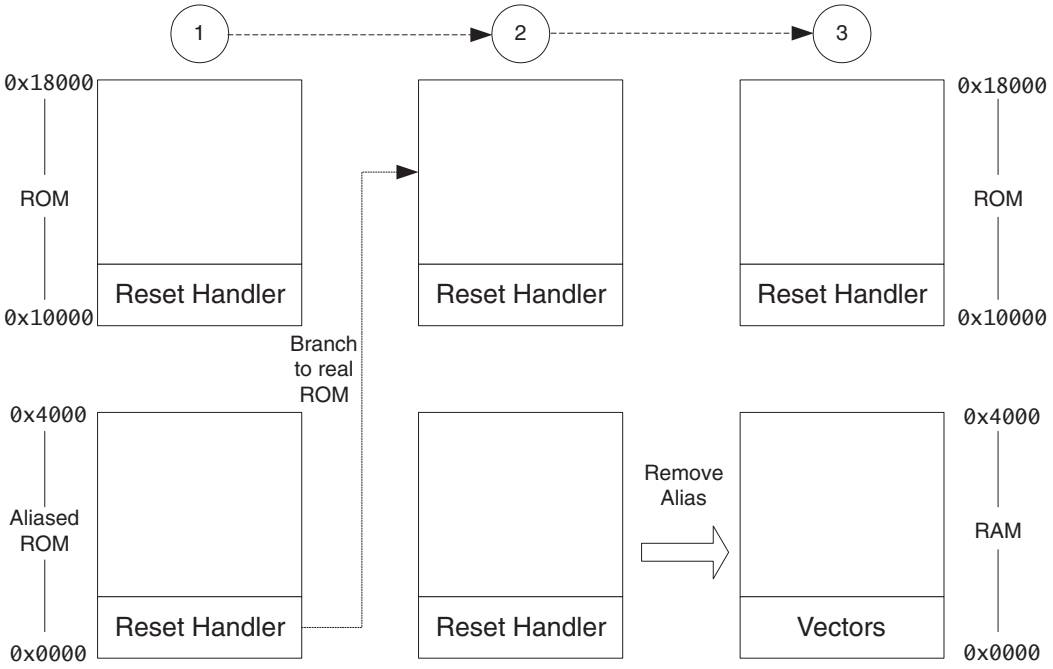


图2-12 ROM/RAM 重映射

示例 2-8 显示了如何在 ARM 汇编程序模块中实现 ROM/RAM 的重映射。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。

示例 2-8 ROM/RAM 重映射

```
; --- 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
```

第一条指令是从 ROM 别名到实际 ROM 的一条跳转指令。可实现是因为标签 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 的别名。

该代码通常在系统重置后立即执行。重映射必须在执行 C 库初始化代码前完成。

注意

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。

2.5.4 局部存储器设置的注意事项

许多 ARM 内核具有片上存储器系统，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注意。

如本章所述，__main 中的 C 库初始化代码负责设置映像在执行时的内存映射。因此，在跳转到 __main 之前，必须设置处理器内核的运行时内存视图。这就是说，在重置处理程序中必须设置并启用所有 MMU 或 MPU。

在跳转到 __main 之前（通常在 MPU/MPU 设置前），还必须启用 TCM，因为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，不必访问启用 TCM 后被屏蔽的存储器。

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，实质上是将指令作为数据进行处理。结果，一些指令可缓存在数据高速缓存中，在此情况下，它们对指令路径来说是不可见的。

为避免这些一致性问题，请在 C 库初始化序列执行完毕后再启用高速缓存。

2.5.5 分散加载和内存设置

在一个内核的重置时内存视图被改变的系统中，不论是通过 ROM/RAM 重映射还是通过 MMU 配置，分散加载描述文件必须描述重映射发生以后的映像内存映射。

示例 2-9 中的描述文件和重映射后的 第2-26 页的*ROM/RAM 重映射* 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
          ...
     }
}
```

载入区 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

重置处理程序至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 第2-30 页的示例 2-10 中，栈位于 stack_base 处。此符号可以是个硬编码地址，也可以在单独的汇编程序源文件中定义并由分散加载描述文件定位。有关如何完成此操作的信息，请参阅*链接器和实用程序指南* 中的 第5-3 页的*使用分散加载描述文件指定堆栈和堆*。

示例 2-10 初始化栈指针

```
Len_FIQ_Stack    EQU     256
Len_IRQ_Stack    EQU     256
:
Reset_Handler
:
; stack_base could be defined above, or located in a scatter 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
MOV     sp, r0

MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit
SUB     r0, r0, #Len_FIQ_Stack
MOV     sp, r0

MSR     CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit
SUB     r0, r0, #Len_IRQ_Stack
MOV     sp, r0
; Leave core in SVC mode
```

示例 2-10 为 FIQ 和 IRQ 模式分配了 256 字节的栈，但您也可对其他执行模式执行此操作。为了设置栈指针，请在中断禁用的情况下进入每种模式，然后为栈指针分配相应的值。

重置处理程序中设置的栈指针值由 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()` 修改此值。

示例 2-11 显示了 `__user_initial_stackheap()` 的实现，可将其与 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
    BX   lr
```

2.5.7 硬件初始化

一般来说，将所有系统初始化代码和主应用程序分开是一种非常好的做法。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 函数包装符插入一个在进入主应用程序前立即执行的例程。这一机制使您能在不改变源代码的情况下扩展函数。

示例 2-12 显示了如何以这种方式使用 $Sub 和 $Super。链接器以调用 $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和启用中断的另一例程。

通过调用 $Super$$main()，代码跳转到实际的 main()。

注意

有关详细信息，请参阅 *链接器和实用程序指南* 中的 第4-21 页的*使用 $Super$$ 和 $Sub$$ 覆盖符号定义*。

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 执行模式的注意事项

必须考虑主应用程序的运行模式。您的选择会影响实现系统初始化的方式。

许多想在启动时在重置处理程序和 $Sub$$main 中实现的功能，仅能在特权模式下执行才能完成，例如，片上存储器操作和启用中断。

如果您想在特权模式下运行应用程序，这不成问题。请确保在退出重置处理程序前切换到适当的模式。

但是，如果想在用户模式下运行应用程序，只有在特权模式下完成必要的任务*之后*，才能切换到用户模式。这在 $Sub$$main() 中最可能发生。

注意

__user_initial_stackheap() 必须设置应用程序模式栈。因此，必须退出系统模式下的使用用户模式寄存器的重置应用程序。然后，__user_initial_stackheap() 在系统模式下执行，因而在进入用户模式时，应用程序的栈和堆仍然存在。

2.5.9 编译代码 4 的示例代码

示例的编译代码 4 可在 Integrator 平台上独立运行。请参阅主示例目录的 ...\emb_sw_dev\build4 中的示例编译文件。

对示例工程的编译代码 3 作了如下修改:

向量表

向量表已添加到工程，并由分散加载描述文件放置。

重置处理程序

init.s 中增加了重置处理程序。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。运行在任意内核的 Integrator 平台上的 ARM7TDMI® 编译代码不包括这些模块。重置后，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内存视图。

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。

利用 Integrator AP 母板上的计时器实现精确的计时器。这产生一个 IRQ，并安装了一个每隔 10 毫秒计数器增一次值的处理程序。

2.6 内存映射的其他注意事项

本章的前几节介绍了如何通过分散加载描述文件放置代码和数据。但是，目标硬件外设的位置和堆栈限制被假定为在源文件或头文件中进行了硬编码。最好是在描述文件中定位所有与目标的内存映射有关的信息，并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工程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内存映射 32 位寄存器的计时器外设。示例 2-13 显示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

```
__attribute__ ((zero_init)) struct {
    volatile unsigned ctrl;           /* timer control */
    volatile unsigned tmr;            /* timer value   */
} timer_regs;
```

要将此结构放在内存映射的特定地址，请创建一个新的执行区来载入该结构。

示例 2-14 中显示的描述文件将 `timer_regs` 结构定位在 `0x40000000`。

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。将执行区标记为 `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 编译代码 5 的示例代码

示例的编译代码 5 与编译代码 4 等效，但是，所有的目标内存映射信息都位于分散加载描述文件中。

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。

更新的分散加载描述文件

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。

请参阅主示例目录的 `...\emb_sw_dev\build5` 中的示例编译文件。

第 3 章
编写与位置无关的代码和数据

本章介绍如何编写与位置无关的代码和数据，使之利用 *《ARM 体系结构的过程调用标准》*(AAPCS)。它包含以下几节:

3.1 与位置无关

ARM® 和 Thumb® 指令集通过使用 PC 相关指令（如 BL）支持与位置无关的代码，或称*可重定位*的代码。

注意

这与可重定位的 ELF 不同，它是由链接器生成的映像类型。

您可以编写可重定位的汇编程序代码，但是它必须不含任何地址常数。指向代码的任何文字形式地址必须是相对 PC 的偏移量。在访问地址之前，应使用 ADD 指令加上 PC 的值。

代码和数据都可以与位置无关：

- 要使代码能在不同的地址执行，它必须与位置无关或可重定位。但是，它在固定地址只能访问一组静态数据。
- 与位置无关的数据要求所有数据访问都是相对于静态基址寄存器 sb 的。这用于实现共享库机制。

RVCT 支持 C 和汇编程序（但不支持 C++）的与位置无关的代码和数据，允许您编写可重定位或可重入的代码。本章的其他部分提供了如何操作的相关信息。

有关详细信息，请参阅：

- *编译程序用户指南*中的第 2-22 页的*与位置无关的限定符*
- *库和浮点支持指南*中的第 2-3 页的*编写可重入且线程安全的代码*。

3.1.1 使用 AAPCS

ARM 体系结构的过程调用标准 (AAPCS) 是*《ARM 体系结构的应用程序二进制接口 (ABI)（基本标准）》*(BSABI) 规范的组成部分。遵循 AAPCS 编写代码可以确保分别编译和汇编的模块能够协同工作。

有关详细信息，请参阅 *install_directory*\Documentation\... 中的 AAPCS 规范。

3.2 只读且与位置无关

如果程序的所有只读段都与位置无关，则该程序为*只读且与位置无关*(ROPI)。

ROPI 段通常是*与位置无关的代码*(PIC)，但也可以是只读数据或 PIC 和只读数据的组合。

注意

ROPI 不是 AAPCS 的组成部分，因为 C++ 中不支持它。但是，您可以使用编译器或汇编程序选项 `--apcs /ropi` 来编译 ROPI 的 C 代码或汇编程序代码。

选择 ROPI 选项可以避免将代码加载到内存中的特定位置。这对以下例程尤其有用：

- 根据运行时事件载入的例程
- 在不同环境下与其他例程的不同组合一起载入内存的例程
- 在执行期间映射到不同地址的例程。

3.2.1 ROPI 的寄存器用法

按照 AAPCS 中的定义，不论使用还是不使用 ROPI，寄存器的用法都相同。

有关详细信息，请参阅*《ARM 体系结构的过程调用标准》*规范文件 `aapcs.pdf`，它位于 *`install_directory`*`\Documentation\Specifications\...` 下。

3.2.2 编写 ROPI 的 C 和汇编程序代码

编写 ROPI 的 C 和汇编程序代码时：

- ROPI 段中的代码对同一个 ROPI 段中的符号的每个引用必须是PC 相对的。AAPCS 不为只读段定义任何其他基址寄存器。一个 ROPI 段中的项的地址不能分配给其他 ROPI 段中的项。
- ROPI 段中的代码对不同 ROPI 段中的符号的每个引用必须是PC 相对的。这两个段必须是相对固定的。
- ROPI 段中的任何其他引用必须是：
  - 绝对地址
  - 对可写数据的相对于 `sb` 的引用，请参阅第3-5 页的*读写且与位置无关*。
- 每当 ROPI 段移动时，寻址 ROPI 段中符号的读写字必须进行调整。

3.2.3 链接代码

使用链接器命令行选项 --ropi 可使包含只读输出节的加载和执行区域与位置无关。通常每个只读输入节必须是只读且与位置无关的。有关详细信息，请参阅*链接器和实用程序指南*。

3.2.4 FPIC 寻址

使用 /fpic 限定符可以生成只读且与位置无关的代码，其中相对地址引用与程序被加载的位置无关。只有在代码使用 System V 共享库时才执行相对寻址。当您的代码使用共享对象时，无需使用 /fpic 进行编译。

有关 RVCT 中支持的 System V 共享库的信息，请参阅*链接器和实用程序指南*中的第 6 章 *BPABI 和 System V 共享库和可执行文件*。

3.2.5 代码示例

有关编写与位置无关代码的信息，请参阅随 RealView Development Suite 提供的 PIC-PID 示例，它位于主示例目录 *install_directory*\RVDS\Examples\picpid 中。

此示例包含置于 ROM 中固定地址上的内核，以及扩展内核功能的应用程序模块集合。应用程序模块紧随内核之后被载入内存中。但是，在链接模块时，该模块将载入的地址未知。因此，模块必须与位置无关（ROPI，PIC）。

该示例包含源代码、makefile 文件、批处理文件以及如何编译和链接不同模块的详细说明，请参阅 readme.txt。

3.3 读写且与位置无关

如果程序的所有读写段都是与位置无关的，则该程序是*读写且与位置无关* (RWPI) 的。

RWPI 段通常是*与位置无关的数据* (PID)。

RWPI 是一个 AAPCS 变体。使用编译器或汇编程序选项 `--apcs /rwpi` 可以避免将数据固定在内存中的特定位置。这对于必须被可重入例程多次实例化的数据尤其有用。

有关详细信息，请参阅*《ARM 体系结构的过程调用标准》*规范文件 `aapcs.pdf`，它位于 *`install_directory`*`\Documentation\Specifications\...` 下。

3.3.1 可重入例程

可重入例程可以同时成为多个进程的*线程*。每个进程有其各自的该例程读写段副本。每个副本的地址由不同的静态基址寄存器 `sb` 值确定。

3.3.2 RWPI 的寄存器用法

寄存器 `r9` 是静态基址寄存器 `sb`。当调用任何外部可见例程时，它必须指向适当的静态数据段基址。

在不使用 `sb` 的例程中，可以将 `r9` 用于其他用途。如果这样做，您必须在进入例程时保存 `sb` 的内容，并且在退出之前恢复它。在调用任何外部例程之前，也必须恢复其内容。

在所有其他方面，不论使用还是不使用 RWPI，寄存器的用法都相同。

3.3.3 与位置无关的数据寻址

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. 链接器从段中一个固定位置计算只读偏移量。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，它用作要加到静态基址寄存器 sb 中的值之上的偏移量。

3.3.4 编写汇编语言的 RWPI

通过 sb 值加上一个固定只读偏移量，构建从只读段到 RWPI 段的引用，请参阅《*汇编程序指南*》中*指令参考*一章的 DCD0。

3.3.5 链接代码

使用链接器命令行选项 --rwpi 可使包含 RW 和 ZI 输出节的加载和执行区域与位置无关。此选项需要 --rw-base 有值。如果未指定 --rw-base，则假设为 --rw-base 0。通常每个可写输入节必须为 RWPI。有关详细信息，请参阅*链接器和实用程序指南*中的第2-11 页的*为映像指定内存映射信息*。

3.3.6 代码示例

有关编写与位置无关代码的详细信息，请参阅随 RealView Development Suite 提供的 PIC-PID 示例，它位于主示例目录 *install_directory*\RVDS\Examples\picpid 中。

此示例包含置于 ROM 中固定地址上的内核，以及扩展内核功能的应用程序模块集合。一个模块实现可多次实例化的一组命名服务，并可通过内核互相调用。当调用某个服务时，内核会创建其静态数据的实例，然后将控制权传递给该服务。但是，随后该服务可能会被回调到内核。因此，模块必须具有与位置无关的数据（RWPI，PID）。

该示例包含源代码、makefile 文件、批处理文件以及如何编译和链接不同模块的详细说明，请参阅 readme.txt。

第 4 章
交互操作 ARM 和 Thumb

本章介绍为执行 Thumb 指令集的处理器编写代码时如何在 ARM® 状态和 Thumb® 状态之间切换。它包含以下几节:

4.1 关于交互操作

使用交互操作，您可以将 ARM 和 Thumb 代码混合使用，以便：

- ARM 例程能够返回到 Thumb 状态调用方
- Thumb 例程能够返回到 ARM 状态调用方。

这表示，如果编译或汇编用于交互操作的代码，则代码可以调用不同模块中的例程，无需考虑它所使用的指令集。

ARM 链接器检测从 Thumb 状态调用 ARM 函数或从 ARM 状态调用 Thumb 函数的情况。ARM 链接器通过改变调用和返回指令，或在必要时插入名为*胶合代码*的小代码段来更改处理器状态。

ARMv5T 及其后的体系结构提供了在不使用任何额外指令的情况下更改处理器状态的方法。通常在 ARMv5T 处理器上交互操作无需任何开销。

注意

编译 ARMv5T 和更高版本时自动假定启用交互操作，并始终生成交互的代码。但是，编译 ARMv5TE 的汇编代码不暗含交互操作，因此您必须使用 `--apcs /interwork` 汇编程序选项编译汇编代码。

4.1.1 使用 AAPCS

您可以根据需要混合使用 ARM 和 Thumb 代码，条件是代码符合 AAPCS 的要求。有关详细信息，请参阅 *《ARM 体系结构的过程调用标准》*规范文件 `aapcs.pdf`，它位于 *`install_directory`*`Documentation\Specifications\...` 下。

如果要编写 ARM 汇编语言模块，必须确保代码符合 AAPCS 的要求。如果要将几个源文件链接在一起，所有的文件必须使用兼容的 AAPCS 选项。如果检测到不兼容的选项，链接器会产生一条错误消息。

4.1.2 何时使用交互操作

为支持 Thumb 指令的 ARM 处理器编写代码时，可能大部分应用程序是按在 Thumb 状态下运行编写的。这会使代码密度最佳。使用 8 位或 16 位宽的内存，还会使性能最佳。但是，可能需要部分应用程序在 ARM 状态下运行，原因如下：

速度 应用程序的某些部分可能对速度要求严格。这些程序段在 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 位内存。ARM 代码可在其中运行，免去了从 8 位或 16 位内存取得每一条指令的开销。

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。需要改为 ARM 状态才能执行以下操作：

- 访问 CPSR 以启用或禁用中断，以及更改模式
- 访问协处理器C 不支持的
- DSP 数学指令。

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最开始必须用 ARM 指令进行编码，即使它重新进入 Thumb 状态来执行异常的主处理。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

支持 Thumb 指令的 ARM 处理器始终在 ARM 状态下启动。要在调试器下运行简单的 Thumb 汇编语言程序，请添加一个 ARM 头文件，在其中改为 Thumb 状态，然后调用主 Thumb 例程。有关示例，请参阅第4-8 页的*ARM 头文件示例*。

4.1.3 使用 /interwork 选项

ARM 编译器和汇编程序中提供 --apcs /interwork 选项。如果设置此选项，则：

- 编译器或汇编程序将交互操作属性记录在目标文件中。
- 链接器为子例程入口提供交互操作胶合代码。
- 在汇编语言中，必须编写返回调用方指令集状态的函数退出代码，例如 BX lr。
- 在 C 或 C++ 中，编译器创建返回调用方指令集状态的函数退出代码。
- 在 C 或 C++ 中，编译器将 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 --apcs /interwork 选项：

- 可能必须要返回到 ARM 代码的 Thumb 子例程
- 可能必须要返回到 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。

注意

如果模块包含用 #pragma arm 或 #pragma thumb 标记的函数，必须使用 --apcs /interwork 对该模块进行编译。这将确保可以从另一个状态（ARM 或 Thumb）成功调用这些函数。

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

- 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。
- 可处理来自 Thumb 代码的异常的异常处理代码。其返回不需要胶合代码。

4.1.4 检测交互操作调用

如果检测到直接的 ARM/Thumb 交互操作调用，而被调用例程未编译为用于交互操作，则链接器会生成一个错误。必须将被调用例程重新编译为用于交互操作。

例如，示例 4-1 显示了在没有使用 --apcs /interwork 选项的情况下编译和链接第4-13 页的示例 4-3 中的 ARM 例程时产生的错误。

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)
```

这些类型的错误表示，在从对象模块对象到例程符号的过程中，检测到 ARM 到 Thumb 或 Thumb 到 ARM 的交互操作调用，但是，被调用的例程未编译为用于交互操作。必须重新编译包含该符号的模块，并指定 --apcs /interwork 选项。

4.1.5 链接器生成的胶合代码

胶合代码是在跳转包括以下内容时，由链接器自动插入的小段代码：

- 状态改变
- 目标超出跳转指令的范围。

胶合代码成为原跳转的目标，然后跳转到目标地址。

链接器可以对同一函数的后续调用重新使用为前一调用生成的胶合代码，条件是两部分都可以跳转到该胶合代码。

有关与胶合代码交互操作的详细信息，请参阅：
- 第4-12 页的*C 和 C++ 交互操作和胶合代码*
- 第4-17 页的*使用胶合代码的汇编语言交互操作*。

胶合代码类型

胶合代码可以是：

long 可选择含有状态更改。

short 仅执行状态更改。

inline 仅执行状态更改，但是添加到要连接的函数的开头。

Veneer$$Code 节

链接器为每个胶合代码创建一个名为 `Veneer$$Code` 的输入节。您可以使用 `*(Veneer$$Code)` 在分散加载描述文件中放置胶合代码。但是，只有在操作安全的情况下链接器才会放置胶合代码。

由于地址范围或执行区大小限制的问题，可能无法将胶合代码输入节分配到该区。如果不能将胶合代码添加到指定的区中，则它将被添加到包含重定位输入节的执行区，此重定位输入节是生成该胶合代码的输入节。

有关详细信息，请参阅*链接器和实用程序指南*。

尽量少用胶合代码

您可以通过以下方式尽量避免使用胶合代码：
- 结构化内存映射，使被调函数在调用方的跳转范围之内
- 通过使调用函数在跳转范围内来鼓励共享胶合代码
- 尽量减少状态更改。

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。这些区域对应 ELF 节。每个区域都可包括 ARM 指令、Thumb 指令或两者兼而有之。

可使用链接器来修复对通过调用方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，请参阅第4-17 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，此方法可使编写的代码更小更快。

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，请参阅第4-10 页的*与 ARM 体系结构 v5T 和更高版本交互操作*。

ARM 和 THUMB 指令指示汇编程序按照相应的指令集汇编指令，请参阅第4-7 页的*更改汇编程序模式*。

4.2.1 跳转和交换指令

BX 指令只可用于支持 Thumb 的内核。该指令跳转到指定寄存器包含的地址，有 4GB 地址范围。跳转地址位 0 的值确定是否继续在 ARM 状态或 Thumb 状态下执行。有关 ARMv5 提供的附加指令，请参阅第4-10 页的*与 ARM 体系结构 v5T 和更高版本交互操作*。

可以按照这种方式使用地址的位 0，原因是：

- 所有的 ARM 指令都是字对齐的，所以任何 ARM 指令的地址位 0 和位 1 都未被使用。
- 所有的 Thumb 指令都是半字对齐的，所以任何 Thumb 指令的地址位 0 都未被使用。

语法

BX 的语法为下列项之一：

Thumb BX R*n*

ARM BX{*cond*} R*n*

其中：

R*n* 是 r0 到 r15 范围内的一个寄存器，包含要跳转到的目标地址。此寄存器中位 0 的值确定处理器的状态：

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

cond 是一个可选的条件代码。仅 BX 的 ARM 版本可有条件地执行。

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。在缺省情况下，它汇编 ARM 代码，除非用 --thumb 选项调用。

因为所有支持 Thumb 的 ARM 处理器均以 ARM 状态启动，因此必须使用 BX 指令跳转和交换到 Thumb 状态，然后使用以下汇编程序指令指示汇编程序切换汇编模式：

THUMB 指示汇编程序将以下指令汇编为 Thumb 指令。这还会导致按两个字节进行边界对齐，即使没有后续指令。

ARM 指示汇编程序恢复为汇编 ARM 指令。这还会导致按四个字节进行边界对齐，即使没有后续指令。

有关这些指令的详细信息，请参阅*汇编程序指南*。

4.2.3 ARM 头文件示例

示例 4-2 包括四段代码。在示例之后将介绍每段代码。

示例 4-2

```
     PRESERVE8

     AREA     AddReg,CODE,READONLY  ; Name this block of code.
     ENTRY                          ; Mark first instruction to call.

; SECTION 1
start
     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.

; SECTION 2
     THUMB                          ; Subsequent instructions are Thumb code.
ThumbProg
     MOVS r2, #2                    ; Load r2 with value 2.
     MOVS r3, #3                    ; Load r3 with value 3.
     ADDS r2, r2, r3                ; r2 = r2 + r3
     ADR r0, ARMProg
     BX  r0                         ; Branch exchange to ARMProg.

; SECTION 3
     ARM                            ; Subsequent instructions are ARM code.
ARMProg
     MOV r4, #4
     MOV r5, #5
     ADD r4, r4, r5

; SECTION 4
stop MOV r0, #0x18                  ; angel_SWIreason_ReportException
     LDR r1, =0x20026               ; ADP_Stopped_ApplicationExit
     SVC 0x123456                   ; ARM semihosting (formerly SWI)
     END                            ; Mark end of this file.
```

SECTION 1 执行将处理器改为 Thumb 状态所需的一小段 ARM 代码头文件。头文件代码使用:

- 加载跳转地址并设置最低有效位的 ADR 指令。ADR 指令通过将 pc+offset+1 值载入 r0 来生成地址。即，ThumbProg 的地址加上 1。

注意

ADR 指令用于同一节内的符号。对于较大的范围，请使用 LDR 伪指令。有关 ADR 和 LDR 指令的详细信息，请参阅*汇编程序指南*。

- 跳转到 Thumb 代码并更改处理器状态的 BX 指令。

代码的 SECTION 2 标记为 ThumbProg，以 THUMB 指令为前缀。这指示汇编程序将其下代码视为 Thumb 代码。Thumb 代码将两个寄存器中的值加在一起。

这段代码再次使用 ADR 指令来获取 ARMProg 标签的地址，但是，这次最低有效位没有被使用。BX 指令将其状态改回 ARM 状态。

代码的 SECTION 3 标记为 ARMProg，这段代码将两个寄存器中的值加在一起。

有关详细信息，请参阅第 9 章 *半主机*。

注意

Thumb 半主机使用操作编号 0xAB。这与 ARM 半主机号 0x123456 不同。

导出符号

如果您导出了引用 Thumb 指令的符号，则链接器自动对 Thumb 代码中的任何标签地址加 1。

如果未导出符号，则必须手动向引用 Thumb 指令的符号加 1。在第4-8 页的示例 4-2 中为 ThumbProg+1。这是因为所有的引用都由汇编程序解析，链接器从不检测符号。

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 来链接文件。
4. 将兼容调试器（如 RealView Debugger）与相应的调试目标配合使用来运行映像。如果每次一条指令地逐步执行程序，将会看到处理器进入 Thumb 状态。请参阅所用调试器的用户文档，找出指示此变更的方法。

4.2.4 与 ARM 体系结构 v5T 和更高版本交互操作

在 ARMv5T 和更高版本中：

- 额外提供以下交互操作指令：

  `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* 在 ARM 代码下必须在 PC 的 32MB 之内，在 Thumb 代码下必须在 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。位 [0] 的值确定新的处理器状态。

  在任何一种情况下，`lr` 的位 [0] 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到正确的处理器状态。

- 如果 `LDR`、`LDM` 或 `POP` 载入 PC，它们会将 `CPSR` 中的 Thumb 位设置为载入 PC 的值的 [0] 位。可以使用此项来更改指令集。这对从子例程返回尤其有用。相同的返回指令可以返回到 ARM 调用方，也可以返回到 Thumb 调用方。

有关详细信息，请参阅*汇编程序指南*和*《ARM 体系结构参考手册》*。

4.2.5 Thumb 代码中的标签

链接器区别指向以下内容的标签：

- ARM 指令
- Thumb 指令
- 数据。

当链接器重定位一个指向 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。这在将以下任何指令用于跳转时都会实现：

- ARMv4T 中的 BX
- ARMv5T 和更高版本中的 BX、BLX 或 LDR。

4.3 C 和 C++ 交互操作和胶合代码

可以自由混合为 ARM 和 Thumb 编译的 C 和 C++ 代码，但是在 ARMv4T 中，ARM 和 Thumb 代码之间需要胶合代码来执行状态更改。ARM 链接器在检测到交互操作调用时生成这些交互操作胶合代码。有关胶合代码的详细信息，请参阅第4-4 页的*链接器生成的胶合代码*。

4.3.1 编译用于交互操作的代码

--apcs /interwork 编译器选项使 ARM 编译器能够编译包含一些例程的 C 和 C++ 模块，所包含的例程可由为其他处理器状态编译的例程调用：

```
armcc --c90 --thumb --apcs /interwork
armcc --c90 --arm --apcs /interwork
armcc --cpp --thumb --apcs /interwork
armcc --cpp --arm --apcs /interwork
```

注意

--arm 是缺省选项。--c90 具有扩展名 .c 的文件的缺省选项，--cpp 是具有扩展名 .cpp 的文件的缺省选项。

为 ARMv4T 上的交互操作编译的模块会生成稍大一些的代码。ARMv5 上的没什么不同。

在叶函数（其函数体不包含函数调用）中，由编译器生成的代码中唯一的更改是用 BX lr 替换 MOV pc,lr。MOV 指令不能产生必要的状态更改。

在 Thumb 模式下为 ARMv4T 编译的非叶函数中，编译器必须将类似如下指令：

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

替换为以下序列：

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

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

--apcs /interwork 选项还会为要将模块编译到其中的代码区域设置交互操作属性。链接器检测此属性并插入适当的胶合代码。

注意

编译为用于交互操作的 ARM 代码只能用于 ARMv4T 和更高版本，因为早期的处理器不执行 BX 指令。

使用链接器选项 --info veneers 可查找胶合代码所占空间大小。

C 交互操作示例

示例 4-3 显示了执行 ARM 子例程交互操作调用的一个 Thumb 例程。ARM 子例程调用对 Thumb 库中的 printf() 进行交互操作调用。提供的这两个模块为 thumbmain.c 和 armsub.c，在主示例目录的 ...\interwork 下。

示例 4-3

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

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

编译并链接这些模块：

1. 若要编译用于交互操作的 Thumb 代码，请输入：

   ```
   armcc --thumb -c -g -O1 --apcs /interwork -o thumbmain.o thumbmain.c
   ```

2. 若要编译用于交互操作的 ARM 代码，请输入：

   ```
   armcc -c -g -O1 --apcs /interwork -o armsub.o armsub.c
   ```

3. 若要链接目标文件，请键入：

   ```
   armlink thumbmain.o armsub.o -o thumbtoarm.axf
   ```

   或者，若要查看示例 4-4 中显示的交互操作胶合代码的大小，请键入：

   ```
   armlink armsub.o thumbmain.o -o thumbtoarm.axf --info veneers
   ```

示例 4-4

```
    Adding TA veneer (4 bytes, Inline) for call to 'arm_function' from thumbmain.o(.text).
    Adding AT veneer (8 bytes, Inline) for call to '__0printf' from armsub.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 '__aeabi_memclr4' from stdio.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_mutex_initialize' from stdio.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__rt_raise' from stdio.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 '__heap_extend' from malloc.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__user_perproc_libspace' from malloc.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 '_fp_init' from lib_init.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '__ARM_argv_veneer' from lib_init.o(.text).
    Adding TA veneer (4 bytes, Inline) for call to '_sys_exit' from abort.o(.text).

14 Veneer(s) (total 80bytes) added to the image.
```

4.3.2 C 和 C++ 交互操作的基本规则

下列规则适用于应用程序内的交互操作：

- 必须使用 --apcs /interwork 命令行选项来编译任何包含可能返回到其他指令集的函数的 C 或 C++ 模块。
- 必须使用 --apcs /interwork 命令行选项来编译任何包含可能间接或虚拟调用其他指令集函数的 C 或 C++ 模块。
- 永远不要从其他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。
- 如果任何输入对象包含 Thumb 代码，则链接器选择 Thumb 运行时库。这些是用于交互操作编译的。

  如果在链接器命令行中显式指定了您自己的一个库，请务必确保它是合适的交互操作库。

注意

如果 C 或 C++ 模块包含标记为 #pragma arm 或 #pragma thumb 的函数，必须使用 --apcs /interwork 对该模块进行编译。这将确保可以从另一个状态（ARM 或 Thumb）成功调用这些函数。

4.3.3 Thumb 状态下函数的指针

如果有 Thumb 函数，即由 Thumb 代码组成的函数并在 Thumb 状态下运行，则该函数的任何指针必须有最低有效位集。这可确保交互操作正常运行。

当链接器重定位一个指向 Thumb 指令的标签的值时，它自动设置该重定位值的最低有效位。如果对 Thumb 函数使用绝对地址，则链接器无法执行此操作。

因此，如果不得不对代码中的 Thumb 函数使用绝对地址，则必须将该地址加 1。例如，您可能有一个 Thumb 函数指针的表，如第4-15 页的示例 4-5 中所示。

有关详细信息，请参阅第4-6 页的*汇编语言交互操作*。

示例 4-5 Thumb 函数的绝对地址

```
typedef int (*FN)();

myfunc() {
    FN fnptrs[] = {
```

```
        (FN)(0x8084 + 1),  // Valid Thumb address
        (FN)(0x8074)       // Invalid Thumb address
    };
    FN* myfunctions = fnptrs;

    myfunctions[0]();    // Call OK
    myfunctions[1]();    // Call Fails
}
```

4.3.4 使用同一函数的两个版本

可以有两个名字相同的函数，一个编译为 ARM，另一个编译为 Thumb。

ARM/Thumb 同义词

链接器允许一个符号的多个定义在映像中共存，只要每个定义与不同的处理器状态相关。当使用 ARM/Thumb 同义词引用符号时，链接器应用以下规则：

- ARM 状态下对符号执行的 B、BL 或 BLX 指令将解析为 ARM 定义。
- Thumb 状态下对符号执行的 B、BL 或 BLX 指令将解析为 Thumb 定义。

任何其他符号引用将解析为链接器遇到的第一个定义。链接器将生成一个警告，指定所选符号。

4.4 使用胶合代码的汇编语言交互操作

第4-6 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。不要求链接器插入交互操作胶合代码。

有关胶合代码的详细信息，请参阅第4-4 页的*链接器生成的胶合代码*。

4.4.1 使用胶合代码的汇编间交互操作

可编写汇编语言 ARM/Thumb 交互操作代码，以利用链接器生成的交互操作胶合代码。要执行此操作，需编写:

- 一个如同任何非交互操作例程的调用方例程，使用 `BL` 指令来进行调用。调用方例程可使用 `--apcs /interwork` 或 `--apcs /nointerwork` 进行汇编。

  注意

  `BL` 指令在 ARM 状态下的范围是 32MB，在 Thumb 状态下的范围是 4MB。在开发过程中，您的应用程序可能会调用跳转范围外的目标，或是调用其他状态下的函数。在这些情况下，链接器将自动插入胶合代码。胶合代码成为原始 `BL` 的中间目标，然后胶合代码将 PC 设为所需的目标地址。

- 使用 `BX` 指令返回的被调用例程。被调用例程必须用 `--apcs /interwork` 进行汇编。另外，您可能需要导出例程的函数标签，例如 `EXPORT ThumbSub`，请参阅第4-18 页的示例 4-6。适当情况下，汇编程序代码必须符合 AAPCS。

通常只有在 ARMv4T 中，或调用方与被调用方分布广泛或在不同的区域中时，这才是必要的。在 ARMv5T 和更高版本中，如果调用方和被调用方距离足够近，则无需胶合代码。

使用胶合代码的汇编语言交互操作示例

示例 4-6 显示了将寄存器 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 4-6

```
    ; *****
    ; arm.s
    ; *****

    PRESERVE8

    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
    SVC 0x123456                 ; ARM semihosting (formerly SWI)
    END

    ; *******
    ; thumb.s
    ; *******
    AREA  Thumb,CODE,READONLY    ; Name this block of code.
    THUMB                        ; Subsequent instructions are Thumb.
    EXPORT ThumbProg
ThumbProg
    MOVS  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 -g arm.s 来汇编 ARM 代码。
2. 键入 armasm --thumb -g --apcs /interwork thumb.s 来汇编 Thumb 代码。

3. 键入 armlink arm.o thumb.o -o count 链接两个目标文件。

4. 将兼容调试器与相应调试目标配合使用来运行映像。

您可以在示例 4-7中显示的反汇编代码中看到链接器插入的交互操作胶合代码。胶合代码被插入到下一个字边界，并从地址 0x0000801C 处开始。

示例 4-7

```
ARMProg:
00008000 E3A00001  MOV      r0,#1
00008004 EB000004  BL       0x801c
00008008 E3A02003  MOV      r2,#3
0000800C E3A00018  MOV      r0,#0x18
00008010 E59F1000  LDR      r1,0x8018
00008014 EF123456  SVC      0x123456
00008018 00020026  <Data> '&' 0x00 0x02 0x00
0000801C E28FC001  ADR      r12,{pc}+9 ; #0x8025
00008020 E12FFF1C  BX       r12
ThumbProg:
00008024     2102  MOV      r1,#2
00008026     4770  BX       r14
```

4.4.2 使用胶合代码的 C、C++ 和汇编语言交互操作

编译为在某一状态下运行的 C 和 C++ 代码可以调用设计为在其他状态下运行的汇编语言代码，反之亦然。为此，请编写调用方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，请参阅第4-20 页的示例 4-8。然后:

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

注意

使用此方式的任何汇编语言代码或用户库代码必须在适当的情况下与 AAPCS 保持一致。

示例 4-8

```
    /**********************
    *       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
    ; *****
    PRESERVE8
    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 -g -c --apcs /interwork thumb.c` 来编译 Thumb 代码。
2. 键入 `armasm -g --apcs /interwork arm.s` 来汇编 ARM 代码。
3. 键入 `armlink arm.o thumb.o -o add --info veneers` 来链接两个目标文件，并查看交互操作胶合代码的大小。
4. 将兼容调试器与相应调试目标配合使用来运行映像。

第 5 章
混合使用 C、C++ 和汇编语言

本章介绍如何混合编写 C、C++ 和 ARM® 汇编语言代码，还介绍如何在 C 和 C++ 中使用 ARM 内联汇编程序和嵌入式汇编程序。它包含以下几节:

5.1 使用内联汇编程序和嵌入式汇编程序

通过 ARM 编译器内置的内联汇编程序和嵌入式汇编程序，您可以使用无法直接从 C 或 C++ 访问的目标处理器功能。例如：

- 饱和算法，请参阅*汇编程序指南*中的第4-93 页的*饱和指令*
- 自定义协处理器
- *程序状态寄存器* (PSR)。

有关详细信息，请参阅*编译器用户指南*中的第 6 章 *使用嵌入式汇编程序和嵌入式汇编*。

5.1.1 内联汇编程序的功能

内联汇编程序支持与 C 和 C++ 非常灵活地进行交互操作。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，以及优化了汇编语言代码。

注意

内联汇编语言受编译器的优化的限制。生成的目标代码可能与汇编代码不完全对应。

ARM 代码内联汇编程序可以执行 ARM 指令集中的大多数指令，包括通用协处理器指令、半字指令和长乘法。

5.1.2 嵌入式汇编程序的功能

嵌入式汇编程序提供对目标处理器不受限制的低级访问，利用它可以使用 C 和 C++ 预处理程序指令，以及轻松访问结构成员偏移量。

通过嵌入式汇编程序，您可以使用完整的 ARM 汇编程序指令集，包括汇编程序指令。嵌入式汇编代码与 C 或 C++ 代码分开进行汇编。生成编译的目标文件，然后与 C 或 C++ 源代码编译的目标文件相结合。

在 ARM 和 Thumb® 代码中都支持嵌入式汇编程序。有关 ARM/Thumb 指令集的详细信息，请参阅*汇编程序指南*。

5.1.3 内联汇编代码与嵌入式汇编代码之间的差异

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 汇编程序指令	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	当大小合适并启用了链接器内联时可进行内联。	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。 使用 `sp (r13)`、`lr (r14)` 和 `pc (r15)` 则会产生错误。
返回指令	必须将其添加到代码中。	自动生成。不支持 `BX`、`BXJ` 和 `BLX` 指令。
`BKPT` 指令	直接支持。	不支持。您可以使用 `__breakpoint` 指令内在函数，请参阅*《编译器参考指南》*中的第4-65 页的*__breakpoint*。

注意

嵌入式汇编程序和 C/C++ 之间的差异的列表在*编译器用户指南*的第 6 章 *使用嵌入式汇编程序和嵌入式汇编* 提供。

5.2 在汇编代码中访问 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 指令执行导入，然后将地址加载到寄存器中。您可以根据变量的类型使用加载和存储指令来访问该全局变量。

例如，对于 **unsigned** 变量：

- LDRB/STRB 用于 **char**
- LDRH/STRH 用于 **short**
- LDR/STR 用于 **int**。

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

少于 8 个字的小型结构可以使用 LDM 和 STM 指令作为整体访问。可以使用适当类型的加载和存储指令来访问结构的单个成员。为了访问成员，必须知道该成员从结构起始地址算起的偏移量。

示例 5-1 将整型全局变量 globvar 的地址加载到 r1，将该地址中包含的值加载到 r0，将它与 2 相加，然后将新值存回 globvar 中。

示例 5-1 访问全局变量

```
    PRESERVE8

    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]
    BX   lr
    END
```

有关 ARM 或 Thumb 代码中可用的指令的信息，请参阅*汇编程序指南*中的第 4 章 *ARM 和 Thumb 指令*。

5.3 在 C++ 中使用 C 头文件

必须先将 C 头文件包装在 extern "C" 指令中，然后才可以在 C++ 中调用 C 头文件。

5.3.1 包含系统 C 头文件

您不必执行任何特殊步骤来包含标准系统 C 头文件，如 stdio.h。标准 C 头文件已经包含适当的 extern "C" 指令。例如：

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

如果使用此语法包含头文件，则所有库名都将置入全局命名空间中。

C++ 标准规定可以通过 C++ 的特定头文件来使用 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\3.0*build_num*\include*platform* 中，可以通过常规方式进行引用。例如：

```
#include <cstdio>
```

在 ARM C++ 中，这些头文件包含 (#include) C 头文件。如果使用此语法包含头文件，则所有 C++ 标准库名都将在命名空间 std 中定义，包括 C 库名。这表示您必须使用以下方法之一来限定所有库名：

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

5.3.2 包含您自己的 C 头文件

要包含您自己的 C 头文件，必须将 #include 指令包装在 extern "C" 语句中。您可以按照以下方式执行此操作：

- 当文件已经被包含 (#include) 时，如示例 5-2 中所示
- 将 extern "C" 语句添加到头文件中，如示例 5-3 中所示。

示例 5-2 包含文件之前使用指令

```
// C++ code

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

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

示例 5-3 在文件头中使用指令

```
/* 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. */
```

5.4 C、C++ 和 ARM 汇编语言交叉调用

本节提供一些示例，帮助您在 C++ 中调用 C 和汇编语言代码，以及在 C 和汇编语言中调用 C++ 代码，还介绍调用约定和数据类型。

只要遵循 AAPCS，就可以混合调用 C、C++ 和汇编语言例程。有关详细信息，请参阅 *《ARM 体系结构的过程调用标准》*规范文件 `aapcs.pdf`，它位于 *`install_directory`*`\Documentation\Specifications\...` 下。

注意

本节中的信息取决于具体的实现情况，可能在将来版本中有所更改。

5.4.1 语言交叉调用的一般规则

以下一般规则适用于 C、C++ 和汇编语言交叉调用。有关详细信息，请参阅*编译器用户指南*。

嵌入式汇编程序以及对 *《ARM 体系结构的应用程序二进制接口 (ABI)（基本标准）》*[BSABI] 的遵循使混合语言编程更易于实现。它们可以在以下方面提供帮助：

- 使用 `__cpp` 关键字进行名称重整
- 传递隐式 **`this`** 参数的方式
- 调用虚拟函数的方式
- 引用的表示形式
- 具有基类或虚拟成员函数的 C++ 类类型的布局
- 非 plain old data 结构的类对象的传递。

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"`，指定它们具有 C 链接。在此版本的 *RealView® 编译工具* (RVCT) 中，具有 C 链接表示定义函数的符号未重整。C 链接可用于以一种语言实现函数，然后在另一种语言中调用它。

  注意

  声明为 `extern "C"` 的函数不能重载。

- 汇编语言模块必须符合适用于应用程序所使用的内存模型的 AAPCS 标准。

以下规则适用于在 C 和汇编语言中调用 C++ 函数：

- 要调用全局 C++ 函数，应将它声明为 `extern "C"`，提供 C 链接。
- 静态和非静态成员函数始终有已重整的名称。使用嵌入式汇编程序的 `__cpp` 关键字，您可以不必手动查找已重整的名称。
- 无法在 C 中调用 C++ 内联函数，除非确保 C++ 编译器生成了函数的外联副本。例如，获取函数地址将导致生成外联副本。
- 非静态成员函数接受隐式 **this** 参数作为 r0 中的第一个自变量，或作为 r1 中的第二个自变量（如果函数返回不类似于 **int** 的结构）。静态成员函数不接受隐式 **this** 参数。

5.4.2 C++ 的特定信息

以下信息专门适用于 C++。

C++ 调用约定

ARM C++ 使用与 ARM C 相同的调用约定，但有一点例外：

- 调用非静态成员函数时，隐式 **this** 参数作为第一个自变量，或者作为第二个自变量（如果被调用函数返回不类似于 **int** 的 **struct**）。这可能在将来实现时有所更改。

C++ 数据类型

ARM C++ 使用与 ARM C 相同的数据类型，但有以下例外和补充：

- 如果 **struct** 或 **class** 类型的 C++ 对象没有基类或虚拟函数，则它们的布局与 ARM C 中预期的布局相同。如果此类 **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 plain old data 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。

符号名称重整

链接器取消消息中符号名称的重整。

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5-5 页的*在 C++ 中使用 C 头文件*。

5.4.3 语言交叉调用的示例

下列各节包含代码示例，说明如何混合调用语言:

- 第5-10 页的*在 C 中调用汇编语言*
- 第5-11 页的*在汇编语言中调用 C*
- 第5-12 页的*在 C++ 中调用 C*
- 第5-13 页的*在 C++ 中调用汇编语言*
- 第5-14 页的*在 C 中调用 C++*
- 第5-15 页的*在汇编语言中调用 C++*
- 第5-17 页的*在 C 或汇编语言中调用 C++*
- 第5-16 页的*在 C 和 C++ 之间传递引用*。

在 C 中调用汇编语言

示例 5-4 和示例 5-5 介绍的 C 程序调用了汇编语言子例程，将一个字符串复制到另一个字符串之前。

示例 5-4 在 C 中调用汇编语言

```
#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 5-5 汇编语言字符串复制子例程

```
    PRESERVE8

    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.
    BX   lr           ; Return.
    END
```

示例 5-4 位于主示例目录 ...\asm 中，文件名为 strtest.c 和 scopy.s。

请按以下步骤在命令行编译该示例：

1. 键入 armasm --debug scopy.s 编译汇编语言源代码。
2. 键入 armcc -c --debug 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. 将兼容调试器与相应调试目标配合使用来运行映像。

在汇编语言中调用 C

示例 5-6 和示例 5-7 介绍如何在汇编语言中调用 C。

示例 5-6 定义 C 函数

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

示例 5-7 汇编语言调用

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

    PRESERVE8

    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
```

在 C++ 中调用 C

示例 5-8 和示例 5-9 介绍如何在 C++ 中调用 C。

示例 5-8 在 C++ 中调用 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 5-9 定义 C 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;
}
```

在 C++ 中调用汇编语言

示例 5-10 和示例 5-11 介绍如何在 C++ 中调用汇编语言。

示例 5-10 在 C++ 中调用汇编语言

```
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 5-11 定义汇编语言函数

```
    PRESERVE8

    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]
    BX  lr
    END
```

在 C 中调用 C++

示例 5-12 和示例 5-13 介绍如何在 C 中调用 C++。

示例 5-12 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 5-13 在 C 中声明并调用函数

```
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;
}
```

在汇编语言中调用 C++

示例 5-14 和示例 5-15 介绍如何在汇编语言中调用 C++。

示例 5-14 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++ 函数的名称，然后使用带链接跳转 (BL) 指令来调用它：

示例 5-15 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 5-16 和示例 5-17 介绍如何在 C 和 C++ 之间传递引用。

示例 5-16 定义 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 5-17 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);
}
```

在 C 或汇编语言中调用 C++

示例 5-18、示例 5-19 和第5-18 页的示例 5-20 中的代码介绍如何在 C 或汇编语言中调用非静态、非虚拟 C++ 成员函数。可以使用编译器的汇编程序输出来查找已重整的函数名。

示例 5-18 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 5-19 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 5-20 在汇编语言中实现该函数

```
    EXPORT cfunc
    AREA foo, CODE
    IMPORT  _ZN1T1fEi

cfunc
    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
```

另外，可以使用嵌入式汇编来实现第5-17 页的示例 5-18 和示例 5-20，如示例 5-21 中所示。在此示例中，使用 __cpp 关键字来引用该函数。因此，您不必知道已重整的函数名。

示例 5-21 在嵌入式汇编中实现该函数

```
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

本章介绍如何处理体系结构 ARMv6 及更早版本以及 v7-A 和 v7-R 配置文件支持的各种类型的异常。

ARMv7-M 处理器（例如，Cortex-M3）使用另一种异常处理模式，请参阅 第 7 章 *处理 Cortex-M3 处理器异常*。

它包含以下几节:

6.1 关于处理器异常

在程序的常规执行流程中，程序计数器 (PC) 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及链接到子例程。

当此正常执行流程改变方向时，会发生处理器异常，使处理器可以处理内部或外部源生成的事件。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，有必要保留处理器先前的状态，以保证在完成适当的异常处理例程后能够恢复产生异常时正在运行的程序，使其继续执行。

6.1.1 异常类型

表6-1 显示了 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
重置	Occurs when the processor reset pin is asserted.在处理器重置引脚生效时发生。This exception is only expected to occur for signaling powerup, or for resetting as if the processor has powered up.仅当出现处理器电源接通信号时或假定电源已接通而重置时才发生此异常。A soft reset can be done by branching to the reset vector, or .跳转到重置向量 0x0000 或 0xFFFF0000 可完成软重置。
未定义指令	在处理器或任何附加的协处理器均不能识别当前执行指令时发生。
超级用户调用 (SVC)	这是一个用户定义的同步中断指令。它使得在用户模式下运行的程序能够请求在超级用户模式下运行的特权操作，如 RTOS 函数。
预取中止	在处理器试图执行一个未获取的指令时发生，因为地址非法，请参阅 *非法地址*。
数据中止	在数据传送指令试图在非法地址加载或存储数据时发生，请参阅 *非法地址*。
IRQ	在处理器外部中断请求引脚生效（低电平信号）且 CPSR 中的 I 位被清除时发生。
FIQ	在处理器外部快速中断请求引脚生效（低电平信号）且 CPSR 中的 F 位被清除时发生。

非法地址

非法虚拟地址是与当前物理内存地址不符的地址，或者是内存管理子系统决定在当前模式下处理器不可访问的地址。

6.1.2 向量表

*向量表*用于控制处理器异常的处理。向量表是一个 32 字节的保留区，通常在内存映射的底端。它为每一类型的异常分配了一个字的空间，目前保留了一个字。

这个空间不足以包含处理程序的全部代码，因此每个异常类型的向量入口通常包含一个跳转指令或加载 PC 指令来继续执行相应的处理程序。

6.1.3 异常使用的模式和寄存器

通常，应用程序在用户模式下运行，但是，异常需要在特权模式下处理。异常改变了处理器模式，这又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*堆栈指针* (sp_*mode*)
- 其自身的 r14 或*链接寄存器* (lr_*mode*)
- 其自身的*保存的程序状态寄存器* (spsr_ *mode*)。

在 FIQ 的情况下，每个异常处理程序还可占用另五个通用寄存器，从 r8_FIQ 到 r12_FIQ。

每个异常处理程序必须保证在退出时将其他寄存器恢复为其原来的内容。可通过将该处理程序必须使用的所有寄存器的内容存储在其堆栈中，并在返回前恢复这些寄存器的内容来实现这一目的。如果您使用的是 RealView ARMulator® ISS，则所需的堆栈已设置。否则，您必须自行设置。

——注意——

提供的汇编程序*没有*预先声明 *register_mode* 格式的符号化寄存器名称。要使用这种格式，必须用 RN 汇编程序指令声明相应的符号化名称，例如，lr_FIQ RN r14 声明了 r14 的符号化寄存器名称为 lr_FIQ。有关 RN 指令的详细信息，请参阅 汇编程序指南 中有关指令的章节。

6.1.4 异常优先级

当多个异常同时发生时，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。所有异常均同时发生是不可能的。例如，“未定义指令”异常和 SVC 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 显示了异常、其对应的处理器模式和处理优先级。

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	重置	超级用户 (SVC)	1
0x4	未定义指令	未定义	6
0x8	超级用户调用 (SVC)	超级用户 (SVC)	6
0xC	预取中止	中止	5
0x10	数据中止	中止	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

由于“数据中止”异常比 FIQ 异常具有更高的优先级，因此“数据中止”实际上在处理 FIQ 之前已被寄存。虽然进入了“数据中止”处理程序，但控制权会立即传给 FIQ 处理程序。处理完 FIQ 后，控制权将返回给“数据中止”处理程序。这意味着不会漏过数据传送错误检测，但如果先处理 FIQ，就有这种可能。

6.2 确定处理器状态

产生异常时，异常处理程序可能需要确定处理器是在 ARM 状态还是在 Thumb® 状态。

特别是 SVC 处理程序，更需要读取处理器状态。通过检查 SPSR 的 T- 位可确定处理器状态。该位在 Thumb 状态时设置，在 ARM 状态时清除。

ARM 和 Thumb 指令集均有 SVC 指令。在 Thumb 状态下调用 SVC 时，必须考虑以下情况：

- 指令的地址在 lr-2，而不在 lr-4。
- 该指令本身为 16 位，因而需要半字加载，请参阅 图6-1。
- 在 ARM 状态下，SVC 编号以 8 位存储，而不是 24 位。

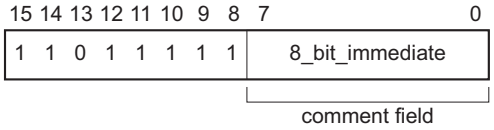


图6-1 Thumb SVC 指令

第6-7 页的示例 6-1 展示了处理两种来源的 SVC 代码的 ARM 代码。

请考虑如下要点：

- 如果需要提高代码的密度，每个 do_svc_x 例程均可执行 Thumb 状态的来回转换。
- 可通过调用一个包含 switch() 语句的 C 函数替换跳转表来实现 SVC。
- 可以根据调用源的不同状态，实现对 SVC 编号的不同处理。
- 动态调用 SVC 可增加 Thumb 状态下可访问的 SVC 编号的范围，如第6-19 页的*SVC 处理程序*中所述。

示例 6-1 SVC 处理程序

```
T_bit   EQU    0x20                    ; Thumb bit of CPSR/SPSR, that is,
                                       ; bit 5.
        :
        :
SVCHandler
        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 SVC number

        CMP     r0, #MaxSVC            ; Rangecheck
        LDRLS   pc, [pc, r0, LSL#2]    ; Jump to the appropriate routine.
        B       SVCOutOfRange
svctable
        DCD     do_svc_1
        DCD     do_svc_2
        :
        :
do_svc_1
        ; Handle the SVC.
        LDMFD   sp!, {r0-r3,r12,pc}^   ; Restore the registers and return.
do_svc_2
        :
```

6.3 进入和退出异常

本节介绍处理器对异常的响应以及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，返回方式也不同，请参阅 第6-3 页的*异常类型*。

支持 Thumb 状态的处理器与不支持 Thumb 状态的处理器使用相同的基本异常处理机制。异常造成下一条指令要从相应的向量表入口提取。

Thumb 状态和 ARM 状态下的异常使用同一向量表。*处理器对异常的响应* 中描述的异常处理过程中增加了切换至 ARM 状态的初始步骤。

在以下说明中，会明确标示在编写支持 Thumb 状态的处理器所适用的异常处理程序时必须额外考虑的因素。

6.3.1 处理器对异常的响应

产生异常时，处理器会执行以下操作：

1. 将要处理的异常所在模式的*当前程序状态寄存器* (CPSR) 复制到*保存的程序状态存储寄存器* (SPSR) 中。这会保存当前模式、中断屏蔽和条件标志。
2. 如果当前指令集状态与异常向量表中使用的指令集状态不匹配，则会自动切换指令集。在 ARM 体系结构 v6T2 之前的版本中，异常向量表始终包含 ARM 指令。自 ARM 体系结构 v6T2 开始，以后版本中的异常向量表只包含 ARM 指令或 Thumb 指令。
3. 改变相应的 CPSR 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。发生异常时，禁用 IRQ。发生 FIQ 和重置时，禁用 FIQ。
4. 如 第6-10 页的*返回地址和返回指令* 所述，将 lr_*mode* 设置为返回地址。
5. 将 PC 设置为异常的向量地址。

6.3.2 从异常处理程序返回

从异常中返回的方法取决于异常处理程序是否使用堆栈操作。无论是否使用，要返回到异常发生处继续执行，异常处理程序必须：

- 从 spsr_*mode* 恢复 CPSR
- 使用 lr_*mode* 中存储的返回地址恢复 PC。

对于不需要从堆栈中恢复目标模式寄存器的简单返回，异常处理程序可通过执行具有以下设置的数据处理指令来完成这些操作：

- 设置 S 标志
- PC 作为目标寄存器。

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6-10 页的*返回地址和返回指令*。

注意

不必从重置处理程序返回，因为重置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栈来存储必须保留的寄存器，则可通过使用带 ^ 限定符的加载多个指令来返回。异常处理程序可使用一条指令返回，例如使用：

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

为此，异常处理程序必须将以下内容保存到堆栈中：

- 调用处理程序时使用的所有工作寄存器
- 为产生与 第6-10 页的*返回地址和返回指令*中介绍的数据处理指令相同的效果而修改的链接寄存器。

^ 限定符指定从 SPSR 恢复 CPSR。它必须只在特权模式下使用。有关堆栈操作的更多常规信息，请参阅 汇编程序指南 中有关如何使用 LDM 和 STM 实现堆栈的说明。

6.3.3 返回地址和返回指令

发生异常时 PC 所指向的实际位置取决于异常的类型。返回地址可能不一定是 PC 指向的下一条指令。

如果异常发生在 ARM 状态下，则处理器将 PC-4 存储在 lr_*mode* 中。但是，对于在 Thumb 状态下发生的异常，处理器会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 ARM 指令都占用 32 位，而 Thumb 或 Thumb-2 指令占用的大小可能会不同。

通过这样的调整，处理器允许处理程序有单条返回指令正确返回，而不用考虑异常发生时指令集的状态。

下面的章节介绍每一种异常类型从处理代码正确返回的指令。

从 SVC 和未定义指令处理程序返回

SVC 和未定义指令异常是由指令本身造成的，因此，发生异常时，PC 保持不变。处理器将 PC-4 存储在 lr_*mode* 中。这使 lr_*mode* 指向下一条要执行的指令。可使用以下指令从链接寄存器恢复 PC:

```
MOVS        pc, lr
```

这将从处理程序返回控制权。

将返回地址存储在堆栈中和在返回时将其弹出的处理程序进入代码和退出代码为:

```
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 PC 改为下一条要执行的指令的地址。这是在 PC-2 处，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，处理器检测中断引脚是否为 LOW（低电平信号），以及是否清除了 CPSR 中的中断禁用位。因此，仅在 PC 被更新后才发生 IRQ 或 FIQ 异常。处理器将 PC-4 存储在 lr_*mode* 中。这使 lr_*mode* 指向发生异常时的末条指令的下一条指令。处理程序完成后，必须从 lr_*mode* 指向的指令的上一条指令处继续执行。该继续执行地址较 lr_*mode* 中的地址少一个字，因此，返回指令为:

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


The handler entry and exit code to stack the return address and pop it on return is:将返回地址存储在堆栈中和在返回时将其弹出的处理程序进入代码和退出代码为:

```
SUB         lr,lr,#4
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 `SUBS pc,lr,#4` 将 PC 改为下一条要执行的指令的地址。由于 PC 是在异常发生之前更新的，因此下一条指令在 PC-4 处。因此，处理器存储在 lr_*mode* 中的值为 PC。

从预取中止处理程序返回

如果处理器试图从非法地址取指令，则该指令被标志为无效。继续执行已经在管道中的指令，直至遇到无效指令为止，此时产生“预取中止”。

异常处理程序将未映射的指令装入物理内存，并使用 MMU（如果有的话）将虚拟内存位置映射到物理内存。然后，处理程序必须返回，重试产生异常的指令。现在加载并执行指令。

由于发出预取中止时 PC 未被更新，因此 `lr_ABT` 指向产生异常的指令的下一条指令。处理程序必须使用下列指令返回到 `lr_ABT-4`:

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

将返回地址推入堆栈中并在返回时将其弹出的处理程序进入和退出代码为:

```
SUB         lr,lr,#4
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 (`SUBS pc,lr,#4`) 将 PC 改为被中止的指令的地址。由于 PC 不是在异常发生之前更新的，因此中止的指令在 (PC-4) 处。因此，处理器存储在 lr_*mode* 中的值为 PC。

从数据中止处理程序返回

当加载或存储指令试图访问内存时，PC 已被更新。`lr_ABT` 中 PC-4 的存储值指向产生异常的地址的下一条指令。MMU（如果有）将相应地址映射至物理内存，处理程序必须返回到原来中止的指令，以便进行第二次执行尝试。因此，返回地址较 `lr_ABT` 中的地址少两个字，返回指令为:

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

将返回地址推入堆栈中并在返回时将其弹出的处理程序进入和退出代码为：

```
SUB         lr,lr,#8
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 `SUBS pc,lr,#8` 将 PC 改为产生中止指令的地址。由于 PC 是在异常发生之前更新的，因此中止指令应在 PC-6 处。因此，处理器存储在 lr_*mode* 中的值为 PC+2。

6.4 处理异常

顶层胶合代码例程必须将处理器的状态和任何所需的寄存器保存在堆栈中。然后可以选择下面的一种方法编写异常处理程序:

- 用 ARM 代码编写整个异常处理程序。
- 如果处理器支持 Thumb-2 指令，则可用 Thumb-2 代码编写整个异常处理程序。
- 执行*跳转和交换* (BX) 从 ARM 代码顶层处理程序转到处理异常的 Thumb 代码例程。然后，该例程必须使用 ARM 或 Thumb-2 代码以便从异常中返回，因为 Thumb-1 指令集没有从 SPSR 恢复 CPSR 所需的指令。

  图6-2 显示了如何实现此策略。

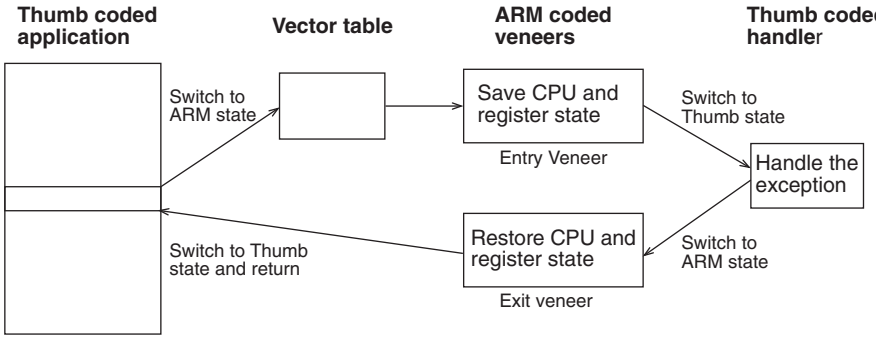


图6-2 在 ARM 或 Thumb 状态下处理异常

有关如何将 ARM 和 Thumb 代码以这种方式结合的信息，请参阅 第 4 章 *交互操作 ARM 和 Thumb*。

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，每当出现相应的异常，都会执行新的处理程序。

6.5.1 安装异常处理程序的方法

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含一个能够转到所需处理程序例程的跳转。但是，这种方法确实有局限性。因跳转指令仅有PC 相对的 32MB 范围，在某些内存组织情况下，该跳转可能无法转到处理程序。

加载 PC 指令

使用该方法，强制 PC 直接指向处理程序，方法如下:

1. 将处理程序的绝对地址存储在合适的内存位置，即在向量地址的 4KB 范围内。
2. 将一条指令存放在向量中，该指令用所选内存位置的内容加载 PC。

6.5.2 在重置时安装处理程序

如果应用程序不依靠调试器或调试监控器来启动程序的执行，则可以使用汇编语言重置或启动代码直接加载向量表。

如果 ROM 是在内存的 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 处理程序包含其中，详情请参阅 第6-29 页的*中断处理程序*。

示例 6-2 显示了如果向量位于 ROM 的零地址时设置向量的代码。可替换加载的跳转语句。

示例 6-2

```
Vector_Init_Block
                LDR     pc, Reset_Addr
                LDR     pc, Undefined_Addr
                LDR     pc, SVC_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
SVC_Addr        DCD     SVC_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
```

重置时，必须使 ROM 处在 0x0 位置。重置代码可将 RAM 重映射到 0x0 位置。这样做之前，必须从 ROM 的某个区域将向量（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 指令确定重置处理程序的地址，以便使重置向量代码能独立定位。

示例 6-3 将第6-14 页的示例 6-2 中给出的向量复制到 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，还可使用分散加载机制来定义向量表的加载和执行地址。在此情况下，C 库将为您复制向量表，详情请参阅 第 2 章 *嵌入式软件开发*。

6.5.3 从 C 安装处理程序

开发过程中有时需要从主应用程序直接将异常处理程序安装到向量中。因此，所需的指令编码必须被写到相应的向量地址中。这可由跳转和加载 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. 取得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量，而不是一个字节的偏移量。
5. 测试其前八位已清除，确保结果的长度仅为 24 位。
6. 将它与 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。

第6-16 页的示例 6-4 显示了实现此算法的 C 函数。

它带有以下几个参数：
- 处理程序的地址
- 将处理程序安装在其中的向量的地址。

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处理程序。

以下代码调用此函数来安装 IRQ 处理程序：

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

在此情况下，将丢弃返回的原始 IRQ 向量内容。

示例 6-4 实现跳转方法

```
unsigned Install_Handler (unsigned routine, unsigned *vector)
/* Updates contents of 'vector' to contain branch instruction */
/* to reach 挾outinei from 捉ectori. Function return value is */
```



```
/* original contents of 'vector'.*/
/* NB: 拎outineí must be within range of 32MB from 捉ectorí.*/

{   unsigned vec, oldvec;
    vec = ((routine - (unsigned)vector - 0x8)>>2);
    if ((vec & 0xFF000000))
    {
        /* diagnose the fault */
        printf ("Installation of Handler failed");
        exit (1);
    }
    vec = 0xEA000000 | vec;
    oldvec = *vector;
    *vector = vec;
    return (oldvec);
}
```

加载 PC 方法

所需的指令可按如下方法构成：

1. 取得包含异常处理程序地址的字的地址。
2. 减去相应向量的地址。
3. 减去 0x8 以便预取。
4. 检查其结果能否能用 12 位表示。
5. 将它与 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. 将处理程序的地址放入该存储位置。

第6-17 页的示例 6-5 显示了实现此方法的 C 例程。

在此示例中，返回的原始 IRQ 向量内容同样会被丢弃，但可用它们来创建一系列处理程序。

示例 6-5 实现加载 PC 的方法

```
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);
}
```

以下代码调用此函数来安装 IRQ 处理程序：

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

注意

如果要使用带单独指令和数据高速缓存的处理器，则必须确保高速缓存一致性问题不会妨碍使用新的向量内容。

为确保新的向量内容写入主内存中，必须清除数据高速缓存或者至少清除含有修改过的向量的条目。然后必须刷新指令高速缓存，以确保可从主内存中读取新的向量内容。

有关清除和刷新高速缓存操作的信息，请参阅目标处理器的技术参考手册。

6.6 SVC 处理程序

本节介绍 SVC 处理程序。

6.6.1 确定要调用的 SVC

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

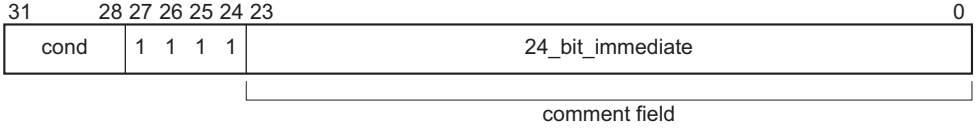


图6-3 ARM SVC 指令

顶层 SVC 处理程序可加载与链接寄存器关联的 SVC 指令 请用汇编语言、C/C++ 内联或嵌入式汇编语言编写。

处理程序必须首先将导致异常的 SVC 指令载入寄存器。此时，lr_SVC 存储 SVC 指令的下一个指令的地址，因此 SVC 载入到了寄存器（此例中为 r0）中，其中使用下面的指令：

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

然后，处理程序可检查注释字段位，以决定所需的操作。通过清除操作码的前八位来提取 SVC 编号：

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

示例 6-6 显示了如何将这些指令结合以形成一个顶层的 SVC 处理程序。

有关处理 ARM 状态和 Thumb 状态下的 SVC 指令的处理程序示例，请参阅 第6-6 页的*确定处理器状态*。

示例 6-6 顶层 SVC 处理程序

```
    PRESERVE8

    AREA TopLevelSVC, CODE, READONLY   ; Name this block of code.
    EXPORT     SVC_Handler
SVC_Handler
    STMFD      sp!,{r0-r12,lr}         ; Store registers.
```

```
    LDR        r0,[lr,#-4]             ; Calculate address of SVC instruction
                                       ; and load it into r0.
    BIC        r0,r0,#0xff000000       ; Mask off top 8 bits of instruction
                                       ; to give SVC number.
    ;
    ; Use value in r0 to determine which SVC routine to execute.
    ;
    LDMFD        sp!, {r0-r12,pc}^     ; Restore registers and return.
    END                                ; Mark end of this file.
```

6.6.2 汇编语言编写的 SVC 处理程序

要调用请求的 SVC 编号的处理程序，最简单的方法是使用跳转表。如果 r0 中包含 SVC 编号，则 示例 6-7 中的代码可插入到 第6-19 页的示例 6-6 中给出的顶层处理程序中，插入位置在 BIC 指令之后。

示例 6-7 SVC 跳转表

```
    CMP    r0,#MaxSVC          ; Range check
    LDRLS  pc, [pc,r0,LSL #2]
    B      SVCOutOfRange
SVCJumpTable
    DCD    SVCnum0
    DCD    SVCnum1
                   ; DCD for each of other SVC routines
SVCnum0            ; SVC number 0 code
    B    EndofSVC
SVCnum1            ; SVC number 1 code
    B    EndofSVC
                   ; Rest of SVC handling code
                   ;
EndofSVC
                   ; Return execution to top level
                   ; SVC handler so as to restore
                   ; registers and return to program.
```

6.6.3 C 语言和汇编语言编写的 SVC 处理程序

虽然顶层的处理程序必须总是用 ARM 汇编语言编写，但处理每一 SVC 的例程却既可用汇编语言编写，又可用 C 语言编写。有关限制情况的说明，请参阅第6-22 页的*在超级用户模式下使用 SVC*。

顶层处理程序使用*带链接的跳转* (BL) 指令可跳转到相应的 C 函数。由于 SVC 编号被汇编语言例程载入了 r0，因此它将作为第一个参数传递给 C 函数。该函数可以使用此值，例如，可用在 switch() 语句中。

可在 第6-19 页的示例 6-6 的 SVC_Handler 例程中加入下列行：

```
    BL    C_SVC_Handler     ; Call C routine to handle the SVC
```

示例 6-8 显示了如何实现该 C 函数。

示例 6-8

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

超级用户堆栈空间可能是有限的，因此要避免使用需要大量堆栈空间的函数。

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

可将值传入和传出用 C 语言编写的 SVC 处理程序，前提是顶层处理程序将堆栈指针值作为第二个参数（在 r1 中）传递给 C 函数，并且更新了 C 函数以访问该值：

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

现在，C 函数在主应用程序代码中遇到 SVC 指令时就可访问存储在寄存器中的值了，请参阅 第6-22 页的图6-4。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];
```

也可向其中回写：

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

这使已更新的值写入了堆栈的相应位置，然后通过顶层处理程序将其恢复到寄存器中。

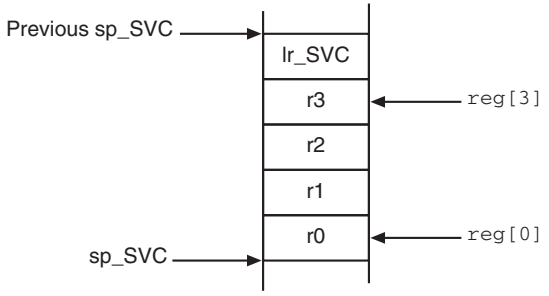


图6-4 访问超级用户堆栈

6.6.4 在超级用户模式下使用 SVC

执行 SVC 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. 返回地址存储在 lr_SVC 中，请参阅 第6-8 页的*处理器对异常的响应*。

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

如果在超级用户模式下调用 SVC，则必须存储 lr_SVC 和 spsr_SVC，以确保链接寄存器和 SPSR 中的原始值不丢失。例如，如果一个特定 SVC 编号的处理程序例程调用了另一个 SVC，则必须确保该处理程序例程将 lr_SVC 和 spsr_SVC 都存储在堆栈中。这可确保处理程序的每一次调用都保存了返回到调用它的 SVC 后面的指令所需要的信息。第6-23 页的示例 6-9 显示了它的实现方法。

示例 6-9 SVC 处理程序

```
        AREA SVC_Area, CODE, READONLY

    PRESERVE8

    EXPORT SVC_Handler
    IMPORT C_SVC_Handler

T_bit EQU 0x20

SVC_Handler

    STMFD    sp!,{r0-r3,r12,lr}   ; Store registers.
    MOV      r1, sp               ; Set pointer to parameters.
    MRS      r0, spsr             ; Get spsr.
    STMFD    sp!, {r0, r3}        ; Store spsr onto stack and another register to maintain
                                  ; 8-byte-aligned stack. Only really needed in case of nested SVCs.

    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 SVC number
        ; r1 now contains pointer to stacked registers

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

    END
```

C 和 C++ 编写的嵌套 SVC

可用 C 或 C++ 语言编写嵌套的 SVC。由 ARM 编译器生成的代码按需要存储和重新加载 lr_SVC。

6.6.5 从应用程序调用 SVC

可从汇编语言或 C/C++ 调用 SVC。

用汇编语言设置所有必需的寄存器值并发出相关的 SVC。例如：

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

几乎像所有的 ARM 指令那样，SVC 指令可被有条件地执行。

在 C/C++ 中，将 SVC 声明为一个 __SVC 函数并调用它。例如：

```
__svc(0) void my_svc(int);
.
.
.
my_svc(65);
```

这确保了 SVC 以内联方式进行编译，无需额外的调用开销，其前提是：

- 所有参数都只传入 r0-r3
- 所有结果都只返回到 r0-r3 中。

参数被传递给 SVC，如同 SVC 是一个真正的函数调用。但是，如果有二到四个返回值，则必须告诉编译器返回值在一个结构中，并使用 __value_in_regs 命令。这是因为基于 **struct** 值的函数通常被视为一个 **void** 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 第6-25 页的示例 6-11 显示了提供 SVC 编号 0x0、0x1、0x2 和 0x3 的 SVC 处理程序。SVC 0x0 和 0x1 每个都有两个整数参数并返回一个结果。SVC 0x2 有四个参数并返回一个结果。SVC 0x3 有四个参数并返回四个结果。此示例位于主示例目录的 ...\svc\main.c 和 ...\svc\svc.h 中。

示例 6-10 main.c

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

unsigned *svc_vec = (unsigned *)0x08;
extern void SVC_Handler(void);

int main( void )
{
    int result1, result2;
    struct four_results res_3;
    Install_Handler( (unsigned) SVC_Handler, svc_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 svc.h

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

struct four_results
{
    int a;
    int b;
    int c;
    int d;
};

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

6.6.6 从应用程序动态调用 SVC

在某些情形下，需要调用直到运行时才会知道其编号的 SVC。例如，当有很多相关操作可对同一目标执行，并且每个操作都有自己的 SVC 时，就会发生这种情况。在这种情况下，前几节中介绍的方法不适用。

对此，有几种解决方法，例如：

- 通过 SVC 编号构建 SVC 指令，将它存储在某处，然后再执行该指令。
- 使用通用的 SVC（作为一个额外的自变量）将一个代码作为对其参数执行的实际操作。通用 SVC 对该操作进行解码并予以执行。

第二种机制可使用汇编语言将所需的操作数传递到寄存器（通常为 r0 或 r12）中来实现。然后可重新编写 SVC 处理程序，对相应寄存器中的值进行处理。

因为有些值必须用注释字段传递给 SVC，所以有可能将这两种方法结合起来使用。

例如，操作系统可能会只用一条 SVC 指令和一个寄存器来传递所需的操作数。这使得其他 SVC 空间可用于应用程序特定的 SVC。在一个特定的应用程序中，如果从指令提取操作数的开销太大，则可使用这个方法。ARM 和 Thumb 半主机指令就是这样实现的。

示例 6-12 显示了如何使用 __svc 将 C 函数调用映射到半主机调用。它派生于主示例目录的 retarget.c，路径为 ...\emb_sw_dev\source\retarget.c。

示例 6-12 将 C 函数映射到半主机调用

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

/* Semihosting call to write a character */
__svc(SemiSVC) void Semihosting(unsigned op, char *c);
#define WriteC(c) Semihosting (0x3,c)

void write_a_character(int ch)
{
```

```
    char tempch = ch;
    WriteC( &tempch );
}
```

编译器含有一个机制，支持使用 r12 来传递所需运算的值。在 AAPCS 下，r12 为 ip 寄存器，并且专用于函数调用。其他时间内可将其用作暂存寄存器。通用 SVC 的参数被传递到 r0-r3 寄存器中，如前面所述，还可以选择在 r0-r3 中返回值，请参阅 第6-24 页的*从应用程序调用 SVC*。由 r12 传递的操作数可以是通用 SVC 调用的 SVC 的编号。但这不是必需的。

第6-27 页的示例 6-13 显示了使用通用或*间接* SVC 的 C 语言程序段。

示例 6-13

```
__svc_indirect(0x80)
    unsigned SVC_ManipulateObject(unsigned operationNumber,
                                  unsigned object,unsigned parameter);

unsigned DoSelectedManipulation(unsigned object,
                                unsigned parameter, unsigned operation)
{ return SVC_ManipulateObject(operation, object, parameter);
}
```

它生成了以下代码：

```
DoSelectedManipulation PROC
        STMFD    sp!,{r3,lr}
        MOV      r12,r2
        SVC      0x80
        LDMFD    sp!,{r3,pc}
        ENDP
```

还可使用 __svc 机制从 C 中传递 r0 中的 SVC 数。例如，如果将 SVC 0x0 用作通用 SVC，且操作 0 为字符读，操作 1 为字符写，那么，就可如下设置：

```
__svc (0) char __ReadCharacter (unsigned op);
__svc (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 r0，则仅有三个寄存器可用于向 SVC 传递参数。通常，在不得不将除 r0-r3 之外的更多参数传递给子例程时，可通过使用堆栈来完成 但是，SVC 处理程序不容易访问堆栈参数，因为通常它们在用户模式的堆栈里而不是在 SVC 处理程序使用的超级用户模式的堆栈里。

作为另一种选择，其中一个寄存器（通常是 r1）可用来指向存储其他参数的内存块。

6.7 中断处理程序

本节介绍如何编写中断处理程序以处理外部中断 FIQ 和 IRQ。

6.7.1 外部中断的级数

ARM 处理器有 FIQ 和 IRQ 两级外部中断，均是由对电平敏感的低 (LOW) 电平信号激活进入内核程序。为了产生中断，CPSR 中相应的禁用位必须清除。

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 会造成禁用 IRQ，使其在 FIQ 处理程序再次将其置为可用状态之前不能得到处理。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 向量是向量表的最后一个入口，地址为 0x1C，因此 FIQ 处理程序可直接放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高速缓存，则向量表和 FIQ 处理程序必须都被锁定在其中的一个块内。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处理程序各次调用之间的状态得以保存，从而又加快了其执行速度。

注意

中断处理程序必须包含清除中断源的代码。

6.7.2 C 语言编写的简单中断处理程序

可使用 __irq 函数声明关键字来编写简单的 C 语言中断处理程序。__irq 关键字可用于简单的单级中断处理程序，也可用于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，因为它不会使 SPSR 得到保存或恢复。本文中，“重入”的含义是处理程序再次激活中断并能将自身中断。有关详细信息，请参阅 第6-31 页的*重入中断处理程序*。

__irq 关键字：

- 保留所有 ATPCS 易损坏的寄存器
- 保留函数使用的所有其他寄存器（不包括任何 NEON/VFP 扩展寄存器）
- 通过将 PC 设置为 lr-4 并恢复 CPSR 的原始值来退出函数。

如果该函数调用了一个子例程，则 __irq 除了保留中断模式的链接寄存器之外，还保留其他易损坏的寄存器。有关详细信息，请参阅 *从中断处理程序调用子例程*。

注意

在编译 Thumb-1 C 代码时，C 语言中断处理程序不能以这种方式生成。编译 Thumb 时，会将指定为 __irq 的所有函数编译为 ARM。

但是，在启用交互操作时，由 __irq 函数调用的子例程可以编译为 Thumb-1。有关交互操作的详细信息，请参阅 第 4 章 *交互操作 ARM 和 Thumb*。

此限制不适用于支持 Thumb-2 的处理器。

从中断处理程序调用子例程

如果从顶层中断处理程序调用子例程，则 __irq 关键字还将从堆栈中恢复 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 显示了这一过程。顶层中断处理程序在 0x80000000 读取内存映射的中断控制器基址的值。如果该地址的值为 1，则顶层处理程序会跳转到一个用 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 会生成以下代码：

```
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
```

6.7.3 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 产生时，存储在 lr_IRQ 中的子例程的返回地址将被破坏。在 C 语言中使用 __irq 关键字不会对 SPSR 进行存储和恢复，而这是重入中断处理程序所要求的，因此必须使用汇编语言编写顶层中断处理程序。

在跳转到嵌套子例程或 C 函数前，重入中断处理程序必须保存 IRQ 状态、切换处理器模式并为新的处理器模式保存状态。还必须确保堆栈对新的处理器模式是 8 字节对齐的，然后才能调用符合 AAPCS 编译的 C 代码，该代码可能使用 LDRD 或 STRD 指令或 8 字节对齐堆栈分配的数据。有关堆栈对齐问题的详细信息，请参阅 ARM 网站上的 *ABI for the ARM Architecture Advisory Note - SP must be 8-byte aligned on entry to AAPCS-conforming functions*（ARM 体系结构的 ABI 告知说明 - 进入符合 AAPCS 的函数时 SP 必须是 8 字节对齐的）(ARM GENC-007024)。

在 ARMv4 及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可能需要的特权访问。有关详细信息，请参阅 第6-43 页的 *系统模式*。相反，在 ARMv4 之前的 ARM 体系结构中，必须切换到超级用户模式。

注意

该方法适用于 IRQ 和 FIQ 中断。但是，因 FIQ 中断是最先得到服务的，而通常仅有一个中断源，所以，可能不必提供重入特性。

IRQ 处理程序中安全地再次激活中断所需的步骤如下:

1. 构造返回地址并保存在 IRQ 堆栈中。
2. 保存工作寄存器、非被调用方保存的寄存器以及 spsr_IRQ。
3. 清除中断源。
4. 切换至系统模式，保持禁用 IRQ。
5. 检查堆栈是否是 8 字节对齐的，在必要时进行调整。
6. 保存用户模式链接寄存器以及所做的调整，体系结构 v4 或 v5TE 的 SP_usr 使用 0 或 4。
7. 再次激活中断并调用 C 中断处理程序函数。
8. C 中断处理程序返回时，禁用中断。
9. 恢复用户模式链接寄存器和堆栈调整值。
10. 必要时重新调整堆栈。
11. 切换到 IRQ 模式。
12. 恢复其他寄存器和 spsr_IRQ。
13. 从 IRQ 返回。

示例 6-15 显示了对于 ARMv4/v5TE 处理器，这些步骤在系统模式下的实现。第6-33 页的示例 6-16 适用于 ARMv6 处理器。

示例 6-15

```
    PRESERVE8
    AREA INTERRUPT, CODE, READONLY
    IMPORT C_irq_handler

IRQ
    SUB     lr, lr, #4              ; construct the return address
    STR     lr, [sp, #-4]!          ; and push the adjusted lr_IRQ
```



```
    MRS     r14, SPSR              ; copy spsr_IRQ to r14
    STMFD   sp!, {r0-r4,r12, r14}  ; save AAPCS regs and spsr_IRQ
    BL identify_and_clear_source
    MSR     CPSR_c, #0x9f          ; switch to SYS mode, IRQ is
                                   ; still disabled. USR mode
                                   ; registers are now current.
    AND     r1, sp, #4             ; test alignment of the stack
    SUB     sp, sp, r1             ; remove any misalignment (0 or 4)
    STMFD   sp!, {r1, lr}          ; store the adjustment and lr_USR
    MSR     CPSR_c, #0x1f          ; enable IRQ
    BL      C_irq_handler          ; branch to C IRQ handler
    MSR     CPSR_c, #0x9f          ; disable IRQ, remain in SYS mode
    LDMFD   sp!, {r1,lr}           ; restore stack adjustment and lr_USR
    ADD     sp, sp, r1             ; add the stack adjustment (0 or 4)
    MSR     CPSR_c, #0x92          ; switch to IRQ mode and keep IRQ
                                   ; disabled. FIQ is still enabled.
    LDMFD   sp!, {r0-r4, r12, r14} ; restore registers and
    MSR     SPSR_csxf, r14         ; spsr_IRQ
    LDMFD   sp!, {pc}^             ; return from IRQ.
    END
```

示例 6-16 嵌套中断（ARMv6，非向量中断）

```
IRQ_Handler
    SUB lr, lr, #4
    SRSFD #0x1f!               ; Save LR_irq and SPSR_irq to System mode stack
    CPS #0x1f                      ; Switch to System mode
    STMFD sp!, {r0-r3,r12}         ; Store other AAPCS registers
    AND r1, sp, #4
    SUB sp, sp, r1
    STMFD sp!, {r1, lr}
    BL identify_and_clear_source
    CPSIE i                        ; Enable IRQ
    BL C_irq_handler
    CPSID i                        ; Disable IRQ
    LDMFD sp!, {r1,lr}
    ADD sp, sp, r1
    LDMFD sp!, {r0-r3, r12}        ; Restore registers
    RFEFD sp!                      ; Return using RFE from System mode stack
```

这些示例假设 FIQ 为永久启用。

6.7.4 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6-35 页的*双通道 DMA 传送*
- 第6-36 页的*中断的优先化*
- 第6-37 页的*上下文切换*.

单通道 DMA 传送

示例 6-17 显示了一个中断处理程序，它执行中断驱动的从 I/O 到内存的软 DMA 传送。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。此代码最适合放在 0x1C。

在该示例代码中：

r8 指向从中读取数据的 I/O 设备的基址。

IOData 是基址到所读取的 32 位数据寄存器的偏移量。读取此寄存器将清除中断。

r9 指向数据传送到的内存位置。

r10 指向要传送到的最后一个地址。

处理正常传送的全部序列为四个指令。位于条件返回后的代码用于通知传送结束。

示例 6-17 FIQ 处理程序

```
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.
```

用载入字节指令替换载入指令可实现字节传送。从内存到 I/O 设备的传送是通过交换载入指令和存储指令的寻址模式来完成的。

双通道 DMA 传送

示例 6-18 类似于第6-34 页的示例 6-17，只不过要处理两个通道。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。该代码最适合放在 0x1c。

在该示例代码中：

r8 指向从中读取数据的 I/O 设备的基址。

IOStat 是从基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 是指示是否第一个端口导致中断的位掩码。否则，将假定第二个端口导致了中断。

IOPort1 上 IOPort2 是要读取的两个寄存器的偏移量。读取数据寄存器将清除相应端口的中断。

r9 指向来自第一个端口的数据要传送到的内存位置。

r10 指向来自第二个端口的数据要传送到的内存位置。

r11 上 r12 指向要传送到的最后一个地址。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有九个指令。位于条件返回后的代码用于通知传送结束。

示例 6-18 FIQ 处理程序

```
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.
```

用载入字节指令替换载入指令可实现字节传送。从内存到 I/O 设备的传送是通过交换条件载入指令和条件存储指令的寻址模式来完成的。

中断的优先化

示例 6-19 为高达 32 个的中断源分派相应的处理程序例程。由于它是为用于正常中断向量 (IRQ) 而设计的，所以从位置 0x18 跳转。

由外部硬件来划分中断的优先级，并在 I/O 寄存器中提供高优先级的活动中断。

在该示例代码中:

IntBase 存储中断控制器的基址。

IntLevel 存储包含最高优先级活动中断的寄存器的偏移量。

r13 假定指向一个小型满降序堆栈。

十条指令后（包括跳转到本代码的指令）启用中断。

再经过两条指令即进入每个中断的特定处理程序，而所有寄存器均保留在堆栈中。

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，因而可以保证从堆栈中安全地恢复 SPSR。

注意

应用程序注释 30，*软中断优先化*介绍了如何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不同。

示例 6-19

```
; 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6-38 页的示例 6-20 实现了在用户模式下的上下文切换。其代码是基于指向一系列它要运行的过程的*过程控制块* (PCB) 的指针。

图6-5 显示了示例所预期的 PCB 布局。

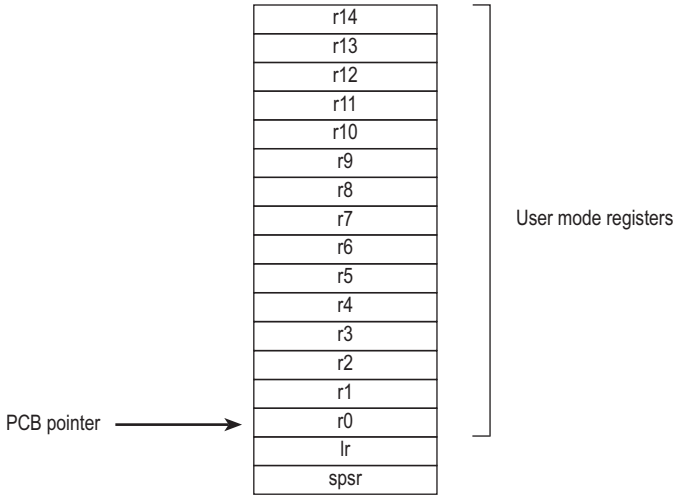


图6-5 PCB 布局

指向下一个要运行的过程的 PCB 指针是由 r12 指定的，且该列表末尾有一零指针。r13 寄存器是指向 PCB 的指针，并保留一小段时间，以便在进入时它指向当前运行过程的 PCB。

示例 6-20

```
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 重置处理程序

重置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。有关详细信息，请参阅 第6-14 页的*安装异常处理程序*。
- 初始化堆栈和寄存器。
- 如果使用 MMU，初始化内存系统。
- 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 语言所需的变量并调用主应用程序。

有关详细信息，请参阅 第 2 章 *嵌入式软件开发*。

6.9 未定义指令处理程序

在以下情况中将产生未定义指令异常:

- 处理器无法识别指令时
- 处理器将指令识别为协处理器指令，但没有协处理器能识别该指令时。

可能的情形是，该指令为协处理器指令，但相关的协处理器（如 VFP）没有加入系统，或已禁用。但是，可能会有一个该协处理器的软件仿真器。

这样的仿真器必须:

1. 为自身分配未定义指令向量并存储旧的内容。
2. 检查该未定义指令是否必须进行仿真。这与 SVC 处理程序提取 SVC 编号的方式类似，但仿真器不是提取底部的 24 位，而是提取 [27:24] 位。

   这些位按如下方式决定该指令是否为协处理器操作:
   - 如果 [27:24] 位为 `b1110` 或 `b110x`，则该指令为协处理器指令。
   - 如果 [8:11] 位显示此协处理器仿真器须处理该指令，则仿真器必须处理该指令并返回用户程序。
3. 否则，该仿真器必须使用仿真器安装时存储的向量，将该异常传递给原始处理程序或链中的下一个仿真器。

仿真器链遍历之后，未定义指令处理程序必须报告错误并退出。

注意

ARMv6T2 之前的 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仿真这类指令。

6.10 预取中止处理程序

如果系统中没有 MMU，预取中止处理程序会报错并退出。否则，造成中止的地址必须存储在物理内存中。lr_ABT 指向造成中止的指令地址下面的指令，因此，要恢复的地址在 lr_ABT-4 处。该地址的虚拟内存错误可以得到处理，并重试取指令操作。因此，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 数据中止处理程序

如果没有 MMU，数据中止处理程序必须报告错误并退出。如果有 MMU，处理程序必须处理虚拟内存错误。

造成中止的指令位于 lr_ABT-8，因为 lr_ABT 指向造成中止的指令的下两条处的指令。

以下类型的指令均可造成该中止：

单个寄存器加载或存储

响应取决于处理器类型：

- 如果中止发生在包括 ARM7TDMI® 在内的 ARM7 处理器上，则地址寄存器已更新，一定要将更新还原。
- 如果中断产生在 ARM9、ARM10、StrongARM 或更高版本的处理器上，则在执行指令之前处理器会恢复该地址的值。撤消此更改不需要任何代码。

交换 (SWP) 此指令不涉及地址寄存器更新。

加载多个或存储多个

响应取决于处理器类型：

- 如果中止发生在 ARM7 处理器上并启用了回写，则会更新基址寄存器，就好像完成了整个传送。

  在寄存器列表中存在含有基址寄存器的 LDM 时，处理器用修改了的基址值替换覆盖值，以便能够恢复。然后可以使用所涉及到的几个寄存器重新计算初始基址。
- 如果中止发生在基于 ARM9、ARM10、StrongARM 或更高版本的处理器上并启用了回写，则基址寄存器恢复到执行指令前它具有的值。

上述三个例子中，MMU 均可将所需的虚拟内存载入物理内存。MMU *故障地址寄存器* (FAR) 包含造成中止的地址。完成后，处理程序可返回并尝试再次执行该指令。

可在主示例目录的 ...\databort 下找到数据中止处理程序的代码示例。

6.12 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，还有其他特权处理器模式，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。

发生处理器异常时，当前 PC 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，PC 被设为一个异常定义的地址，以开始执行异常处理程序。

更改 PC 之前，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 移到 PC (MOV pc,lr)。

所有这些操作说明，处理异常的 ARM 模式必须确保如果调用子例程，不会发生另一个同类型的异常，因为该子例程的返回地址会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从特权模式改为用户模式。第一个解决方法限制太多，第二个则意味着可能没有正确运行所需的特权。

ARMv4 和之后的版本提供了一个被称为 *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。特权模式任务可在此模式下运行，且异常不再覆盖链接寄存器。

注意

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。有关示例，请参阅 第6-31 页的*重入中断处理程序*。

第 7 章
处理 Cortex-M3 处理器异常

本章介绍如何处理 Cortex-M3 处理器支持的各种类型的异常。

它包含以下几节:

7.1 关于 Cortex-M3 处理器异常

在程序的常规执行流程中，程序计数器 (PC) 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及链接到子例程。

当此正常执行流程改变方向时，会发生处理器异常，使处理器可以处理内部或外部源生成的事件。此类事件的示例有:

- 外部产生的中断
- 处理器试图执行未定义的指令
- 访问特权操作系统函数。

处理异常时，执行流程将跳转到*向量表*中的地址。处理的特定异常决定了向量表中所用地址的位置。有关详细信息，请参阅第7-4 页的*异常编号*和第7-5 页的*向量表*。

注意

与多数 ARM 处理器不同的是，Cortex-M3 处理器的向量表包含地址，而不包含指令。

处理此类异常时，有必要保留处理器先前的状态，以保证在完成适当的异常处理例程后能够恢复产生异常时正在运行的程序，使其继续执行。

保存过程是由 Cortex-M3 处理器上的硬件自动完成的。因为 CPSR 和 AAPCS 易损坏的寄存器会自动保存，多数异常处理程序可以用 C 或 C++ 编写，并且通常不要求用汇编语言编写低级处理程序。

Cortex-M3 处理器基于 ARM v7-M 体系结构。它只执行 Thumb-2 指令，不支持 ARM 指令集。因此，异常处理代码在 Thumb 状态下运行。

7.1.1 操作模式和执行模式

Cortex-M3 处理器支持：

- 两种操作模式，即线程模式和处理程序模式
- 两种执行模式，即特权模式和用户模式。

重置时会进入线程模式，并且通常从异常中返回时也会进入该模式。在线程模式下，代码可以在特权模式或用户模式下执行。

发生异常后将进入处理程序模式。所有代码都在特权模式下执行。发生异常时，内核会自动切换到特权模式。

特权模式具有完全访问权限。

用户模式具有有限的访问权限。这些限制包括：

- 指令用法的限制，如 MSR 指令中可以使用哪些字段
- 某些协处理器寄存器用法的限制
- 基于系统设计对内存和外围设备访问的限制
- MPU 配置施加的对内存和外围设备访问的限制。

从异常中返回时，通过修改链接寄存器 (R14) 中 EXC_RETURN 值可以从特权线程模式更改为用户线程模式。通过使用 MSR 指令清除 CONTROL[0]，也可以从特权线程模式更改为用户线程模式。不过，不通过异常处理（例如 SVC）无法直接从非特权模式更改为特权模式，请参阅第7-16 页的*超级用户调用*。

7.1.2 主堆栈和进程堆栈

Cortex-M3 处理器支持两个不同堆栈，主堆栈和进程堆栈。它有两个堆栈指针 (SP)，分别用于两个堆栈。一次只能看见一个堆栈指针，具体取决于正在使用的堆栈。

重置以及进入异常处理程序时使用主堆栈。要使用进程堆栈，必须选择该堆栈。可以通过以下方法之一执行此操作：

- 退出处理程序模式时，可通过使用 EXC_RETURN 值
- 在线程模式中时，可通过使用 MSR 指令写入 CONTROL[1]。

——注意——

您的初始化代码或上下文切换代码必须对进程堆栈指针进行初始化。

7.1.3 异常编号

表7-1 介绍 Cortex-M3 处理器识别的不同类型的异常。有关导致每种类型异常的事件的信息，请参阅第7-7 页的*事件*。

表7-1 异常编号

异常编号	异常	优先级	禁用	说明
1	重置	-3	否	
2	NMI	-2	否	不可屏蔽中断
3	HardFault	-1	否	其他异常未涉及的所有错误
4	MemManage	可配置	可禁用	内存保护错误（指令或数据）
5	BusFault	可配置	可禁用	其他内存错误
6	UsageFault	可配置	可禁用	除内存错误以外的指令执行错误
7-10	保留	-	-	
11	SVCall	可配置	否	执行 SVC 指令导致的同步超级用户调用
12	调试监控器	可配置	可禁用	同步调试事件
13	保留	-	-	
14	PendSV	可配置	否	异步超级用户调用
15	SysTick	可配置	否	系统计时器滴答声
16	外部中断 (0)	可配置	可禁用	外部中断
...	...	...	...	
16 + *n*	外部中断 (*n*)	可配置	可禁用	外部中断

7.1.4 向量表

重置时，向量表位于地址 0x00000000。

如果您的代码在特权模式下执行，则可以执行以下操作：

- 可以从*向量表偏移量寄存器* (VTOR) 中读取向量表 *vectorbaseaddress* 的当前位置
- 您可以通过写入 VTOR 将它重定位到其他地址。

有关 VTOR 的信息，请参阅第7-6 页的*向量表偏移量寄存器*，有关特权的详细信息，请参阅第7-3 页的*操作模式和执行模式*。

向量表包含每种类型异常的处理程序的地址。异常编号 *n* 的处理程序位于 (*vectorbaseaddress* + 4 * *n*)。有关每个异常的 *n* 值，请参阅第7-4 页的*异常编号*。

注意

这与其他 ARM 处理器中的向量表有所不同。其他 ARM 处理器中的向量表包含指令，而不包含地址。

所有处理程序地址的 [0] 位必须设置为 1，因为 ARMv7-M 处理器只能执行 Thumb 代码。

位于 *vectorbaseaddress* 的字包含主堆栈指针的重置值。

向量表偏移量寄存器

此寄存器的内存映射地址是 0xE000ED08。重置时，它包含 0x00000000。

保留 VTOR 的 [31:30] 位和 [7:0] 位，并且都为 0。如果 [29] 位为 0，则向量表位于内存的代码区；否则，向量表位于内存的 RAM 区。

向量表地址必须始终为 32 字对齐。如果使用的外部中断多于 16 个，则地址对齐必须大于 32 位。表7-2 介绍任意数量外部中断的对齐方式。ARMv7-M 不支持外部中断多于 240 个。

表7-2

外部中断数	*vectorbaseaddress* 的对齐方式
0-16	32 字
17-48	64 字
49-112	128 字
113-240	256 字

7.1.5 事件

每种类型异常的可能原因如下所示：

重置 重置是一个特殊异常，当重置信号有效时，会以不可恢复的方式终止执行。有两种级别的重置：

- *上电重置* (POR) 会重置内核、系统控制空间以及调试逻辑。
- 局部重置会重置内核以及大多数系统控制空间。不会重置系统控制空间中某些与调试相关的资源，也不重置调试逻辑。

当重置信号无效时，将从固定点重新开始执行。

NMI 不可屏蔽中断。

HardFault 这是其他异常处理机制无法处理的任何一般错误。它通常用于不可恢复的错误。

HardFault 用于错误升级。

MemManage

进行数据和指令访问时，内存保护单元产生的内存保护错误会引发 MemManage 异常。如果禁用了此错误，则内存保护错误会升级为 HardFault。

BusFault MemManage 处理的错误以外的内存相关错误会引发 BusFault 异常。这些错误通常是在系统总线上检测到的。它们可以为同步或异步。如果禁用了此错误，则这些错误会升级为 HardFault。

UsageFault 除内存错误以外的指令执行错误。这可能为：

- 试图执行未定义的指令
- 执行指令后导致无效状态
- 异常返回时出错
- 试图使用不可用或已禁用的协处理器
- 除零（如果配置为导致 UsageFault）
- 非对齐地址（如果配置为导致 UsageFault）。

SVCall 执行 SVC 指令导致的同步超级用户调用。应用程序代码使用 SVC 指令请求需要对系统进行特权访问的服务。

调试监控器

同步调试事件。这可能是执行了*断点* (BKPT) 指令。

PendSV 异步超级用户调用。

SysTick 系统计时器事件。

外部中断 (n)

嵌套向量中断控制器发出的信号。请参阅第7-8 页的*嵌套向量中断控制器*。

7.1.6 异常优先级和抢先能力

具有低优先级编号的异常可以比具有更高优先级编号的异常抢先执行。也就是说，当处理器处于处理程序模式时，如果某异常比当前正在处理的异常具有更高的优先级编号，则会先处理该异常。任何具有相同或更低优先级编号的异常将被*暂挂*。

当异常处理程序终止时:

- 如果没有暂挂的异常，则处理器会返回到线程模式，并继续执行应用程序。
- 如果有暂挂的异常，则转为执行优先级编号最低的暂挂异常的处理程序。如果有两个暂挂异常具有相同的最低优先级编号，则先处理异常编号最低的异常。

重置、不可屏蔽中断和 HardFault 具有优先级 -3、-2 和 -1。所有其他异常具有可配置的优先级，即 0 或更高。

7.1.7 嵌套向量中断控制器

根据具体实现情况，*嵌套向量中断控制器* (NVIC) 最多可支持 240 个外部中断，每个外部中断最多可具有 256 个可重新动态划分的不同优先级别。它支持优先级别中断源和脉冲中断源。当进入中断时，处理器状态会自动保存在硬件中，并在退出中断时恢复。NVIC 还支持末尾连锁中断。

Cortex-M3 中使用 NVIC 表示 Cortex-M3 的向量表与以前的 ARM 内核有很大不同。Cortex-M3 向量表包含异常处理程序和 ISR 的地址，而多数其他 ARM 内核包含指令。初始堆栈指针以及重置处理程序的地址必须分别位于 0x0 和 0x4。重置时，这些地址会加载到适当的 CPU 寄存器中。

7.2 编写异常表

填充向量表最简单的方法是使用分散文件将函数指针的 C 数组放置到内存地址 0x0。您可以使用 C 数组来配置初始堆栈指针、映像入口点以及异常处理程序的地址。

示例 7-1 异常处理程序的 C 结构示例

```
/* Filename: exceptions.c */
typedef void(* const ExecFuncPtr)(void);
/* Place table in separate section */
#pragma arm section rodata="exceptions_area"
ExecFuncPtr exception_table[] = {
    (ExecFuncPtr)&Image$$ARM_LIB_STACKHEAP$$ZI$$Limit,
                        /* Initial Stack Pointer, from linker-generated symbol
*/
    (ExecFuncPtr)&__main,   /* Initial PC, set to entry point  */
    &NMIException,
    &HardFaultException,
    &MemManageException,
    &BusFaultException,
    &UsageFaultException,
    0, 0, 0, 0,             /* Reserved */
    &SVCHandler,
    &DebugMonitor,
    0,                      /* Reserved */
    &PendSVC,
    (ExecFuncPtr)&SysTickHandler,
    /* Configurable interrupts start here...*/
    &InterruptHandler,
    &InterruptHandler,
    &InterruptHandler
    /*
    :
    */
};
#pragma arm section
```

此结构中的前两项为初始堆栈指针和映像入口点。示例 7-1 将 C 库入口点 (__main) 用作映像的入口点。

异常表还具有自己的段。它使用:

```
#pragma arm section rodata="exceptions_area"
```

这会指示编译器将:

```
#pragma arm section rodata="exceptions_area"
```

和

```
#pragma arm section
```

之间的所有 RO 数据放置到自己的段中，该段名为 exceptions_area。然后，您可以在分散文件中引用此结构，将异常表放置在内存映射的正确位置（地址 0x0）。

7.3 编写异常处理程序

发生异常时，内核会保存系统状态，并在返回时恢复该状态。异常处理程序无需保存或恢复系统状态，它们可以作为与 ABI 兼容的普通 C 函数进行编写。

ARM 体系结构的应用程序二进制接口 (ABI) 要求堆栈必须在所有外部接口上 8 字节对齐，如在不同源文件中互调函数。不过，代码无需保持内部 8 字节堆栈对齐，例如在叶函数中。

这表示当发生 IRQ 时，堆栈可能无法正确地 8 字节对齐。 当发生异常时，版本 1 以及更高版本的 Cortex-M3 处理器可以自动对齐堆栈指针。您可以通过设置配置控制寄存器中的 STKALIGN（第 9 位）来启用此行为，地址为 0xE000ED14。

如果您使用的是版本 0 Cortex-M3 处理器，则此调整不在硬件中执行。编译器会在您的 IRQ 处理程序中生成正确对齐堆栈的代码。要执行此操作，您必须在 IRQ 处理程序前附加前缀 __irq 并使用 --cpu=Cortex-M3-rev0 编译器开关，而不是 --cpu=Cortex-M3。

示例 7-2 简单 C 异常处理程序

```
__irq void SysTickHandler(void)
{
    printf("----- SysTick Interrupt -----");
}
```

注意

中断服务例程必须清除中断源。

Cortex-M3 处理器完全在内核中划分异常优先级别、处理异常嵌套以及保存易损坏的寄存器，从而使处理更有效。这表示当进入每个异常处理程序时，中断仍然由内核保持启用状态。

7.4 放置异常表

由于异常表已放置在目标文件中异常表自己的段中，可以使用分散文件轻松地将它放置在 0x0。

示例 7-3 在分散文件中放置异常表

```
LOAD_REGION 0x00000000 0x00200000
{
    ;; Maximum of 256 exceptions (256*4 bytes == 0x400)
    VECTORS 0x0 0x400
    {
        exceptions.o (exceptions_area, +FIRST)
    }
}
```

注意

+FIRST 确保 exceptions_area 放置在区中的起始位置，并防止链接器删除未使用段的机制删除向量表。

7.5 配置系统控制空间寄存器

系统控制空间 (SCS) 寄存器位于 0xE000E000。由于有大量单独寄存器，因此最好使用一个结构来表示这些寄存器。您可以使用与异常表类似的方法向分散文件添加结构，将此结构放置到正确的内存位置。示例 7-4 介绍 SCS 寄存器的结构示例。

示例 7-4 SCS 寄存器结构

```
typedef volatile struct {
    int MasterCtrl;
    int IntCtrlType;
    int zReserved008_00c[2];                    /* Reserved space */
    struct {
        int Ctrl;
        int Reload;
        int Value;
        int Calibration;
    } SysTick;
    int zReserved020_0fc[(0x100-0x20)/4];      /* Reserved space */
    /* Offset 0x0100 */
    struct {
        int Enable[32];
        int Disable[32];
        int Set[32];
        int Clear[32];
        int Active[64];
        int Priority[64];
    } NVIC;
    int zReserved0x500_0xcfc[(0xd00-0x500)/4];      /* Reserved space */
    /* Offset 0x0d00 */
    int CPUID;
    int IRQcontrolState;
    int ExceptionTableOffset;
    int AIRC;
    int SysCtrl;
    int ConfigCtrl;
    int SystemPriority[3];
    int SystemHandlerCtrlAndState;
    int ConfigurableFaultStatus;
    int HardFaultStatus;
    int DebugFaultStatus;
    int MemManageAddress;
    int BusFaultAddress;
    int AuxFaultStatus;
    int zReserved0xd40_0xd90[(0xd90-0xd40)/4];      /* Reserved space */
    /* Offset 0x0d90 */
```

```
    struct {

        int Type;
        int Ctrl;
        int RegionNumber;
        int RegionBaseAddr;
        int RegionAttrSize;
    } MPU;
} SCS_t;
```

注意

这可能没有包含您的设备中的所有 SCS 寄存器。请参阅您的设备的硅制造商提供的参考手册。

7.6 配置单个 IRQ

在中断设置启用寄存器中，每个 IRQ 有单独的启用位，中断设置启用寄存器是 NVIC 寄存器的组成部分。要启用中断，请将中断设置启用寄存器中相应的位设置为 1。有关中断设置启用寄存器的特定信息，请参阅您使用的设备的参考手册。

示例 7-5 介绍第7-13 页的示例 7-4 中描述的 SCS 结构的中断启用代码示例。

示例 7-5 IRQ 启用函数

```
void NVIC_enableISR(unsigned isr)
{
    /* The isr argument is the number of the interrupt to enable. */
    SCS.NVIC.Enable[ (isr/32) ] = 1<<(isr % 32);
}
```

注意

NVIC 区中的某些寄存器只能在特权模式下访问。

您可以通过将中断清除启用寄存器中适当的位清除为 0 来禁用单个 IRQ。

7.6.1 中断优先级

您可以使用中断优先级寄存器为每个中断分配优先级别。根据具体实现情况，最多可为每个中断分配 256 个不同的优先级别。最多可使用 8 个位来表示优先级别。四个中断优先级组成一组，存储在内存的一个字中。

0 表示最高优先级，255 表示最低优先级。

7.7 超级用户调用

SVC 指令（以前为 SWI）生成*超级用户调用*（以前称为“软件中断”）。超级用户调用通常用于在操作系统上请求特权操作或访问系统资源。

SVC 指令中嵌入了一个数字，该数字通常称为 SVC 编号。在多数 ARM 处理器上，这可以用于表示正在请求的服务。不过，Cortex-M3 处理器会在进入首次异常时，将自变量寄存器保存到堆栈中。

在 SVC 处理程序执行第一个指令之前较晚处理的异常可能会损坏 R0 到 R3 中仍保存的参数副本。这表示参数的堆栈副本必须由 SVC 处理程序使用。还必须通过修改堆栈中的寄存器值将任何返回值传递回调用方。为此，必须实现一小段汇编代码作为 SVC 处理程序的开头部分。这可以标识将寄存器保存到哪个堆栈，从指令中提取 SVC 编号，然后将该编号和指针传递到用 C 编写的处理程序的程序体的参数中。

示例 7-6 介绍 SVC 处理程序示例。此代码会测试处理器设置的 EXC_RETURN 值，确定当调用 SVC 时，正在使用哪个堆栈指针。在多数系统上无需执行此测试，因为在典型系统设计中，只能通过用户代码执行超级用户调用，用户代码使用的是进程堆栈。在这种情况下，汇编代码可以包含单个 MSR 指令，后接指向处理程序 C 程序体的尾调用跳转（B 指令）。

示例 7-6 SVC 处理程序示例

```
__asm void SVCHandler(void)
{
    IMPORT SVCHandler_main
    TST lr, #4
    MRSEQ r0, MSP
    MRSNE r0, PSP
    B SVCHandler_main
}
void SVCHandler_main(unsigned int * svc_args)
{
    unsigned int svc_number;
    /*
    * Stack contains:
    * r0, r1, r2, r3, r12, r14, the return address and xPSR
    * First argument (r0) is svc_args[0]
    */
    svc_number = ((char *)svc_args[6])[-2];
    switch(svc_number)
    {
        case SVC_00:
            /* Handle SVC 00 */
```

```
            break;
        case SVC_01:
            /* Handle SVC 01 */
            break;
        default:
            /* Unknown SVC */
            break;
    }
}
```

示例 7-7 介绍如何为多个 SVC 进行不同声明。**__svc** 是一个编译器关键字，它将函数调用替换为包含指定编号的 SVC 指令。

示例 7-7 在 C 代码中调用 SVC 的示例

```
#define SVC_00 0x00
#define SVC_01 0x01
void __svc(SVC_00) svc_zero(const char *string);
void __svc(SVC_01) svc_one(const char *string);

int call_system_func(void)
{
    svc_zero("String to pass to SVC handler zero");
    svc_one("String to pass to a different OS function");
}
```

7.8 系统计时器

SCS 中包含系统计时器 SysTick，操作系统可使用它使从其他平台移植的操作更加轻松。软件可以轮询 SysTick，或者您可以将它配置为生成中断。SysTick 中断在向量表中有自己的条目，因此可以有自己的处理程序。

表7-3 介绍用于配置 SysTick 的四个寄存器。

表7-3

名称	地址	说明
SysTick 控制和状态	0xE000E010	SysTick 的基本控制：启用、时钟源、中断或轮询
SysTick 重新加载值	0xE000E014	当前值寄存器为 0 时要加载的值
SysTick 当前值	0xE000E018	倒计时的当前值
SysTick 校准值	0xE000E01C	包含生成 10 毫秒间隔时间的滴答声数目以及其他信息

7.8.1 配置 SysTick

要配置 SysTick，请将 SysTick 事件间所需的间隔时间加载到 SysTick 重新加载值寄存器。SysTick 控制和状态寄存器中的计时器中断（即 COUNTFLAG 位）在从 1 转换为 0 时被激活，因此它会激活每个 *n*+1 时钟滴答声。如果所需的时间间隔为 100，请将 99 写入 SysTick 重新加载值寄存器中。SysTick 重新加载值寄存器支持 1 到 0x00FFFFFF 之间的值。

如果想使用 SysTick 以固定的时间间隔生成事件，例如 1 毫秒，则可以使用 SysTick 校准值寄存器来缩放重新加载寄存器的值。SysTick 校准值寄存器是只读寄存器，包含 10 毫秒的脉冲数，它在 TENMS 字段 [23:0] 位中。

此寄存器还具有 SKEW 位。[30] 位 == 1 表示 TENMS 段中 10 毫秒的校准值不是精确的 10 毫秒，因为存在时钟频率问题。[31] 位 == 1 表示提供了参考时钟。

控制和状态寄存器可以读取 COUNTFLAG [16] 位或通过 SysTick 生成中断，来轮询计时器。

缺省情况下，SysTick 配置为轮询模式。在这种模式下，用户代码会轮询 COUNTFLAG，确定是否已发生 SysTick 事件。这由是否已设置 COUNTFLAG 来指示。读取控制和状态寄存器会清除 COUNTFLAG。要将 SysTick 配置为生成中断，请将 SysTick 控制和状态寄存器的 TICKINT [1] 位设置为 1。您还必须在 NVIC 中启用适当的中断，然后使用 CLKSOURCE [2] 位选择时钟源。将此位设置为 1 会选择内核时钟；设置为 0 会选择外部参考时钟。

通过将 SysTick 状态和控制寄存器的 [0] 位设置为 1 来启用计时器。

7.9 移植为其他 ARM 处理器编写的异常处理代码

您的异常处理程序必须适用于 Cortex-M3。

通常情况下不需要用汇编语言编写低级处理程序，因为重入由内核处理。如果低级处理程序执行其他工作，则可能需要将重入分割成新处理程序可以调用的单独函数。为了清晰起见，务必使用 `__irq` 关键字来标记 IRQ 处理程序，并确保编译器可保持 Cortex-M3 版本 0 硬件的堆栈对齐。

Cortex-M3 没有 FIQ 输入，因此 ARM7TDMI 工程上发出 FIQ 信号的任何外围设备都必须移到高优先级的向量化中断中。可能需要确定此类中断的处理程序不会使用编组的 FIQ 寄存器，因为 Cortex-M3 处理程序没有编组的 FIQ 寄存器，并且对于任何其他常规 IRQ 处理程序，必须将 R8-R12 存入堆栈。

最后，您必须编写新的初始化函数来配置 NVIC，包括中断优先级。然后，您可以启用中断，然后输入主应用程序代码。

7.9.1 关键段和异常行为

在 Cortex-M3 处理器上，完全由内核来划分异常优先级别、处理异常嵌套以及保存易损坏的寄存器，从而使处理非常有效并使中断等待时间最短。这表示当进入每个异常处理程序时，中断仍然由内核保持启用状态。另外，如果从异常中返回时禁用了中断，则这些中断不会由处理器自动重新启用。无法执行启用中断然后从异常中返回的原子操作。如果在处理程序中临时禁用中断，则必须在执行用于返回的指令之前用一个单独的指令重新启用这些中断。因此，在异常返回前可能会立即发生异常。

异常模型的这些功能可能会影响代码中的关键段，具体取决于系统设计。关键段是指在执行过程中要求禁用中断的段，以便将它们作为不中断的块执行，例如操作系统中的上下文切换代码。某些遗留代码可能会假定进入异常处理程序时中断已禁用，并只在完成关键段后才显式启用中断。在 Cortex-M3 的新异常模型下，这些假定不成立，因此可能需要重新编写此类代码以便将此因素考虑在内。

第 8 章
调试通信通道

本章介绍如何使用*调试通信通道* (DCC)。它包含以下几节:

8.1 关于调试通信通道

ARM® 内核中的 EmbeddedICE® 逻辑单元包含调试通信通道。这可使数据通过 JTAG 端口和协议转换器在目标和主机调试器之间传递，同时无需停止程序流或进入调试状态。本章介绍目标上运行的程序和主机调试器如何访问 DCC。

8.2 目标数据传送

目标使用 ARM 指令 MCR 和 MRC 将 DCC 作为内核上的协处理器 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

用于接收来自调试器的数据的 32 位宽寄存器。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

用于向调试器发送数据的 32 位宽寄存器。以下指令将 Rn 中的值写到写入寄存器中：

```
MCR p14, 0, Rn, c1, c0
```

注意

有关访问 ARM10 和 ARM11 内核 DCC 寄存器的信息，请参阅相应的技术参考手册。ARM9 之后的各处理器中，所用指令、状态位位置以及对状态位的解释都有所不同。

8.3 轮询调试通信

除了通信数据读取寄存器和写入寄存器之外，DCC 还提供了通信数据控制寄存器。

8.3.1 通信数据控制寄存器

以下指令在 Rd 中返回控制寄存器的值：

```
MRC p14, 0, Rd, c0, c0
```

此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） 从目标的角度表示通信数据写入寄存器是否空闲：
- **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） 从目标的角度表示通信数据读取寄存器中是否有新数据：
- **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。

注意

调试器不能利用协处理器 14 直接访问调试通信通道，因为这对调试器无意义。但调试器可使用扫描链读写 DCC 寄存器。DCC 数据和控制寄存器可映射到 EmbeddedICE 逻辑单元中的地址。若要查看 EmbeddedICE 逻辑寄存器，请参阅您的调试器和调试目标的相关文档。

8.3.2 目标到调试器的通信

这是运行于 ARM 内核上的应用程序与运行于主机上的调试器之间的通信事件顺序：

1. 目标应用程序检查 DCC 写入寄存器是否空闲可用。为此，目标应用程序使用 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，则通信数据写入寄存器已清空，应用程序对协处理器 14 使用 MCR 指令将字写入通信数据写入寄存器。写入寄存器操作会自动设置 W 位。如果 W 位已设置，则表明调试器尚未清空通信数据写入寄存器。此时，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 轮询通信数据控制寄存器。如果调试器发现 W 位已设置，则它可以读 DCC 数据寄存器，以读取应用程序发送的信息。读取数据的进程会自动清除通信数据控制寄存器中的 W 位。

示例 8-1 显示了这一过程。您可从主示例目录的 ...\dcc\outchan.s 中获得该示例代码。

示例 8-1

```
     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
     SVC   0x123456       ; ARM semihosting (formerly SWI)
outdata
     DCB "Hello there!"
     END
```

执行该示例：

1. 汇编 outchan.s：
   ```
   armasm --debug outchan.s
   ```
2. 链接输出对象：
   ```
   armlink outchan.o -o outchan.axf
   ```
   该链接步骤将创建可执行文件 outchan.axf
3. 载入并执行该映像。有关信息，请参阅您的调试器文档。

8.3.3 调试器到目标的通信

这是运行于主机上的调试器向运行于内核上的应用程序传输消息的事件顺序：

1. 调试器轮询通信数据控制寄存器的 R 位。如果 R 位已清除，则通信数据读取寄存器已清空，可将数据写入此寄存器，以供目标应用程序读取。
2. 调试器通过扫描链 2 将数据扫描到通信数据读取寄存器中。此操作会自动设置通信数据控制寄存器中的 R 位。
3. 目标应用程序轮询通信数据控制寄存器中的 R 位。如果该位已经设置，则通信数据读取寄存器中已经有数据，应用程序可使用 MRC 指令从协处理器 14 读取该数据。同时，读取指令还会清除 R 位。

第8-7 页的示例 8-2 中显示的目标应用程序代码演示了这一过程。您可从主示例目录的 ...\dcc\inchan.s 中获得该示例代码。

执行该示例：

1. 在主机上创建一个输入文件，其中包含如 And goodbye! 等。
2. 汇编 inchan.s：
   ```
   armasm --debug inchan.s
   ```
3. 链接输出对象：
   ```
   armlink inchan.o -o inchan.axf
   ```
   该链接步骤将创建可执行文件 inchan.axf
4. 载入并执行该映像。有关信息，请参阅您的调试器文档。

示例 8-2

```
     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
     SVC   0x123456       ; ARM semihosting (formerly SWI)

     AREA  Storage, DATA, READWRITE
indata
     DCB   "Duffmessage#"
     END
```

8.4 中断驱动调试通信

第8-4 页的*轮询调试通信*中提供的示例演示了如何轮询 DCC。通过将 Embedded ICE 逻辑单元中的 COMMRX 和 COMMTX 信号连接到您的中断控制程序，可以将这些示例转化为中断驱动示例。

然后，就可将 第8-5 页的示例 8-1 和 第8-7 页的示例 8-2 中的读写代码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6-29 页的*中断处理程序*。

8.5 从 Thumb 状态访问

对于 ARMv6T2 以前的处理器，如果内核处于 Thumb 状态，由于没有协处理器指令，因此不能使用调试通信通道。

对于这个问题，有三种可能的解决方法：

- 可将每个轮询例程写入一个 SVC 处理程序，然后就可在无论 ARM 还是 Thumb 状态下执行该处理程序。进入 SVC 处理程序后立即将内核转为 ARM 状态，在该状态下可以使用协处理器指令。有关 SVC 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 ARM 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 *交互操作 ARM 和 Thumb*。
- 使用中断驱动通信，而不使用轮询通信。中断处理程序运行在 ARM 指令集状态，因此可以直接访问协处理器指令。

第 9 章
半主机

本章介绍半主机机制。它包含以下几节:

I.1 关于半主机

半主机可使运行在 ARM® 目标上的代码使用运行 RealView® 调试器的主机上的 I/O 功能。这些功能包括键盘输入、屏幕输出和磁盘 I/O 等。

I.1.1 什么是半主机？

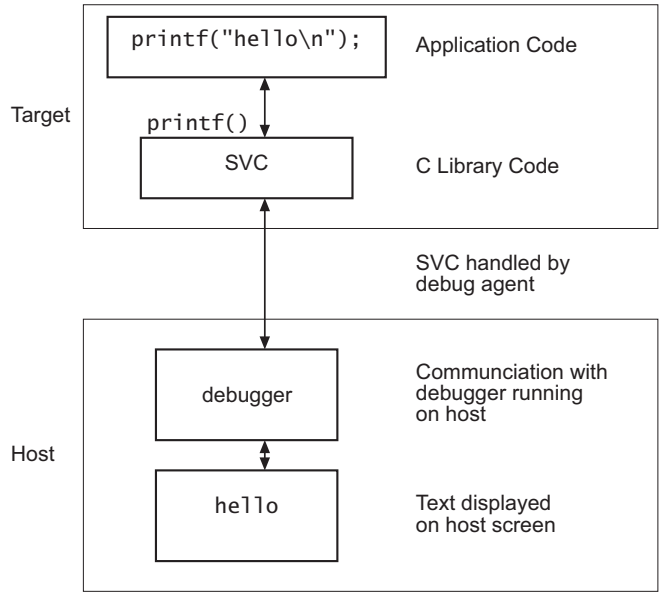
半主机是用于 ARM 目标的一种机制，可将来自应用程序代码的输入/输出请求传送至运行调试器的主机。例如，使用此机制可以启用 C 库中的函数，如 `printf()` 和 `scanf()`，来使用主机的屏幕和键盘，而不是在目标系统上配备屏幕和键盘。

这种机制很有用，因为开发时使用的硬件通常没有最终系统的所有输入和输出设备。半主机可让主机来提供这些设备。

半主机是通过一组定义好的软件指令（如 SVC）来实现的，这些指令通过程序控制生成异常。应用程序调用相应的半主机调用，然后调试代理处理该异常。调试代理提供与主机之间的必需通信。

半主机接口对 ARM 公司提供的所有调试代理都是通用的。在无需移植的情况下使用 *RealView ARMulator® ISS* (RVISS)、*指令集系统模型* (ISSM)、RealView ICE 或 RealMonitor 时，会执行半主机操作，请参阅第 A-3 页的图I-1。

在很多情况下，半主机由库函数内的代码调用。应用程序还可以直接调用半主机操作。有关 ARM C 库中半主机支持的详细信息，请参阅*库指南*中的第 2 章 *C 和 C++ 库*。



图I-1 半主机概述

注意

ARMv7 之前的 ARM 处理器使用 SVC 指令（以前称为 SWI 指令）进行半主机调用。但是，如果要为 v6-M 或 v7-M 处理器（如 Cortex-M1 或 Cortex-M3）进行编译，则半主机通过使用 BKPT 指令来实现。

I.1.2 半主机接口

ARM 和 Thumb® SVC 指令包含一个字段，该字段对应用程序代码使用的 SVC 编号进行编码。系统 SVC 处理程序可以解码此编号。

注意

如果要为 v6-M 或 v7-M 处理器（如 Cortex-M1 或 Cortex-M3）进行编译，则使用 Thumb BKPT 指令，而不是 Thumb SVC 指令。BKPT 和 SVC 都使用 8 位直接值。在所有其他方面，半主机对于所有支持的 ARM 处理器都是一样的。

半主机操作的请求使用单个 SVC 编号，其他编号可供应用程序或操作系统使用。用于半主机的 SVC 编号取决于目标机体系结构或处理器：

SVC 0x123456 对于所有体系结构，在 ARM 状态下。

SVC 0xAB　在 Thumb 状态（除 ARMv7-M 以外）和 ARM 状态下。不保证来自 ARM 或第三方的*所有*调试目标实现这种行为。

BKPT 0xAB　对于 ARMv7-M 来说，仅 Thumb 状态。

另请参阅第 A-4 页的*更改半主机操作编号*。

SVC 编号向调试代理指示 SVC 指令是半主机请求。为辨别操作，操作类型在 r0 中传递。所有其他参数在 r1 指向的块中传递。

结果在 r0 中返回，或者是显式的返回值，或者是指向数据块的指针。即使未返回结果，也假定 r0 被破坏。

在 r0 中传递的可用半主机操作编号分配如下：

0x00-0x31　由 ARM 使用

0x32-0xFF　ARM 保留，以备将来使用。

0x100-0x1FF　为用户应用程序保留。这些编号不由 ARM 使用。

但是，如果要编写自己的 SVC 操作，建议使用其他 SVC 编号，而不要使用半主机 SVC 编号和这些操作类型编号。

0x200-0xFFFFFFFF

未定义和目前未使用的编号。建议不要使用这些编号。

在以下章节中，在操作名称之后括号中的编号是放入 r0 中的值，例如 SYS_OPEN (0x01)。

如果从汇编语言代码中调用 SVC，ARM 建议您使用在 semihost.h 中定义的操作名。它随 RealView ARMulator 扩展套件安装。可以用 EQU 指令定义操作名称。例如：

```
SYS_OPEN    EQU 0x01
SYS_CLOSE   EQU 0x02
```

更改半主机操作编号

强烈建议不要更改半主机操作编号。如果要更改，必须进行以下操作：

- 更改系统中的所有代码（包括库代码），使其使用新编号
- 重新配置调试器，使其使用新编号。

I.2 半主机实现

半主机提供的功能通常对所有调试代理都相同。但半主机的实现因主机而异。

本节介绍不同调试代理上的半主机实现。

I.2.1 RealView ARMulator ISS

遇到半主机请求时，RVISS 直接捕获 SVC，不执行向量表中 SVC 入口的指令。

若要在 RVISS 中关闭对半主机的支持，请将 `default.ami` 文件中的 `Default_Semihost` 改为 `No_Semihost`。

有关详细信息，请参阅*《RealView ARMulator ISS 用户指南》*。

I.2.2 RealView ICE

使用 RealView ICE DLL 时，可用实际的 SVC 异常处理程序处理半主机，也可通过使用断点模拟处理程序来处理半主机。有关使用 RealView ICE 的半主机的详细信息，请参阅*《RealView ICE 和 RealView Trace 用户指南》*。

I.2.3 指令集系统模型

遇到半主机请求时，ISSM 直接捕获该请求，不执行向量表中 SVC 入口的指令。有关使用 ISSM 的半主机的详细信息，请参阅调试器文档。

若要在 ISSM 中关闭对半主机的支持，请在调试器中配置目标，或在 `default.smc` 文件中更改相应的入口：

```
...Name="semihosting-enable" Type="Bool">1</param>
```

I.2.4 RealMonitor

RealMonitor 实现的 SVC 处理程序必须与系统相集成才能启用半主机支持。

目标执行半主机 SVC 指令时，RealMonitor SVC 处理程序执行与主机之间所需的通信。

有关详细信息，请参阅 RealMonitor 随附的文档。

I.3 半主机操作

本部分列出了可在主机和 ARM 目标之间启用调试 I/O 功能的半主机操作。

I.3.1 angel_SWIreason_EnterSVC (0x17)

将处理器设置为超级用户模式，通过设置新 CPSR 中的两个中断掩码位来禁用所有中断。使用 RealView ICE 或 RealMonitor，“用户”堆栈指针 r13_USR 被复制到“超级用户”堆栈指针 r13_SVC，并设置当前 CPSR 中的 I 位和 F 位，从而禁用正常和快速中断。

注意

如果用 RVISS 进行调试：

- r0 设置为零，表明没有可用于返回用户模式的函数
- 用户模式堆栈指针 *不* 被复制到超级用户堆栈指针。

入口

未使用寄存器 r1。CPSR 可指定是用户模式还是超级用户模式。

返回值

退出时，r0 包含为返回到用户模式而要调用的函数的地址。该函数具有以下原型：

```
void ReturnToUSR(void)
```

如果在用户模式下调用 EnterSVC，则此例程使调用方返回到用户模式并恢复中断标记。否则，此例程的操作未定义。

如果进入了用户模式，由于会复制用户堆栈指针，所以超级用户堆栈将丢失。返回到用户例程可将 r13_SVC 恢复为超级用户模式下的堆栈值，但是此堆栈决不能由应用程序使用。

执行完 SVC 之后，当前链接寄存器是 r14_SVC，而不是 r14_USR。如果调用之后需要 r14_USR 的值，则必须在调用之前将其存入堆栈，调用之后弹出，与 BL 函数调用相同。

I.3.2 angel_SWIreason_ReportException (0x18)

应用程序可以直接调用此 SVC 向调试器报告异常。最常见的用途是用 ADP_Stopped_ApplicationExit 来报告执行已完成。

入口

进入时，r1 设为表I-1 和第A-8 页的表I-2 中列出的值之一。这些值在 angel_reasons.h 中定义。

如果将调试器变量 vector_catch 设置为捕获某种异常类型，并且调试代理能够报告该异常类型，则会产生该类硬件异常。

表I-1 硬件向量原因代码

名称	十六进制值
ADP_Stopped_BranchThroughZero	0x20000
ADP_Stopped_UndefinedInstr	0x20001
ADP_Stopped_SoftwareInterrupt	0x20002
ADP_Stopped_PrefetchAbort	0x20003
ADP_Stopped_DataAbort	0x20004
ADP_Stopped_AddressException	0x20005
ADP_Stopped_IRQ	0x20006
ADP_Stopped_FIQ	0x20007

作为缺省操作，异常处理程序可以在处理程序链的末尾使用这些 SVC，以表明未处理该异常。

表I-2 软件原因代码

名称	十六进制值
ADP_Stopped_BreakPoint	0x20020
ADP_Stopped_WatchPoint	0x20021
ADP_Stopped_StepComplete	0x20022
ADP_Stopped_RunTimeErrorUnknown	*0x20023

表I-2 软件原因代码 (续)

名称	十六进制值
ADP_Stopped_InternalError	*0x20024
ADP_Stopped_UserInterruption	0x20025
ADP_Stopped_ApplicationExit	0x20026
ADP_Stopped_StackOverflow	*0x20027
ADP_Stopped_DivisionByZero	*0x20028
ADP_Stopped_OSSpecific	*0x20029

在表I-2 中，值旁边的 * 表示 ARM 调试器不支持该值。调试器对这些值报告 Unhandled ADP_Stopped exception。

返回值

这些调用不返回任何值。但是，通过执行 RDI_Execute 请求或等效请求，调试器可以请求应用程序继续执行。在这种情况下，继续使用进入 SVC 时或者随后被调试器修改的寄存器来执行。

I.3.3 SYS_CLOSE (0x02)

关闭主机系统上的文件。句柄必须引用使用 SYS_OPEN 打开的文件。

入口

进入时，r1 包含一个指向一字自变量块的指针：

字 1 包含打开文件的句柄。

返回值

退出时，r0 包含：

- 0，如果调用成功
- -1，如果调用不成功。

I.3.4 SYS_CLOCK (0x10)

返回自执行开始以来的百分秒数。

因为通信开销或其他代理程序特有因素，对于一些基准目的，由这个 SVC 返回的值的作用有限。例如，使用 RealView ICE 时，请求被传递回主机以便执行。这可能导致在传送和进程调度中不可预测的延迟。

使用此函数计算时间间隔的方法是计算计时下和不计时下代码序列的间隔差异。

一些系统启用更准确的计时，请参阅第A-11 页的*SYS_ELAPSED (0x30)* 和第A-19 页的*SYS_TICKFREQ (0x31)*。

入口

寄存器 r1 必须包含零。没有其他参数。

返回值

退出时，r0 包含：

- 自任意时间点开始的百分秒数，如果调用成功
- -1，如果调用失败，例如由于通信错误。

I.3.5 SYS_ELAPSED (0x30)

返回自执行开始后已发生的目标滴答声数目。使用 SYS_TICKFREQ 可确定滴答声的频率。

入口

进入时，r1 指向用于返回已发生滴答声数目的二字数据块：

字 1 在双字值中较不重要的字。

字 2 较重要的字。

返回值

退出时：

- 如果 r1 确实指向包含已发生滴答声数目的双字，则 r0 包含 -1。RealView ICE 不支持此 SVC，并始终在 r0 中返回 -1。
- r1 指向包含已发生滴答声数目的双字（低位字在前）。

I.3.6 SYS_ERRNO (0x13)

返回 C 库 errno 变量的值，该变量与半主机 SVC 的主机实现相关。很多 C 库半主机函数可以设置 errno 变量，这些函数包括：

- SYS_REMOVE
- SYS_OPEN
- SYS_CLOSE
- SYS_READ
- SYS_WRITE
- SYS_SEEK。

errno 是否设置以及设置为何值完全是特定于主机的，ISO C 标准只是定义了该行为在何处发生。

入口

没有参数。寄存器 r1 必须为零。

返回值

退出时，r0 包含 C 库 errno 变量的值。

I.3.7 SYS_FLEN (0x0C)

返回指定文件的长度。

入口

进入时，r1 包含一个指向一字自变量块的指针：

字 1 先前打开的可搜索文件对象的句柄。

返回值

退出时，r0 包含：

- 文件对象的当前长度，如果调用成功
- -1，如果出错。

I.3.8 SYS_GET_CMDLINE (0x15)

返回用于调用可执行程序的命令行，即 argc 和 argv。

入口

进入时，r1 指向用于返回命令字符串及其长度的二字数据块：

字 1 指向至少为字 2 中指定大小的缓冲区的指针。

字 2 以字节为单位的缓冲区长度。

返回值

退出时：

- 寄存器 r1 指向一个二字数据块：

  字 1 一个指针，指向命令行的空终止字符串。

  字 2 字符串的长度。

  调试代理可能对可传送字符串的最大长度有所限制。但是，代理必须能够传送至少 80 字节的命令行。

- 寄存器 r0 包含错误代码：
  - 0，如果调用成功
  - -1，如果调用失败，例如由于通信错误。

I.3.9 SYS_HEAPINFO (0x16)

返回系统栈和堆参数。通常，返回值为初始化过程中 C 库所使用的值。对于 RVISS，返回值为 peripherals.ami 中提供的值。对于 RealView ICE，返回值为映像位置和内存顶部。

C 库可以覆盖这些值。有关 C 库中内存管理方面的详细信息，请参阅*库指南*中的第2-78 页的*调整存储管理*。

使用 top_of_memory 调试器变量，主机调试器可确定要返回的实际值。

入口

进入时，r1 包含指向四字数据块的指针的地址。数据块的内容由函数填充。有关数据块的结构和返回值的信息，请参阅示例 I-1。

示例 I-1

```
struct block {
    int heap_base;
    int heap_limit;
    int stack_base;
    int stack_limit;
};
struct block *mem_block, info;
mem_block = &info;
AngelSWI(SYS_HEAPINFO, (unsigned) &mem_block);
```

注意

如果数据块的字 1 的值为零，则 C 库用 Image$$ZI$$Limit 替换零。这个值对应内存映射中数据区的顶部。

返回值

退出时，r1 包含指向结构的指针的地址。

如果结构中的一个值为 0，则系统无法计算实值。

I.3.10 SYS_ISERROR (0x08)

确定从另一个半主机调用返回的代码是否为错误状态。调用中传入一个参数块，其中包含要加以检验的错误代码。

入口

进入时，r1 包含一个指向一字数据块的指针：

字 1 要检查的所需状态字。

返回值

退出时，r0 包含：

- 0，如果状态字不是错误指示
- 非零值，如果状态字是错误指示。

I.3.11 SYS_ISTTY (0x09)

检查文件是否连接到交互设备。

入口

进入时，r1 包含一个指向一字自变量块的指针：

字 1 先前打开的文件对象的句柄。

返回值

退出时，r0 包含：

- 1，如果该句柄标识交互设备
- 0，如果该句柄标识文件
- 不等于 1 或 0 的值，如果出错。

I.3.12 SYS_OPEN (0x01)

打开主机系统上的文件。根据主机操作系统的路径约定，文件路径可指定为相对于主机进程当前目录的相对路径或绝对路径。

ARM 目标将特殊的路径名 :tt 解释为控制台输入流（用于打开-读取操作）或控制台输出流（用于打开-写入操作）。对于引用 C stdio 流的应用程序，打开这些流是作为标准启动代码的一部分来执行的。

入口

进入时，r1 包含一个指向三字自变量块的指针：

字 1 一个指针，指向包含文件或设备名的空终止字符串。

字 2 一个指定文件打开模式的整数。表I-3 给出该整数的有效值，以及这些值对应的 ISO C fopen() 模式。

字 3 给出字 1 指向的字符串长度的整数。

该长度不包含必须存在的终止空字符。

表I-3 模式的值

模式	0	1	2	3	4	5	6	7	8	9	10	11
ISO C fopen 模式[a]	r	rb	r+	r+b	w	wb	w+	w+b	a	ab	a+	a+b

a. 不支持非 ANSI 选项。

返回值

退出时，r0 包含：

- 非零句柄，如果调用成功
- -1，如果调用不成功。

I.3.13 SYS_READ (0x06)

将文件内容读取到缓冲区。指定文件位置的两种方式：

- 显式，由 SYS_SEEK 指定
- 隐式，为之前 SYS_READ 或 SYS_WRITE 请求后的一个字节。

打开文件时，文件位置在其起始处；关闭文件时，文件位置丢失。只要有可能，就将文件操作作为单个操作执行。例如，不要将 16KB 的读操作块分为四个 4KB 的块，除非别无选择。

入口

进入时，r1 包含一个指向四字数据块的指针：

字 1 包含先前使用 SYS_OPEN 打开的文件的句柄。

字 2 指向缓冲区。

字 3 包含要从文件读取到缓冲区的字节数。

返回值

退出时：

- 如果调用成功，r0 包含零。
- 如果 r0 包含与字 3 相同的值，则调用失败并认为已到达文件末尾。
- 如果 r0 包含小于字 3 的值，则调用部分成功。认为没有错误，但是未填充缓冲区。

如果句柄用于交互设备，即 SYS_ISTTY 返回 -1。从 SYS_READ 返回一个非零值表示读行未填充缓冲区。

I.3.14 SYS_READC (0x07)

从控制台读取一个字节。

入口

寄存器 r1 必须包含零。不可能有其他参数或值。

返回值

退出时，r0 包含从控制台读取的字节。

I.3.15 SYS_REMOVE (0x0E)

小心

删除主机文件编排系统上的指定文件。

入口

进入时，r1 包含一个指向二字自变量块的指针：

字 1 指向空终止字符串，该字符串给出要删除的文件的路径名。

字 2 字符串的长度。

返回值

退出时，r0 包含：

- 0，如果删除成功
- 非零的主机特定错误代码，如果删除失败。

I.3.16 SYS_RENAME (0x0F)

重命名指定文件。

入口

进入时，r1 包含一个指向四字数据块的指针：

字 1 指向旧文件的名称的指针。

字 2 旧文件名的长度。

字 3 指向新文件名的指针。

字 4 新文件名的长度。

两个字符串都是空终止的。

返回值

退出时，r0 包含：

- 0，如果重命名成功
- 非零的主机特定错误代码，如果重命名失败。

I.3.17 SYS_SEEK (0x0A)

从文件起始处算起，使用指定的偏移量搜索到文件的指定位置。假定文件是字节数组，偏移量按照字节给出。

入口

进入时，r1 包含一个指向二字数据块的指针：

字 1 可搜索文件对象的句柄。

字 2 要搜索到的绝对字节位置。

返回值

退出时，r0 包含：

- 0，如果请求成功
- 负值，如果请求失败。SYS_ERRNO 可用于读取描述错误的主机 errno 变量的值。

注意

未定义在文件对象当前范围之外的搜索结果。

I.3.18 SYS_SYSTEM (0x12)

将命令传递给主机命令行解释程序。它可用于执行系统命令，如 dir、ls 或 pwd。终端 I/O 在主机上，对于目标来说是不可见的。

小心

传递给主机的命令在主机上执行。您需确保所有被传递的命令不出现意外结果。

入口

进入时，r1 包含一个指向二字自变量块的指针：

字 1 指向一个字符串，该字符串将被传递给主机命令行解释程序。

字 2 字符串的长度。

返回值

退出时，r0 包含返回状态。

I.3.19 SYS_TICKFREQ (0x31)

返回滴答声的频率。

入口

进入此例程时，寄存器 r1 必须包含零。

返回值

退出时，r0 包含以下值之一：

- 每秒的滴答声数目
- -1，如果目标不知道一个滴答声的值。RealView ICE 不支持此 SVC，并始终在 r0 中返回 -1。

I.3.20 SYS_TIME (0x11)

返回自 1970 年 1 月 1 日 00:00 到现在的秒数。这是真实世界的时间，与任何 RVISS、ISSM 或 RealView ICE 配置无关。

入口

没有参数。

返回值

退出时，r0 包含秒数。

I.3.21 SYS_TMPNAM (0x0D)

返回系统文件识别符所标识的文件的临时名称。

入口

进入时，r1 包含一个指向三字自变量块的指针：

字 1 指向缓冲区的指针。

字 2 此文件名的目标识别符。其值必须为 0 到 255 之间的一个整数。

字 3 包含缓冲区的长度。该长度必须至少等于主机系统上 L_tmpnam 的值。

返回值

退出时，r0 包含：

- 0，如果调用成功
- -1，如果出错。

r1 指向的缓冲区包含文件名，该文件名以相应的目录名为前缀。

如果再次使用相同的目标识别符，则返回相同文件名。

注意

返回的字符串必须是空终止的。

I.3.22 SYS_WRITE (0x05)

将缓冲区中的内容写入位于当前文件位置的指定文件。指定文件位置的两种方式：

- 显式，由 SYS_SEEK 指定
- 隐式，为之前 SYS_READ 或 SYS_WRITE 请求后的一个字节。

打开文件时，文件位置在其起始处；关闭文件时，文件位置丢失。

只要有可能，就将文件操作作为单个操作执行。例如，不要将 16KB 的写操作块分为四个 4KB 的块，除非别无选择。

入口

进入时，r1 包含一个指向三字数据块的指针：

字 1 包含先前使用 SYS_OPEN 打开的文件的句柄。

字 2 指向包含要写入的数据的内存。

字 3 包含要从缓冲区写入文件的字节数。

返回值

退出时，r0 包含：

- 0，如果调用成功
- 未写入的字节数，如果出错。

I.3.23 SYS_WRITEC (0x03)

将 r1 指向的字符字节写入调试通道。在 ARM 调试器下执行时，字符出现在主机调试器控制台上。

入口

进入时，r1 包含一个指向字符的指针。

返回值

无。寄存器 r0 被破坏。

I.3.24 SYS_WRITE0 (0x04)

将空终止的字符串写入调试通道。在 ARM 调试器下执行时，这些字符出现在主机调试器控制台上。

入口

进入时，r1 包含一个指向字符串第一个字节的指针。

返回值

无。寄存器 r0 被破坏。

I.4 调试代理交互 SVC

除了第A-7 页的*半主机操作*中介绍的 C 库半主机函数之外，以下 SVC 支持与调试代理之间进行交互：

- 第A-7 页的*angel_SWIreason_EnterSVC (0x17)*
- 第A-8 页的*angel_SWIreason_ReportException (0x18)*。