RealView® 编译工具

4.0 版

开发指南

ARM®

ARM DUI 0203IC

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	ARM® RealView® Developer Suite 2.0.1 版
2004 年 1 月	D	非保密	RealView Developer Suite 2.1 版
2004 年 12 月	E	非保密	RealView Developer Suite 2.2 版
2005 年 5 月	F	非保密	RealView Developer Suite 2.2 SP1 版
2006 年 3 月	G	非保密	RealView Development Suite 3.0 版
2007 年 3 月	H	非保密	RealView Development Suite 3.1 版
2008 年 9 月	I	非保密	RealView Development Suite 4.0 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。

受限访问是一种 ARM 内部分类。

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

前言

本前言介绍 ARM® 《RealView® 编译工具开发指南》。本章分为以下几节：

- 第viii 页的*关于本手册*
- 第xi 页的*反馈*

关于本手册

本手册包含可帮助开发适用于 ARM 系列处理器的代码的信息。本手册中的各章节和所用示例都假设在开发代码时使用最新版本的 ARM RealView 编译工具。

适用对象

本手册是为所有使用 RealView 编译工具生成应用程序的开发人员编写的。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》中介绍的 ARM 工具。

使用本手册

本手册由以下章节组成：

第 1 章 ***简介***

本章介绍 RealView 编译工具。

第 2 章 ***为 ARM 处理器开发代码***

本章介绍每个体系结构类型的重要功能，并指出在使用 RealView 编译工具时应注意的某些要点。

第 3 章 ***嵌入式软件开发***

本章介绍如何用 RealView 编译工具开发嵌入式应用程序。本章介绍在没有目标系统情况下的缺省 RealView 编译工具行为，以及如何调整 C 库和映像内存映射来适应目标系统。

第 4 章 ***混合使用 C、C++ 和汇编语言***

本章介绍如何为 ARM 体系结构编写 C、 C++ 和 ARM 汇编语言混合代码。还介绍如何在 C 和 C++ 文件中使用 ARM 指令内在函数、内联汇编器和嵌入式汇编器。

第 5 章 ***交互操作 ARM 和 Thumb***

本章介绍在为实现 Thumb 指令集的处理器编写代码时，如何在 ARM 状态和 Thumb® 状态之间切换。

第 6 章 ***处理处理器异常***

本章介绍如何处理 ARM 处理器所支持的各类异常。

第 7 章 ***调试通信通道***

本章介绍如何使用 *调试通信通道* (DCC)。

第 8 章 *半主机*

本章介绍半主机机制。通过半主机，运行在 ARM 目标上的代码可以使用运行 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 公司将定期对其文档进行更新和更正。有关最新勘误表、附录和 ARM 常见问题 (FAQ)，请访问 http://infocenter.arm.com/help/index.jsp。

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。该套件中包含的其他出版物有：

- 《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 链接器用户指南》 (ARM DUI 0206)
- 《RVCT 链接器参考指南》 (ARM DUI 0381)
- 《RVCT 实用程序指南》 (ARM DUI 0382)
- 《RVCT 汇编器指南》 (ARM DUI 0204)

有关基本标准、软件接口和 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：

- 《ARM 体系结构参考手册》， ARMv7-A 和 ARMv7-R 版 (ARM DDI 0406)
- 《ARMv7 体系结构参考手册》 (ARM DDI 0403)
- 《ARMv6-M 体系结构参考手册》 (ARM DDI 0419)
- 《ARM 体系结构参考手册》 (ARM DDI 0100)
- 您的硬件设备的 ARM 数据手册或技术参考手册

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》 (*ARM System Developer's Guide: Designing and Optimizing System Software*) （2004 年出版）。Morgan Kaufmann， ISBN 1-558-60874-5。

有关面向使用 ARM 处理器的片上系统设计人员和使用 ARM 体系结构的工程师的要点参考手册，请参阅 Steve Furber 编著的《ARM 片上系统体系结构》(*ARM system-on-chip architecture*) （2000 年的第二版）。Addison Wesley， ISBN 0-201-67519-6。

反馈

ARM Limited 欢迎提供有关 RealView 编译工具及其文档的反馈。

对 RealView 编译工具的反馈

如果您对 RealView 编译工具有任何问题，请与供应商联系。为便于供应商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
- 能说明问题的示例输出
- 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
- 您要对其发表意见的页码
- 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

本章介绍 ARM® RealView® 编译工具。

本章分为以下几节：

1.1 关于 RealView 编译工具

RealView 编译工具包含一组应用程序以及支持文档和示例，使用这些工具可以为 ARM 系列处理器编写应用程序。您可以使用 RealView 编译工具生成 C、C++ 和 ARM 汇编语言程序。

本手册介绍有助于开发 ARM 处理器代码的信息。本手册中的各章节和所用示例都假设，开发代码时使用的是最新版本的 RealView 编译工具。

如果要从旧版本升级为 RealView 编译工具，请务必阅读 《RealView 编译工具要点指南》，以了解此版本的新功能和增强功能的相关信息。

如果不熟悉 RealView 编译工具，请阅读 《RealView 编译工具要点指南》，这样可对 ARM 工具及其在开发项目中的使用方法有大致了解。

有关早期版本的 RealView 编译工具的信息，请参阅 《RealView 编译工具要点指南》中的附录 A。

有关 RealView 编译工具文档套件中介绍 ARM 汇编器、 ARM 编译器、 ARM 链接器和支持软件相关信息的其他手册，请参阅第 ix 页的*ARM 公司出版物*。

1.2 使用示例

本手册使用 RealView Development Suite 附带的示例。这些示例位于示例目录 *install_directory*\RVDS\Examples 中。有关这些示例的摘要，请参阅 《RealView Development Suite 入门指南》。

第 2 章
为 ARM 处理器开发代码

本章介绍每个体系结构版本的重要功能，指出在使用 ARM RealView 编译工具时应注意的某些要点。

本章分为以下几节：

2.1 关于 ARM 体系结构

本节概述各种 ARM 体系结构，以及在为特定处理器开发代码时应注意的相关功能。

ARM 体系结构支持 32 位 ARM 和 16 位 Thumb® 指令集体系结构以及体系结构扩展，以提供对*紧耦合存储器* (TCM)、内存管理、*单指令多数据* (SIMD) 和 NEON™ 技术的支持。

ARM 体系结构不断改进，以满足前沿应用程序开发人员日益增长的要求，同时保留了必要的向后兼容性，以保护软件开发投资。

有关详细信息，请参阅处理器的 《技术参考手册》 (*Technical Reference Manual*) 或 《ARM 体系结构参考手册》。

表 2-1 简要列出了 ARM 处理器的部分重要功能。

表 2-1 重要功能

处理器	体系结构	紧耦合存储器	内存管理	Thumb-2
ARM7TDMI®	ARMv4T	-	-	-
ARM920T™	ARMv4T	-	MMU	-
ARM922T™	ARMv4T	-	MMU	-
ARM926EJ-S™	ARMv5TEJ	支持	MMU	-
ARM946E-S™	ARMv5TE	支持	MPU	-
ARM966E-S™	ARMv5TE	支持	-	-
ARM11™ MPCore™	ARMv6K	-	MMU	-
ARM1136J-S™/ARM1136JF-S™	ARMv6K	支持	MMU	-
ARM1156T2-S™/ARM1156T2F-S™	ARMv6T2	亢峭	MPU	亢峭
ARM1176JZ-S™/ARM1176JZF-S™	ARMv6Z	支持	MMU	-
Cortex™-M1	ARMv6-M	支持	-	-
Cortex-A8	ARMv7-A	-	MMU	支持

表 2-1 重要功能 （续）

处理器	体系结构	紧耦合存储器	内存管理	Thumb-2
Cortex-A9	ARMv7-A	-	MMU	支持
Cortex-R4 和 Cortex-R4F	ARMv7-R	可变	MPU	支持
Cortex-M3	ARMv7-M	-	MMU （可选）	仅 Thumb-2

2.1.1 多重处理系统

ARM 体系结构 v6K 首次引入了对多达 4 个 CPU 及关联硬件的 MPCore 处理器支持。应用程序必须专为在多重处理系统上运行而设计，以便优化性能。例如，一个 CPU 可专用于单线程应用程序中的某个特定任务，也可用于多线程环境中的并行处理。高效的多重处理系统与单 CPU 系统相比，功耗更低、散热量更少，响应速度更快，但是也更加复杂，因此更难以调试。

设计多重处理系统时应考虑如下几点：

- 使用 LDREX/STREX 进行同步，以创建互斥量或信号量，从而保护重要代码段和不可共享的资源。
- 强制高速缓存一致性，以实现对称多重处理
- 在不同的线程中执行重复性任务
- 将大任务拆分为多个并行执行的线程
- 设置主 CPU，使用 CP15 CPU ID 寄存器执行初始化任务
- 确定中断优先级
- 将位掩码用于中断抢先
- 配置触发计时器或看门狗的周期计数

注意

这些任务通常是由操作系统处理的。

2.1.2 紧耦合存储器

TCM 是一段始终有效的连续内存区域 （如果启用了 TCM）。TCM 用作系统的物理内存映射的一部分，不必由物理地址相同的外部存储器来支持。因此，TCM 的行为与标记为直写可高速缓存的内存区域的高速缓存不同。在这类区域中，向 TCM 中的内存位置写入时，不会发生任何外部写入。

TCM 用于向处理器提供低延迟内存，它没有高速缓存特有的不可预测性。可以使用 TCM 来存放重要例程，如中断处理例程或者极需要避免高速缓存不确定性的实时任务。此外，可以使用 TCM 来保存暂时寄存器数据、局部属性不适合高速缓存的数据类型，以及中断堆栈等重要数据结构。

有关 TCM 的完整体系结构描述，请参阅 《ARM 体系结构参考手册》以及处理器的 《技术参考手册》 (*Technical Reference Manual*)。

2.1.3 内存管理

ARM 内存管理选项为：

MMU *内存管理单元* (MMU) 用于对内存系统进行细粒度控制。大多数细粒度控制是通过保存在内存中的转换表实现的。这些表中的条目定义不同内存区域的属性。其中包括：

- 虚拟地址到物理地址的映射
- 内存访问权限
- 内存类型

MPU *内存保护单元* (MPU) 提供了比 MMU 更为简单的替代方式。在不需要所有 MMU 功能的系统中，使用 MPU 可以简化硬件和软件。可以使用 MPU 将外部存储器分为多个大小和属性不相同的连续区域。还可以控制不同内存区域的访问权限和内存特性。

MPU 不需要外部存储器来存放转换表，在启用高速缓存之前，必须先启用 MPU。

有关 MMU 或 MPU 的完整体系结构描述，请参阅 《ARM 体系结构参考手册》和处理器的 《技术参考手册》 (*Technical Reference Manual*)。

2.1.4 Thumb-2

ARMv6T2 及更高版本的体系结构提供 Thumb-2 技术。Thumb-2 是对 16 位 Thumb 指令集的重要增强。它增加的 32 位指令可在程序中与 16 位指令任意混合使用。通过新增的 32 位指令， Thumb-2 可实现 ARM 指令集的全部功能。通过这些 32 位指令，Thumb-2 同时兼具早期版本 Thumb 的代码密度和 ARM 指令集的性能。

Thumb-2 指令集和 ARM 指令集之间最重要的区别是，大多数 32 位 Thumb 指令是无条件的，而大多数 ARM 指令是有条件的。Thumb-2 引入了条件执行指令 `IT`，该指令是逻辑 if-then-else 操作，可应用于后续指令以使其按条件执行。

有关该指令集的详细信息，请参阅 《ARM 体系结构参考手册》和处理器的《技术参考手册》 (*Technical Reference Manual*)。

2.1.5 浮点生成选项

以下准则有助于为应用程序选择最适合的浮点生成选项。

ARM 和 Thumb 浮点 （ARMv6 及更早版本）

对于以 ARM 状态代码和 Thumb 状态代码执行浮点运算的代码，有几个编译选项：

仅 ARM 选择选项 `--fpu vfpv2` 可使编译器仅为包含浮点运算的函数生成 ARM 代码。

如果选择选项 `--fpu vfpv2`，则不论是为 ARM 编译还是为 Thumb 编译，编译器都会为任何包含浮点运算的函数生成 ARM 代码。

因为 Thumb 代码不能包含 VFP 指令也不能访问 VFP 寄存器，所以包含浮点运算的函数以及为 Thumb 编译的函数会编译为 ARM 代码。这样会使用硬件 VFP 链接。

如果只进行 ARM 编译，应使用 `--fpu=vfp`，而不是 `--fpu=softvfp+vfp`。 软件链接会增加在 VFP 和 ARM 之间传输值的开销，导致传输速度降低，且需要更多的指令。

混合 ARM/Thumb

选择选项 `--fpu softvfp+vfpv2` 可使编译器生成混合的 ARM/Thumb 代码。

如果选择选项 `--fpu softvfp+vfpv2`，则所有函数均使用软件浮点链接进行编译。这意味着，传递给函数和从函数返回的浮点参数是保存在整数寄存器中的。

Thumb 指令集不包含 VFP 指令，无法访问 VFP 寄存器。因此，对于 Thumb 代码，如果使用 `--fpu=softvfp+vfpv2`，则编译器生成对库函数的调用来执行 VFP 操作。这些库函数必须使用软件链接，因为 Thumb 代码无法访问使用硬件链接所需的 VFP 寄存器。

RVCT 库包含为 ARM 编译的软件浮点函数版本，并使用将用于 `--fpu=softvfp+vfpv2` 的 VFP 指令。与完全软件浮点函数相比，这些库函数提高了性能，并减小了代码大小。

能实现最佳代码大小或性能的选项取决于所编译的代码。进行 ARM 编译时，最好试用选项 `--fpu softvfp+vfpv2` 和 `--fpu vfpv2`，以确定哪个选项可以实现所需的代码大小和性能属性。

如果代码中混合了 ARM 和 Thumb，您可能会希望尝试 `--fpu` 选项来获得最佳结果。

ARM 和 Thumb-2 浮点 （ARMv7、 RealView Development Suite v3.0 及更高版本）。

混合 ARM/Thumb-2

选择选项 `--fpu softvfp+vfpv3` 可使编译器生成混合 ARM/Thumb 代码。

如果选择选项 `--fpu softvfp+vfpv3`，则使用软件浮点链接编译所有函数。这意味着浮点参数传递到并返回自 ARM 整数寄存器中的函数。

借助软件浮点链接，可以与通过软件浮点链接自行构建的通用库和遗留代码进行链接。

仅 ARM 选择选项 `--arm --fpu vfpv3` 可使编译器只生成 ARM 代码。这样会使用硬件 VFP 链接。

仅 Thumb-2

选择 `--thumb --fpu vfpv3` 选项可使编译器为整个程序只生成 Thumb-2 代码。Thumb-2 支持 VFP 指令。因此，不需要切换到 ARM 状态来执行 VFP 操作。这样会使用硬件 VFP 链接。

注意

此选项仅适用于具有 VFPv3 的 ARMv7 处理器 （例如 Cortex-A8），在这类处理器中，可用 ARM 和 Thumb-2 指令集直接访问 VFP。

2.2 ARM 体系结构 v4T

本节概述 RealView 工具对 ARMv4T 的支持。ARM 体系结构的这一版本提供 16 位 Thumb 指令，即 32 位 ARM 指令集的一个子集。它同时支持 ARM 和 Thumb 指令集。

表 2-2 有用的命令行选项

命令行选项	说明
`--cpu=4T`	具有 Thumb 的 ARMv4。
`--cpu=`*name*	其中， *name* 是特定的 ARM 处理器。例如 `ARM7TDMI?`
`--apcs=`*qualifier*	其中 *qualifier* 表示一个或多个表示交互操作和位置无关性的限定符。 例如 `--apcs=/interwork`。

2.2.1 重要功能

在为 ARMv4T 编译代码时，编译器支持附加 Thumb 指令来实现更大的代码密度，但是有以下限制：

- 对于给定任务， Thumb 代码通常要使用更多指令。因此，为使对时间要求严格的代码达到最佳性能，最好使用 ARM 代码。
- 进行异常处理时，需要使用 ARM 状态和关联的 ARM 指令。

2.2.2 对齐支持

所有加载和存储指令必须指定在自然对齐边界上对齐的地址。例如：

- `LDR` 和 `STR` 地址必须在字边界上对齐
- `LDRH` 和 `STRH` 地址必须在半字边界上对齐
- `LDRB` 和 `STRB` 地址可与任意边界对齐

如果访问的地址不位于自然对齐边界上，则会导致无法预测的行为。若要对此加以控制，在需要访问未对齐地址时，必须使用 `__packed` 通知编译器，以便编译器生成安全的代码。请参阅 《编译器参考指南》中第4-11 页的*__packed*。

注意

未对齐访问 （如果允许）视为循环对齐访问。

2.2.3 端支持

使用编译器命令行选项 `--littleend` 和 `--bigend` 可以分别生成小端代码和大端代码。

ARMv4T 支持以下端模式：

LE 小端格式

BE-32 旧大端格式

2.3 ARM 体系结构 v5TE

本节概述 RealView 工具对 ARMv5TE 的支持。ARM 体系结构的这一版本对*数字信号处理* (DSP) 算法提供增强算法支持。它同时支持 ARM 和 Thumb 指令集。

表 2-3 有用的命令行选项

命令行选项	说明
`--cpu=5TE`	具有 Thumb、交互操作、 DSP 乘法以及双字指令的 ARMv5
`--cpu=5TEJ`	具有 Thumb、交互操作、 DSP 乘法、双字指令以及 Jazelle® 扩展[a] 的 ARMv5
`--cpu=`*name*	其中， *name* 是特定的 ARM 处理器。例如： • `ARM926EJ-S`，它基于具有 Thumb、Jazelle 扩展、物理映射高速缓存和 MMU 的 ARMv5。

a. ARM 编译器不能生成 Jazelle 字节代码。

2.3.1 重要功能

为 ARMv5TE 编译代码时，编译器：

- 支持 ARM 和 Thumb 之间的改进交互操作，例如 `BLX`。
- 为指定的处理器执行指令调度。指令经过重新排序，以尽量减少互锁并提高性能。
- 使用对 16 位数据项执行操作的乘法和乘法累加指令。
- 使用指令内在函数来生成执行饱和签名算法的加法和减法指令。如果计算结果溢出正常整数范围，则饱和算法将生成最大正数值或负数值，而不是绕回结果。
- 使用操作双字数据的加载 (`LDRD`) 和存储 (`STRD`) 指令。
- 使用预加载数据指令 `PLD`。

2.3.2 对齐支持

所有加载和存储指令必须指定在自然对齐边界上对齐的地址。例如：

- LDR 和 STR 地址必须在字边界上对齐
- LDRH 和 STRH 地址必须在半字边界上对齐
- LDRD 和 STRD 地址必须在双字边界上对齐
- LDRB 和 STRB 地址可与任意边界对齐。

如果访问的地址不位于自然对齐边界上，则会导致无法预测的行为。若要对此加以控制，在需要访问未对齐地址时，必须使用 __packed 通知编译器，以便编译器生成安全的代码。请参阅 《编译器参考指南》中第4-11 页的*__packed*。

所有 LDR 和 STR 指令 （除 LDRD 和 STRD 外）必须指定字对齐地址，否则指令会中止。

注意

未对齐访问 （如果允许）视为循环对齐访问。

另请参阅

- 处理器的 《技术参考手册》 (*Technical Reference Manual*)
- 《编译器用户指南》中第5-23 页的*对齐数据*
- 《编译器参考指南》中第2-113 页的*--[no_]unaligned_access*

2.3.3 端支持

使用编译器命令行选项 --littleend 和 --bigend 可以分别生成小端代码和大端代码。

ARMv5TE 支持以下端模式：

LE 小端格式

BE-32 旧大端格式

2.4 ARM 体系结构 v6

本节概述 RealView 工具对 ARMv6 的支持。ARM 体系结构的这一版本扩展了原始 ARM 指令集，可支持多重处理操作，并添加了一些额外的内存模型功能。它同时支持 ARM 和 Thumb 指令集。

表 2-4 有用的命令行选项

选项	说明
`--cpu=6`	具有 Thumb、交互操作、DSP 乘法、双字指令、未对齐和混合端支持、Jazelle 扩展以及多媒体扩展的 ARMv6
`--cpu=6Z`	具有安全扩展的 ARMv6
`--cpu=6T2`	具有 Thumb-2 的 ARMv6
`--cpu=`*name*	其中，*name* 是特定的 ARM 处理器。例如： • 如果是 `ARM1136J-S`，则通过软件 VFP 支持为 ARM1136J-S 生成代码 • 如果是 `ARM1136JF-S`，则通过硬件 VFP 为 ARM1136J-S 生成代码

2.4.1 重要功能

为 ARMv6 编译代码时，编译器：

- 为指定的处理器执行指令调度。指令经过重新排序，以尽量减少互锁并提高性能。
- 需要时生成显式 `SXTB`、`SXTH`、`UXTB`、`UXTH` 字节或半字扩展指令。
- 如果可以推断 C 表达式将执行端反转，则生成端反转指令 `REV`、`REV16` 和 `REVSH`。
- 生成 ARMv6 中可用的其他 Thumb 指令，例如 `CPS`、`CPY`、`REV`、`REV16`、`REVSH`、`SETEND`、`SXTB`、`SXTH`、`UXTB` 和 `UXTH`。
- 使用某些专为 ARMv6 优化的函数，例如 `memcpy()`。

编译器无法生成 SIMD 指令，因为这些指令未正确映射到 C 表达式。必须使用汇编语言或内在函数才能进行 SIMD 代码生成。

某些增强的指令可用于改善异常处理：

- `SRS` 和 `RFE` 指令，用于保存和恢复*链接寄存器* (LR) 和*保存的程序状态寄存器* (SPSR)

- CPS 简化了在*当前程序状态寄存器* (CPSR) 中更改状态以及修改 I 和 F 位的操作。
- 借助向量中断控制器为向量中断提供的体系结构支持
- 低延迟中断模式
- ARM1156T2-S 可在 Thumb 状态下使用 Thumb-2 代码进入异常。

2.4.2 对齐支持

缺省情况下，编译器使用 ARMv6 未对齐访问支持，通过允许 LDR 和 STR 指令对非自然字边界对齐的字进行加载和存储，来加快对压缩结构的访问。结构保持为非压缩，除非用 __packed 显式限定。表 2-5 演示在为 ARMv6 及更早版本的体系结构编译代码时的单字节对齐的效果。

表 2-5 单字节对齐

```
__packed struct
{
    int i;
    char ch;
    short sh;
} foo;
```

为 ARMv6 之前的版本编译代码：

```
MOV R4,R0
BL __aeabi_uread4
LDRB R1, [R4,#4]
LDRSB R2,[R4,#5]
LDRB R12,[R4,#6]
ORR R2,R12,R2 LSL#8
```

为 ARMv6 及更高版本编译代码：

```
LDR R0, [R4,#0]
LDRB  R1,[R4,#4]
LDRSH R2,[R4,#5]
```

仅当处理器启用未对齐数据访问支持时，为 ARMv6 编译的代码才能正确运行。可以使用 CP15 寄存器 c1 中的 U 和 A 位，或者对处理器的 **UBITINIT** 输入键入 HIGH 来控制对齐。

通过使用编译器选项 --no_unaligned_access 可以生成使用 ARMv6 之前版本的未对齐数据访问行为的代码。

注意

在 BE-32 端模式中，不能进行未对齐数据访问。

LDRD 和 STRD 可以为字对齐。

另请参阅

- 处理器的 《技术参考手册》 (*Technical Reference Manual*)
- 《编译器用户指南》中第5-23 页的*对齐数据*
- 《编译器参考指南》中第2-113 页的*--[no_]unaligned_access*

2.4.3 端支持

使用编译器命令行选项 `--littleend` 和 `--bigend` 可以分别生成小端代码和大端代码。

ARMv6 支持以下端模式：

LE 小端格式

BE8 大端格式

BE-32 旧大端格式

通过使用 `SETEND` 和 `REV` 指令，还可以实现混合端系统。

为 ARMv6 端模式 BE8 编译

缺省情况下，在为 ARMv6 进行大端编译时，编译器将生成 BE8 大端代码。编译器在代码中设置一个标记，将代码标记为 BE8。因此，若要在 ARM 处理器中启用 BE8 支持，通常必须在 CPSR 中设置 `E` 位。

将旧代码与 ARMv6 代码链接起来，可以在基于 ARMv6 的处理器上运行。不过，在这种情况下，链接器会将旧代码的字节顺序切换为 BE8 模式。得到的映像为 BE8 模式。

针对 ARMv6 旧端模式 BE32 编译

若要使用 ARMv6 之前的或旧的 BE32 模式，必须将 `BIGENDINIT` 输入绑定到处理器 `HIGH`，或者设置 CP15 寄存器 c1 的 `B` 位。

注意

必须使用链接器选项 `--be32` 链接 BE32 兼容代码。否则， ARMv6 属性会导致生成 BE8 映像。

有关详细信息，请参阅：

- 第2-12 页的*对齐支持*
- 《编译器参考指南》中第2-16 页的*--bigend*
- 《编译器参考指南》中第2-76 页的*--littleend*
- 《编译器参考指南》中第2-113 页的*--[no_]unaligned_access*
- 《链接器参考指南》中第2-4 页的*--be8*
- 《链接器参考指南》中第2-5 页的*--be32*

2.5 ARM 体系结构 v6-M

本节概述 RealView 工具对 ARMv6-M 的支持。通过寄存器硬件堆栈以及对使用高级语言写入中断处理程序的支持，微控制器架构实现了专为快速中断处理而设计的程序员模型。它用于需要将小型处理器集成到 FPGA 中的深度嵌入式应用程序，并支持 Thumb 指令集以及少量 32 位 Thumb-2 指令。

表 2-6 有用的命令行选项

命令行选项	说明
`--cpu=6-M`	只有 Thumb 的 ARMv6 微控制器架构，以及处理器状态指令
`--cpu=6S-M`	只有 Thumb 的 ARMv6 微控制器架构，以及处理器状态指令和操作系统扩展
`--cpu=`*name*	其中，*name* 是特定的 ARM 处理器。例如： • `Cortex-M1`，它基于只有 Thumb、外加处理器状态指令、操作系统扩展以及 BE8 和 LE 数据端标记支持的 ARMv6。

2.5.1 重要功能

ARMv6-M 的重要功能：

- 编译器支持使用 Thumb-2 技术的 Thumb 指令集的扩展。例如，`BL`、`DMB`、`DSB`、`ISB`、`MRS` 和 `MSR`。

2.5.2 对齐支持

缺省情况下，编译器使用 ARMv6 未对齐访问支持，通过允许 `LDR` 和 `STR` 指令对非自然字边界对齐的字进行加载和存储，来加快对压缩结构的访问。

未对齐数据访问会转换为两次或三次对齐访问，具体取决于未对齐访问的大小和对齐方式。这将暂停所有后续访问，直到未对齐访问完成为止。可以使用 DCode 和系统总线接口来控制对齐。

另请参阅

- *《Cortex-M1 技术参考手册》(Cortex-M1 Technical Reference Manual)*
- 《编译器用户指南》中第5-23 页的*对齐数据*
- 《编译器参考指南》中第2-113 页的*--[no_]unaligned_access*

2.5.3 端支持

使用编译器命令行选项 `--littleend` 和 `--bigend` 可以分别生成小端代码和大端代码。

ARMv6-M 支持以下端模式：

LE 小端格式

BE8 大端格式

2.6 ARM 体系结构 v7-A

本节概述 RealView 工具对 ARMv7-A 的支持。应用程序架构通过多模式和对基于 MMU 的虚拟内存系统体系结构的支持，实现传统 ARM 体系结构。这些架构同时支持 ARM 和 Thumb 指令集。

表 2-7 有用的命令行选项

命令行选项	说明
`--cpu=7`	只有 Thumb-2 且没有硬件除法器的 ARMv7[a]
`--cpu=7-A`	支持基于虚拟 MMU 的内存系统的 ARMv7 应用程序架构，具有 ARM、 Thumb、Thumb-2 和 Thumb-2EE 指令集、 NEON™ 支持以及 32 位 SIMD 支持
`--cpu=`*`name`*	其中， *name* 是特定的 ARM 处理器。例如： • `Cortex-A8`，它基于具有 ARM、 Thumb、 Thumb-2、硬件 VFP、 NEON 支持和 32 位 SIMD 支持的 ARMv7。

a. ARM v7 不是具体的 ARM 体系结构。它指的是所有 ARMv7-A、 ARMv7-R 和 ARMv7-M 体系结构都具有的功能。

2.6.1 重要功能

ARMv7-A 的重要功能：

- 支持高级 SIMD 扩展
- 支持 *Thumb 执行环境* (ThumbEE)

2.6.2 对齐支持

在 ARMv4 和 ARMv7 中， ARM 体系结构所支持的数据对齐行为有明显不同。ARMv7 实现必须支持未对齐数据访问。可以使用 CP15 寄存器 c1 中的 `A` 位控制加载和存储指令的对齐要求。

注意

ARMv7 体系结构不支持 ARMv6 之前版本的对齐。

另请参阅

- 处理器的 《技术参考手册》 (*Technical Reference Manual*)
- 《编译器用户指南》中第5-23 页的*对齐数据*
- 《编译器参考指南》中第2-113 页的`--[no_]unaligned_access`

2.6.3 端支持

使用编译器命令行选项 `--littleend` 和 `--bigend` 可以分别生成小端代码和大端代码。

ARMv7-A 支持以下端模式：

LE 小端格式

BE8 ARMv6 和 ARMv7 所用的大端格式

ARMv7 不支持旧 BE-32 模式。如果针对 ARMv7 处理器的旧代码包含大端字节顺序的指令，则必须执行字节顺序反转。请参阅 《ARM 体系结构参考手册》。

2.7 ARM 体系结构 v7-R

本节概述 RealView 工具对 ARMv7-R 的支持。实时架构通过多模式和对基于 MPU 的受保护内存系统体系结构的支持，实现传统 ARM 体系结构。ARMv7-R 体系结构同时支持 ARM 和 Thumb 指令集。

表 2-8 有用的命令行选项

命令行选项	说明
`--cpu=7`	只有 Thumb-2 但没有硬件除法器的 ARMv7[a]
`--cpu=7-R`	具有 ARM、 Thumb、 Thumb-2（可选）、 VFP、 32 位 SIMD 支持和硬件除法器的 ARMv7 实时架构
`--cpu=`*name*	其中， *name* 是特定的 ARM 处理器。例如： • `Cortex-R4F`，它基于具有 ARM、Thumb、Thumb-2、硬件 VFP、硬件除法器和 SIMD 支持的 ARMv7。

a. ARM v7 不是具体的 ARM 体系结构。它指的是所有 ARMv7-A、 ARMv7-R 和 ARMv7-M 体系结构都具有的功能。

2.7.1 重要功能

ARMv7-R 的重要功能：

- 支持 `SDIV` 和 `UDIV` 指令

2.7.2 对齐支持

在 ARMv4 和 ARMv7 中， ARM 体系结构所支持的数据对齐行为有明显不同。ARMv7 实现通过使用 `LDR`、`STR`、`LDRH` 和 `STRH` 提供对某些未对齐数据访问的硬件支持。其他数据访问必须使用 `LDM`、 `STM`、 `LDRD`、 `STRD`、 `LDC`、 `STC`、 `LDREX`、 `STREX` 和 `SWP` 保持对齐。

可以使用 CP15 寄存器 c1 中的 `A` 位控制加载和存储指令的对齐要求。

另请参阅

- 处理器的 《技术参考手册》 (*Technical Reference Manual*)
- 《编译器用户指南》中第5-23 页的*对齐数据*
- 《编译器参考指南》中第2-113 页的*--[no_]unaligned_access*

2.7.3 端支持

使用编译器命令行选项 `--littleend` 和 `--bigend` 可以分别生成小端代码和大端代码。

ARMv7-R 支持以下端模式：

LE 小端格式

BE8 大端格式

ARMv7 不支持旧 BE-32 模式。如果针对 ARM v7 处理器的旧代码包含大端字节顺序的指令，则必须执行字节顺序反转。

ARMv7-R 支持可选字节顺序反转硬件作为复位的静态选项。请参阅 《ARM 体系结构参考手册》， ARMv7-A 和 ARMv7-R 版。

2.8 ARM 体系结构 v7-M

本节概述 RealView 工具对 ARMv7-M 的支持。通过寄存器硬件堆栈以及对使用高级语言写入中断处理程序的支持，微控制器架构实现了专为快速中断处理而设计的程序员模型。它实现了一种 ARMv7 受保护内存系统体系结构，只支持 Thumb-2 指令集。

表 2-9 有用的命令行选项

命令行选项	说明
`--cpu=7`	仅具有 Thumb-2 且没有硬件除法器的 ARMv7[a]
`--cpu=7-M`	ARMv7 微控制器架构，只有 Thumb-2，并且带硬件除法器
`--cpu=`*name*	其中，*name* 是特定的 ARM 处理器。例如： • `Cortex-M3`，它基于只有 Thumb-2，具有硬件除法器、 ARMv6 样式的 BE8 和 LE 数据端标记支持以及未对齐访问的 ARMv7。

a. ARM v7 不是具体的 ARM 体系结构。它指的是所有 ARMv7-A、 ARMv7-R 和 ARMv7-M 体系结构都具有的功能。

2.8.1 重要功能

ARMv7-M 的重要功能：

- 支持 `SDIV` 和 `UDIV` 指令。
- 使用中断内在函数生成用于更改当前抢先优先级的 `CPSIE` 或 `CPSID` 指令（请参阅表 2-10）。例如，在使用 `__disable_irq` 内在函数时，编译器会生成 `CPSID i` 指令，该指令将 `PRIMASK` 设置为 `1`。这将把执行优先级提升为 `0`，并阻止具有可配置优先级的异常进入。请参阅 《ARMv7-M 体系结构参考手册》。

表 2-10 中断内在函数

内在函数	操作码	PRIMASK	FAULTMASK
`__enable_irq`	`CPSIE i`	0	
`__disable_irq`	`CPSID i`	1	
`__enable_fiq`	`CPSIE f`		0
`__disable_fiq`	`CPSID f`		1

2.8.2 对齐支持

在 ARMv4 和 ARMv7 中， ARM 体系结构所支持的数据对齐行为有明显不同。ARMv7 实现必须支持未对齐数据访问。可以使用 CP15 寄存器 c1 中的 A 位控制加载和存储指令的对齐要求。

注意

ARMv7 体系结构不支持 ARMv6 之前版本的对齐。

2.8.3 端支持

使用编译器命令行选项 `--littleend` 和 `--bigend` 可以分别生成小端代码和大端代码。

ARMv7-M 支持以下端模式：

LE 小端格式

BE8 大端格式

ARMv7 体系结构不支持旧 BE-32 模式。如果针对 ARM v7 处理器的旧代码包含大端字节顺序的指令，则必须执行字节顺序反转。请参阅 《ARM 体系结构参考手册》。

第 3 章
嵌入式软件开发

本章介绍在有或没有目标系统的情况下，如何使用 ARM RealView 编译工具开发嵌入式应用程序。

本章分为以下几节：

3.1 关于嵌入式软件开发

大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中的资源是有差异的。因此，很有必要考虑将嵌入式应用程序从依赖于开发工具或调试环境的系统移植到在目标硬件上独立运行的系统所涉及的过程。

使用 RealView 编译工具开发嵌入式软件时，必须考虑以下几个方面：

- 了解编译工具的缺省行为，以便理解从缺省生成转为完全独立的应用程序的必要步骤。
- 某些 C 库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。
- RealView 编译工具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。除了 RealView 编译工具 C 库初始化例程外，完整的初始化序列还需要您实现的代码。

3.1.1 示例代码

为了演示本章中所述的主题，示例目录 `...\RVDS\Examples\...\emb_sw_dev\` 中提供了关联示例项目。每个生成都对应一个单独的目录，提供了本章后面各节所述方法的示例。`readme.txt` 文件提供了有关每个生成的特定信息。

生成 1 生成 1 是 Dhrystone 基准程序的缺省生成，遵循缺省 RealView 编译工具行为。

有关详细信息，请参阅第3-4 页的*缺省编译工具行为*。

生成 2 此示例对生成 1 进行了修改，将 Versatile 板用于时钟计时和字符串 I/O。

有关详细信息，请参阅第3-9 页的*根据目标硬件调整 C 库*。

生成 3 此示例实现了一个分散加载描述文件，以调整堆栈和堆的放置。

有关详细信息，请参阅第3-11 页的*根据目标硬件调整映像内存映射*。

生成 4 此示例可以在 Versatile 板上独立运行。实现了一个向量表和复位处理程序。有关详细信息，请参阅第3-16 页的*复位和初始化*。

生成 5 此示例与生成 4 等效，但所有目标内存映射信息都在分散加载描述文件中。

有关详细信息，请参阅第3-22 页的*目标硬件和内存映射*。

Dhrystone 基准程序为示例项目提供代码基准。这些示例进行了调整，以便在 Versatile 板上运行。但是，原则可适用于任何目标硬件。有关板连接和设置的详细信息，请参阅板的《用户指南》(*User Guide*) 中的“入门”(*Getting Started*) 一节。

——注意——

本章不专门针对 Dhrystone 程序进行重点介绍，而是主要介绍要使该程序在完全独立的系统上运行所需要采取的步骤。有关使用 Dhrystone 作为基准工具的其他信息，请参阅 《应用程序说明 93：以 ARMulator® 为基准程序》 (Application Note 93 - *Benchmarking with ARMulator®*)。在 ARM 网站 `http://www.arm.com` 的“文档” (Documentation) 区域可以找到 ARM 应用程序说明。

3.2 缺省编译工具行为

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范。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。

为了让您在不了解这些详细信息之前就能继续软件开发，编译工具提供了缺省行为，使您可以立即开始生成和调试应用程序代码。了解这种缺省行为非常有用，这样您会理解从缺省生成转为完全独立的应用程序的必要步骤。

在 ARM C 库中，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提供的。提供此功能的机制称为*半主机*。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。

有关详细信息，请参阅第 8 章 *半主机*。

3.2.1 C 库结构

在概念上，C 库可划分为两类函数，一类是 ISO C 标准函数，另一类为 ISO C 标准提供支持。

例如，第3-5 页的图 3-1 演示通过写入调试器控制台窗口来实现函数 `printf()` 的 C 库。这种实现是通过调用 `_sys_write()`（执行半主机调用的支持函数）来提供的，这会导致使用调试器 （而不是目标外围设备）的缺省行为。

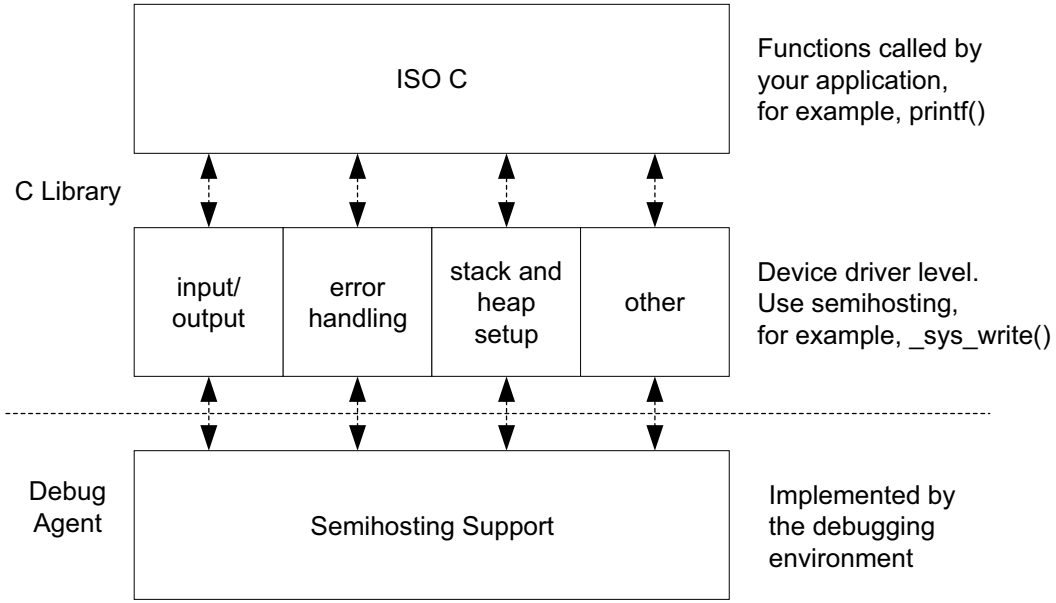

图 3-1 C 库结构

3.2.2 缺省内存映射

对于没有描述内存映射的映像，链接器根据缺省内存映射放置代码和数据，如第3-6 页的图 3-2 所示。

注意

基于 ARMv6-M 和 ARMv7-M 体系结构的处理器具有固定内存映射。这样，在基于这些处理器的不同系统之间移植软件会更加方便。有关详细信息，请参阅《Cortex-M1 技术参考手册》 (*Cortex-M1 Technical Reference Manual*) 和《Cortex-M3 技术参考手册》 (*Cortex-M3 Technical Reference Manual*)。

下面介绍缺省内存映射：

- 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。

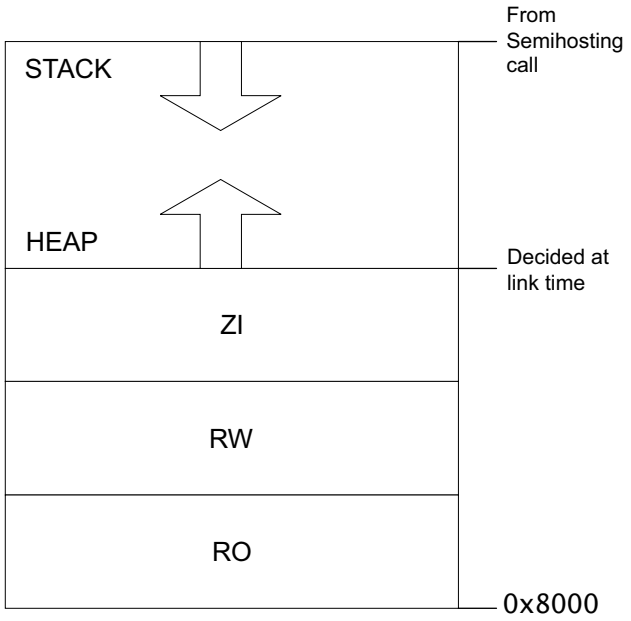

图 3-2 缺省内存映射

在确定将代码和数据置于内存中什么位置时，链接器遵守一组规则，如第3-7 页的图 3-3 所示。通常，链接器依次按属性、按名称、按输入列表中的位置对输入节进行排序。有关详细信息，请参阅 《链接器用户指南》中第3-2 页的*指定映像结构*和第3-7 页的 *节放置*。

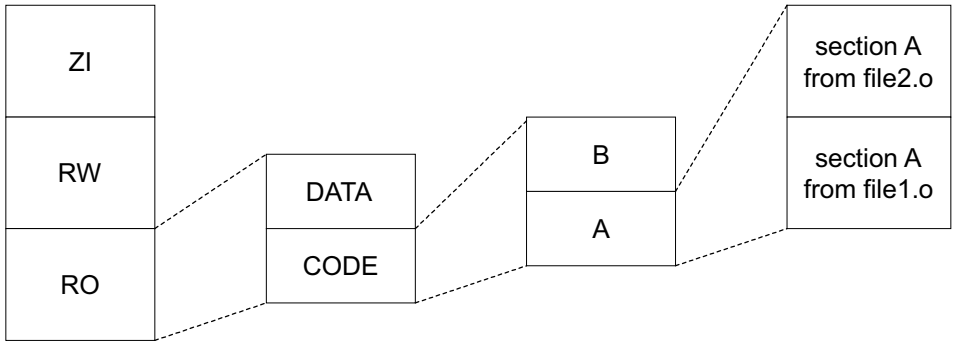

图 3-3 链接器放置规则

若要完全控制代码和数据的放置，必须使用分散加载机制。有关详细信息，请参阅第3-11 页的*根据目标硬件调整映像内存映射*。

3.2.3 应用程序启动

在大多数嵌入式系统中，都会在执行主任务前，通过执行初始化序列来设置系统。第3-8 页的图 3-4 演示缺省 RealView 编译工具初始化序列。

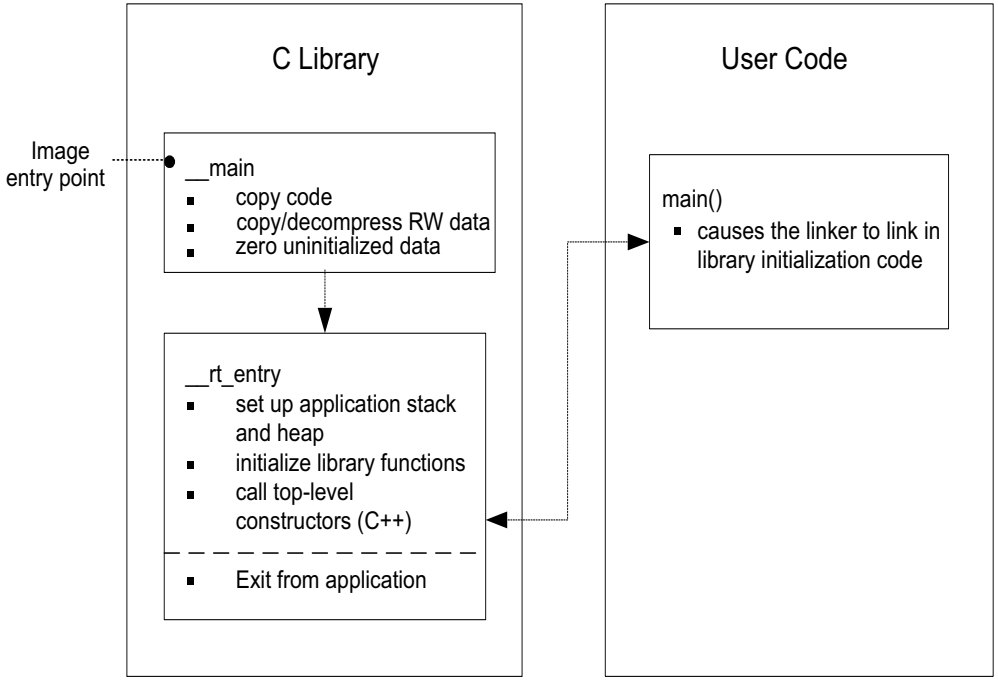

图 3-4 缺省初始化序列

__main 负责设置内存，而 __rt_entry 负责设置运行时环境。

__main 执行代码和数据复制、解压缩以及 ZI 数据的零初始化。然后，它跳转到 __rt_entry，设置堆栈和堆、初始化库函数和静态数据，并调用任何顶级 C++ 构造函数。然后，__rt_entry 跳转到应用程序的入口 main()。主应用程序结束执行后，__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，则没有链接到初始化序列，因而部分标准 C 库功能得不到支持。有关将备选 C 库与 __main 之外的启动符号一起使用的详细信息，请参阅 《链接器参考指南》中第2-55 页的*--[no_]startup=symbol*。

3.3 根据目标硬件调整 C 库

缺省情况下， C 库使用半主机来提供设备驱动程序级功能，使主机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。

您可以自己实现 C 库函数来使用目标硬件并自动链接到支持 C 库实现的映像。这一过程称为重定向 C 库目标，如图 3-5 所示。

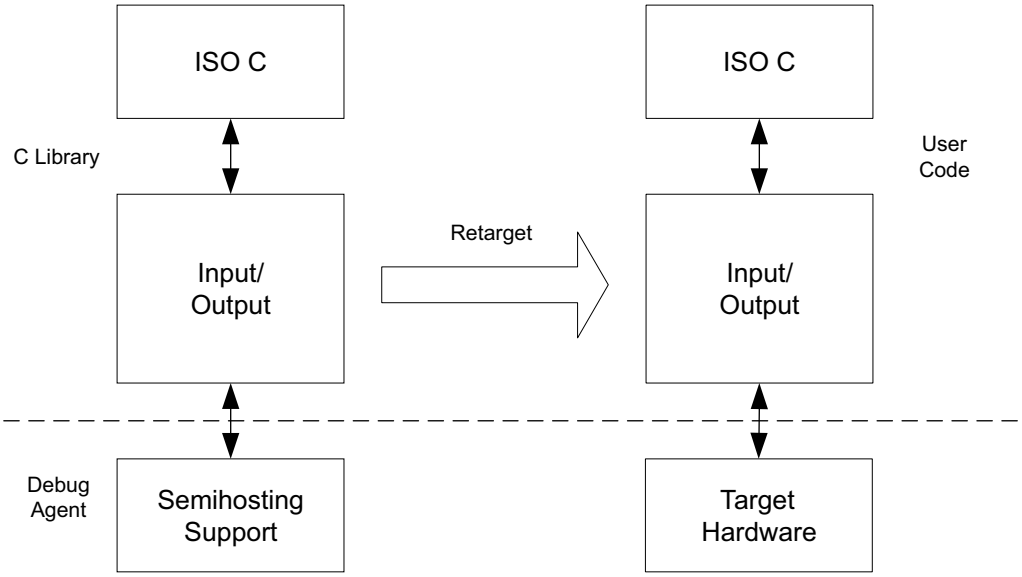

图 3-5 重定向 C 库的目标

例如，您有一个外围 I/O 设备（如 LCD 屏幕），希望重写 `fputc()` 的库实现，即从原来的写入调试器控制台变为输出到 LCD。因为 `fputc()` 的实现与最终映像链接，所以整个 `printf()` 系列函数都输出到 LCD。

第3-10 页的示例 3-1 演示 `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()`（假定该函数是在一个单独的源文件中实现的）。这样， `fputc()` 充当目标相关输出函数和 C 库标准输出函数之间的抽象层。

示例 3-1 fputc() 的实现

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

在独立应用程序中，不太可能支持半主机操作。因此，必须删除对半主机函数的所有调用，或者使用非半主机函数重新实现这些调用。有关详细信息，请参阅 《库和浮点支持指南》中第2-19 页的*构建用于非半主机环境的应用程序*。

有关使用半主机的 C 库函数的完整列表，请参阅第 8 章 *半主机*。

3.4 根据目标硬件调整映像内存映射

在最终嵌入式系统中，如果没有半主机功能，则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，必须在加载和运行时使用不同的内存视图。

通过分散加载功能，可以在一个文本描述文件（称为“分散加载描述文件”）中描述代码和数据在加载和运行时的内存位置。在命令行使用 --scatter 选项可将此文件传递给链接器。例如：

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

分散加载定义两个内存区类型：

- 加载区，包含复位和加载时的应用程序代码和数据。
- 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，会从每个加载区创建一个或多个执行区。

单个代码或数据节只能放置在单个执行区中。不能进行拆分。

在启动过程中，__main 中的 C 库初始化代码执行必要的代码/数据复制和数据清零，以从映像加载视图转为执行视图。

3.4.1 分散加载描述文件

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 3-6 演示这种文件语法。

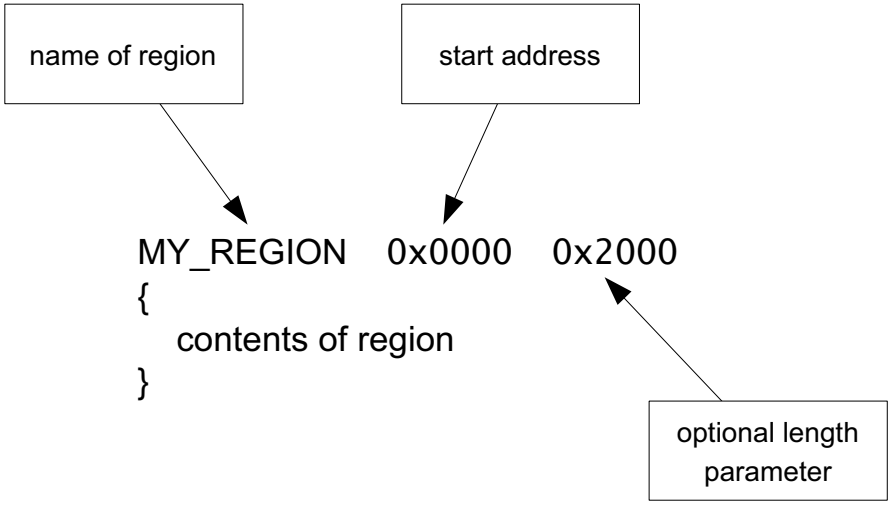

图 3-6 分散加载描述文件语法

区域由至少包含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。
- 执行区必须至少包含一个代码或数据节，由 EMPTY 属性声明的区域除外。非 EMPTY 区域通常包含对象或库代码。使用通配符 (*) 语法可以对未在分散加载描述文件中其他位置指定的、具有给定属性的所有节进行分组。

有关不同内存映射的更多示例和详细信息，请参阅 《链接器用户指南》中第5-5 页的*具有简单内存映射的映像*。

有关形式语法的详细信息，请参阅 《链接器参考指南》中的第 3 章 *分散加载描述文件的形式语法*。

3.4.2 根区

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

分散加载受到的一个限制是负责创建执行区的代码和数据不能将自身复制到另一位置。因此，根区中必须包含以下几节：

- 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`
- `Region$$Table` 节，它包含要复制或压缩的代码和数据的地址。

因为这些节定义为只读，是用 `* (+RO)` 通配符语法进行分组的。因此，如果在非根区中指定 `* (+RO)`，则必须使用 `InRoot$$Sections` 在根区中显式声明这些节。

有关详细信息，请参阅 《链接器用户指南》中第5-25 页的*为根区分配节*。

3.4.3 放置堆栈和堆

分散加载机制提供了一种方法，用于指定如何在映像中放置代码和静态分配数据。应用程序的堆栈和堆是在 C 库初始化过程中设置的。通过使用特别命名的 `ARM_LIB_HEAP`、`ARM_LIB_STACK` 或 `ARM_LIB_STACKHEAP` 执行区，可以调整堆栈和堆的放置。此外，如果不使用分散加载描述文件，则可以重新实现 `__user_initial_stackheap()` 函数。

有关详细信息，请参阅 《链接器用户指南》中第5-3 页的*使用分散加载描述文件指定堆栈和堆*。

运行时内存模型

RealView 编译工具提供以下运行时内存模型：

单区模型

应用程序的堆栈和堆在同一内存区中往对方的方向增长。请参阅第3-14 页的图 3-7。在此运行时内存模型中，分配新的堆空间时（例如调用 `malloc()`），根据堆栈指针的值对堆进行检查。

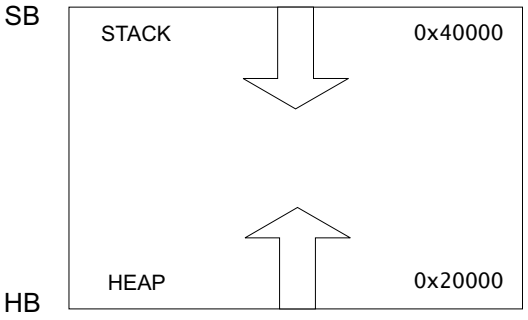

图 3-7 单区模型

示例 3-2 单区模型例程

```
LOAD_FLASH ...
{
    ...
    ARM_LIB_STACKHEAP 0x20000 EMPTY 0x20000  ; Heap and stack growing towards
    { }                                      ; each other in the same region
    ...
}
```

双区模型

堆栈和堆放置在不同的内存区中。例如，您可能有一小块高速 RAM，希望只供堆栈使用。对于双区模型，必须导入 `__use_two_region_memory`。

在此运行时内存模型中，分配新的堆空间时，根据堆限制对堆进行检查。

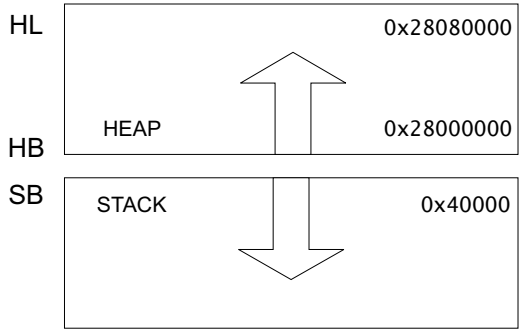

图 3-8 双区模型

示例 3-3 双区模型例程

```
LOAD_FLASH ...
{
    ...
    ARM_LIB_STACK 0x40000 EMPTY -0x20000  ; Stack region growing down
    { }                                   ;
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
    { }
    ...
}
```

在两种运行时内存模型中，都不对堆栈的增长进行检查。

有关详细信息，请参阅 《库和浮点支持指南》中第2-67 页的*调整运行时内存模型*。

3.5 复位和初始化

到目前为止，本章假设是从 C 库初始化例程的入口点 __main 开始执行的。事实上，目标硬件上的任何嵌入式应用程序在启动时都会执行一些系统级初始化。本节将对此进行详细介绍。

图 3-9 演示一个基于 ARM 体系结构的嵌入式系统的初始化序列。如果使用分散加载描述文件调整堆栈和堆放置，则链接器创建 __user_initial_stackheap() 函数，并使用链接器定义的符号作为这些区域的名称。有关详细信息，请参阅《链接器用户指南》中第5-3 页的*使用分散加载描述文件指定堆栈和堆*。或者，您也可以创建自己的实现。

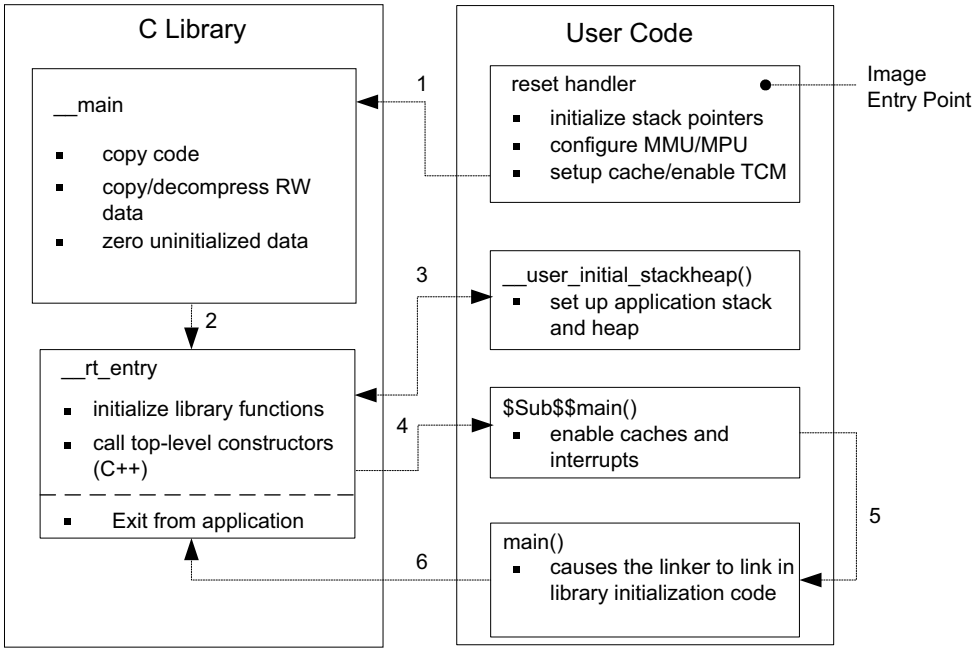

图 3-9 初始化序列

复位处理程序是在汇编器中编写的短模块，系统一启动就立即执行。复位处理程序最少要为应用程序的运行模式初始化堆栈指针。对于具有本地内存系统（如缓存、 TCM、 MMU 和 MPU）的处理器，某些配置必须在初始化过程的这一阶段完成。复位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。标有 $Sub$$main() 的代码块紧接在主应用程序开始执行之前执行这些任务。有关详细信息，请参阅《链接器用户指南》中第4-14 页的*使用 $Super$$ 和 $Sub$$ 覆盖符号定义*。

3.5.1 向量表

所有 ARM 系统都有一个向量表。向量表不是初始化序列的组成部分，但是，要处理任何异常都必须使用向量表。向量表必须放置于特定地址，通常为 0x0。为此，可以使用分散加载 +FIRST 指令，请参阅示例 3-4。

示例 3-4 将向量表放置于特定地址

```
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
  }
}
```

微控制器架构的向量表与大多数 ARM 体系结构截然不同。有关处理器的向量表的示例，请参阅：

- 第6-4 页的*向量表*，该向量表适用于 ARMv6 及更早版本以及 ARMv7-A 和 ARMv7-R 架构
- 第6-30 页的*向量表*，该向量表适用于 ARMv6-M 和 ARMv7-M 架构。

3.5.2 ROM 和 RAM 重映射

注意

本节内容不适用于 ARMv6-M 和 ARMv7-M 架构。

您必须考虑在地址 0x0 （执行第一条指令的地址）处的系统内存是什么类型。

注意

本节假设 ARM 处理器在 0x0 处开始取指令。这是基于 ARM 处理器的系统的标准。但是，某些 ARM 处理器可配置为从 0xFFFF0000 处开始取指令。

启动时， 0x0 处必须有一条有效指令，因此在复位时， 0x0 处必须是非易失性的存储器。一种实现方法是使 0x0 在 ROM 中。但是，这样配置有几个缺点。

示例 3-5 演示另一种在复位时实现 ROM/RAM 重映射的方法。此示例中所示的常量是特定于 Versatile 板的，但该方法适用于通过类似方法实现重映射的任何平台。分散加载描述文件必须描述重映射后的内存映射。

示例 3-5 ROM/RAM 重映射

```
; --- System memory locations
Versatile_ctl_reg     EQU 0x101E0000 ; Address of control register
DEVCHIP_Remap_bit     EQU 0x100      ; Bit 8 is remap bit of control register
    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 control register
; Clear the DEVCHIP_Remap_bit by writing 1 to bit 8 of the control register
        LDR     R1, =Versatile_ctl_reg
        LDR     R0, [R1]
        ORR     R0, R0, #DEVCHIP_Remap_bit
        STR     R0, [R1]
; RAM is now at 0x0.
; The exception vectors must be copied from ROM to RAM
; The copying is done later by the C library code inside __main
; Reset_Handler follows on from here
```

3.5.3 本地存储器设置的注意事项

很多 ARM 处理器都具有片上内存管理系统，如 MMU 或 MPU。这些设备通常是在系统启动过程中进行设置和启用的。因此，需要对带有本地内存系统的处理器的初始化序列特别注意。

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，必须设置处理器的运行时内存视图。也就是说，必须在复位处理程序中设置并启用所有 MMU 或 MPU。

在跳转到 __main 之前 （通常在 MPU/MPU 设置之前），还必须启用 TCM，因为通常情况下需要将代码和数据分散加载到 TCM 中。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。

3.5.4 堆栈指针初始化

复位处理程序至少必须为应用程序所使用的所有执行模式的堆栈指针分配初始值。

在示例 3-6 中，堆栈位于 stack_base 处。此符号可以是一个硬编码地址，也可以在单独的汇编器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 《链接器用户指南》中第5-3 页的*使用分散加载描述文件指定堆栈和堆*。

示例 3-6 初始化堆栈指针

```
; *****************************************************************
; This example does not apply to ARMv6-M and ARMv7-M profiles
; *****************************************************************
Len_FIQ_Stack    EQU     256
Len_IRQ_Stack    EQU     256
stack_base      DCD      0x18000
;
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    ; Interrupts disabled
MOV     sp, R0
SUB     R0, R0, #Len_FIQ_Stack
MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit    ; Interrupts disabled
MOV     sp, R0
SUB     R0, R0, #Len_IRQ_Stack
MSR     CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit    ; Interrupts disabled
MOV     sp, R0
; Leave processor in SVC mode
```

第3-19 页的示例 3-6 为 FIQ 和 *中断请求* (IRQ) 模式分配了 256 字节的堆栈，您也可对其他执行模式执行该操作。若要设置堆栈指针，请在中断禁用的情况下进入每种模式，然后为堆栈指针分配相应的值。

复位处理程序中设置的堆栈指针值由 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。

3.5.5 硬件初始化

注意

本节内容不适用于 ARMv6-M 和 ARMv7-M 架构。

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。

利用 `$Sub` 和 `$Super` 函数包装符可以插入一个紧接主应用程序之前执行的例程。这一机制使您能在不改变源代码的情况下扩展函数。

第3-21 页的示例 3-7 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 《链接器用户指南》中第4-14 页的 *使用 $Super$$ 和 $Sub$$ 覆盖符号定义*。

示例 3-7 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()
}
```

3.5.6 执行模式的注意事项

注意

本节内容不适用于 ARMv6-M 和 ARMv7-M 架构。

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。

您可能希望在启动时实现 （在复位处理程序和 `$Sub$$main` 中）的许多功能，只有在特权模式下执行才能完成，例如，片上内存操作和启用中断。

如果您想在特权模式下运行应用程序，这不成问题。请确保在退出复位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，必须退出在系统模式下使用用户模式寄存器的复位应用程序。然后，`__user_initial_stackheap()` 在系统模式下执行，因而在进入用户模式时，应用程序的堆栈和堆仍然存在。

3.6 目标硬件和内存映射

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。

按惯例，外围寄存器的地址在项目源文件或头文件中进行硬编码。也可声明映射到外围寄存器的结构，并将这些结构放置在分散加载描述文件中。

例如，目标可以有一个具有两个内存映射 32 位寄存器作为外围设备的计时器。示例 3-8 演示映射到这些寄存器的 C 结构。

示例 3-8 映射到外设寄存器

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

若要将此结构放置在内存映射中的特定地址上，可以创建一个执行区，其中包含定义该结构的模块。示例 3-9 演示一个名为 TIMER 的执行区，该执行区将 timer_regs 结构放置在 0x40000000 处。

应用程序启动过程中不对这些寄存器的内容进行零初始化是非常重要的，因为这样可能改变系统的状态。使用 UNINIT 属性对执行区进行标记可避免 __main 对该区域中的 ZI 数据进行零初始化。

示例 3-9 放置映射结构

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

第 4 章
混合使用 C、 C++ 和汇编语言

本章介绍如何为 ARM® 体系结构编写 C、 C++ 和汇编语言的混合代码。还介绍如何在 C 和 C++ 文件中使用 ARM 指令内在函数、内联汇编器和嵌入式汇编器。

本章分为以下几节：

4.1 使用指令内在函数、内联汇编器和嵌入式汇编器

指令内在函数、内联汇编器和嵌入式汇编器内置在 ARM 编译器中，通过它们，可以使用通常无法直接从 C 或 C++ 访问的目标处理器功能。例如：

- 饱和算法
- 自定义协处理器
- *程序状态寄存器* (PSR)

指令内在函数

指令内在函数提供了一种简便的手段，用于在 C 和 C++ 源代码中采用目标处理器功能，而不必借助复杂的汇编语言实现。指令内在函数具有 C 或 C++ 函数调用形式，不过在编译过程中，会由汇编语言指令替换。

注意

指令内在函数是 ARM 指令集特有的，因而不能移植到其他体系结构。

内联汇编器

内联汇编器支持与 C 和 C++ 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。内联汇编器还扩展复杂指令，优化汇编语言代码。

注意

输出对象代码可能因编译器优化而与输入不完全对应。

嵌入式汇编器

通过嵌入式汇编器，可以使用完整的 ARM 汇编器指令集，包括汇编器指令。嵌入式汇编代码与 C 或 C++ 代码分开进行汇编。生成编译的对象代码，然后与 C 或 C++ 源代码编译的对象代码相结合。

表 4-1 总结了指令内在函数、内联汇编器和嵌入式汇编器之间的主要差异。

表 4-1 差异

功能	指令内在函数	内联汇编器	嵌入式汇编器
指令集	ARM 和 Thumb®。	仅 ARM。	ARM 和 Thumb。
ARM 汇编器指令	不支持。	不支持。	全部支持。
C/C++ 表达式	完整 C/C++ 表达式。	完整 C/C++ 表达式。	仅常数表达式。
优化汇编代码	全部优化。	全部优化。	不优化。
内联	自动内联。	自动内联。	如果大小合适并启用了链接器内联，可由链接器进行内联。
寄存器访问	物理寄存器，包括 PC、LR 和 SP。	除 PC、LR 和 SP 之外的虚拟寄存器。	物理寄存器，包括 PC、LR 和 SP。
返回指令	自动生成。	自动生成。不支持 BX、BXJ 和 BLX 指令。	必须将其添加到代码中。
BKPT 指令	支持。	不支持。	支持。

有关详细信息，请参阅：

- 《编译器用户指南》中第4-2 页的*内在函数*
- 《编译器参考指南》中第4-66 页的*指令内在函数*
- 《编译器用户指南》中第 7 章 *使用嵌入式汇编器和嵌入式汇编*
- 《汇编器指南》中第4-89 页的*饱和指令*

4.2 在汇编代码中访问 C 全局变量

全局变量只能通过地址间接访问。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。根据变量的类型，可以使用加载和存储指令来访问全局变量。

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。

使用 LDM 和 STM 指令可以将少于 8 个字的小型结构作为整体进行访问。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 4-1 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 4-1 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
    LDR  R0, [R1]       ; load value of globvar
    ADD  R0, R0, #2
    STR  R0, [R1]       ; store new value into globvar
    BX   lr
    END
```

有关可在 ARM 或 Thumb 代码中使用的指令的信息，请参阅 《汇编器指南》中的第 4 章 *ARM 和 Thumb 指令*。

4.3 在 C++ 中使用 C 头文件

若要在 C++ 代码中包含 C 头文件，必须将这些 C 头文件包含在 `extern "C"` 指令中。

4.3.1 包含系统 C 头文件

标准系统 C 头文件已包含相应的 `extern "C"` 指令，因此不必采取特殊步骤来包含这些文件。不同的 #include 语法确定要使用的命名空间，从而确定可以访问的类型。

例如：

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

如果使用此语法包含头文件，则所有库名称都包含在全局命名空间中。

C++ 标准规定，通过 C++ 的特定头文件可以使用 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\...\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`。

4.3.2 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 4-2 中所示
- 将 extern "C" 语句添加到头文件中，如示例 4-3 中所示

示例 4-2 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 4-3 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. */
```

4.4 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 *ARM 体系结构的过程调用标准* (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 *install_directory*\Documentation\Specifications\... 中的 AAPCS 规范 aapcs.pdf。

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。

4.4.1 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》。

通过嵌入式汇编器并且遵循 《ARM 体系结构的基本标准应用程序二进制接口》(*Base Standard Application Binary Interface for the ARM Architecture*) (BSABI)，能更方便地实现混合语言编程。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
- 调用虚函数的方式
- 引用的表示形式
- 具有基类或虚成员函数的 C++ 类类型的布局
- 非 *普通旧式数据结构* (PODS) 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"，以指定这些函数具有 C 链接。在本 RealView® 编译工具版本中，具有 C 链接意味着定义函数的符号未进行重整。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。使用嵌入式汇编器的 `__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。

4.4.2 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** 形参作为第一个实参，或者作为第二个实参 （如果被调用函数返回不非 **int** 数据 （如 **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++ 对象没有基类或虚函数，则这些对象的布局与 ARM C 中所需的布局相同。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，则是普通旧式数据结构。
- 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4-5 页的*在 C++ 中使用 C 头文件*。

4.4.3 语言交叉调用的示例

下列各节包含一些代码示例，说明如何混合调用语言：

- 第4-10 页的*在 C 中调用汇编语言*
- 第4-11 页的*在汇编语言中调用 C*
- 第4-12 页的*在 C++ 中调用 C*
- 第4-13 页的*在 C++ 中调用汇编语言*
- 第4-14 页的*在 C 中调用 C++*
- 第4-15 页的*在汇编语言中调用 C++*
- 第4-17 页的*在 C 或汇编语言中调用 C++*
- 第4-16 页的*在 C 和 C++ 之间传递引用*

在 C 中调用汇编语言

示例 4-4 和示例 4-5 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 4-4 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' re 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 4-5 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 null terminator.
    BNE  strcopy      ; Keep going if not.
    BX   lr           ; Return.
    END
```

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

请按以下步骤在命令行生成该示例：

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 链接对象代码。

4. 通过对相应调试目标使用兼容调试器来运行映像。

在汇编语言中调用 C

示例 4-6 和示例 4-7 演示如何在汇编语言中调用 C。

示例 4-6 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 4-7 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 4-8 和示例 4-9 演示如何在 C++ 中调用 C。

示例 4-8 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 4-9 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 4-10 和示例 4-11 演示如何在 C++ 中调用汇编语言。

示例 4-10 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 4-11 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 4-12 和示例 4-13 演示如何在 C 中调用 C++。

示例 4-12 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 4-13 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 4-14 和示例 4-15 演示如何在汇编语言中调用 C++。

示例 4-14 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，导入 C++ 函数的名称，然后使用*带链接的跳转* (BL) 指令来调用该函数：

示例 4-15 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 4-16 和示例 4-17 演示如何在 C 和 C++ 之间传递引用。

示例 4-16 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++ function to be called from C.
    return 7 * r;
}
int f() {
    int i = 3;
    return cfunc(i);    // passes a pointer to 'i'
}
```

示例 4-17 定义 C 函数

```
extern int cppfunc(const int*);
/* declaration of the C++ function 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 4-18、示例 4-19 和第4-18 页的示例 4-20 中的代码演示如何在 C 或汇编语言中调用非静态、非虚 C++ 成员函数。可以使用编译器的汇编器输出来查找已重整的函数名。

示例 4-18 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 4-19 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 4-20 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4-17 页的示例 4-18 和示例 4-20，如示例 4-21 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 4-21 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 5 章
交互操作 ARM 和 Thumb

本章介绍在为实现 ARM 和 Thumb 指令集的处理器编写代码时如何在 ARM® 状态与 Thumb® 状态之间切换。

注意

本章不适用于 ARMv6-M 和 ARMv7-M。

本章分为以下几节：

5.1 关于交互操作

通过交互操作，可以混合使用 ARM 和 Thumb 代码，以便：

- ARM 例程返回 Thumb 状态调用方
- Thumb 例程返回 ARM 状态调用方。

这意味着，如果编译或汇编交互操作代码，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 编译器和 ARM 汇编器都使用 `--apcs=/interwork` 命令行选项来启用交互操作。

只要代码符合 AAPCS，可以随意混合为 ARM 和 Thumb 编译或汇编的代码。请参阅 *install_directory*`\Documentation\Specifications\...\PDF\aapcs.pdf` 中的规范。

如果链接器检测到以下情况，则会生成错误：

- 直接进行 ARM 或 Thumb 交互操作调用，但被调用例程不是为交互操作生成的
- 汇编语言源文件使用不兼容的 AAPCS 选项。

从另一种状态调用交互操作函数时，ARM 链接器进行检测。通过更改调用和返回指令，并在必要时插入称为中间代码的小代码段，更改处理器状态。有关详细信息，请参阅 《链接器用户指南》中第3-17 页的*中间代码*。

ARM 体系结构 v5T 及更高版本提供了一些方法，可以在不使用任何额外指令的情况下更改处理器状态。在 ARMv5T 及更高版本处理器上，交互操作几乎没有任何开销。

注意

为 ARMv5T 及更高版本体系结构进行编译时，自动假定进行交互操作，始终生成交互操作安全的代码。但是，为 ARMv5T 生成的汇编代码不包含交互操作，因此必须使用 `--apcs=/interwork` 汇编器选项生成汇编代码。

5.1.1 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生成为在 Thumb 状态下运行。这样可获得最佳代码密度。使用 8 位或 16 位宽的内存，还会使性能最佳。但是，出于以下考虑，可能希望部分应用程序在 ARM 状态下运行：

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 状态下的运行效率可能比在 Thumb 状态下更高。

有些系统具有少量快速 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，请参阅 《汇编器指南》中第4-134 页的*CPS*
- 访问协处理器
- 执行不能用 C 语言执行的*数字信号处理器* (DSP) 数学运算指令。

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，异常处理程序的开始部分必须用 ARM 指令编写代码，即使该处理程序重新进入 Thumb 状态来执行异常的主处理也是如此。完成此类处理时，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。若要运行简单的 Thumb 汇编语言程序，请添加一个 ARM 头文件，该头文件将状态更改为 Thumb 状态，然后调用主 Thumb 例程。有关示例，请参阅第5-7 页的*汇编语言交互操作*。

——注意——

因速度或功能原因而更改为 ARM 状态主要与支持 Thumb 而不支持 Thumb-2 的处理器有关。Thumb-2 指令集提供了几乎与 ARM 指令集完全一样的功能。

5.2 汇编语言交互操作

通过 --apcs=/interwork 命令行选项， ARM 汇编器可以对可从其他处理器状态调用的代码进行汇编：

```
armasm --thumb --apcs=/interwork
armasm --arm --apcs=/interwork
```

在汇编语言源文件中，可以有多个区域。这些区域对应于 ARM *可执行和链接格式* (ELF) 节。每个区域都可包括 ARM 指令和/或 Thumb 指令。

通过使用链接器，可以实现对调用方使用不同指令集的例程的调用，以及从这些例程返回。要执行此操作，请使用 BL 来调用例程，请参阅第5-8 页的示例 5-3。

如果您愿意，可以编写代码，显式地更改指令集。如果采用这种方法，在某些情况下，可以编写更小更快的代码。可以使用 BX、BLX、LDR、LDM 和 POP 指令执行处理器状态更改，请参阅第5-7 页的示例 5-2。有关详细信息，请参阅 《汇编器指南》中第4-111 页的*B*、 *BL*、 *BX*、 *BLX 和 BXJ*。

ARM 汇编器可汇编 Thumb 代码和 ARM 代码。缺省情况下，除非用 --thumb 选项调用，汇编器都汇编 ARM 代码。

因为所有支持 Thumb 的 ARM 处理器都是以 ARM 状态启动的，所以必须使用 BX 指令跳转并切换到 Thumb 状态，然后使用以下汇编指令指示汇编器切换汇编模式。

ARM 和 THUMB 指令指示汇编器根据相应指令集对指令进行汇编，请参阅 《汇编器指南》中第7-59 页的*ARM*、 *THUMB*、 *THUMBX*、 *CODE16 和 CODE32*。

5.3 C 和 C++ 交互操作

通过 --apcs=/interwork 命令行选项， ARM 编译器可以对可从另一种处理器状态调用的 C 和 C++ 代码进行编译：

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

在叶函数 （函数体不包含任何函数调用的函数）中，编译器生成返回指令 BX lr。

在 Thumb 状态下为 ARMv4T 生成的非叶函数中，编译器必须进行替换，例如将单个返回指令：

```
POP  {R4-R7,pc}
```

替换为序列：

```
POP  {R4-R7}
POP  {R3}
BX   R3
```

这对性能有一点影响。

如果使用 --apcs=/interwork 选项，则还会为要将模块编译到其中的代码区设置交互操作属性。链接器检测此属性并插入适当的中间代码。若要确定中间代码所占空间大小，可以使用链接器命令行选项 --info=veneers。

建议将所有源模块编译为用于交互操作，除非您确信这些模块不会用于交互操作。

注意

为交互操作而编译的 ARM 代码只能用于 ARMv4T 和更高版本，因为早期的处理器不实现 BX 指令。

5.3.1 Thumb 状态下函数的指针

指向 Thumb 函数 （由 Thumb 代码组成的函数，在 Thumb 状态下运行）的任何指针都必须设置最低有效位。这样可确保交互操作正常运行。

链接器重定位一个引用 Thumb 指令的标签的值时，会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，如果必须在代码中对 Thumb 函数使用绝对地址，则必须将该地址加 1，请参阅第5-6 页的示例 5-1。

示例 5-1 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
}
```

5.3.2 使用同一函数的两个版本

可以使用两个名字相同的函数，一个编译为 ARM，另一个编译为 Thumb。

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。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。

5.4 交互操作示例

以下是交互操作的示例：

- 示例 5-2 演示汇编语言交互操作
- 第5-8 页的示例 5-3 演示使用中间代码的汇编语言交互操作
- 第5-10 页的示例 5-4 演示 C 和 C++ 语言交互操作
- 第5-11 页的示例 5-5 演示使用中间代码的 C、 C++ 和汇编语言交互操作。

此外， RealView Development Suite 也附带了一些交互操作示例。有关详细信息，请参阅 *install_directory*\RVDS\Examples\...\interwork 中的 readme.txt 文件。

示例 5-2 汇编语言交互操作

此示例实现一小段后跟 ADR 指令的头文件节 (SECTION 1)，以获取标签 THUMBProg 的地址，并设置该地址的最低有效位。BX 指令将状态更改为 Thumb 状态。

在 SECTION2 中， Thumb 代码将两个寄存器的内容相加，使用 ADR 指令获取标签 ARMProg 的地址，并清除最低有效位。BX 指令将状态改回 ARM 状态。

在 SECTION3 中， ARM 代码将两个寄存器中的值相加，然后结束。

```
     ; ********
     ; addreg.s
     ; ********

     PRESERVE8
     AREA     AddReg,CODE,READONLY  ; Name this block of code.
     ENTRY                          ; Mark first instruction to call.
; SECTION1
start
     ADR R0, ThumbProg:OR:1         ; Generate branch target address
                                    ; and set bit 0, hence arrive
                                    ; at target in Thumb state.
     BX  R0                         ; Branch exchange to ThumbProg.
; SECTION2
     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.
; SECTION3
     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
     END                            ; Mark end of this file.
```

按照以下步骤编译并链接模块：

1. 若要汇编源文件以进行交互操作，请键入：
   ```
   armasm --debug --apcs=/interwork addreg.s
   ```
2. 若要链接对象文件，请键入：
   ```
   armlink addreg.o -o addreg.axf
   ```
   或者，若要查看交互操作中间代码的大小，请键入：
   ```
   armlink addreg.o -o addreg.axf --info=veneers
   ```
3. 通过对相应调试目标使用兼容调试器来运行映像。

示例 5-3 使用中间代码的汇编语言交互操作

此示例演示汇编代码中的源代码交互操作，其功能是将寄存器 R0 至 R2 分别设置为值 1、 2 和 3。寄存器 R0 和 R2 由 ARM 代码设置。R1 由 Thumb 代码设置。链接器会自动添加交互操作中间代码。使用中间代码：

- 必须用 --apcs=/interwork 选项来汇编代码
- 使用 BX lr 指令返回，而不是 MOV pc,lr。

```
     ; *****
     ; 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 the ARM function.
    END                         ; Mark end of this file.
```

按照以下步骤编译并链接模块：

1. 若要汇编用于交互操作的 ARM 代码，请键入：

   ```
   armasm --debug --apcs=/interwork arm.s
   ```

2. 若要汇编用于交互操作的 Thumb 代码，请键入：

   ```
   armasm --thumb --debug --apcs=/interwork thumb.s
   ```

3. 若要链接对象文件，请键入：

   ```
   armlink arm.o thumb.o -o count.axf
   ```

   或者，若要查看交互操作中间代码的大小，请键入：

   ```
   armlink arm.o thumb.o -o count.axf --info=veneers
   ```

4. 通过对相应调试目标使用兼容调试器来运行映像。

示例 5-4 C 和 C++ 语言交互操作

此示例演示一个 Thumb 例程，它执行对 ARM 子例程的交互操作调用。该 ARM 子例程对 Thumb 库中的 printf() 进行交互操作调用。

```
/**********************
*       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");
}
```

按照以下步骤编译并链接模块：

1. 若要编译用于交互操作的 Thumb 代码，请输入：

   ```
   armcc --thumb -c --debug --apcs=/interwork thumbmain.c -o thumbmain.o
   ```

2. 若要编译用于交互操作的 ARM 代码，请输入：

   ```
   armcc -c --debug --apcs=/interwork armsub.c -o armsub.o
   ```

3. 若要链接对象文件，请键入：

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

   或者，若要查看交互操作中间代码的大小，请键入：

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

4. 通过对相应调试目标使用兼容调试器来运行映像。

示例 5-5 使用中间代码的 C、 C++ 和汇编语言交互操作

此示例演示 C 语言 Thumb 代码与汇编语言 ARM 代码之间的交互操作。

```
/**********************
*       thumb.c      *
**********************/

#include <stdio.h>
extern int arm_function(int);
int main(void)
{
     int i = 1;
     printf("i = %d\n", i);
     printf("And i+4 = %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
```

按照以下步骤编译并链接模块：

1. 若要编译用于交互操作的 Thumb 代码，请输入：

   ```
   armcc --thumb --debug -c --apcs=/interwork thumb.c
   ```

2. 若要汇编用于交互操作的 ARM 代码，请键入：

   ```
   armasm --debug --apcs=/interwork arm.s
   ```

3. 若要链接对象文件，请键入：

   ```
   armlink arm.o thumb.o -o add.axf
   ```

   或者，若要查看交互操作中间代码的大小，请键入：

   ```
   armlink arm.o thumb.o -o add.axf --info=veneers
   ```

4. 通过对相应调试目标使用兼容调试器来运行映像。

第 6 章
处理处理器异常

本章介绍如何处理 ARM® 体系结构支持的各种类型的异常。

本章分为以下几节：

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) 在其地址空间内顺序增加，还可以跳转至附近标签或通过子例程链接进行跳转。

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

图 6-1 演示异常处理过程。

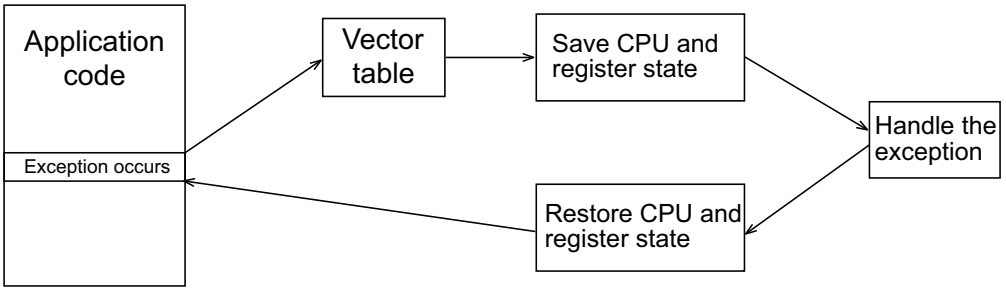

图 6-1 处理异常

当异常发生时，控制权在称为向量表的内存区域中进行传递。这是一个保留区域，通常位于内存映射的底端。在向量表中，每个异常类型都分配有一个字。这个字包含跳转指令，如果是 ARMv6-M 和 ARMv7-M，则包含相关异常处理程序的地址。

根据处理器所支持的指令集，可以使用 ARM 或 Thumb®-2 代码编写异常处理程序。对于 ARMv7-M 和 ARMv6-M 架构，处理器进入向量表中所指定的异常处理程序。对于所有其他 ARM 处理器，必须从顶层处理程序跳转到异常处理代码。如果需要更改状态，请使用*分支交换* (BX)（有关详细信息，请参阅第 5 章*交互操作 ARM 和 Thumb*）。在处理异常时，必须保存当前处理器模式、状态和寄存器，以便在完成相应异常处理例程后继续执行程序。

6.2 ARMv6 及更早版本、 ARMv7-A 和 ARMv7-R 架构

本节介绍如何处理 ARM 体系结构 v6 及更早版本以及 ARMv7-A 和 ARMv7-R 架构支持的各类异常。

注意

微控制器架构使用另一种异常处理模型。有关详细信息，请参阅第6-28 页的 *ARMv6-M 和 ARMv7-M 架构*。

6.2.1 异常类型

表 6-1 列出了 ARMv6 及更早版本以及 ARMv7-A 和 ARMv7-R 架构可识别的各类异常。当多个异常同时发生时，按固定的优先级顺序进行处理。在返回到原始应用程序前，依次处理每个异常。同时发生所有异常是不可能的。例如，*未定义指令* (Undef) 和*超级用户调用* (SVC) 异常是互斥的，因为两者均是通过执行一条指令触发的。

进入异常时：

- 对所有异常禁用 *中断请求* (IRQ)。
- 对 FIQ 和重置异常禁用 *快速中断请求* (FIQ)。

表 6-1 按优先级排序的异常类型

优先级（1=高，6=低）	异常类型	异常模式	说明
1	复位	超级用户	在处理器复位引脚生效时发生。仅当接通电源或复位（处理器已接通电源）时才会发生此异常。通过跳转到复位向量可以完成软复位。
2	数据中止	中止	当数据传输指令试图在非法地址加载或存储数据时发生[a]。
3	FIQ	FIQ	当处理器外部快速中断请求引脚生效 （低电平），且清除了 CPSR 中的 F 位时发生。
4	IRQ	IRQ	当处理器外部中断请求引脚生效 （低电平），且清除了 CPSR 中的 I 位时发生。

表 6-1 按优先级排序的异常类型 （续）

优先级（1=高，6=低）	异常类型	异常模式	说明
5	预取中止	中止	当处理器试图执行还未获取的指令时发生，因为地址是非法的[a]。
6	SVC	超级用户	这是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。
6	未定义指令	未定义	在处理器或任何附加的协处理器均不能识别当前执行的指令时发生。

a. 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确定在当前模式下处理器不可访问的地址。

因为“数据中止”异常比 FIQ 异常具有更高的优先级，所以“数据中止”实际上在处理 FIQ 之前已被寄存。尽管进入了“数据中止”处理程序，但控制权会立即传递给 FIQ 处理程序，因为在处理“数据中止”时，FIQ 保持为启用状态。处理完 FIQ 后，控制权返回到“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。

6.2.2 向量表

ARMv6 及更早版本以及 ARMv7-A 和 ARMv7-R 架构的向量表由相关处理程序的跳转或加载 PC 指令构成。如有必要，可以在向量表末尾包含 FIQ 处理程序，以确保尽可能高效地对其进行处理，请参阅示例 6-1。使用文字池意味着以后可以在需要时轻松地修改地址。

示例 6-1 使用文字池的典型向量表

```
                AREA vectors, CODE, READONLY
                ENTRY
Vector_Table
                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
FIQ_Handler
```

```
                ; FIQ handler code - max 4kB in size

Reset_Addr      DCD Reset_Handler
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
                ...
                END
```

此示例假设 ROM 在复位时位于 `0x0` 位置。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。这种情况下， C 库将复制向量表。有关分散加载的详细信息，请参阅 《链接器用户指南》中的第 5 章 *使用分散加载描述文件*。

注意

ARMv6 及更早版本体系结构的向量表仅支持 ARM 指令。ARMv6T2 及更高版本体系结构的向量表同时支持 Thumb-2 和 ARM 指令。这不适用于 ARMv6-M 和 ARMv7-M 架构。

6.2.3 处理器模式和寄存器

ARM 体系结构定义了一个非特权用户模式，该模式包含 15 个通用寄存器、一个 PC 和一个 CPSR。此外，还存在其他特权模式，每个特权模式都包含一个 SPSR 和一些编组的寄存器。

通常，应用程序在用户模式下运行，但处理异常需要特权模式。异常会更改处理器模式，这意味着每个异常处理程序都可以访问编组寄存器的某个子集：

- 自身的*堆栈指针* (SP)
- 自身的 LR
- 自身的 SPSR
- 五个其他通用寄存器 （仅 FIQ）

每个异常处理程序都必须确保在退出时将其他寄存器恢复为其原来的内容。通过将处理程序必须使用的所有寄存器的内容保存到其堆栈中，并在返回前恢复这些寄存器的内容，可以实现这一目的。

系统模式

在处理多个同一类型的异常时，链接寄存器损坏可能会造成问题。请参阅第6-9 页的*重入中断处理程序*。

ARMv4 及之后的体系结构包含一个称为*系统*模式的特权模式，用于解决这一问题。系统模式使用与用户模式相同的寄存器，在这种模式中，可以运行需要进行特权访问的任务，并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 才能进入系统模式。有关示例，请参阅第6-9 页的*重入中断处理程序*。

6.2.4 处理异常

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主程序。必须确保异常处理程序在发生异常时保存系统状态，并在返回时恢复系统状态。

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。如果出现异常，则需要从相应的向量表入口获取下一条指令。

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 CPSR 复制到相应的 SPSR 中。这会保存当前模式、中断屏蔽和条件标记。
2. 如果当前状态与异常向量表中使用的指令集不匹配，则会自动切换状态。
3. 更改相应的 CPSR 模式位，以便：
   - 更改为适当的模式，并在该模式的相应编组寄存器中进行映射。
   - 禁用中断。发生任何异常时，都会禁用 IRQ。在复位时发生 FIQ，会禁用 FIQ。
4. 将相应 LR 设置为返回地址。
5. 将 PC 设置为异常的向量地址。

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：

- 从相应 SPSR 恢复 CPSR
- 使用相应 LR 中的返回地址恢复 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 标记
- PC 作为目标寄存器。

所需的返回指令取决于异常的类型。

注意

不必从复位处理程序返回，因为复位处理程序直接执行主代码。

处理异常时，如果异常处理程序入口代码使用了堆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。它只能在特权模式下使用。有关详细信息，请参阅 《汇编器指南》中有关如何使用 LDM 和 STM 实现堆栈的说明。

注意

不能使用任何 16 位 Thumb 指令从异常返回，因为这些指令无法恢复 CPSR。

6.2.5 重置处理程序

重置处理程序执行的操作取决于开发的软件所面向的系统。

例如，它可以：

- 设置异常向量。有关详细信息，请参阅第6-4 页的*向量表*。
- 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 3 章 *嵌入式软件开发*。

6.2.6 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，该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 ARM7™（包括 ARM7TDMI®）上，地址寄存器已更新，因此必须撤消更改。
- 如果中止发生在 ARM9™ 或更高版本的处理器上，则处理器会将地址恢复为开始执行指令之前的值。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 ARM9 或更高版本的处理器上并启用了回写，则基址寄存器恢复为开始执行指令前的值。

上述三个例子中，MMU 均可将所需的虚拟内存加载物理内存。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。

在示例目录的 ...\databort 中可以找到数据中止处理程序的示例。

6.2.7 中断处理程序

本节介绍如何编写中断处理程序。

外部中断的级别

ARM 处理器有 FIQ 和 IRQ 两级外部中断，它们都是由对电平敏感的低电平 (LOW) 信号激活进入处理器的。为了产生中断， CPSR 中的相应禁用位必须清零。

FIQ 的优先级比 IRQ 高，具体表现如下：

- 当发生多个中断时，首先处理 FIQ。
- 处理 FIQ 会导致禁用 IRQ 和后续 FIQ，在 FIQ 处理程序启用之前，不会处理 IRQ 和后续 FIQ。这通常是通过在处理程序结束时从 SPSR 恢复 CPSR 来完成的。

FIQ 向量是向量表的最后一个入口，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。这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 处理程序可能都锁定在高速缓存中的一个块内。这一点非常重要，因为 FIQ 就是为了尽快处理中断而设计的。通过五个额外的 FIQ 模式编组寄存器，可以保存处理程序各次调用的状态，这也加快了执行速度。

注意

中断处理程序必须包含清除中断源的代码。

重入中断处理程序

如果中断处理程序在调用子例程前启用中断，并且发生了另一个中断，则在第二个 IRQ 发生时，会损坏存储在 IRQ 模式 LR 中的子例程的返回地址。这是因为处理器会自动将返回地址保存在 the IRQ mode LR 中，使得新中断可以覆盖子例程的返回地址。如果原先的中断中的子例程试图返回，则会导致无限循环。

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、切换处理器模式并保存新处理器模式的状态。还必须确保堆栈对新处理器模式是 8 字节对齐的，然后才能调用按照 AAPCS 编译的 C 代码，这种代码可能使用 LDRD 或 STRD 指令，或者 8 字节对齐堆栈分配数据。有关堆栈对齐问题的详细信息，请参阅《ARM 体系结构的 ABI 告知说明 1 - 进入符合 AAPCS 的函数时 SP 必须是 8 字节对齐的》(*ABI for the ARM Architecture Advisory Note 1- SP must be 8-byte aligned on entry to AAPCS-conforming functions*) (ARM IHI 0046A)。

在 C 语言中使用 __irq 关键字不会对 SPSR 进行保存和恢复，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 ARMv4 或更高版本中，如果需要进行特权访问，可以切换为系统模式。有关详细信息，请参阅第6-6 页的*系统模式*。

注意

这一方法适用于 IRQ 和 FIQ 中断。但是，因为 FIQ 中断需要尽快处理，而且通常只有一个中断源，所以，可能不必提供重入特性。

在 IRQ 处理程序中安全地启用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 IRQ 模式 SPSR。
3. 清除中断源。
4. 切换为系统模式，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 LR 以及在用户模式 SP 上所使用的调整 （用于体系结构 v4 的 0，或用于 v5TE 的 4）。
7. 启用中断并调用 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 LR 和堆栈调整值。
10. 必要时重新调整堆栈。
11. 切换为 IRQ 模式。
12. 恢复其他寄存器和 IRQ 模式 SPSR。

13. 从 IRQ 返回。

示例 6-2 和示例 6-3 演示如何在系统模式下实现这些步骤。

示例 6-2 ARMv4/v5TE 的重入中断处理程序

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

IRQ_Handler
    SUB     lr, lr, #4              ; construct the return address
    PUSH    {lr}                    ; and push the adjusted lr_IRQ
    MRS     lr, SPSR                ; copy spsr_IRQ to lr
    PUSH    {R0-R4,R12,lr}          ; 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)
    PUSH    {R1,lr}                 ; store the adjustment and lr_USR
    MSR     CPSR_c, #0x1F           ; enable IRQ
    BL      C_irq_handler
    MSR     CPSR_c, #0x9F           ; disable IRQ, remain in SYS mode
    POP     {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.
    POP     {R0-R4,R12,lr}          ; restore registers and
    MSR     SPSR_cxsf, lr           ; spsr_IRQ
    LDM     sp!, {pc}^              ; return from IRQ.
    END
```

示例 6-3 ARMv6 的重入中断 （非向量中断）

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

IRQ_Handler
    SUB         lr, lr, #4
    SRSDB       sp!,#31          ; Save LR_irq and SPSR_irq to System mode
```

```
stack
    CPS #031                    ; Switch to System mode
    PUSH        {R0-R3,R12}     ; Store other AAPCS registers
    AND         R1, sp, #4
    SUB         sp, sp, R1
    PUSH        {R1, lr}
    BL          identify_and_clear_source
    CPSIE       i               ; Enable IRQ
    BL          C_irq_handler
    CPSID i                     ; Disable IRQ
    POP         {R1,lr}
    ADD         sp, sp, R1
    POP         {R0-R3, R12}    ; Restore registers
    RFEIA       sp!             ; Return using RFE from System mode stack
    END
```

这两个示例假设 FIQ 保持为永久启用。

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。下面几节提供了几个示例：

- *单通道 DMA 传送*
- 第6-13 页的*双通道 DMA 传送*
- 第6-14 页的*中断的优先化*
- 第6-16 页的*上下文切换*

单通道 DMA 传送

第6-13 页的示例 6-4 演示一个中断处理程序，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。

处理常规传送的全部序列为四条指令。位于条件返回后的代码用于通知传送结束。

示例 6-4 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.
```

用加载字节指令替换加载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加载指令和存储指令的寻址模式来完成的。

双通道 DMA 传送

第6-14 页的示例 6-5 类似于示例 6-4，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。该代码最适合放置在位置 0x1C 处。

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, 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, 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 6-5 FIQ 处理程序

```
LDR     sp, [R8, #IOStat]      ; Load status register to find which port
                               ; caused the interrupt.
TST     sp, #IOPort1Active
LDREQ   sp, [R8, #IOPort1]     ; Load port 1 data.
LDRNE   sp, [R8, #IOPort2]     ; Load port 2 data.
STREQ   sp, [R9], #4           ; Store to buffer 1.
STRNE   sp, [R10], #4          ; Store to buffer 2.
CMP     R9, R11                ; Reached the end?
CMPLE   R10, R12               ; On either channel?
SUBSNE  pc, lr, #4             ; Return
                               ; Insert transfer complete code here.
```

用加载字节指令替换加载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加载指令和条件存储指令的寻址模式来完成的。

中断的优先化

第6-15 页的示例 6-6 最多可为 32 个中断源调度其相应处理程序。因为它用于常规中断向量 (IRQ)，所以从位置 0x18 跳转。

外部*向量中断控制器* (VIC) 硬件用于确定中断优先级，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 保存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——

应用程序说明 30：《软件中断优先级》(*Software Prioritization of Interrupts*) 介绍如何使用软件确定多个中断源的优先级，与此处所述的用 VIC 硬件确定优先级的方式不同。

示例 6-6 向处理程序调度中断

```
    ; first save the critical state
    SUB     lr, lr, #4              ; Adjust the return address
                                    ; before we save it.
    STMDB   sp!, {lr}               ; Stack return address
    MRS     lr, SPSR                ; get the SPSR ...
    PUSH    {R12,lr}                ; ... 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     lr, APSR                ; Read the status register.
    BIC     lr, lr, #0x80           ; Clear the I bit
                                    ; (use 0x40 for the F bit).
    MSR     CPSR_c, lr              ; 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
    PUSH    {R0-R11}                ; Save other working registers.
                                    ; Insert handler code here.
; ...
    POP     {R0-R11}                ; Restore working registers (not R12).
                                    ; Now read-modify-write the CPSR
                                    ; to disable interrupts.
    MRS     R12, APSR               ; 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.
    POP     {r12,lr}                ; Restore R12 and get SPSR.
    MSR     SPSR_cxsf, lr           ; Restore status register from R14.
    LDM     sp!, {pc}^              ; Return from handler.
Priority1Handler
; ...
```

上下文切换

示例 6-7 执行对用户模式进程的上下文切换。这段代码基于一组指针，这些指针指向要运行的进程的*进程控制块* (PCB)。

图 6-2 是该示例的 PCB 布局。

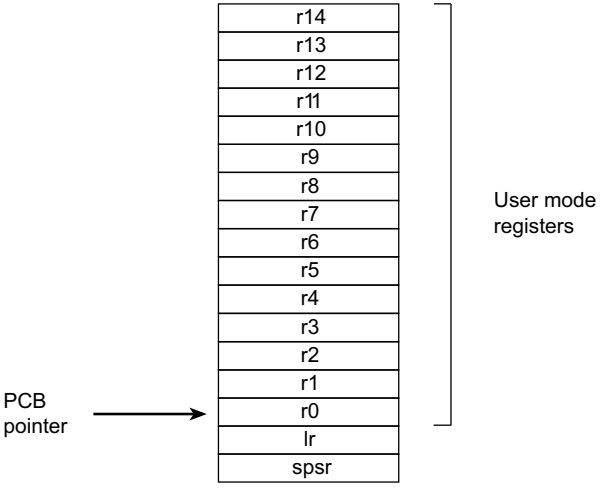

图 6-2 PCB 布局

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，它会保留一小段时间，以便在进入时指向当前运行进程的 PCB。

示例 6-7 对用户模式进程的上下文切换

```
STM     sp,{R0-lr}^             ; Dump user registers above R13.
MRS     R0, SPSR                ; Pick up the user status
STMDB   sp, {R0, lr}            ; and dump with return address below.
LDR     sp, [R12], #4           ; Load next process info pointer.
CMP     sp, #0                  ; If it is zero, it is invalid
LDMDBNE sp, {R0, lr}            ; Pick up status and return address.
MSRNE   SPSR_cxsf, R0           ; Restore the status.
LDMNE   sp, {R0 - lr}^          ; Get the rest of the registers
NOP
SUBSNE pc, lr, #4               ; and return and restore CPSR.
                                ; Insert "no next process code" here.
```

6.2.8 SVC 处理程序

发生异常时，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 状态时为 1，在 ARM 状态时为 0。

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 6-3。
- 在 ARM 状态下， SVC 编号以 8 位存储，而不是 24 位。

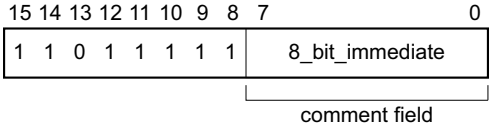

图 6-3 Thumb SVC 指令

示例 6-8 演示处理 SVC 异常的 ARM 代码。通过动态调用 SVC，可以增大 Thumb 状态下可访问的 SVC 编号的范围。

示例 6-8 SVC 处理程序

```
    PRESERVE8
    AREA SVC_Area, CODE, READONLY
    EXPORT SVC_Handler
    IMPORT C_SVC_Handler
T_bit   EQU    0x20                  ; Thumb bit (5) of CPSR/SPSR.
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
    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 main part of handler
LDMFD   sp!, {r0, r3}          ; Get spsr from stack
MSR     SPSR_cxsf, r0               ; Restore spsr
LDMFD   sp!, {r0-r3, r12, pc}^ ; Restore registers and return
END
```

确定要调用的 SVC

进入 SVC 处理程序后，必须确定要调用哪个 SVC。此信息可存储在指令本身的 0-23 位，如图 6-4 所示，或将其传递给某个整数寄存器，通常为 R0-R3 中的一个。

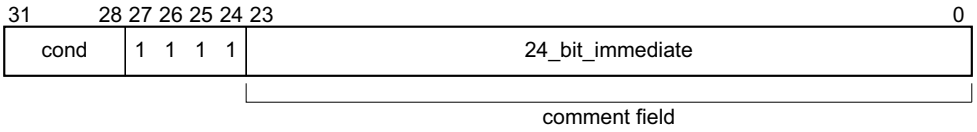

图 6-4 ARM SVC 指令

顶层 SVC 处理程序可以相对于 LR 加载 SVC 指令 请用汇编语言、 C/C++ 内联或嵌入式汇编器编写。

处理程序必须首先将导致异常的 SVC 指令加载到寄存器中。此时，SVC LR 保存 SVC 指令的下一个指令的地址，这样 SVC 加载到了寄存器 （本例中为 R0）中，代码如下：

```
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
```

第6-19 页的示例 6-9 演示如何用这些指令编写顶层 SVC 处理程序。有关在 ARM 状态和 Thumb 状态下处理 SVC 指令的处理程序示例，请参阅第6-17 页的示例 6-8。

示例 6-9 顶层 SVC 处理程序

```
    PRESERVE8
    AREA TopLevelSVC, CODE, READONLY   ; Name this block of code.
    EXPORT     SVC_Handler
SVC_Handler
    PUSH       {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.
    ;
    LDM         sp!, {R0-R12,pc}^      ; Restore registers and return.
    END
```

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 包含 SVC 编号，则示例 6-10 中的代码可以插入到示例 6-9 提供的顶层处理程序中，插入位置在 BIC 指令之后。

示例 6-10 SVC 跳转表

```
    AREA SVC_Area, CODE, READONLY
    PRESERVE8
    IMPORT SVCOutOfRange
    IMPORT MaxSVC
    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.
    END
```

C 语言和汇编语言编写的 SVC 处理程序

尽管顶层处理程序始终必须用 ARM 汇编语言编写，处理每个 SVC 的例程既可用汇编语言编写，也可用 C 语言编写。有关限制的说明，请参阅第6-21 页的*在超级用户模式下使用 SVC*。

顶层处理程序使用 BL 指令可跳转到相应的 C 函数。因为 SVC 编号是由汇编例程加载到 R0 中的，所以作为第一个参数传递给 C 函数。举例来说，函数可以将此参数值用在 switch() 语句中，请参阅示例 6-11。

若要调用此 C 函数，可以向第6-19 页的示例 6-9 中 SVC_Handler 例程添加下面的代码行：

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

示例 6-11 C 函数中的 SVC 处理程序

```
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 */
    }
}
```

超级用户模式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.
```

用 C 语言编写的 SVC 处理程序可以传入和传出值，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 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 图 6-5。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到堆栈的相应位置，然后通过顶层处理程序恢复到寄存器中。

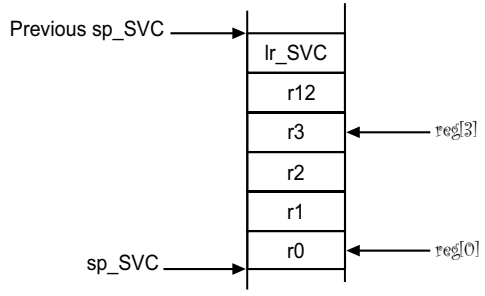

图 6-5 访问超级用户模式堆栈

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 存储到 SVC SPSR 中。
3. 返回地址存储在 SVC LR 中，请参阅第6-6 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，则会损坏 SVC LR 和 SPSR。

如果在超级用户模式下调用 SVC，则必须存储 SVC LR 和 SPSR，以确保 LR 和 SPSR 的原始值不会丢失。例如，如果某个特定 SVC 编号的处理程序例程调用另一个 SVC，则必须确保该处理程序例程将 SVC LR 和 SPSR 都存储在堆栈中。这可确保处理程序的每一次调用都保存返回到调用它的 SVC 后面的指令所需要的信息。示例 6-12 演示如何实现这一点。

示例 6-12 SVC 处理程序

```
    AREA SVC_Area, CODE, READONLY
    PRESERVE8
    EXPORT SVC_Handler
    IMPORT C_SVC_Handler
T_bit EQU 0x20
SVC_Handler
    PUSH     {R0-R3,R12,lr}       ; Store registers.
    MOV      R1, sp               ; Set pointer to parameters.
    MRS      R0, SPSR             ; Get SPSR.
    PUSH     {R0,R3}              ; Store SPSR onto stack and another register to maintain
                                  ; 8-byte-aligned stack. Only required for nested SVCs.
    TST      R0,#0x20             ; Occurred in Thumb state?
    LDRHNE   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.
    POP      {R0,R3}              ; Get SPSR from stack.
    MSR      SPSR_cf, R0          ; Restore SPSR.
    LDM      sp!, {R0-R3,R12,pc}^ ; Restore registers and return.
    END
```

用 C 和 C++ 编写的嵌套 SVC

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

从应用程序调用 SVC

可用汇编语言或 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，它的第一个参数必须是存放结果结构的地址。

示例 6-13 和 第6-24 页的示例 6-14 演示一个 SVC 处理程序，该处理程序提供 SVC 编号 0x0、0x1、0x2 和 0x3。SVC 0x0 和 SVC 0x1 都采用两个整数参数并返回一个结果。SVC 0x2 采用四个参数并返回一个结果。SVC 0x3 采用四个参数并返回四个结果。此示例位于示例目录 （...\svc\main.c 和 ...\svc\svc.h）中。

示例 6-13 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 6-14 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);
```

从应用程序动态调用 SVC

在某些情况下，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 （作为一个额外的参数）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 6-15 演示如何使用 __svc 将 C 函数调用映射到半主机调用。该示例是根据示例目录中的 retarget.c 编写的，该文件路径为 ...\emb_sw_dev\source\retarget.c。

示例 6-15 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6-22 页的*从应用程序调用 SVC*。在 R12 中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。

示例 6-16 演示使用通用或*间接* SVC 的 C 程序段。

示例 6-16 使用间接 SVC

```
__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);
}
```

它生成以下代码：

```
DoSelectedManipulation
        PUSH    {R4,lr}
        MOV     R12,R2
        SVC     #0x80
        POP     {R4,pc}
        END
```

还可使用 __svc 机制从 C 中传递 R0 中的 SVC 编号。例如，如果将 SVC 0x0 用作通用 SVC，操作 0 为字符读，操作 1 为字符写，则可以进行如下设置：

```
__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，因为这些参数通常在用户模式堆栈中，而不是在 SVC 处理程序使用的超级用户模式堆栈中。

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。

6.2.9 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告错误并退出。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，处理程序返回该指令而不是它之后的那个指令，例如：

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

6.2.10 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，这类协处理器可能有一个软件仿真器。

这样的仿真器必须：

1. 使自身附加到未定义指令向量并存储旧的内容。
2. 检查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。

   这些位确定指令是否为协处理器操作的方法如下：

   - 如果 [27:24] 位为 b1110 或 b110x，则指令为协处理器指令。
   - 如果 [8:11] 位显示此协处理器仿真器必须处理该指令，则仿真器必须处理该指令并返回用户程序。
3. 否则，仿真器必须使用仿真器安装时存储的向量，将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，因此不需要未定义指令处理程序来仿真这类指令。

6.3 ARMv6-M 和 ARMv7-M 架构

本节介绍如何处理微控制器架构 （例如 Cortex™-M1 和 Cortex-M3）支持的各类异常。

微控制器架构支持：

- 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。

复位时会进入线程模式，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，处理器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。

通过使用 MSR 指令清除 CONTROL[0]，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 （例如 SVC）无法直接从用户模式更改为特权模式，请参阅第6-35 页的*超级用户调用*。

6.3.1 主堆栈和进程堆栈

微控制器架构支持两个不同的堆栈，即主堆栈和进程堆栈。它有两个堆栈指针，分别用于两个堆栈。任何时候只有一个堆栈指针可见，具体取决于正在使用的堆栈。

复位以及进入异常处理程序时使用主堆栈。若要使用进程堆栈，则必须选择该堆栈。在线程模式中时，可通过使用 MSR 指令写入 CONTROL[1] 来实现这一目的。

——注意——

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。

6.3.2 异常类型

表 6-2 列出了微控制器架构可以识别的不同类型的异常。同时发生的异常是按固定的优先级顺序处理的。在返回到原始应用程序前，依次处理每个异常。

表 6-2 按优先级排序的异常类型

位置	异常	优先级	禁用	说明
1	重置	-3	否	
2	NMI	-2	否	*不可屏蔽中断* (NMI)
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 （仅适用于 ARMv7-M）
5	BusFault	可配置	可禁用	其他内存错误 （仅适用于 ARMv7-M）
6	UsageFault	可配置	可禁用	除内存错误以外的指令执行错误 （仅适用于 ARMv7-M）
7-10	暨椒	-	-	
11	SVCall	可配置	可禁用	执行 SVC 指令导致的同步 SVC 调用
12	调试监控器	可配置	可禁用	同步调试事件 （仅适用于 ARMv7-M）
13	保留	-	-	
14	PendSV	可配置	可禁用	异步 SVC 调用
15	SysTick	可配置	可禁用	系统计时器滴答声
16 及以上	外部中断	可配置	可禁用	外部中断

异常的优先级编号越低，其优先级越高。例如，当处理器处于处理程序模式时，如果某个异常的优先级编号低于当前正在处理的异常的优先级编号，则会先处理该异常。如果是优先级编号相同或更高的异常，则会将其*暂挂*。

当异常处理程序终止时：

- 如果没有暂挂的异常，则处理器返回线程模式，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。

6.3.3 向量表

微控制器架构的向量表由相关处理程序的地址组成。异常编号 *n* 的处理程序位于 (*vectorbaseaddress* + 4 * *n*)。

在 ARMv7-M 处理器中，可以在*向量表偏移量寄存器* (VTOR) 中指定 *vectorbaseaddress* 以重定位向量表。复位时的缺省位置为 `0x0` （CODE 空间）。对于 ARMv6-M，向量表基址固定为 `0x0`。有关每个异常的 *n* 值，请参阅第6-29 页的*异常类型*。位于 *vectorbaseaddress* 的字保存主堆栈指针的复位值。

注意

向量表中每个地址的最低有效位 （位 [0]）必须进行设置，否则会生成 HardFault 异常。如果向量表使用了 Thumb 符号名称，则 RealView 工具通常会为您实现此功能。

向量表偏移量寄存器 （仅适用于 ARMv7-M）

向量表偏移量寄存器将向量表置于 CODE 或 SRAM 空间中。在设置另一个位置时，偏移量必须根据表中的异常的数量进行对齐。这意味着最小对齐为 32 个字，可以用于最多 16 个中断。对于更多中断，必须通过向上舍入为 2 的下一次幂调整对齐。例如，如果需要 21 个中断，则必须在 64 字边界上对齐，因为表的大小是 37 个字， 2 的下一次幂是 64。

编写异常表

填充向量表的最简单方法是使用分散加载描述文件将函数指针的 C 数组放置到内存地址 `0x0`。可以使用 C 数组来配置初始堆栈指针、映像入口点和异常处理程序的地址，请参阅第6-31 页的示例 6-17。

注意

ARMv6-M 不具有第6-31 页的示例 6-17 所示的部分功能。若要维持对齐，必须通过在向量表中输入 0 来保留空间。

有关分散加载的详细信息，请参阅 《链接器用户指南》中的第 5 章 *使用分散加载描述文件*。

示例 6-17 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,                 /* ARMv7-M only (0 for ARMv6-M)
     */
    &BusFaultException,                  /* ARMv7-M only (0 for ARMv6-M)
     */
    &UsageFaultException,                /* ARMv7-M only (0 for ARMv6-M)
     */
    0, 0, 0, 0,                          /* Reserved */
    &SVCHandler,                         /* Only available with OS extensions
*/
    &DebugMonitor,                       /* ARMv7-M only (0 for ARMv6-M)
     */
    0,                                   /* Reserved */
    &PendSVC,                            /* Only available with OS extensions
*/
    (ExecFuncPtr)&SysTickHandler,        /* Only available with OS extensions
*/

    /* Configurable interrupts start here...*/
    &InterruptHandler,
    &InterruptHandler,
    &InterruptHandler
    /*
    :
    */
};
#pragma arm section
```

6.3.4 嵌套向量中断控制器

根据具体实现，*嵌套向量中断控制器* (NVIC) 可以支持：

ARMv6-M 1、 8、 16 或 32 个外部中断，分为 4 个优先级。

ARMv7-M 最多 240 个外部中断，最多 256 个优先级 （可重新动态确定优先级）。NVIC 还支持末尾连锁中断。

微控制器架构支持电平和脉冲中断源。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。

在微控制器架构中使用 NVIC 意味着向量表与包含地址 （而不包含指令）的其他 ARM 处理器有很大不同。初始堆栈指针和复位处理程序的地址必须分别位于 `0x0` 和 `0x4`。复位时，处理器将这些地址加载到 SP 和 PC 寄存器中。

6.3.5 处理异常

在微控制器架构上，完全由处理器来处理异常优先级、异常嵌套和易损坏寄存器的保存，从而极大地提高处理效率，尽量减少中断等待时间。在进入每个异常处理程序时，会自动启用中断，这意味着必须从为其他处理器编写的项目中删除任何顶层重入代码。如果需要禁用中断，则必须在代码中进行此操作，并确保从异常返回时启用这些中断。

注意

异常处理程序必须清除中断源。

微控制器架构没有 FIQ 输入。在其他处理器的项目上发出 FIQ 信号的任何外围设备都必须移到高优先级的外部中断。可能需要检查此类中断的处理程序是否不使用编组的 FIQ 寄存器，因为微控制器架构没有编组的寄存器，对于任何其他常规 IRQ 处理程序，必须对 R8-R12 进行堆栈处理。

微控制器架构还提供一个无法禁用的*不可屏蔽中断* (NMI)。

简单 C 异常处理程序

微控制器架构的异常处理程序不需要保存或恢复系统状态，可以编写为 ABI 兼容的普通 C 函数。不过，建议使用 `__irq` 关键字将函数标识为中断例程，请参阅第6-33 页的示例 6-18。

示例 6-18 简单 C 异常处理程序

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

8 字节堆栈对齐

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（如不同源文件中函数之间的调用）上 8 字节对齐。不过，代码不必保持内部 8 字节堆栈对齐，例如在叶函数中。这意味着当发生 IRQ 时，堆栈可能无法正确地 8 字节对齐。

当发生异常时， ARMv7-M 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。

ARMv6-M 处理器始终启用此行为，不过建议您手动设置 STKALIGN （位 9），以便映像向前兼容 ARMv7-M 处理器。

注意

如果要使用修订版 0 Cortex-M3 处理器，则不支持 STKALIGN，因此不会在硬件中执行调整，而是需要通过软件完成。编译器会在 IRQ 处理程序中生成正确对齐堆栈的代码。若要执行此操作，必须在 IRQ 处理程序前使用前缀 __irq 并使用 --cpu=Cortex-M3-rev0 编译器开关，而不是 --cpu=Cortex-M3。

6.3.6 配置系统控制空间寄存器

系统控制空间 (SCS) 寄存器位于 0xE000E000。结构可以用于表示大量单个寄存器及相关偏移量，请参阅示例 6-19。通过与向量表类似的方法，可以使用分散加载描述文件将结构放置到正确的内存位置。

在示例目录 （*install_directory*\RVDS\Examples\..\Example3）中，可以找到适用于 Cortex-M1 和 Cortex-M3 处理器的此代码示例。

示例 6-19 SCS 寄存器结构和定义

```
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
     * Additional space allocated to ensure alignment
     */

    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;                                    /* ARMv7-M only   */
    int ConfigCtrl;                                 /* ARMv7-M only   */
    int SystemPriority[3];                          /* ARMv7-M only   */

    int zReserved0xd40_0xd90[(0xd90-0xd40)/4];      /* Reserved space */
    /* Offset 0x0d90 */

    struct {
        int Type;                                   /* ARMv7-M only   */
        int Ctrl;                                   /* ARMv7-M only   */
        int RegionNumber;                           /* ARMv7-M only   */
        int RegionBaseAddr;                         /* ARMv7-M only   */
        int RegionAttrSize;                         /* ARMv7-M only   */
    } MPU;                                          /* ARMv7-M only   */
} SCS_t;

/*
 * System Control Space (SCS) Registers
 * in separate section so it can be placed correctly using scatter file
```

```
 */
#pragma arm section zidata="scs_registers"
SCS_t SCS;
#pragma arm section
```

注意

根据具体的实现， SCS 寄存器的内容可能不同。例如，如果未实现操作系统扩展，则可能没有 SysTick 寄存器。

6.3.7 配置单个 IRQ

在中断设置启用寄存器中，每个 IRQ 都有单独的启用位，中断设置启用寄存器是 NVIC 寄存器的组成部分。若要启用或禁用 IRQ，必须将中断设置启用寄存器中相应位分别设置为 1 或 0。有关中断设置启用寄存器的具体信息，请参阅设备的参考手册。

示例 6-20 是一个典型的函数，该函数为第6-33 页的示例 6-19 所示的 SCS 结构启用 IRQ。

示例 6-20 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。

除了具有固定优先级的硬故障、*不可屏蔽中断* (NMI) 和复位，可以使用中断优先级寄存器向每个中断分配优先级。

6.3.8 超级用户调用

与更早版本的 ARM 处理器一样，有一条 SVC 指令可生成 SVC。SVC 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，这个数字通常称为 SVC 编号。在大多数 ARM 处理器上，此编号用于指示要请求的服务。在微控制器架构上，处理器在最初进入异常时，将参数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，必须在 SVC 处理程序的开头实现一小段汇编代码。这样可以确定寄存器的保存位置，从指令中提取 SVC 编号，并将编号和参数指针传递到用 C 编写的处理程序主体中。

示例 6-21 是一个 SVC 处理程序示例。这段代码测试处理器所设置的 EXC_RETURN 值，确定在调用 SVC 时，使用的是哪个堆栈指针。这对于重入 SVC 可能很有用，但在大多数系统上是不必要的，因为在典型系统设计中，只会从使用进程堆栈的用户代码中调用 SVC。在这种情况下，汇编代码可以包含单条 MSR 指令，后跟指向处理程序 C 程序体的尾调用跳转 （B 指令）。

示例 6-21 SVC 处理程序示例

```
__asm void SVCHandler(void)
{
    IMPORT SVCHandler_main
    TST lr, #4
    ITE EQ
    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 6-22 演示如何为很多 SVC 进行不同声明。**__svc** 是编译器关键字，该关键字将函数调用替换为包含指定编号的 SVC 指令。

示例 6-22 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");
}
```

6.3.9 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 6-3 介绍用于配置 SysTick 的四个寄存器。

表 6-3

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	包含倒计时的当前值

配置 SysTick

若要配置 SysTick，请将 SysTick 事件间所需的时间间隔加载到 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 `0x1` 至 `0x00FFFFFF` 之间的值。

如果想使用 SysTick 以固定的时间间隔 （例如 1 毫秒）生成事件，则可以使用 SysTick 校准值寄存器来调整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 表示提供的是参考时钟。

——注意——

对于 Cortex-M1 处理器， TENMS 字段为 0，因为校准值是未知的。

控制和状态寄存器可以通过读取 COUNTFLAG [16] 位或由 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。若要配置 SysTick 以生成中断，请将 TICKINT （SysTick 控制和状态寄存器的位 [1]）设置为 1。还必须在 NVIC 中启用相应的中断，并使用 CLKSOURCE（位 [2]）选择时钟源。如果将此位设置为 1，则选择处理器时钟；如果设置为 0，则选择外部参考时钟。

——注意——

对于 ARMv6-M 处理器，CLKSOURCE 字段为 1，因为 SysTick 始终使用处理器时钟。

通过设置 SysTick 状态和控制寄存器的 [0] 位，可以启用计时器。

第 7 章
调试通信通道

本章介绍如何使用 *调试通信通道* (DCC)。

本章分为以下几节：

- 第7-2 页的*关于调试通信通道*
- 第7-3 页的*目标和主机调试工具之间的 DCC 通信*
- 第7-5 页的*从 Thumb 状态访问*

7.1 关于调试通信通道

ARM® 处理器中的 EmbeddedICE® 逻辑包含调试通信通道。这使得数据能在目标和主机调试工具之间传递。本章介绍目标上运行的程序和主机调试器如何访问 DCC。

为了演示本章中所述的 DCC 的用法，请参阅示例目录 *install_directory*\RVDS\Examples\...\dcc\ 中的示例代码。详细信息可在 readme.txt 中找到。

注意

ARM RealView® Debugger 的最新版本提供了对 DCC 查看器的支持。可以在 RealView Debugger 中运行可执行映像，还可以使用 DCC 查看器来发送和接收目标中的数据。

7.2 目标和主机调试工具之间的 DCC 通信

目标使用 ARM 指令 MCR 和 MRC 作为处理器上的协处理器 14 访问 DCC。图 7-1 演示三个 DCC 寄存器在目标和主机调试工具之间控制和传输数据。

读取寄存器

供目标读取从主机调试工具发出的数据。

写入寄存器

供目标将消息写入主机调试工具。

控制寄存器

为目标和主机调试工具提供握手信息。

对于 ARMv6 以前的处理器：

位 1 （W 位） 在目标可发送数据时清零。

位 0 （R 位） 在有数据可供目标读取时置 1。

对于 ARMv6 及更高版本的处理器：

位 29 （W 位） 在目标可发送数据时清零。

位 30 （R 位） 在有数据可供目标读取时置 1。

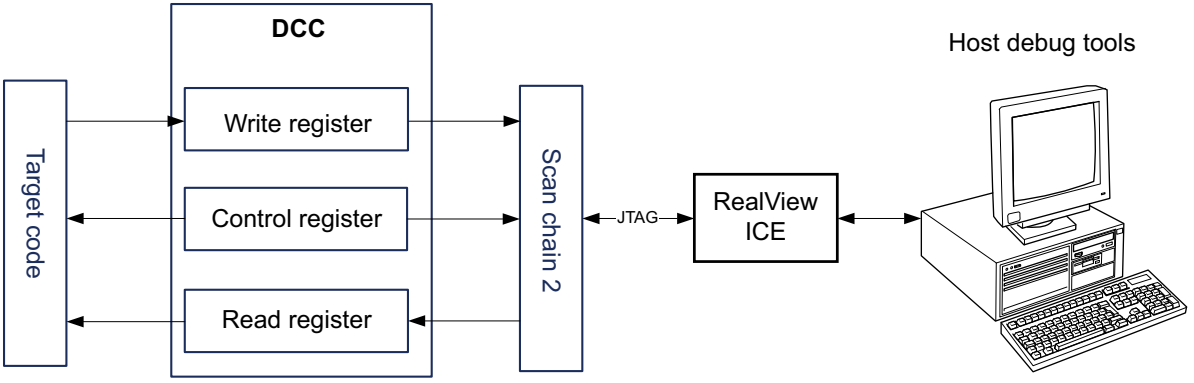

图 7-1 目标和主机调试工具之间的 DCC 通信

注意

有关访问 DCC 寄存器的信息，请参阅处理器的 《技术参考手册》 (*Technical Reference Manual*)。

7.2.1 中断驱动调试通信

示例 7-1 所示的代码片段演示了一个简单的 DCC 例程。从调试工具发出的文本从目标回显，并更改大小写。从该示例 （在 *install_directory*\RVDS\Examples\...\dcc\ 中）生成可执行映像，并使用 JTAG 端口在目标上运行该映像。可以使用 RealView Debugger 中的“Comms Channel/通信通道”视图与目标通信。有关详细信息，请参阅 《RealView Debugger 用户指南》。

示例 7-1 目标和主机调试工具之间的 DCC 通信

```
   AREA DCC, CODE, READONLY
   ENTRY
pollin
   MRC   p14,0,r3,$SCReg,0 ; Read Debug Status and Control Register
   TST   r3, $TestFull
   BEQ   pollin            ; If R bit clear then loop
read
   MRC   p14,0,r0,$DReg,0  ; read word into r0
char_masks
   MOV   r4, #0x20         ; EOR mask to invert case of a char by flipping bit
6
   MOV   r5, #0xC0         ; AND mask to clear all but top 2 bits of each char
changeCase
   TST   r0, r5            ; Check whether character value >0x3F
   EORNE r0, r0, r4        ; If character value >0x3F, flip bit 6 to invert
case
   MOV   r5, r5, LSL #0x8  ; Shift the character mask left by 1 char
   MOVS  r4, r4, LSL #0x8  ; Shift the case inverter pattern left by 1 char
   BNE   changeCase        ; Branch to do the next char
pollout
   MRC   p14,0,r3,$SCReg,0 ; Read Debug Status and Control Register
   TST   r3, $TestEmpty
   BNE   pollout           ; if W set, register still full
write
   MCR   p14,0,r0,$DReg,0  ; Write word from r0
   B     pollin            ; Loop for more words to read
   END
```

如果从 Embedded ICE 逻辑发出的 COMMRX 和 COMMTX 信号连接到中断控制器上，则可以将此类型的轮询示例转换为中断驱动的示例。然后可在中断处理程序中使用读取代码和写入代码。有关编写中断处理程序的信息，请参阅第6-9 页的 *中断处理程序*。

7.3 从 Thumb 状态访问

在体系结构早于 ARM 体系结构 v6T2 的处理器上，不可在处理器处于 Thumb® 状态时使用调试通信通道，因为没有 Thumb 协处理器指令。

对于这个问题，有三种可能的解决方法：

- 将每个轮询例程写入一个 SVC 处理程序，这样在 ARM 或 Thumb 状态下都可以调用该处理程序。进入 SVC 处理程序后立即将处理器转为 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 5 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 8 章
半主机

本章介绍半主机机制。

本章分为以下几节：

8.1 关于半主机

通过半主机，运行在 ARM® 目标上的代码可以使用运行 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。

8.1.1 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、*指令集系统模型* (ISSM)、*实时系统模型* (RTSM)、RealView ICE 或 RealMonitor 时，会执行半主机操作，请参阅第 8-3 页的图 8-1。

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 库中的半主机支持的详细信息，请参阅 《库和浮点支持指南》中的第 2 章 *C 和 C++ 库*。

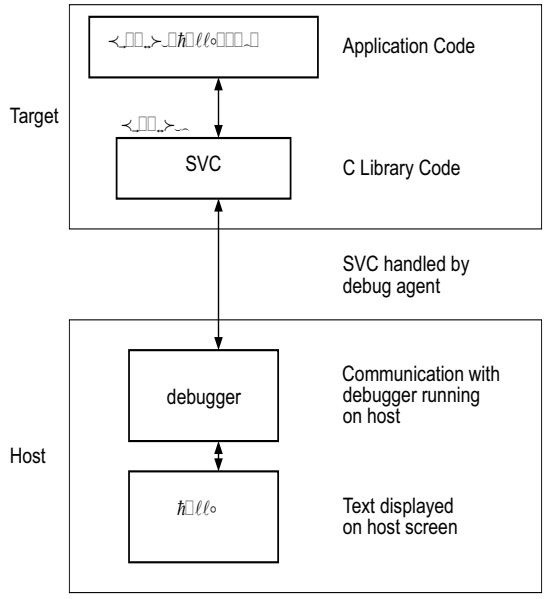

图 8-1 半主机概述

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 ARMv6-M 或 ARMv7-M （如 Cortex™-M1 或 Cortex-M3 处理器）进行编译，请使用 BKPT 指令来实现半主机。

8.1.2 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 ARMv6-M 或 ARMv7-M 进行编译，请使用 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 ARM 状态和 Thumb 状态下（不包括 ARMv6-M 和 ARMv7-M）。不能确保 ARM 或第三方的所有调试目标都具有这种行为。

BKPT 0xAB　对于 ARMv6-M 和 ARMv7-M，仅限于 Thumb 状态。

另请参阅*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

8.2 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。

8.2.1 RealView ARMulator ISS

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

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

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

8.2.2 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》(*RealView ICE and RealView Trace User Guide*)。

8.2.3 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>
```

8.2.4 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。

8.3 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。

8.3.1 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 时，用户堆栈指针 R13_USR 会复制到超级用户模式堆栈指针 R13_SVC，并设置当前 CPSR 中的 I 位和 F 位，从而禁用常规和快速中断。

注意

如果使用 RealView ARMulator ISS 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会复制到超级用户模式堆栈指针

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，复制用户堆栈指针会导致丢失超级用户模式堆栈。返回到用户例程会将 R13_SVC 恢复为超级用户模式堆栈值，但是应用程序不能使用此堆栈。

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。

8.3.2 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 8-1 和表 8-2 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 8-1 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 8-2 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 8-2 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

在第8-8 页的表 8-2 中，值旁边的 * 表示该值不受 ARM 调试器支持。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。

8.3.3 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 包含一个指向单字参数块的指针：

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

8.3.4 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8-11 页的*SYS_ELAPSED (0x30)* 和第8-19 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

8.3.5 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双字数据块：

字 1 双字值中的最低有效字

字 2 最高有效字

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）。

8.3.6 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。

8.3.7 SYS_FLEN (0x0C)

返回指定文件的长度。

入口

进入时，R1 包含一个指向单字参数块的指针：

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

8.3.8 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双字数据块：

字 1 指向一个缓冲区的指针，该缓冲区至少为字 2 中指定的大小

字 2 缓冲区长度，以字节为单位的

返回值

退出时：

- 寄存器 R1 指向一个双字数据块：

  字 1 命令行中指向以 null 终止的字符串的指针

  字 2 字符串的长度。

  调试代理可能对可传送的最大字符串长度有所限制。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

8.3.9 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 RealView ARMulator ISS，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 库中的内存管理的详细信息，请参阅 《库和浮点支持指南》中第2-64 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 8-1。

示例 8-1

```
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。

8.3.10 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 包含一个指向单字数据块的指针：

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

8.3.11 SYS_ISTTY (0x09)

检查文件是否连接到交互设备。

入口

进入时，R1 包含一个指向单字参数块的指针：

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

8.3.12 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 包含一个指向三字参数块的指针：

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 8-3 列出了该整数的有效值，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 8-3 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。

8.3.13 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。

8.3.14 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。

8.3.15 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 包含一个指向双字参数块的指针：

字 1 指向以 null 终止的字符串，该字符串提供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

8.3.16 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 指向旧文件名的指针

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

两个字符串都是以 null 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

8.3.17 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 包含一个指向双字数据块的指针：

字 1 可搜索的文件对象的句柄

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。

8.3.18 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 包含一个指向双字参数块的指针：

字 1 指向要传递给主机命令行解释程序的字符串

字 2 字符串的长度

返回值

退出时，R0 包含返回状态。

8.3.19 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 0。

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。

8.3.20 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 RealView ARMulator ISS、 ISSM、 RTSM 或 RealView ICE 配置无关。

入口

没有参数。

返回值

退出时， R0 包含秒数。

8.3.21 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 包含一个指向三字参数块的指针：

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。

8.3.22 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

8.3.23 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。

8.3.24 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。

8.4 调试代理交互 SVC

除了第8-7 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8-7 页的*angel_SWIreason_EnterSVC (0x17)*
- 第8-8 页的*angel_SWIreason_ReportException (0x18)*