

Cortex-M3 权威指南

Joseph Yiu 著

宋岩 译

www.ouravr.com热心网友 校对

网络版初稿的译序

我接触 ARM 的历史约 4 年，早期是很欣赏这类处理器，到了后来切身使用它们的机会越来越多，慢慢地有了感觉，也更加喜欢了。在偶然听说 Cortex-M3 后，我就冥冥地感到它不寻常。只是因为其它工作一直没有去了解它，直到今年初才进一步学习，很快就觉得相知恨晚。当时只能看 ARM 官方的重量级资料，在看到这本书的英文原稿后，更感觉被电到了一样，于是突然有了把它翻译成中文的冲动。经过累计约 150 个小时的奋战，终于有了此译稿。在翻译过程中，我始终采用下列指导思想：

1. 尽量使用短句，并且把句子口语化。我认为高深的道理不一定要用高级的语法句型才能表达。想想看，即使是几位博士互相聊天讨论一个课题，也还是使用口语吧，而且火花往往就是在这种讨论中产生呢！
2. 多用修辞方法，并且常常引用表现力强的词汇——甚至包括网络用语和脍炙人口的歌词。另外，有时会加工句子，使得风格像是对话。这样做的目的是整个文风更鲜活——有点像为写出高分作文而努力的样子。这点可能与很多学术著作的“严肃、平实”文风不同，也是一次大胆的尝试。还希望读者不吝给予反馈。
3. 在“宏观”上直译，在“微观”上意译。英语不仅单一句子的语法和汉语不同，并且句子的连贯方式也与汉语不同。因此在十几个到几十个单词的范围内，我先把它们翻译成脑子里的“中间语言”，再把中间语言翻译成汉语。这样，就最大地避免了贻笑大方的“英式汉语”。
4. 有些术语名词不方便翻译成汉语，或者目前的翻译方式不统一，或者与其它术语翻译的结果很接近，如 **error** 和 **fault**，就只能用英语意会。此时我就保留英文单词，相信这样比硬生生地翻译成汉语还好。这些词汇主要是:**retarget**, **fault**, **region** 等。另外，英文中有一个很能精练表达“两者都”意思的单词及其用法：“**both**”，我也常常保留之。
5. 图表对颜色的使用比较丰满，尤其是比较大型的插图，相信这样能帮助读者分析和理解。插图是从原图直接复制的，因此矢量图变成了位图，无法再适应任何比例的缩放。不过，我在复制原图时，把原图以 200% 的比例放大，从而提高了图片的质量。
6. 我在很多地方加了译注。比较短的译注就直接以“()”加在文字后面。比较长的译注则为它开出一个“文字池”，放到相应的“**.text**”后面并与之相临。早期的译注多用于解释一些不是很广为人知的术语，后期的译注则更多是我认为有必要补充的内容，包括读者在阅读过程中可能会产生的问题，容易混淆的概念，深入理解等。
7. 我对少量自然段作了改编。也有个别部分译自 ARM 提供的权威文档。

本书的翻译工作在 40% 进度的时候是最困难的时期，有一种好像长跑中遇到了所谓“极限”的感觉。望着距离掉下去还有那么高的滚动条，甚至都有停住的自我暗示了。那天刚好是哀悼日的第一天，我本来情绪很低沉，但在我看到默哀完毕，天安门广场上排山倒海般地呐喊“中国加油”时，我突然有了强烈的共振感觉，那是一种热泪盈眶的激动和感叹，甚至觉得他们就是在鼓励我！让我一下子振作起来，找回了比刚开始还要强烈的干劲，并且更加信心满满。这种精神力量一直推动我翻译完最后一个字，并且还有“余勇可贾”的快感^_^

整个翻译的时间跨度是在 2008.05.10-2008.06.07，共计 28 天。不知这是否算得上很“仓促”。想必有很多句式还能改得更好，甚至还有错别字等低级错误。我使用了五笔输入法，可能错别字会错得很离奇，不过肯定逃不过读者雪亮的双眼的。希望读者在发现错误后批评指正。反馈地址是：rock.song@hotmail.com，也可以通过 QQ:9471202/9312500。

本译稿草稿完成后，我交给几位好友去试读和审校，得以揪出了很多大大小小的 bugs。他们是：浮云，土豆波，美眉 Y，李天后。在这里以点名表示感谢！

宋 岩 2008.07.02

出版致谢 (按时间顺序)

在本译文的酝酿期和翻译的全过程，我的兄弟魏国平一直鼓励我，相信这是对社会有益的事，并且给从精神和物质上给我打气和支招。这只是他八年来与我兄弟之间的一段小掠影，在此感谢之情已难以言表！他的能力也是我非常佩服的，他仅在加入国内视频监控业的老大“海康威视”一年后，就在只有二人当选的最佳员工中榜上有名。

在本书翻译的后期，我告知了父母。虽然他们知道这会冲击我的工作，但依然支持我继续下去。对父母的感谢早已超出任何语言和行动所能及的程度，因此不多提也罢。

本译文完成后，我交给了 4 位好友去试读。在这里为他/她们“正名”：黄强，杨波，马铭遥，李武华。他们不仅评估了译文在“文笔”上的质量，还找出了一些错误。尤其是黄强，他和他的“先锋突击队”率先“软硬兼施”地实践过基于 CM3 的 STM32 单片机，并且从人文的层面上对各种读者的口味都很了解，因此为我提出最多的技术上和文风上的建议。在内测期和译文发表后，我多次和他一起讨论如何改进。另外，他也是带我认知 CM3 的第一人。

后来，我的恩师吴建德在看过译文后，就组织他带领的研究生去在新项目中使用基于 CM3 的单片机，并且研究在电源与拖动控制器中，从 16 位 DSP 转到 CM3 上的一些课题，还鼓励我继续做类似的工作。他和实验室的精英们以实际的先锋行动给了我很大的鼓励。

在“内部测试”期间和网上发表后，我的好友高明和、李小林、梁纪荣、沈争、王金成、杨福双、叶枫和于艳良都支持我发表，并且在我工作之余为我计划未来的蓝图，王金成还经常是我吐苦水时的受害者。他们的友情扫除了我在工作中的枯燥，并且让我更加信心满满！

在经过“内部测试版”后，本译文的初稿是发表在 www.ouravr.com 的论坛上的。阿莫站长在帖子发表后的第一时间（已经是凌晨）就置“酷”，后来又置顶，再后来还专门开出一个 Cortex-M3 技术讨论区，并任命我为第一任版主。阿莫给了我一个大舞台，这也是一个大家畅所欲言、求医问道、展现自己闪亮的大舞台。这里自由开放，甚至不需要登录就可以下载资源！在这里再次向阿莫站长致敬！

帖子发表后，ouravr 的很多热心网友回贴鼓励我，顶我的帖子直到顶到置顶。难能可贵的是 watercat 还第一个警示我不要乐昏了头，提出我的这种翻译方式和文风会面临的风险。这在我校对的过程中起了很多指导作用。

帖子发表后才两天，北航出版社的胡晓柏主任就联系我，与我讨论出版的事宜。胡主任其实早就慧眼识中了这本书的原版，从而北航出版社购得本书的简体版版权。于是，本书中文版的出版简直是大路通天般的顺畅！我以前没有出版过书，胡主任一直耐心地解答我的每个疑问，无论在邮件中还是电话里，都平易近人。在商务上，和胡主任与北航出版社的合作也轻松愉快，比我谈的绝大多数项目都容易得多。推荐大家有好作品就去找他，最近的一本热销书《匠人手记》也是胡主任“执着”的成果。

后来我得知，是周立功先生在看到译文后，推荐使用这个译文出版的。后来周公还来信鼓励我再接再厉。我以前读过周公的一篇讲学习点 RTOS 的文章后曾热血沸腾，在此对周公的大度和鼓励我非常感激。

后来在上海沁科的王永虹经理主办的一次活动中，我和英蓓特的周麒相叙多时。我希望能由 ARM 中国的总裁谭军先生写序，经过周麒的表奏后，谭总欣然同意了。这样落地有声的肯定，我在继续前进的路上还有什么好犹豫的！我现在觉得我加入了一个充满战斗力并且温暖和谐的精英先锋队，为了让 32 位单片机新生代的春风化雨早日润遍人间而激情战斗！

谭总把此事告知了原作者，他的中文名是姚文祥先生。他也很高兴地重新写了中文版前言，并且还把今年夏天 Cortex-M3 最新修订版的更新内容发给我，以使此书与时俱进。姚先生的敬业和热情是令人肃然起敬的！

感谢纸版的读者您。您的阅读就是对我工作的肯定和鼓励，并且我也因为祖国的嵌入式领域在向次世代挺进的过程中，又多了一位生力军而兴奋不已！

原序

谁是最节能，最擅长把好钢用在刀刃上的人？要让我说，我一定得表一表单片机的开发者。他们使出浑身解数写出精妙玲珑的代码，把单片机点点滴滴的力量汇集起来，让它如同涌泉一般尽情地迸发，灌溉滋养着各行各业。是什么灵丹妙药赐予了他们这么神奇的力量？天资聪慧是主观方面。在客观方面，除了有好的处理器之外，还要配合好的开发环境和工具链。也正出于此，在设计ARM7TDMI处理器时，ARM的工具链工程师们和CPU设计师们强强联手，为了让它的内部结构更优化、更精练、更到位而并肩奋战了很多日日夜夜，终于有了ARM7TDMI的无限辉煌，并且久经岁月的洗礼依旧光芒绽放。

珠联璧合的最新果实，是破茧而出的ARM Cortex-M3处理器。这个小尤物，处处闪耀着ARM体系结构最激动人心的新突破。它基于最新最好的32位ARMv7架构——这个架构支持高度成功的Thumb-2指令集，还有很多时尚、前卫甚至崭新的特性，充满了新生代的气息。它在很好、很强大的同时，编程模型却变得更加清新爽洁了。不管你是祖国的花朵、是人民教师、还是精明的商人，也无所谓是新手还是骨灰级玩家，Cortex-M3都将尽情展现它的秀外慧中，带给你喜出望外的收获和“激活”！

ARM嵌入式解决方案主任

Wayne Lyons

前 言

不管你是做软件的还是做硬件的，只要相中了ARM的Cortex-M3处理器，这本书就是为你而写。以前Cortex-M3的资料只有两个大部头，分别是：

- 《Cortex-M3技术参考手册》 (Cortex-M3 Technical Reference Manual, 简称 Cortex-M3 TRM)
- 《ARMv7-M应用程序级架构参考手册》 (ARMv7-M Application Level Architecture Reference Manual)

虽然这它俩差不多是权威到“古文观止”级的，但实在是太深入了，以至于对新手来说那简直就是天书。本书则是一个精简版，并且叙述的前后更有条理。目标读者包括：一线程序员，嵌入式产品设计师，片上系统 (SoC) 工程师，嵌入式系统发烧友，学院研究员，还包括所有涉猎过单片机和微处理器领域，慧眼识珍看中了Cortex-M3的人民大众们。

本书要给Cortex-M3的架构做一个简介，浏览一下指令系统，写几个段代码练练手，说一些硬件特性，再表一表该处理器精深的调试系统。本书还给出了应用程序范例，手把手地教你使用开发工具，包括ARM的工具和GNU的工具链。如果你以前是ARM7TDMI的玩家，正准备着升级装备到Cortex-M3，本书也非常解渴，里面讲述了两者的不同，以及鸟枪换炮的升级过程。

致谢

我要感谢下面的人们，他们有人帮我检查了本书，还有人提供了我建议和反馈

Alan Tringham, Dan Brook, David Brash, Haydn Povey, Gary Campbell, Kevin McDermott, Richard Earnshaw, Samin Ishtiaq, Shyam Sadasivan, Simon Axford, Simon Craske, Simon Smith, Stephen Theobald and Wayne Lyons.

我还要感谢CodeSourcery为我提供技术支持，以及Luminary Micro提供封面图（英文原书）。当然，Elsevier的朋友们为出版本书作出了专业的工作，在这里我也要一并感谢。

最后，特别感谢读者您，以及Peter Cole与Ivan Yardley，他们鼓励我写这本书。

缩略语

缩写代号	含义
ADK	AMBA设计套件
AHB	先进高性能总线
AHB-AP	AHB访问端口
AMBA	先进单片机总线架构
APB	先进外设总线
ARM ARM	ARM架构参考手册
ASIC	行业领域专用集成电路
ATB	先进跟踪总线
BE8	字节不变式大端模式
CPI	每条指令的周期数
CPU	中央处理单元
DAP	调试访问端口
DSP	数字信号处理器 / 数字信号处理
DWT	数据观察点及跟踪
ETM	嵌入式跟踪宏单元
FPB	闪存地址重载及断点
FSR	Fault状态寄存器
HTM	CoreSight AHB跟踪宏单元
ICE	在线仿真器
IDE	集成开发环境
IRQ	中断请求 (通常是指外部中断的请求)
ISA	指令系统架构
ISR	中断服务例程
ITM	仪器化跟踪宏单元
JTAG	连结点测试行动组 (一个关于测试和调试接口的标准)
JTAG-DP	JTAG调试端口
LR	连接寄存器
LSB	最低有效位
LSU	加载/存储单元
MCU	微控制器单元 (俗称单片机)
MMU	存储器管理单元
MPU	存储器保护单元
MSB	最高有效位
MSP	主堆栈指针
NMI	不可屏蔽中断
NVIC	嵌套向量中断控制器
OS	操作系统
PC	程序计数器
PSP	进程堆栈指针
PPB	私有外设总线

本书大面积地使用了如下的排版字体约定：

- 普通汇编代码

MOV R0, R1 ; 把寄存器R1中的数据移至R0

- 以模式化语法表示的汇编代码——编程时必须使用真实的寄存器名字

MRS <reg>, <special_reg> ;

- C 程序代码

for (i=0;i<3;i++) { func1(); }

- 伪代码

if (a > b) { ... }

- 数值：

1. 4'hC, 0x123 都表示16进制数

2. #3表示数字3 (e.g., IRQ #3 就是指3号中断)

3. #immed_12表示一个12位的立即数

4. 寄存器位。通常是表示一个位段的数值，例如

bit[15:12] 表示位序号从15往下数到12，这一段的数值。

- 寄存器访问类型

1. R 表示只读

2. W表示只写

3. RW 表示可读可写 (前3条好像地球人都知道)

4. R/Wc 表示可读，但是写访问将使之清 0

其它参考资料

1. *Cortex-M3 Technical Reference Manual (TRM) (Cortex-M3技术参考手册)*

请从www.arm.com/documentation/ARMProcessor_Cores/index.html下载

2. *ARMv7-M Architecture Application Level Reference Manual(ARMv7-M应用级架构参考手册)*

请从www.arm.com/products/CPUs/ARM_Cortex-M3_v7.html下载

3. *CoreSight Technology System Design Guide (CoreSight技术系统设计指导)*

请从www.arm.com/documentation/Trace_Debug/index.html下载

4. *AMBA Specification (AMBA 规格书)*

请从www.arm.com/products/solutions/AMBA_Spec.html下载

5. *AAPCS Procedure Call Standard for the ARM Architecture(AAPCS ARM 架构过程调用标准)*

请从www.arm.com/pdfs/aapcs.pdf下载

6. *RVCT 3.0 Compiler and Library Guide(RVCT 3.0编译器及库向导)*

请从www.arm.com/pdfs/DUI0205G_rvct_compiler_and_libraries_guide.pdf下载

7. *ARM Application Note 179: Cortex-M3 Embedded Software Development(ARM应用笔记179: Cortex-M3嵌入式软件开发)*

请从www.arm.com/documentation/Application_Notes/index.html下载

译注：这些资料都不是什么“省油的灯”，阅读起来可能比较吃力，条理性也未必很明显。因此不必强求自己一下子读完。最好把它们当作后备参考资料，遇到疑难时再诉诸于它们找答案。另外，3号和4号资料更倾向于芯片设计师的口味。

介绍

- ARM Cortex-M3处理器初探
- ARM的各种架构版本
- 指令集的开发
- Thumb-2指令集架构(ISA)
- Cortex-M3的舞台
- 本书组织
- 深入研究用的读物

1.1 ARM Cortex-M3 处理器初探

单片机市场的规模可以用“巨无霸”来形容，预计到2010时每年能有20G片的出货量。世界各地的器件供应商纷纷亮出自己的得意之作，他们提供的器件和架构也是各具特色。业界内部可谓是百花齐放，热闹非凡，好戏不断。各行各业对单片机能力的要求也一直“得寸进尺”，而且还要马儿跑，又要马儿不吃草——处理器必须在不怎么增加主频和功耗的条件下干更多的活儿。另一方面，处理器之间的互连也在加深，看这一串串熟悉的字眼：串口，USB，以太网，无线数传……处理器如欲支持这些数据通道，就必须在片上塞进更多的外设。软件方面的情况也如出一辙：应用程序的功能一直在花样翻新，性能需求也是变本加厉：更高的运算速度，更硬的实时能力，更多的功能模块，更炫的图形界面，……所有这些要求单片机都得照单全收。在这个大环境下，ARM Cortex-M3处理器，作为Cortex系列的处女作，为了让32位处理器入主作庄单片机市场，轰轰烈烈地诞生了！由于采用了最新的设计技术，它的门数更低，性能却更强。许多曾经只能求助于高级32位处理器或DSP的软件设计，都能在CM3上跑得很快很欢。嵌入式处理器市场正在32位化，相信用不了多久，CM3就一定会在这美丽新世界中脱颖而出。比当年8051推动整个业界还有过之而无不及，再次放飞工程师们的梦想，让深埋于心底多年的夙愿迎来dreams come true的激动！

CM3的招牌功夫包括：

- 性能强劲。在相同的主频下能做处理更多的任务，全力支持劲爆的程序设计。
- 功耗低。延长了电池的寿命——这简直就是便携式设备的命门（如无线网络应用）。
- 实时性好。采用了很前卫甚至革命性的设计理念，使它能极速地响应中断，而且响应中断所需的周期数是确定的。
- 代码密度得到很大改善。一方面力挺大型应用程序，另一方面为低成本设计而省吃俭用。
- 使用更方便。现在从8位/16位处理器转到32位处理器之风刮得越来越猛，更简单的编程模型和更透彻的调试系统，为与时俱进的人们大大减负。
- 低成本的整体解决方案。让32位系统比和8位/16位的还便宜，低端的Cortex-M3单片机甚至还卖不到1美元。
- 遍地开花的优秀开发工具。免费的，便宜的，全能的，要什么有什么。

基于Cortex-M3内核的处理器已渐成气候，以处处满溢的先进特性力压群芳。而且架构师们还在不停地求索降低成本的出路，同时很多组织也在尝试着实现“器件聚合”（device aggregation），使

一个单一的小强芯片可以抵得上以前3、4块传统的单片机。

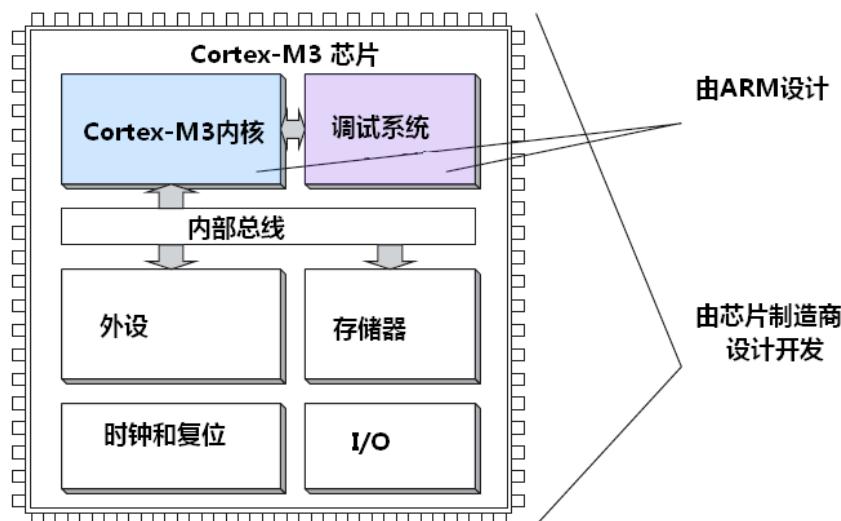
降低成本还有一招，就是使基础代码在所有系统中都可以重用，至少要方便移植。CM3的内核架构非常精工细作，使它与C语言成为了一个梦幻绝配。优质的C程序代码三下五除二就可以移植并重用，使升级和移植一下子从拦路虎变成了纸老虎。

值得一提的是，CM3并不是第一个被拿去做万金油型处理器的内核。那廉颇虽老却依然骁勇的ARM7/ARM9处理器，在通用嵌入式处理器市场中德高望重，至今拥有无数铁杆粉丝。半导体业界的群英们，像NXP（philips）、TI、Atmel、OKI、ST等，都以ARM为内核，做出了各自身怀绝技的32位MCU。ARM7作为最受欢迎的32位嵌入式处理器，被载入了亮煌煌的几页史册——每年超过10亿片出货量，为各行各业的嵌入式设备中都找得到它们的身影。

CM3作为ARM7的后继者，大刀阔斧地改革了设计架构。从而显著地简化了编程和调试的复杂度，处理能力也更加强大。除此之外，CM3还突破性地引入了很多时尚的甚至崭新的技术，专门满足单片机应用程序的需求。比如，服务于“使命-关键”应用的不可屏蔽中断，极度敏捷并且拥有确定性的嵌套向量中断系统，原子性质的位操作，还有一个可选的内存保护单元。这些令人惊艳和振奋的新特性，让老的ARM玩家们再次找到“初恋”时烈焰迸发的感觉，也使萍水相逢就有激爽触电般的体验！相信读者一旦有机会用到了它，就会为它的秀外慧中而赞叹，爱不释手！

1.1.1 从 Cortex-M3 处理器内核到基于 Cortex-M3 的 MCU

Cortex-M3处理器内核是单片机的中央处理单元（CPU）。完整的基于CM3的MCU还需要很多其它组件。在芯片制造商得到CM3处理器内核的使用授权后，它们就可以把CM3内核用在自己的硅片设计中，添加存储器，外设，I/O以及其它功能块。不同厂家设计出的单片机会有不同的配置，包括存储器容量、类型、外设等都各具特色。本书主讲处理器内核本身。如果想要了解某个具体型号的处理器，还需查阅相关厂家提供的文档。



1.1.2 ARM 及 ARM 架构的背景

1.1.2.1 一路走来

让我们回顿一下ARM的进化史，你会知道为什么会有品种如此之多的ARM处理器和ARM架构。ARM在1990年成立，当初的名字是“Advanced RISC Machines Ltd.”，当时它是三家公司的合资——

它们分别是苹果电脑, Acorn电脑公司, 以及VLSI技术(公司)。在1991年, ARM推出了ARM6处理器家族, VLSI则是第一个吃螃蟹的人。后来, 陆续有其它巨头: 包括TI, NEC, Sharp, ST等, 都获取了ARM授权, 它们真正地把ARM处理器大面积地铺开, 使得ARM处理器在手机, 硬盘控制器, PDA, 家庭娱乐系统以及其它消费电子中都大展雄才。

现如今, ARM芯片的出货量每年都比上一年多20亿片以上。不像很多其它的半导体公司, ARM从不制造和销售具体的处理器芯片。取而代之的, 是ARM把处理器的设计授权给相关的商务合作伙伴, 让他们去根据自己的强项设计具体的芯片, 这些伙伴可都是巨头啊。基于ARM低成本和高效的处理器设计方案, 得到授权的厂商生产了多种多样的处理器、单片机以及片上系统(SoC)。这种商业模式就是所谓的“知识产权授权”。

除了设计处理器, ARM也设计系统级IP和软件IP。为了挺它们, ARM开发了许多配套的基础开发工具、硬件以及软件产品。使用这些工具, 合作伙伴可以更加舒心地开发他们自己的产品。

1.2 ARM 的各种架构版本

ARM十几年如一日地开发新的处理器内核和系统功能块。这些包括流行的ARM7TDMI处理器, 还有更新的高档产品ARM1176TZ(F)-S处理器, 后者能拿去做高档手机。功能的不断进化, 处理水平的持续提高, 年深日久造就了一系列的ARM架构。要说明的是, 架构版本号和名字中的数字并不是一码事。比如, ARM7TDMI是基于ARMv4T架构的(T表示支持“Thumb指令”); ARMv5TE架构则是伴随着ARM9E处理器家族亮相的。ARM9E家族成员包括ARM926E-S和ARM946E-S。ARMv5TE架构添加了“服务于多媒体应用增强的DSP指令”。

后来又出了ARM11, ARM11是基于ARMv6架构建成的。基于ARMv6架构的处理器包括ARM1136J(F)-S, ARM1156T2(F)-S, 以及ARM1176JZ(F)-S。ARMv6是ARM进化史上的一个重要里程碑: 从那时候起, 许多突破性的新技术被引进, 存储器系统加入了很多的崭新的特性, 单指令流多数据流(SIMD)指令也是从v6开始首次引入的。而最前卫的新技术, 就是经过优化的Thumb-2指令集, 它专为低成本的单片机及汽车组件市场。

ARMv6的设计中还有另一个重大的决定: 虽然这个架构要能上能下, 从最低端的MCU到最高端的“应用处理器”都通吃, 但不能因此就这也会, 那也会, 但就是都不精。仍须定位准确, 使处理器的架构能胜任每个应用领域。结果就是, 要使ARMv6能够灵活地配置和剪裁。对于成本敏感市场, 要设计一个低门数的架构, 让她有极强的确定性; 另一方面, 在高端市场上, 不管是要有丰富功能的还是要有高性能的, 都要有拿得出手的好东西。

最近的几年, 基于从ARMv6开始的新设计理念, ARM进一步扩展了它的CPU设计, 成果就是ARMv7架构的闪亮登场。在这个版本中, 内核架构首次从单一款式变成3种款式。

- 款式A: 设计用于高性能的“开放应用平台”——越来越接近电脑了
- 款式R: 用于高端的嵌入式系统, 尤其是那些带有实时要求的——又要快又要实时。
- 款式M: 用于深度嵌入的, 单片机风格的系统中——本书的主角。

让我们再进距离地考察这3种款式:

- 款式A (ARMv7-A): 需要运行复杂应用程序的“应用处理器”^[译注1]。支持大型嵌入式操作系统(不一定实时——译注), 比如Symbian(诺基亚智能手机用), Linux, 以及微软的Windows CE和智能手机操作系统Windows Mobile。这些应用需要劲爆的处理性能, 并且需要硬件MMU实现的完整而强大的虚拟内存机制, 还基本上会配有Java支持, 有时还要求一个安全程序执行环境(用于电子商务——译注)。典型的产品包括高端手机和手持仪器, 电子钱包以及金融事务处理机。

[译注1]: 这里的“应用”尤指大型应用程序, 像办公软件, 导航软件, 网页浏览器等。这些软件的使用习惯和开发模式都很像PC上的软件, 但是基本上没有实时要求。

- 款式R (ARMv7-R): 硬实时且高性能的处理器。标的是高端实时^[注1]市场。那些高级的玩意,

像高档轿车的组件，大型发电机控制器，机器手臂控制器等，它们使用的处理器不但要很好很强大，还要极其可靠，对事件的反应也要极其敏捷。

- 款式M (ARMv7-M)：认准了旧世代单片机的应用而量身定制。在这些应用中，尤其是对于实时控制系统，低成本、低功耗、极速中断反应以及高处理效率，都是至关重要的。

Cortex系列是v7架构的第一次亮相，其中Cortex-M3就是按款式M设计的。

[注1]：通用处理器能否胜任实时系统的控制，常遭受质疑，并且在这方面的争论从没停止过。从定义的角度讲，“实时”就是指系统必须在给定的死线（deadline，亦称作“最后期限”）内做出响应。在一个以ARM处理器为核心的系统中，决定能否达到“实时”这个目标的，有很多因素，包括是否使用“实时操作系统”，中断延迟，存储器延时，以及当时处理器是否在运行更高优先级的中断服务例程。

本书认准了Cortex-M3就一猛子扎下去。到目前为止，Cortex-M3也是款式M中被抚养成人的独苗。其它Cortex家族的处理器包括款式A的Cortex-A8（应用处理器），款式R的Cortex-R4（实时处理器）。

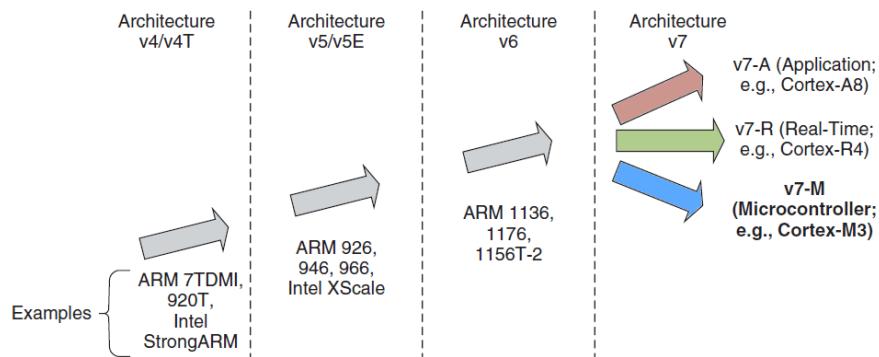


图1.2 ARM处理器架构进化史

ARMv7-M的私房秘密都记录在《The ARMv7-M Architecture Application Level Reference Manual》中（本书也讲了很多“System Level”的内容——译注），ARM已经将其公开。《Cortex M3 Technical Reference Manual》中则记录了实现v7-M时的很多细节和花絮。又因为v7M中列出的指令有一些是可选的，而CM3裁掉了一部分，所以在这个文档中重新列出了被CM3支持的指令集。

1.2.1 处理器命名法

以前，ARM使用一种基于数字的命名法。在早期（1990s），还在数字后面添加字母后缀，用来进一步明细该处理器支持的特性。就拿ARM7TDMI来说，T代表Thumb指令集，D是说支持JTAG调试(Debugging)，M意指快速乘法器，I则对应一个嵌入式ICE模块。后来，这4项基本功能成了任何新产品的标配，于是就不再使用这4个后缀——相当于默许了。但是新的后缀不断加入，包括定义存储器接口的，定义高速缓存的，以及定义“紧耦合存储器（TCM）”的，于是形成了新一套命名法，这套命名法也是一直在使用的。

表1.1 ARM处理器名字

处理器名字	架构版本号	存储器管理特性	其它特性
ARM7TDMI	v4T		
ARM7TDMI-S	v4T		
ARM7EJ-S	v5E		DSP,Jazelle ^[译注3]
ARM920T	v4T	MMU	
ARM922T	v4T	MMU	
ARM926EJ-S	v5E	MMU	DSP,Jazelle
ARM946E-S	v5E	MPU	DSP
ARM966E-S	v5E		DSP
ARM968E-S	v5E		DMA,DSP
ARM966HS	v5E	MPU (可选)	DSP
ARM1020E	v5E	MMU	DSP
ARM1022E	v5E	MMU	DSP
ARM1026EJ-S	v5E	MMU 或 MPU ^[译注2]	DSP, Jazelle
ARM1136J(F)-S	v6	MMU	DSP, Jazelle
ARM1176JZ(F)-S	v6	MMU+TrustZone	DSP, Jazelle
ARM11 MPCore	v6	MMU+多处理器缓存支持	DSP
ARM1156T2(F)-S	v6	MPU	DSP
Cortex-M3	v7-M	MPU (可选)	NVIC
Cortex-R4	v7-R	MPU	DSP
Cortex-R4F	v7-R	MPU	DSP+浮点运算
Cortex-A8	v7-A	MMU+TrustZone	DSP, Jazelle

[译注2]：Jazelle是ARM处理器的硬件Java加速器。

[译注3]：MMU，存储器管理单元，用于实现虚拟内存和内存的分区保护，这是应用处理器与嵌入式处理器的分水岭。电脑和数码产品所使用的处理器几乎清一色地都带MMU。但是MMU也引入了不确定性，这有时是嵌入式领域——尤其是实时系统不可接受的。然而对于安全关键(safety-critical)的嵌入式系统，还是不能没有内存的分区保护的。为解决矛盾，于是就有了MPU。可以把MPU认为是MMU的功能子集，它只支持分区保护，不支持具有“定位决定性”的虚拟内存机制。

到了架构7时代，ARM改革了一度使用的，冗长的、需要“解码”的数字命名法，转到另一种看起来比较整齐的命名法。比如，ARMv7的三个款式都以Cortex作为主名。这不仅更加澄清并且“精装”了所使用的ARM架构，也避免了新手对架构号和系列号的混淆。例如，ARM7TDMI并不是一款ARMv7的产品，而是辉煌起点——v4T架构的产品。

1.3 指令集的发展

为了增强和扩展指令系统的能力而奋斗，多少年来这一直是ARM锲而不舍的精神动力。

由于历史原因（从ARM7TDMI开始），ARM处理器一直支持两种形式上相对独立的指令集，它们分别是：

- 32位的ARM指令集。对应处理器状态: ARM状态
- 16位的Thumb指令集。对应处理器状态: Thumb状态

可见,这两种指令集也对应了两种处理器执行状态。在程序的执行过程中,处理器可以动态地在两种执行状态之中切换。实际上,Thumb指令集在功能上是ARM指令集的一个子集,但它能带来更高的代码密度,给目标代码减肥。这对于要勒紧裤腰带的应用还是很经济的。

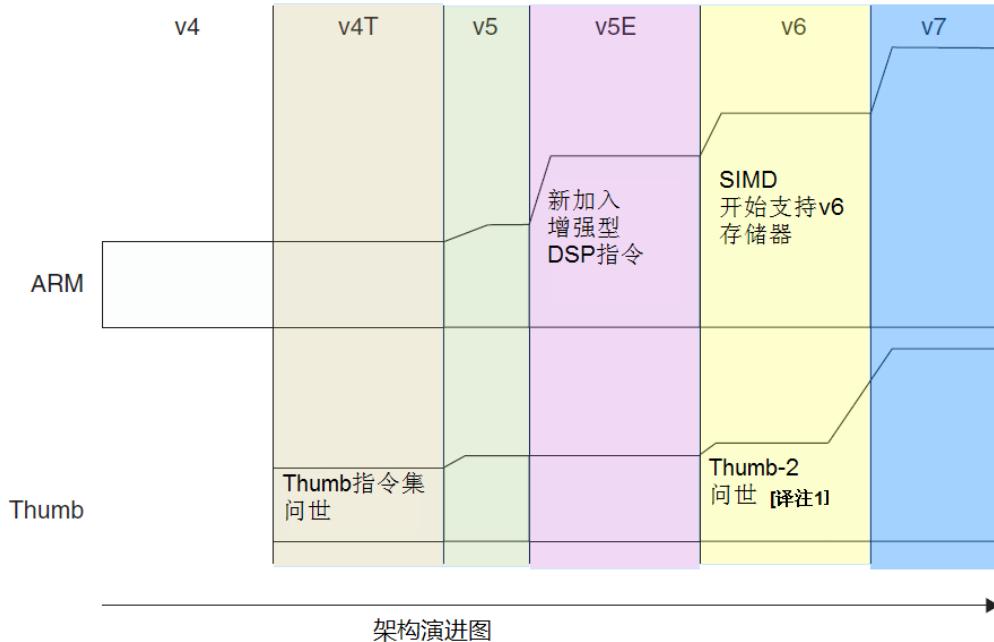


图1.3 指令集演进图

译注1: 原书把Thumb-2的问世时间放到v7中,但根据其它权威文献的记录,似有误,应在v6中问世。
(如《ARM and Thumb-2 Instruction Set Quick Reference Card》中的描述)

随着架构版本号的更新,新好指令不断地加入ARM和Thumb指令集中。附录2中给出的内容,就是Thumb指令在架构进化过程中的改变记录。Thumb-2是2003年盛夏的果实,它是Thumb的超集,它支持both 16位和32位指令。

指令集的详细说明在《The ARM Architecture Reference Manual》(简称为ARMARM)中。每次ARM出新版本时此手册都有更新。到了v7时,因为以前的单一架构被分成了3个款式,这个规格书也就跟着变成了3本。为Corex-M3的ARMv7-M架构而写的那本叫《ARMv7-M Architecture Application Level Reference Manual(Ref2)》,对于软件开发人员,那里面把该说的都说了。

1.4 Thumb-2 指令集体系结构 (ISA)

Thumb-2真不愧是一个突破性的指令集。它强大,它易用,它轻佻,它高效。Thumb-2是16位Thumb指令集的一个超集,在Thumb-2中,16位指令首次与32位指令并存,结果在Thumb状态下可以做的事情一下子丰富了许多,同样工作需要的指令周期数也明显下降。

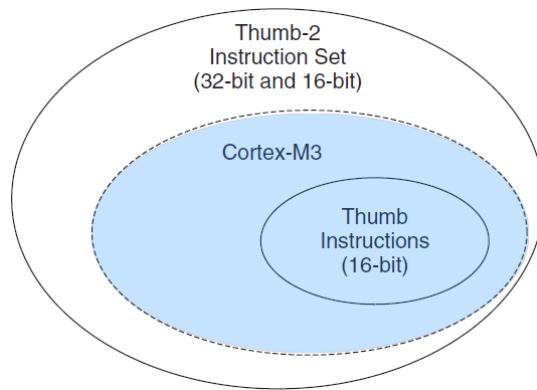


图1.4 Thumb-2指令集与Thumb指令集的关系

从图中可见, Cortex-M3勇敢地拒绝了32位ARM指令集,却把自己的处理能力以身相许般地全托给Thumb-2指令集。这可能有些令人意外,但事实上这却见证了Cortex-M3的用情专一:在内核水平上,就已经为适应单片机和小内存器件而抉择、取舍过了。但她没有嫁错郎,因为Thumb-2完全胜任在这个领域挑大梁。不过,这也意味着Cortex-M3作为新生代处理器,不是向后兼容的。因此,为ARM7写的ARM汇编语言程序不能直接移植到CM3上来。不过,CM3支持绝大多数传统的Thumb指令,因此用Thumb指令写的汇编程序就从善如流了。

在支持了both 16位和32位指令之后,就无需烦心地把处理器状态在Thumb和ARM之间来回的切换了。这种事在ARM7和ARM9是司空见惯的,尤其是在使用大型条件嵌套,以及执行复杂运算的时候,能精妙地移形换影于不同状态之间,那可是当年要成为大虾的基本功。

Cortex-M3是ARMv7架构的掌上明珠。和曾经红透整个业界的老一辈ARM7相比,Cortex-M3则是新生代的偶像,处处闪耀着青春的光芒活力。比如,硬件除法器被带到CM3中;乘法方面,也有好几条新指令闪亮登场,用于提升data-crunching的性能。CM3的出现,还在ARM处理器中破天荒地支持了“非对齐数据访问支持”。

1.5 Cortex-M3 处理器的舞台

- 高性能+高代码密度+小硅片面积, 3璧合一, 使得CM3大面积地成为理想的处理平台:
- 低成本单片机: CM3与生俱来就适合做单片机, 甚至简单到用于做玩具和小电器的单片机, 都能使用CM3作为内核。这里本是8位机和16位机统治最牢固的腹地, 但是CM3更便宜, 更高性能, 更易使用, 所以值得开发者们转到这个新生的ARM32位系统中来, 哪怕花点时间重新学习。
- 汽车电子: CM3也是汽车电子的好佬。CM3同时拥有非常高的性能和极低的中断延迟, 打入实时领域的大门。CM3处理器能支持多达240个外部中断, 内建了嵌套向量中断控制器, 还可以选择配上一个存储器保护单元 (MPU)。所有这些, 使它用于高集成度低成本的汽车应用最合适不过了。
- 数据通信: CM3的低成本+高效率, 再加上Thumb-2的强大位操作指令s, 使CM3非常理想地适合于很多数据通信应用, 尤其是无线数传和Ad-Hoc网络, 如ZigBee和蓝牙等。
- 工业控制: 在工控场合, 关键的要素在于简洁、快速响应以及可靠。再一次地, CM3处理器的中断处理能力, 低中断延迟, 强化的故障处理能力 (fault-handling, 以后fault就不再译成中文了——译注), 足以让它能昂首挺胸地踏入这片热土。
- 消费类产品: 以往, 在许多消费产品中, 都必须使用一块甚至好几块高性能的微处理器。你别

看CM3只是个小处理器，它的高性能和MPU机制可是足以让复杂的软件跑起来的，同时提供健壮的存储器保护。

目前在市场上已经有了好多基于Cortex-M3内核的处理器产品，最便宜的还不到1美元，让ARM终于比很多8位机还便宜了。

本书的组织

Chpt 1和2,	Cortex-M3的介绍和概览
Chpt 3-6,	Cortex-M3的基础知识
Chpt 7-9,	异常与中断
Chpt 10和11,	论述在Cortex-M3的编程
Chpt 12-14,	Cortex-M3的硬件特性
Chpt 15-16,	Cortex-M3的调试支持
Chpt 17-20,	在Cortex-M3上的应用软件开发
附录s	

1.6 深入研究用的读物

本书并没有面面俱到地谈及Cortex-M3的技术细节。本书靠前的章节用来做Cortex-M3新手的敲门砖，同时也是CM3处理器的增值参考资料。如果要进一步地学习，就需要从ARM网站下载下面这些重量级的权威资料：

《The Cortex-M3 Technical Reference Manual》，深入了处理器的内心，编程模型，存储器映射，还包括了指令时序。

《 The ARMv7-M Architecture Application Level Reference Manual》第2版，对指令集和存储器模型都提供了最不嫌繁的说明。

其它半导体厂家提供的，基于CM3单片机的数据手册。

如想了解更多总线协议的细节，可以去看《AMBA Specification 2.0》第 4版），它讲了更多AMBA接口的内幕。

对于C程序员，可以从《ARM Application Note 179: Cortex-M3 Embedded Software Development》（第7版）中得到一些编程技巧和提示。

本书假设你已经涉足过嵌入式编程，有一些基本知识和经验。如果你是位产品经理或者是想先浅浅地尝一尝，请先读第2章，试着找找感觉再决定要不要深入学习。这一章浓缩了全书的精华，走马观花地讲了Cortex-M3内核。

Cortex-M3 概览

内容提要:

- 简介
- 寄存器组
- 操作模式和特权级别
- 内建的嵌套向量中断控制器
- 存储器映射
- 总线接口
- 存储器保护单元
- 指令系统
- 中断和异常
- 调试支持
- 小结

2.1 简介

Cortex-M3 是一个 32 位处理器内核。内部的数据路径是 32 位的，寄存器是 32 位的，存储器接口也是 32 位的。CM3 采用了哈佛结构，拥有独立的指令总线和数据总线，可以让取指与数据访问并行不悖。这样一来数据访问不再占用指令总线，从而提升了性能。为实现这个特性，CM3 内部含有好几条总线接口，每条都为自己的应用场合优化过，并且它们可以并行工作。但是另一方面，指令总线和数据总线共享同一个存储器空间（一个统一的存储器系统）。换句话说，不是因为有两条总线，可寻址空间就变成 8GB 了。

比较复杂的应用可能需要更多的存储系统功能，为此 CM3 提供一个可选的 MPU，而且在需要的情况下也可以使用外部的 cache。另外在 CM3 中，Both 小端模式和大端模式都是支持的。

CM3 内部还附赠了好多调试组件，用于在硬件水平上支持调试操作，如指令断点，数据观察点等。另外，为支持更高级的调试，还有其它可选组件，包括指令跟踪和多种类型的调试接口。

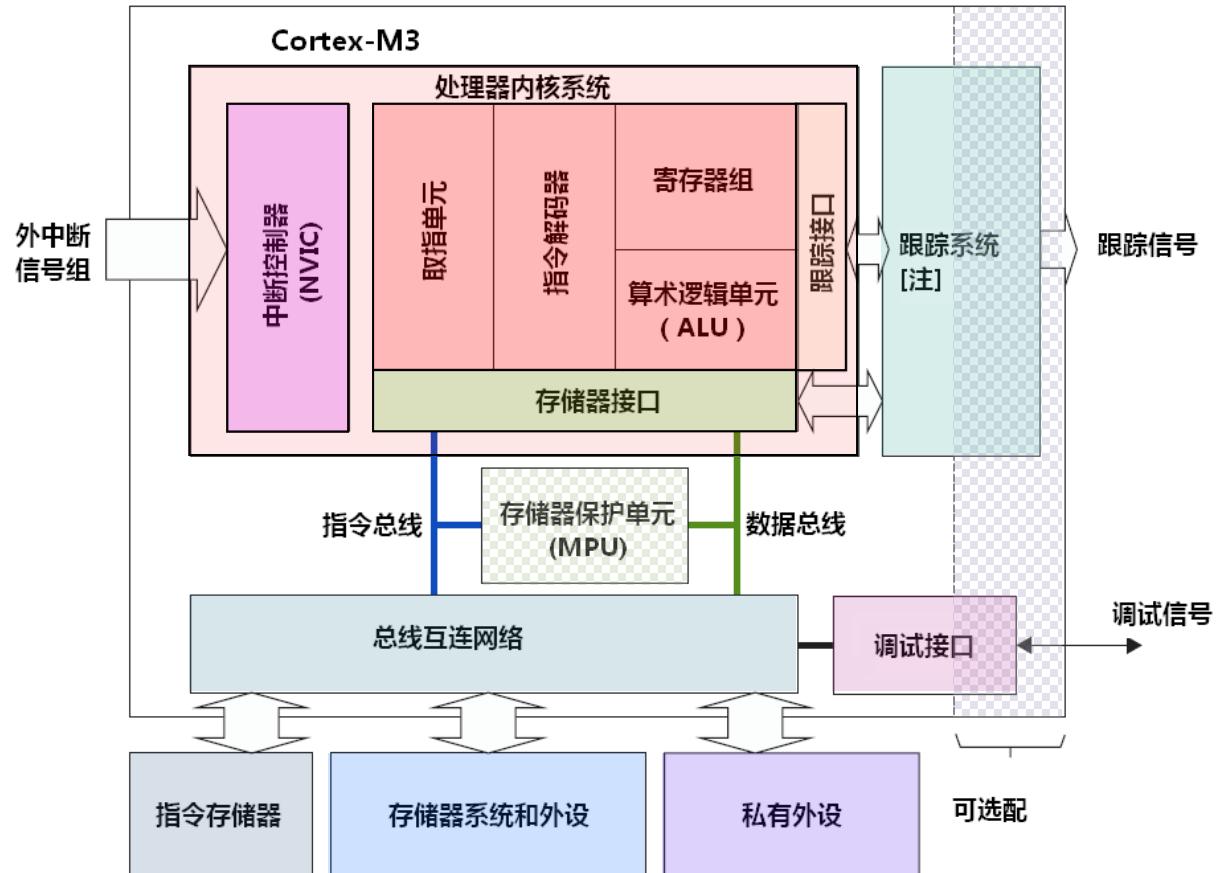


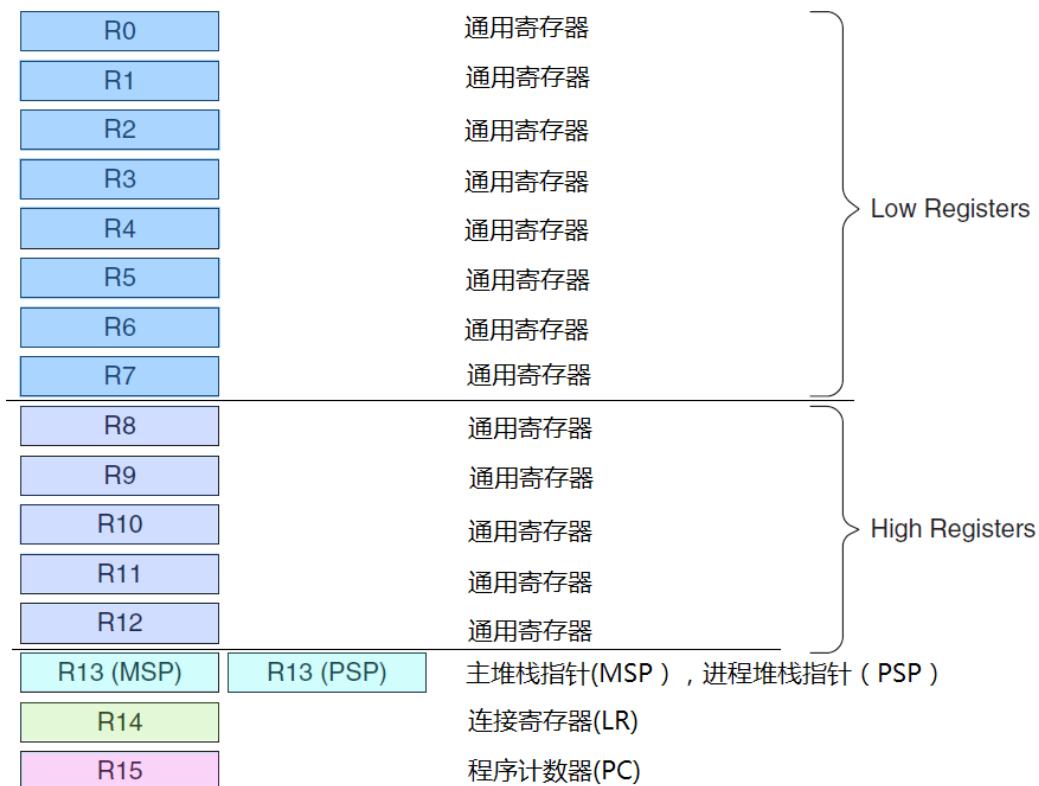
图 2.1 Cortex-M3 的一个简化视图

[图 2.1 注：

1. 原文为“Debug System”。但从图中看，疑似有误，更像“Trace System”，故改为“跟踪系统”
2. 天花板方块标示的是可选部分。其中 MPU 全部可选，而跟踪系统与调试接口的组件则有一部分是可选的。

2.2 寄存器组

Cortex-M3 处理器拥有 R0-R15 的寄存器组。其中 R13 作为堆栈指针 SP。SP 有两个，但在同一时刻只能有一个可以看到，这也就是所谓的“banked”寄存器。



2.2.1 R0-R12: 通用寄存器

R0-R12 都是 32 位通用寄存器，用于数据操作。但是注意：绝大多数 16 位 Thumb 指令只能访问 R0-R7，而 32 位 Thumb-2 指令可以访问所有寄存器。

2.2.2 Banked R13: 两个堆栈指针

Cortex-M3 拥有两个堆栈指针，然而它们是 **banked**，因此任一时刻只能使用其中的一个。

- **主堆栈指针 (MSP)**: 复位后缺省使用的堆栈指针，用于操作系统内核以及异常处理例程（包括中断服务例程）
- **进程堆栈指针 (PSP)**: 由用户的应用程序代码使用。

堆栈指针的最低两位永远是 0，这意味着堆栈总是 4 字节对齐的。

在 ARM 编程领域中，凡是打断程序顺序执行的事件，都被称为异常(exception)。除了外部中断外，当有指令执行了“非法操作”，或者访问被禁的内存区间，因各种错误产生的 fault，以及不可屏蔽中断发生时，都会打断程序的执行，这些情况统称为异常。在不严格的上下文中，异常与中断也可以混用。另外，程序代码也可以主动请求进入异常状态的（常用于系统调用）。

2.2.3 R14: 连接寄存器

当呼叫一个子程序时，由 R14 存储返回地址

不像大多数其它处理器，ARM 为了减少访问内存的次数（访问内存的操作往往要 3 个以上指令周期，带 MMU 和 cache 的就更加不确定了），把返回地址直接存储在寄存器中。这样足以使很多只有 1 级子程序调用的代码无需访问内存（堆栈内存），从而提高了子程序调用的效率。如果多于 1 级，则需要把前一级的 R14 值压到堆栈里。在 ARM 上编程时，应尽量只使用寄存器保存中间结果，迫不得以时才访问内存。在 RISC 处理器中，为了强调访内操作越

过了处理器的界线，并且带来了对性能的不利影响，给它取了一个专业的术语：溅出。

2.2.4R15：程序计数寄存器

指向当前的程序地址。如果修改它的值，就能改变程序的执行流（很多高级技巧就在这里面——译注）。

2.2.5特殊功能寄存器

Cortex-M3 还在内核水平上搭载了若干特殊功能寄存器，包括

程序状态字寄存器组（PSRs）

中断屏蔽寄存器组（PRIMASK, FAULTMASK, BASEPRI）

控制寄存器（CONTROL）

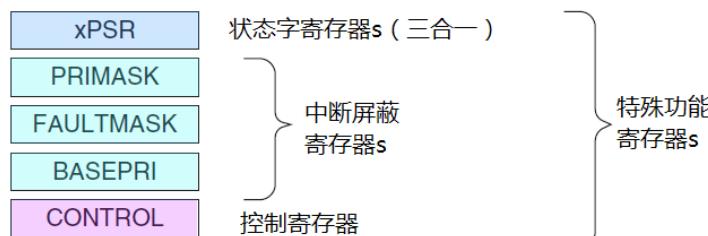


图 2.3：Cortex-M3 中的特殊功能寄存器集合

表 2.1 寄存器及其功能

寄存器	功能
xPSR	记录 ALU 标志（ θ 标志，进位标志，负数标志，溢出标志），执行状态，以及当前正服务的中断号
PRIMASK	除能所有的中断——当然了，不可屏蔽中断（NMI）才不甩它呢。
FAULTMASK	除能所有的 fault——NMI 依然不受影响，而且被除能的 faults 会“上访”，见后续章节的叙述。
BASEPRI	除能所有优先级不高于某个具体数值的中断。
CONTROL	定义特权状态（见后续章节对特权的叙述），并且决定使用哪一个堆栈指针

第 3 章对此有展开的叙述。

2.3 操作模式和特权级别

Cortex-M3 处理器支持两种处理器的操作模式，还支持两级特权操作。

两种操作模式分别为：处理器模式（`handler mode`，以后不再把 `handler` 中译——译注）和线

程模式（`thread mode`）。引入两个模式的本意，是用于区别普通应用程序的代码和异常服务例程的代码——包括中断服务例程的代码。

Cortex-M3 的另一个侧面则是特权的分级——特权级和用户级。这可以提供一种存储器访问

的保护机制，使得普通的用户程序代码不能意外地，甚至是恶意地执行涉及到要害的操作。处理器支持两种特权级，这也是一个基本的安全模型。

译注：“用户级”其实是从“user”译来的。有些时候英文文档也使用术语“Unprivileged”，后者如果直译，则称为“非特权级”。为统一术语，本译文一律使用“用户级”。

	特权级	用户级
异常handler的代码	handler模式	错误的用法
主应用程序的代码	线程模式	线程模式

图 2.4 Cortex-M3 下的操作模式和特权级别

在 CM3 运行主应用程序时（线程模式），既可以使用特权级，也可以使用用户级；但是异常服务例程必须在特权级下执行。复位后，处理器默认进入线程模式，特权级访问。在特权级下，程序可以访问所有范围的存储器（如果有 MPU，还要在 MPU 规定的禁地之外），并且可以执行所有指令。

在特权级下的程序可以为所欲为，但也可能会把自己给玩进去——切换到用户级。一旦进入用户级，再想回来就得走“法律程序”了——用户级的程序不能简简单单地试图改写 CONTROL 寄存器就回到特权级，它必须先“申诉”：执行一条系统调用指令(SVC)。这会触发 SVC 异常，然后由异常服务例程（通常是操作系统的一部分）接管，如果批准了进入，则异常服务例程修改 CONTROL 寄存器，才能在用户级的线程模式下重新进入特权级。

事实上，从用户级到特权级的唯一途径就是异常：如果在程序执行过程中触发了一个异常，处理器总是先切换入特权级，并且在异常服务例程执行完毕退出时，返回先前的状态（也可以手工指定返回的状态——译注）。

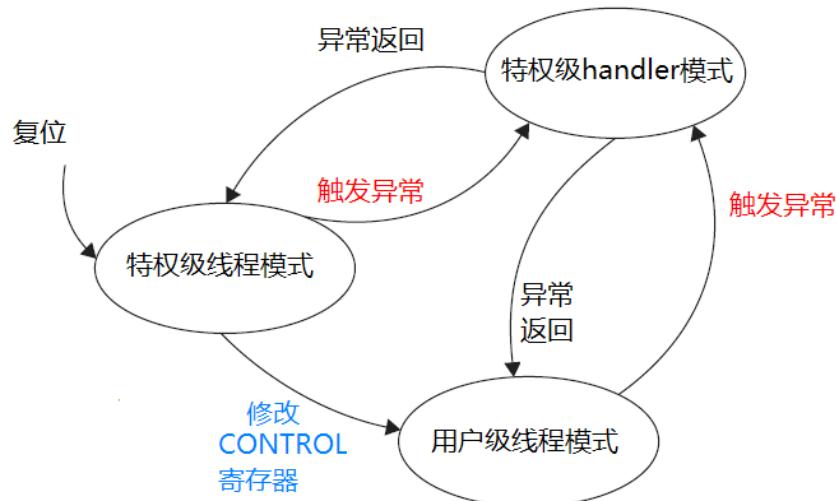


图 2.5 合法的操作模式转换图

通过引入特权级和用户级，就能在硬件水平上限制某些不受信任的或者还没有调试好的程序，不让它们随便地配置涉及要害的寄存器，因而系统的可靠性得到了提高。进一步地，如果配了 MPU，它还可以作为特权机制的补充——保护关键的存储区域不被破坏，这些区域通常是操作系统的区域。

举例来说，操作系统的内核通常都在特权级下执行，所有没有被 MPU 禁掉的存储器都可以访问。在操作系统开启了一个用户程序后，通常都会让它在用户级下执行，从而使系统不会因某个程序的崩溃或恶意破坏而受损。

2.4 内建的嵌套向量中断控制器

Cortex-M3 在内核水平上搭载了一颗中断控制器——**嵌套向量中断控制器 NVIC**(Nested Vectored Interrupt Controller)。它与内核有很深的“亲密接触”——与内核是紧耦合的。NVIC 提供如下的功能：

- 可嵌套中断支持
- 向量中断支持
- 动态优先级调整支持
- 中断延迟大大缩短
- 中断可屏蔽

2.4.1 可嵌套中断支持

可嵌套中断支持的作用范围很广，覆盖了所有的外部中断和绝大多数系统异常。外在表现是，这些异常都可以被赋予不同的优先级。当前优先级被存储在 xPSR 的专用字段中。当一个异常发生时，硬件会自动比较该异常的优先级是否比当前的异常优先级更高。如果发现来了更高优先级的异常，处理器就会中断当前的中断服务例程（或者是普通程序），而服务新来的异常——即立即抢占。

2.4.2 向量中断支持

当开始响应一个中断后，CM3 会自动定位一张向量表，并且根据中断号从表中找出 ISR 的入口地址，然后跳转过去执行。不需要像以前的 ARM 那样，由软件来分辨到底是哪个中断发生了，也无需半导体厂商提供私有的中断控制器来完成这种工作。这么一来，中断延迟时间大为缩短。

2.4.3 动态优先级调整支持

软件可以在运行时期更改中断的优先级。如果在某 ISR 中修改了自己所对应中断的优先级，而且这个中断又有新的实例处于悬起中（pending），也不会自己打断自己，从而没有重入（reentry）〔译注 7〕 风险。

〔译注 7〕：所谓的重入，就是指某段子程序还没有执行完，就因为中断或者是多任务操作系统的调度原因，导致该子程序在一个新的寄存器上下文中被执行（请不要把重入与递归混淆，它们有本质的区别）。这种情况常常会闹出乱子，因此有“可重入性”的研究。

2.4.4 中断延迟大大缩短

Cortex-M3 为了缩短中断延迟，引入了好几个新特性。包括自动的现场保护和恢复，以及其它的措施，用于缩短中断嵌套时的 ISR 间延迟。详情请见后面关于“咬尾中断”和“晚到中断”的讲述。

2.4.5 中断可屏蔽

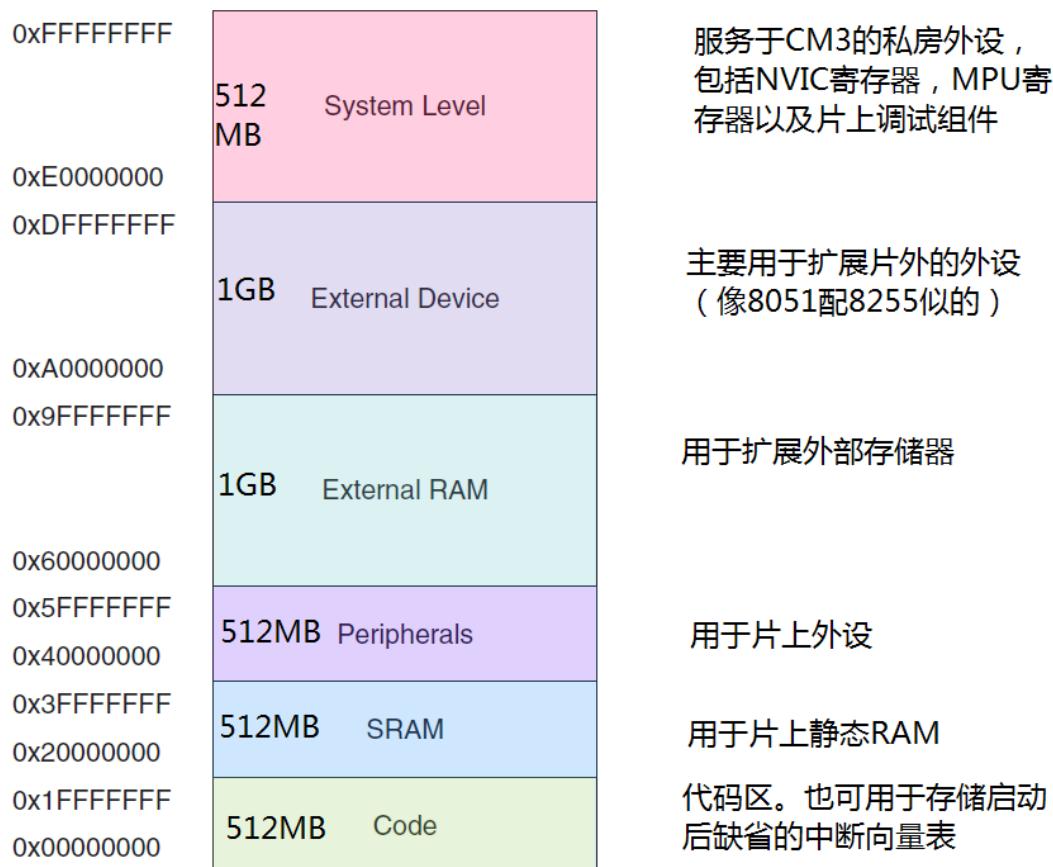
既可以屏蔽优先级低于某个阈值的中断/异常〔译注 8〕（设置 BASEPRI 寄存器），也可以全体封杀（设置 PRIMASK 和 FAULTMASK 寄存器）。这是为了让时间关键（time-critical）的任务能在死线

(**deadline**, 或曰最后期限)到来前完成, 而不被干扰。

[译注 8]: 鉴于(外部)中断的常见性, 以后译文中如果没有特殊说明, 凡是提到“异常”, 均指除了外部中断之外的异常, 而使用“中断”来表示所有外部中断——也就是对于处理器来说是异步的中断。

2.5 存储器映射

总体来说, Cortex-M3 支持 4GB 存储空间, 如图 2.6 所示地被划分成若干区域。



从图中可见, 不像其它的 ARM 架构, 它们的存储器映射由半导体厂家说了算, Cortex-M3 预先定义好了“粗线条的”存储器映射。通过把片上外设的寄存器映射到外设区, 就可以简单地以访问内存的方式来访问这些外设的寄存器, 从而控制外设的工作。结果, 片上外设可以使用 C 语言来操作。这种预定义的映射关系, 也使得对访问速度可以做高度的优化, 而且对于片上系统的设计而言更易集成(还有一个重要的, 不用每学一种不同的单片机就要熟悉一种新的存储器映射——译注)。

Cortex-M3 的内部拥有一个总线基础设施, 专用于优化对这种存储器结构的使用。在此之上, CM3 甚至还允许这些区域之间“越权使用”。比如说, 数据存储器也可以被放到代码区, 而且代码也能够在外部 RAM 区中执行(但是会变慢不少——译注)。

处于最高地址的系统级存储区, 是 CM3 用于藏“私房钱”的——包括中断控制器、MPU 以及各种调试组件。所有这些设备均使用固定的地址(本书第 5 章讨论存储器系统)。通过把基础设施的地址定死, 就至少在内核水平上, 为应用程序的移植扫清了障碍。

2.6 总线接口

Cortex-M3 内部有若干个总线接口，以使 CM3 能同时取址和访内（访问内存），它们是：

- 指令存储区总线（两条）
- 系统总线
- 私有外设总线

有两条代码存储区总线负责对代码存储区的访问，分别是 **I-Code** 总线和 **D-Code** 总线。前者用于取指，后者用于查表等操作，它们按最佳执行速度进行优化。

系统总线用于访问内存和外设，覆盖的区域包括 **SRAM**，片上外设，片外 **RAM**，片外扩展设备，以及系统级存储区的部分空间。

私有外设总线负责一部分私有外设的访问，主要就是访问调试组件。它们也在系统级存储区。

2.7 存储器保护单元 (MPU)

Cortex-M3 有一个可选的存储器保护单元。配上它之后，就可以对特权级访问和用户级访问分别施加不同的访问限制。当检测到犯规（**violated**）时，MPU 就会产生一个 **fault** 异常，可以由 **fault** 异常的服务例程来分析该错误，并且在可能时改正它。

MPU 有很多玩法。最常见的就是由操作系统使用 MPU，以使特权级代码的数据，包括操作系统本身的数据不被其它用户程序弄坏。MPU 在保护内存时是按区管理的（“区”的原文是 **region**，以后不再中译此名词——译注）。它可以把某些内存 **region** 设置成只读，从而避免了那里的内容意外被更改；还可以在多任务系统中把不同任务之间的数据区隔离。一句话，它会使嵌入式系统变得更加健壮，更加可靠（很多行业标准，尤其是航空的，就规定了必须使用 MPU 来行使保护职能——译注）。

2.8 指令集

Cortex-M3 只使用 **Thumb-2** 指令集。这是个了不起的突破，因为它允许 32 位指令和 16 位指令水乳交融，代码密度与处理性能两手抓，两手都硬。而且虽然它很强大，却依然易于使用。

在过去，做 ARM 开发必须处理好两个状态。这两个状态是井水不犯河水的，它们是：32 位的 ARM 状态和 16 位的 **Thumb** 状态。当处理器在 ARM 状态下时，所有的指令均是 32 位的（哪怕只是个“NOP”指令），此时性能相当高。而在 **Thumb** 状态下，所有的指令均是 16 位的，代码密度提高了一倍。不过，**thumb** 状态下的指令功能只是 ARM 下的一个子集，结果可能需要更多条的指令去完成相同的工作，导致处理性能下降。

为了取长补短，很多应用程序都混合使用 ARM 和 **Thumb** 代码段。然而，这种混合使用是有额外开销（**overhead**）的，时间上的和空间上的都有，主要发生在状态切换之时。另一方面，ARM 代码和 **Thumb** 代码需要以不同的方式编译，这也增加了软件开发管理的复杂度。

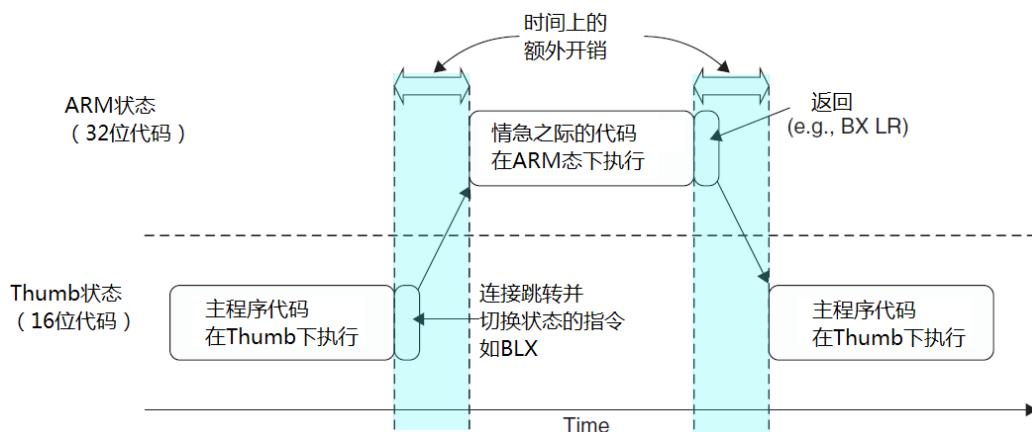


图 2.7 在诸如 ARM7 处理器上的状态切换模式图

伴随着 Thumb-2 指令集的横空出世, 终于可以在单一的操作模式下搞定所有处理了, 再也没有来回切换的事来烦你了。事实上, Cortex-M3 内核干脆都不支持 ARM 指令, 中断也在 Thumb 态下处理 (以前的 ARM 总是在 ARM 状态下处理所有的中断和异常)。这可不是小便宜, 它使 CM3 在好几个方面都比传统的 ARM 处理器更先进:

- 消灭了状态切换的额外开销, 节省了 both 执行时间和指令空间。
- 不再需要把源代码文件分成按 ARM 编译的和按 Thumb 编译的, 软件开发的管理大大减负。
- 无需再反复地求证和测试: 究竟该在何时何地切换到何种状态下, 我的程序才最有效率。开发软件容易多了。

不少有趣和强大的指令为 Cortex-M3 注入了新鲜的青春血液, 下面给出几个例子:

- **UBFX, BFI, BFC:** 位段提取, 位段插入, 位段清零。支持 C 位段, 也简化了外设寄存器操作。
- **CLZ, RBIT:** 计算前导零指令和位反转指令。二者组合使用能实现一些特技
- **UDIV, SDIV:** 无符号除法和带符号除法指令。
- **SEV, WFE, WFI:** 发送事件, 等待事件以及等待中断指令。用于实现多处理器之间的任务同步, 还可以进入不同的休眠模式。
- **MSR, MRS:** 通向禁地——访问特殊功能寄存器。

因为 CM3 专情于最新的 Thumb-2, 旧的应用程序需要移植和重建。对于大多数 C 源程序, 只需简单地重新编译就能重建, 汇编代码则可能需要大面积地修改和重写, 才能使用 CM3 的新功能, 并且融入 CM3 新引入的统一汇编器框架(unified assembler framework)中。

请注意: CM3 并不支持所有的 Thumb-2 指令, ARMv7-M 的规格书只要求实现 Thumb-2 的一个子集。举例来说, 协处理器指令就被裁掉了 (可以使用外部的数据处理引擎来替代)。CM3 也没有实现 SIMD 指令集。旧世代的一些 Thumb 指令不再需要, 因此也被排除。不支持指令还包括 v6 中引入的 SETEND 指令。如欲查出一个完整的指令列表, 可以去看附录 A。

2.9 中断和异常

ARMv7-M 开创了一个全新的异常模型, CM3 采用了它。请你一定要划清界线: 这种异常模型跟传统 ARM 处理器使用的完全是两码事。新的异常模型“使能”了非常高效的异常处理。它支持

16-4-1=11 种系统异常（保留了 4+1 个档位），外加 240 个外部中断输入。在 CM3 中取消了 FIQ 的概念（v7 前的 ARM 都有这个 FIQ，快中断请求），这是因为有了更新更好的机制——中断优先级管理以及嵌套中断支持，它们被纳入 CM3 的中断管理逻辑中。因此，支持嵌套中断的系统就更容易实现 FIQ。

CM3 的所有中断机制都由 NVIC 实现。除了支持 240 条中断之外，NVIC 还支持 16-4-1=11 个内部异常源，可以实现 fault 管理机制。结果，CM3 就有了 256 个预定义的异常类型，如表 2.2 所示。

表 2.2 Cortex-M3 异常类型

编号	类型	优先级	简介
0	N/A	N/A	没有异常在运行
1	复位	-3(最高)	复位
2	NMI	-2	不可屏蔽中断 (来自外部 NMI 输入脚)
3	硬(hard) fault	-1	所有被除能的 fault , 都将 “上访” 成硬 fault
4	MemManage fault	可编程	存储器管理 fault , MPU 访问犯规以及访问非法位置
5	总线 fault	可编程	总线错误 (预取流产 (Abort) 或数据流产)
6	用法(usage) Fault	可编程	由于程序错误导致的异常
7-10	保留	N/A	N/A
11	SVCall	可编程	系统服务调用
12	调试监视器	可编程	调试监视器 (断点 , 数据观察点 , 或者是外部调试请求)
13	保留	N/A	N/A
14	PendSV	可编程	为系统设备而设的 “可悬挂请求” (pendable request)
15	SysTick	可编程	系统滴答定时器 (也就是周期性溢出的时基定时器——译注)
16	IRQ #0	可编程	外中断#0
17	IRQ #1	可编程	外中断#1
...
255	IRQ #239	可编程	外中断#239

虽然 CM3 是支持 240 个外中断的，但具体使用了多少个是由芯片生产商决定。CM3 还有一个 NMI（不可屏蔽中断）输入脚。当它被置为有效（assert）时，NMI 服务例程会无条件地执行。

2.9b 低功耗与高能效(r2p0 修订版)

为了使我们的产品功耗更低，以及能源利用效率更高，Cortex-M3 在设计时加入了很多针对性的功能。

首先，在节能模式上，它提供了睡眠模式和深度睡眠模式。芯片以及整个系统在设计时通过与内核的节能模式相呼应，就可以根据应用的要求，在空闲时降低功耗。第二，它精练的设计使得门数很低，并且在工作状态下电路的活动更少，所以 CM3 自己也是“身先士卒”地以身作则了。而且，由于 CM3 的程序代码密度高，程序容量也可以变得更少；同时，再加上它强大的性能减少了程序执行时间，使得系统能以最快的速度回到睡眠中，以削低对能源的用量。综上所述，Cortex-M3 的能效要高于大多的 8 位或 16 位单片机。

2.10 调试支持

Cortex-M3 在内核水平上搭载了若干种调试相关的特性。最主要的就是程序执行控制，包括停机(halting)、单步执行(stepping)、指令断点、数据观察点、寄存器和存储器访问、性能速写(profiling) 以及各种跟踪机制。

Cortex-M3 的调试系统基于 ARM 最新的 CoreSight 架构。不同于以往的 ARM 处理器，内核本身不再含有 JTAG 接口。取而代之的，是 CPU 提供称为“调试访问接口(DAP)”的总线接口。通过这个总线接口，可以访问芯片的寄存器，也可以访问系统存储器，甚至是在内核运行的时候访问！对此总线接口的使用，是由一个调试端口(DP)设备完成的。DPs 不属于 CM3 内核，但它们是在芯片的内部实现的。目前可用的 DP 包括 SWJ-DP(既支持传统的 JTAG 调试，也支持新的串行线调试协议)，另一个 SW-DP 则去掉了对 JTAG 的支持。另外，也可以使用 ARM CoreSight 产品家族的 JTAG-DP 模块。这下就有 3 个 DP 可以选了，芯片制造商可以从中选择一个，以提供具体的调试接口（通常都是选 SWJ-DP）。

此外，CM3 还能挂载一个所谓的“嵌入式跟踪宏单元(ETM)”。ETM 可以不断地发出跟踪信息，这些信息通过一个被称为“跟踪端口接口单元(TPIU)”的模块而送到内核的外部，再在芯片外面使用一个“跟踪信息分析仪”，就可以把 TIPU 输出的“已执行指令信息”捕捉到，并且送给调试主机——也就是 PC。

在 Cortex-M3 中，调试动作能由一系列的事件触发，包括断点，数据观察点，fault 条件，或者是外部调试请求输入的信号。当调试事件发生时，Cortex-M3 可能会停机，也可能进入调试监视器异常 handler。具体如何反应，则根据与调试相关寄存器的配置。

与调试相关的还有其它的绝活。现在要介绍的是“仪器化跟踪宏单元(ITM)”，它也有自己的办法把数据送往调试器。通过把数据写到 ITM 的寄存器中，调试器能够通过跟踪接口来收集这些数据，并且显示或者处理它。此法不但容易使用，而且比 JTAG 的输出速度更快。

所有这些调试组件都可以由 DAP 总线接口来控制，CM3 内核提供 DAP 接口。此外，运行中的程序也能控制它们。所有的跟踪信息都能通过 TPIU 来访问到。

2.11 Cortex-M3 的品性简评

讲了这么多，究竟是拥有了什么，使 Cortex-M3 成为如此有突破性的新生代处理器？Cortex-M3 到底在哪里先进了？本节就给出一个小小的简评。

2.11.1 高性能

- 许多指令都是单周期的——包括乘法相关指令。并且从整体性能上，Cortex-M3 比得过绝大多数其它的架构。
- 指令总线和数据总线被分开，取值和访内可以并行不悖
- Thumb-2 的到来告别了状态切换的旧世代，再也不需要花时间来切换于 32 位 ARM 状态和 16 位 Thumb 状态之间了。这简化了软件开发和代码维护，使产品面市更快。
- Thumb-2 指令集为编程带来了更多的灵活性。许多数据操作现在能用更短的代码搞定，这意味着 Cortex-M3 的代码密度更高，也就对存储器的需求更少。
- 取指都按 32 位处理。同一周期最多可以取出两条指令，留下了更多的带宽给数据传输。
- Cortex-M3 的设计允许单片机高频运行(现代半导体制造技术能保证 100MHz 以上的速度)。即使在相同的速度下运行，CM3 的每指令周期数(CPI)也更低，于是同样的 MHz 下可以做更多的工作；另一方面，也使同一个应用在 CM3 上需要更低的主频。

2.11.2 先进的中断处理功能

- 内建的嵌套向量中断控制器支持多达 240 条外部中断输入。向量化的中断功能剧烈地缩短了中断延迟，因为不再需要软件去判断中断源。中断的嵌套也是在硬件水平上实现的，不需要软件代码来实现。
- Cortex-M3 在进入异常服务例程时，自动压栈了 R0-R3，R12，LR，PSR 和 PC，并且在返回时自动弹出它们，这多清爽！既加速了中断的响应，也再不需要汇编语言代码了（第 8 章有详述）。
- NVIC 支持对每一路中断设置不同的优先级，使得中断管理极富弹性。最粗线条的实现也至少要支持 8 级优先级，而且还能动态地被修改。
- 优化中断响应还有两招，它们分别是“咬尾中断机制”和“晚到中断机制”。
- 有些需要较多周期才能执行完的指令，是可以被中断一继续的——就好比它们是一串指令一样。这些指令包括加载多个寄存器 (LDM)，存储多个寄存器 (STM)，多个寄存器参与的 PUSH，以及多个寄存器参与的 POP。
- 除非系统被彻底地锁定，NMI（不可屏蔽中断）会在收到请求的第一时间予以响应。对于很多安全-关键(safety-critical)的应用，NMI 都是必不可少的（如化学反应即将失控时的紧急停机）。

2.11.3 低功耗

- Cortex-M3 需要的逻辑门数少，所以先天就适合低功耗要求的应用（功率低于 0.19mW/MHz）

- 在内核水平上支持节能模式（**SLEEPING** 和 **SLEEPDEEP** 位）。通过使用“等待中断指令（**WFI**）”和“等待事件指令（**WFE**）”，内核可以进入睡眠模式，并且以不同的方式唤醒。另外，模块的时钟是尽可能地分开供应的，所以在睡眠时可以把 **CM3** 的大多数“官能团”给停掉。
- **CM3** 的设计是全静态的、同步的、可综合的。任何低功耗的或是标准的半导体工艺均可放心饮用。

2.11.4 系统特性

- 系统支持“位寻址带”操作（**8051** 位寻址机制的“威力大幅加强版”），字节不变的大端模式，并且支持非对齐的数据访问。
- 拥有先进的 **fault** 处理机制，支持多种类型的异常和 **faults**，使故障诊断更容易。
- 通过引入 **banked** 堆栈指针机制，把系统程序使用的堆栈和用户程序使用的堆栈划清界线。如果再配上可选的 **MPU**，处理器就能彻底满足对软件健壮性和可靠性有严格要求的应用。

2.11.5 调试支持

- 在支持传统的 **JTAG** 基础上，还支持更新更好的串行线调试接口。
- 基于 **CoreSight** 调试解决方案，使得处理器哪怕是在运行时，也能访问处理器状态和存储器内容。
- 内建了对多达 6 个断点和 4 个数据观察点的支持。
- 可以选配一个 **ETM**，用于指令跟踪。数据的跟踪可以使用 **DWT**
- 在调试方面还加入了以下的新特性，包括 **fault** 状态寄存器，新的 **fault** 异常，以及闪存修补（**patch**）操作，使得调试大幅简化。
- 可选 **ITM** 模块，测试代码可以通过它输出调试信息，而且“拎包即可入住”般地方便使用。

Cortex-M3 基础

- 寄存器组
- 特殊功能寄存器组
- 操作模式
- 异常和中断
- 向量表
- 存储器保护单元
- 堆栈区的操作
- 复位序列

3.1 寄存器组

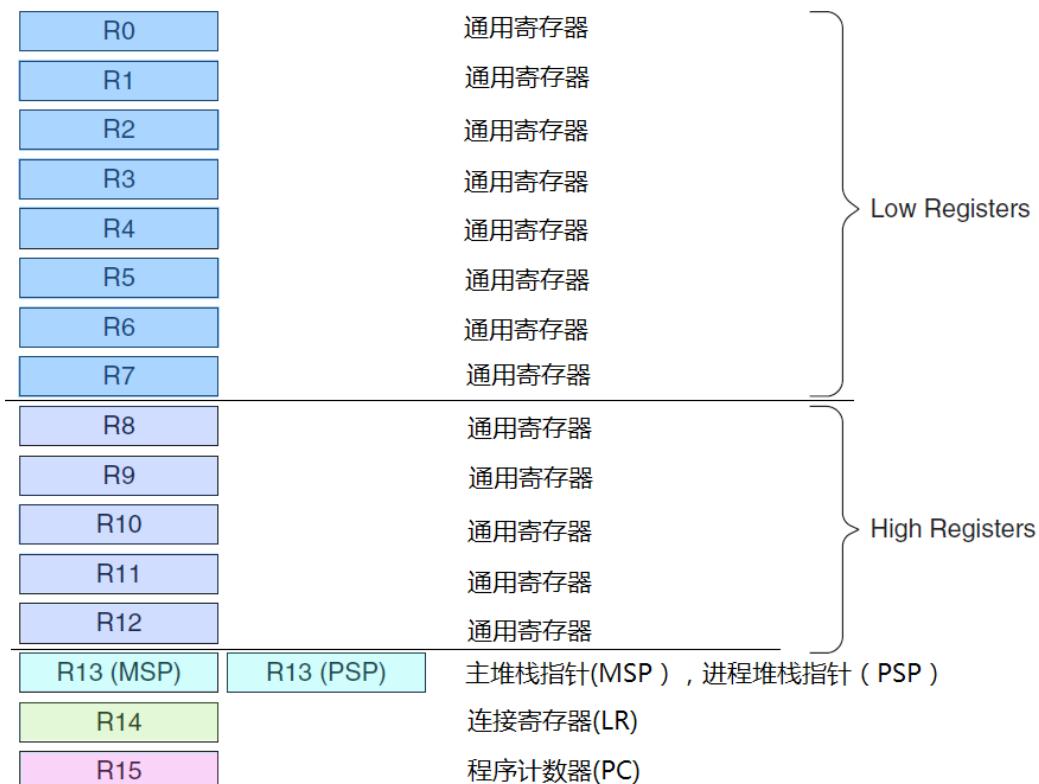
如我们所见, CM3 拥有通用寄存器 R0-R15 以及一些特殊功能寄存器。R0-R12 是最“通用目的”的, 但是绝大多数的 16 位指令只能使用 R0-R7 (低组寄存器), 而 32 位的 Thumb-2 指令则可以访问所有通用寄存器。特殊功能寄存器有预定义的功能, 而且必须通过专用的指令来访问。

3.1.1 通用目的寄存器 R0-R7

R0-R7 也被称为低组寄存器。所有指令都能访问它们。它们的字长全是 32 位, 复位后的初始值是不可预料的。

3.1.2 通用目的寄存器 R8-R12

R8-R12 也被称为高组寄存器。这是因为只有很少的 16 位 Thumb 指令能访问它们, 32 位的 thumb-2 指令则不受限制。它们也是 32 位字长, 且复位后的初始值是不可预料的。



3.1.3 特殊功能寄存器:

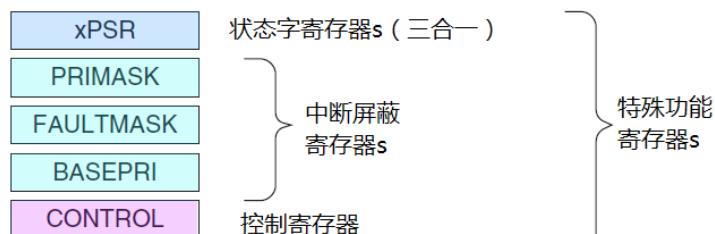


图 3.1 Cortex-M3 的寄存器组

3.1.4 堆栈指针 R13

R13 是堆栈指针。在 CM3 处理器内核中共有两个堆栈指针，于是也就支持两个堆栈。当引用 R13 (或写作 SP) 时，引用到的是当前正在使用的那个，另一个必须用特殊的指令来访问 (MRS, MSR 指令)。这两个堆栈指针分别是：

- **主堆栈指针 (MSP)**，或写作 **SP_main**。这是缺省的堆栈指针，它由 OS 内核、异常服务例程以及所有需要特权访问的应用程序代码来使用。
- **进程堆栈指针 (PSP)**，或写作 **SP_process**。用于常规的应用程序代码 (不处于异常服用例程中时)。

译注：在本章中，如无特殊说明，“异常”与“中断”都是指当发生“事件”时，处理器改变正常执行流，去响应该事件的情况。只不过异常之于处理器是同步的，中断之于处理器是异步的。因此常混合使用二术语，ISR 和 ESR 也混合使用，请读者不必工于辨析这两个术语的不同，在这里这不是重点。

要注意的是，并不是每个程序都要用齐两个堆栈指针才算圆满。简单的应用程序只使用 MSP 就

够了。堆栈指针用于访问堆栈，并且 **PUSH** 指令和 **POP** 指令默认使用 **SP**。

堆栈的 PUSH 与 POP

堆栈是一种存储器的使用模型。它由一块连续的内存和一个栈顶指针组成，用于实现“后进先出”的缓冲区。其最典型的应用，就是在数据处理前先保存寄存器的值，再在处理任务完成后从中恢复先前保护的这些值。

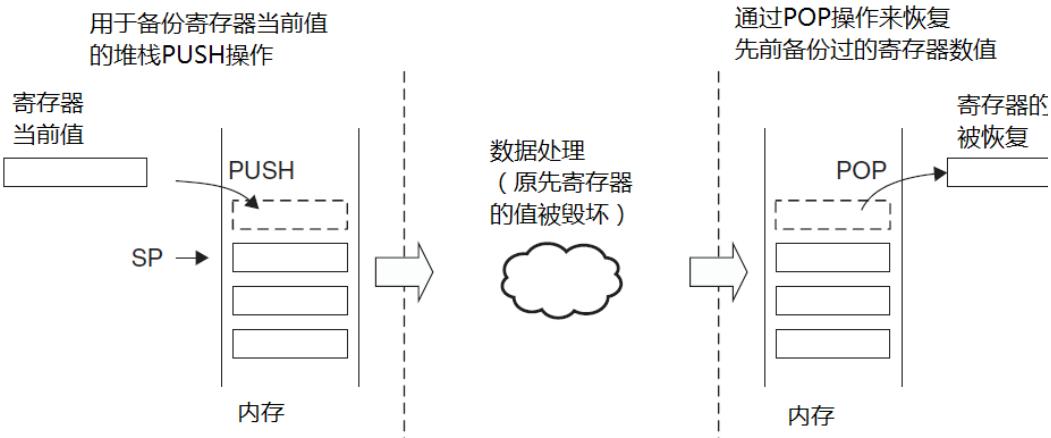


图 1.2 堆栈内存的基本概念

在执行 **PUSH** 和 **POP** 操作时，那个通常被称为 **SP** 的地址寄存器，会由硬件自动调整它的值，以避免后续操作破坏先前的数据。本书的后续章节还要围绕着堆栈展开更详细的论述。

在 Cortex-M3 中，有专门的指令负责堆栈操作——**PUSH** 和 **POP**。它们的汇编语言语法如下例所演示

```
PUSH {R0} ; *(--R13)=R0。R13 是 long* 的指针
POP {R0} ; R0= *R13++
```

请注意后面 C 程序风格的注释，它诠释了所谓的“向下生长的满栈”（本章后面在讲到堆栈内存操作时还要展开论述），**Cortex-M3** 就是以这种方式使用堆栈的。因此，在 **PUSH** 新数据时，堆栈指针先减一个单元。通常在进入一个子程序后，第一件事就是把寄存器的值先 **PUSH** 入堆栈中，在子程序退出前再 **POP** 曾经 **PUSH** 的那些寄存器。另外，**PUSH** 和 **POP** 还能一次操作多个寄存器，如下所示：

```
subroutine_1
PUSH {R0-R7, R12, R14} ; 保存寄存器列表
...
POP {R0-R7, R12, R14} ; 恢复寄存器列表
BX R14 ; 返回到主调函数
```

在程序中为了突出重点，可以一直把 **R13** 写作 **SP**。在程序代码中，**both MSP** 和 **PSP** 都被称为 **R13/SP**。不过，我们可以通过 **MRS/MSR** 指令来指名道姓地访问具体的堆栈指针。

MSP，亦写作 **SP_main**，这是复位后缺省使用堆栈指针，服务于操作系统内核和异常服务例程；而 **PSP**，亦写作 **SP_process**，典型地用于普通的用户线程中。

寄存器的 PUSH 和 POP 操作永远都是 4 字节对齐的——也就是说他们的地址必须是 0x4,0x8,0xc,……。事实上, R13 的最低两位被硬线连接到 0,并且总是读出 0 (Read As Zero)。

3.1.5 连接寄存器 R14

R14 是连接寄存器 (LR)。在一个汇编程序中, 你可以把它写作 both LR 和 R14。LR 用于在调用子程序时存储返回地址。例如, 当你在使用 BL(分支并连接, Branch and Link)指令时, 就自动填充 LR 的值。

```
main      ; 主程序
...
BL function1      ; 使用“分支并连接”指令呼叫 function1
;   PC= function1, 并且 LR=main 的下一条指令地址
...
Function1
...
; function1 的代码
BX LR      ; 函数返回 (如果 function1 要使用 LR, 必须在使用前 PUSH,
; 否则返回时程序就可能跑飞了——译注)
```

尽管 PC 的 LSB 总是 0 (因为代码至少是字对齐的), LR 的 LSB 却是可读可写的。这是历史遗留的产物。在以前, 由位 0 来指示 ARM/Thumb 状态。因为其它有些 ARM 处理器支持 ARM 和 Thumb 状态并存, 为了方便汇编程序移植, CM3 需要允许 LSB 可读可写。

3.1.6 程序计数器 R15

R15 是程序计数器, 在汇编代码中一般我们都叫它的外号 “PC”。因为 CM3 内部使用了指令流水线, 读 PC 时返回的值是当前指令的地址+4。比如说:

```
0x1000:      MOV      R0,      PC      ;  R0 = 0x1004
```

如果向 PC 中写数据, 就会引起一次程序的分支 (但是不更新 LR 寄存器)。CM3 中的指令至少是半字对齐的, 所以 PC 的 LSB 总是读回 0。然而, 在分支时, 无论是直接写 PC 的值还是使用分支指令, 都必须保证加载到 PC 的数值是奇数 (即 LSB=1), 用以表明这是在 Thumb 状态下执行。倘若写了 0, 则视为企图转入 ARM 模式, CM3 将产生一个 fault 异常。

3.2 特殊功能寄存器组

Cortex-M3 中的特殊功能寄存器包括:

- 程序状态寄存器组 (PSRs 或曰 xPSR)
- 中断屏蔽寄存器组 (PRIMASK, FAULTMASK, 以及 BASEPRI)
- 控制寄存器 (CONTROL)

它们只能被专用的 MSR/MRS 指令访问, 而且它们也没有与之相关联的访问地址。

```
MRS      <gp_reg>,      <special_reg> ; 读特殊功能寄存器的值到通用寄存器
```

MSR <special_reg>, <gp_reg> ;写通用寄存器的值到特殊功能寄存器

3.2.1 程序状态寄存器 (PSRs 或曰 PSR)

程序状态寄存器在其内部又被分为三个子状态寄存器：

- **应用程序 PSR (APSR)**
- **中断号 PSR (IPSR)**
- **执行 PSR (EPSR)**

通过 MRS/MSR 指令，这 3 个 PSRs 即可以单独访问，也可以组合访问（2 个组合，3 个组合都可以）。当使用三合一的方式访问时，应使用名字“xPSR”或者“PSR”。

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q											
IPSR																Exception Number
EPSR						ICI/IT	T			ICI/IT						

图 3.3 Cortex-M3 中的程序状态寄存器 (xPSR)

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT						Exception Number

图 3.4 合体后的程序状态寄存器(xPSR)

3.2.2 PRIMASK, FAULTMASK 和 BASEPRI

这三个寄存器用于控制异常的使能和除能。

表 3.2 Cortex-M3 的屏蔽寄存器组

名字	功能描述
PRIMASK	这是个只有单一比特的寄存器。在它被置 1 后，就关掉所有可屏蔽的异常，只剩下 NMI 和硬 fault 可以响应。它的缺省值是 0，表示没有关中断。
FAULTMASK	这是个只有 1 个位的寄存器。当它置 1 时，只有 NMI 才能响应，所有其它的异常，甚至是硬 fault，也通通闭嘴。它的缺省值也是 0，表示没有关异常。
BASEPRI	这个寄存器最多有 9 位（由表达优先级的位数决定）。它定义了被屏蔽优先级的阈值。当它被设成某个值后，所有优先级号大于等于此值的中断都被关（优先级号越大，优先级越低）。但若被设成 0，则不关闭任何中断，0 也是缺省值。

对于时间-关键任务而言，恰如其分地使用 PRIMASK 和 BASEPRI 来暂时关闭一些中断是非常重要的，而 FAULTMASK 则可以被 OS 用于暂时关闭 fault 处理机能，这种处理在某个任务崩溃时可能需要。因为在任务崩溃时，常常伴随着一大堆 faults。在系统料理“后事”时，通常不再需要响应这些 fault ——人死帐清。总之 FAULTMASK 就是专门留给 OS 用的。

要访问 PRIMASK, FAULTMASK 以及 BASEPRI，同样要使用 MRS/MSR 指令，如：

```

MRS    R0,      BASEPRI      ; 读取 BASEPRI 到 R0 中
MRS    R0,      FAULTMASK    ; 似上
MRS    R0,      PRIMASK     ; 似上
MSR    BASEPRI, R0          ; 写入 R0 到 BASEPRI 中
MSR    FAULTMASK, R0        ; 似上
MSR    PRIMASK, R0          ; 似上

```

只有在特权级下，才允许访问这 3 个寄存器。

译者添加：

其实，为了快速地开关中断，CM3 还专门设置了一条 CPS 指令，有 4 种用法

```

CPSID  I      ; PRIMASK=1,      ; 关中断
CPSIE  I      ; PRIMASK=0,      ; 开中断
CPSID  F      ; FAULTMASK=1,    ; 关异常
CPSIE  F      ; FAULTMASK=0    ; 开异常

```

3.2.3 控制寄存器 (CONTROL)

控制寄存器有两个用途，其一用于定义特权级别，其二用于选择当前使用哪个堆栈指针。由两个比特来行使这两个职能。

表 3.3 Cortex-M3 的 CONTROL 寄存器

位	功能
CONTROL[1]	<p>堆栈指针选择</p> <p>0=选择主堆栈指针 MSP (复位后的缺省值)</p> <p>1=选择进程堆栈指针 PSP</p> <p>在线程或基础级 (没有在响应异常——译注)，可以使用 PSP。在 handler 模式下，只允许使用 MSP，所以此时不得往该位写 1。</p>
CONTROL[0]	<p>0=特权级的线程模式</p> <p>1=用户级的线程模式</p> <p>Handler 模式永远都是特权级的。</p>

CONTROL[1]

在 Cortex-M3 的 handler 模式中，CONTROL[1]总是 0。在线程模式中则可以为 0 或 1。

因此，仅当处于特权级的线程模式下，此位才可写，其它场合下禁止写此位。改变处理器的模式也有其它的方式：在异常返回时，通过修改 LR 的位 2，也能实现模式切换。这是 LR 在异常返回时的特殊用法，颠覆了对 LR 的传统使用方式，将在第 5 章中展开论述。

CONTROL[0]

仅当在特权级下操作时才允许写该位。一旦进入了用户级，唯一返回特权级的途径，就是触发一个（软）中断，再由服务例程改写该位。

CONTROL 寄存器也是通过 MRS 和 MSR 指令来操作的：

```
MRS      R0,      CONTROL
MSR      CONTROL, R0
```

3.3 操作模式

Cortex-M3 支持 2 个模式和两个特权等级。

	特权级	用户级
异常handler的代码	handler模式	错误的用法
主应用程序的代码	线程模式	线程模式

图 3.6 操作模式和特权等级

当处理器处在线程状态下时，既可以使用特权级，也可以使用用户级；另一方面，handler 模式总是特权级的。在复位后，处理器进入线程模式+特权级。

在线程模式+用户级下，对系统控制空间 (SCS) 的访问将被阻止——该空间包含了配置寄存器组以及调试组件的寄存器组。除此之外，还禁止使用 MRS/MSR 访问刚才讲到的，除了 APSR 之外的特殊功能寄存器。如果以身试法，则对于访问特殊功能寄存器的，访问操作被忽略；而对于访问 SCS 空间的，将 fault 伺候。

译注：原文的意思是越权访问一律产生 fault (If a program running at the user access level tries to access SCS or special registers, a fault exception will occur)。但译者使用 Keil MDK 开发环境的模拟器和 STM32 单片机作实验时却发现，对特殊功能寄存器越权访问时，仅忽略访问操作，并不产生 fault。另外，译者发现，当使用模拟器时，即使访问了 SCS 中的地址（译者使用的地址是 0xE000E100），模拟器竟然也允许读写！后来译者又使用 STM32 单片机来实验，STM32 单片机则的确产生了总线 fault 并上访成了硬 fault。因此，如果使用指令模拟器，则要小心。附：译者使用的 MDK 版本号是 3.20

在特权级下的代码可以通过置位 CONTROL[0] 来进入用户级。而不管是任何原因产生了任何异常，处理器都将以特权级来运行其服务例程，异常返回后，系统将回到产生异常时所处的级别。用户级下的代码不能再试图修改 CONTROL[0] 来回到特权级。它必须通过一个异常 handler，由那个异常 handler 来修改 CONTROL[0]，才能在返回到线程模式后拿到特权级。

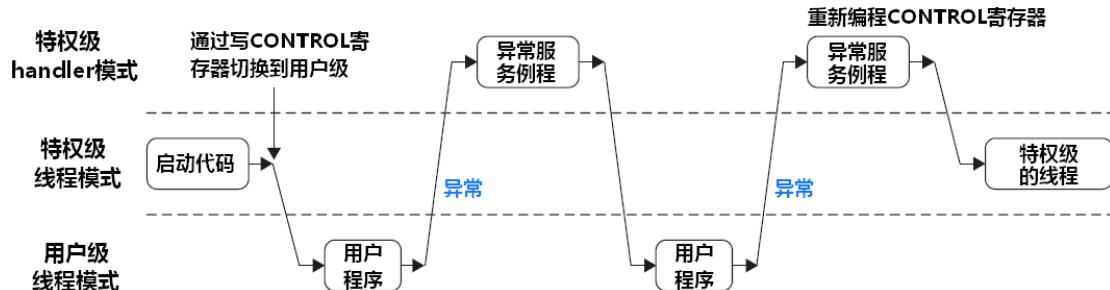


图 3.7 特权级和处理器模式转换图

把代码按特权级和用户极分开对待，有利于使 CM3 的架构更加安全和健壮。例如，当某个用户程序代码出问题时，不会让它成为害群之马，因为用户级的代码是禁止写特殊功能寄存器和 NVIC 中寄存器的。另外，如果还配有 MPU，保护力度就更大，甚至可以阻止用户代码访问不属于它的内

存区域。

为了避免系统堆栈因应用程序的错误使用而毁坏, 我们可以给应用程序专门配一个堆栈, 不让它共享操作系统内核的堆栈。在这个管理制度下, 运行在线程模式的用户代码使用 **PSP**, 而异常服务例程则使用 **MSP**。这两个堆栈指针的切换是智能全自动的, 就在异常服务的始末由 **CM3** 硬件处理。第 8 章将详细讨论此主题。

如前所述, 特权等级和堆栈指针的选择均由 **CONTROL** 负责。当 **CONTROL[0]=0** 时, 在异常处理的始末, 只发生了处理器模式的转换, 如下图所示。

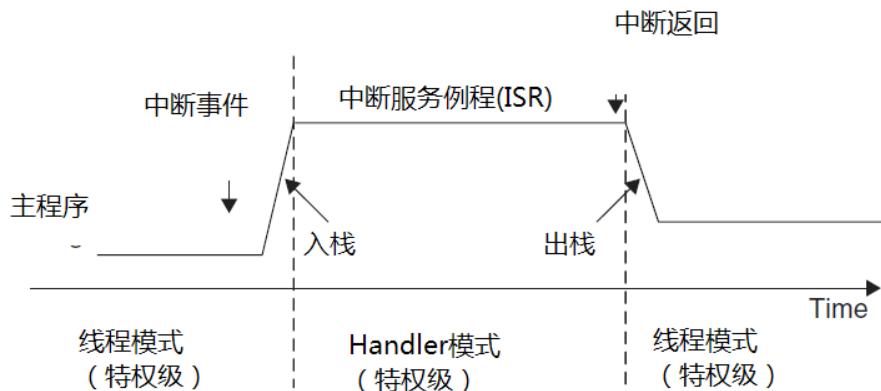


图 3.8 中断前后的状态转换

但若 **CONTROL[0]=1**（线程模式+用户级），则在中断响应的始末，**both** 处理器模式和特权等级都要发生变化，如下图所示。

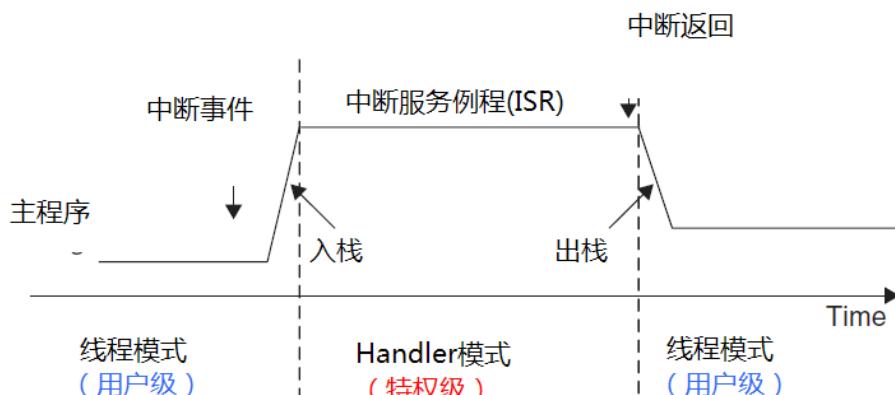


图 3.9 中断前后的状态转换+特权等级切换

CONTROL[0] 只有在特权级下才能访问。用户级的程序如想进入特权级, 通常都是使用一条“系统服务呼叫指令 (SVC)”来触发“SVC 异常”, 该异常的服务例程可以视具体情况而修改 **CONTROL[0]**。

3.4 异常与中断

Cortex-M3 支持大量异常, 包括 $16-4-1=11$ 个系统异常, 和最多 240 个外部中断——简称 **IRQ**。具体使用了这 240 个中断源中的多少个, 则由芯片制造商决定。由外设产生的中断信号, 除了 **SysTick** 的之外, 全都连接到 **NVIC** 的中断输入信号线。典型情况下, 处理器一般支持 16 到 32 个中断, 当然也有在此之外的。

作为中断功能的强化, NVIC 还有一条 NMI 输入信号线。NMI 究竟被拿去做什么, 还要视处理器的设计而定。在多数情况下, NMI 会被连接到一个看门狗定时器, 有时也会是电压监视功能块, 以便在电压掉至危险级别后警告处理器。NMI 可以在任何时间被激活, 甚至是在处理器刚刚复位之后。

表 3.4 列出了 Cortex-M3 可以支持的所有异常。有一定数量的系统异常是用于 fault 处理的, 它们可以由多种错误条件引发。NVIC 还提供了一些 fault 状态寄存器, 以便于 fault 服务例程找出导致异常的具体原因。

表 3.4 Cortex-M3 中的异常类型

编号	类型	优先级	简介
0	N/A	N/A	没有异常在运行
1	复位	-3 (最高)	复位
2	NMI	-2	不可屏蔽中断 (来自外部 NMI 输入脚)
3	硬(hard) fault	-1	所有被除能的 fault, 都将 “上访” 成硬 fault。除能的原因包括当前被禁用, 或者被 PRIMASK 或 BASPRI 掩蔽。
4	MemManage fault	可编程	存储器管理 fault, MPU 访问犯规以及访问非法位置均可引发。企图在 “非执行区” 取指也会引发此 fault
5	总线 fault	可编程	从总线系统收到了错误响应, 原因可以是预取流产 (Abort) 或数据流产, 或者企图访问协处理器
6	用法(usage) Fault	可编程	由于程序错误导致的异常。通常是使用了一条无效指令, 或者是非法的状态转换, 例如尝试切换到 ARM 状态
7-10	保留	N/A	N/A
11	SVCall	可编程	执行系统服务调用指令 (SVC) 引发的异常
12	调试监视器	可编程	调试监视器 (断点, 数据观察点, 或者是外部调试请求)
13	保留	N/A	N/A
14	PendSV	可编程	为系统设备而设的 “可悬挂请求” (pendable request)
15	SysTick	可编程	系统滴答定时器 (也就是周期性溢出的时基定时器——译注)
16	IRQ #0	可编程	外中断#0
17	IRQ #1	可编程	外中断#1
...
255	IRQ #239	可编程	外中断#239

第 7-9 章给出了异常操作的详细信息。

3.5 向量表

当 CM3 内核响应了一个发生的异常后, 对应的异常服务例程(ESR)就会执行。为了决定 ESR 的入口地址, CM3 使用了 “向量表查表机制”。这里使用一张向量表。向量表其实是一个 WORD (32 位整数) 数组, 每个下标对应一种异常, 该下标元素的值则是该 ESR 的入口地址。向量表在地址空间中的位置是可以设置的, 通过 NVIC 中的一个重定位寄存器来指出向量表的地址。在复位后, 该寄存器的值为 0。因此, 在地址 0 处必须包含一张向量表, 用于初始时的异常分配。

表 3.5 向量表结构

异常类型	表项地址	异常向量
偏移量		

0	0x00	MSP 的初始值
1	0x04	复位
2	0x08	NMI
3	0x0C	硬 fault
4	0x10	MemManage fault
5	0x14	总线 fault
6	0x18	用法 fault
7-10	0x1c-0x28	保留
11	0x2c	SVC
12	0x30	调试监视器
13	0x34	保留
14	0x38	PendSV
15	0x3c	SysTick
16	0x40	IRQ #0
17	0x44	IRQ #1
18-255	0x48-0x3FF	IRQ #2 - #239

举个例子，如果发生了异常 11 (SVC)，则 NVIC 会计算出偏移量是 $11 \times 4 = 0x2C$ ，然后从那里取出服务例程的入口地址并跳入。要注意的是这里有个另类：0 号类型并不是什么入口地址，而是给出了复位后 MSP 的初值。

3.6 栈内存操作

在 Cortex-M3 中，除了可以使用 PUSH 和 POP 指令来处理堆栈外，内核还会在异常处理的始末自动地执行 PUSH 与 POP 操作。本节让我们来检视一下具体的动作，第 9 章则讨论异常处理时的自动栈操作。

3.6.1 堆栈的基本操作

笼统地讲，堆栈操作就是对内存的读写操作，但是访问地址由 SP 给出。寄存器的数据通过 PUSH 操作存入堆栈，以后用 POP 操作从堆栈中取回。在 PUSH 与 POP 的操作中，SP 的值会按堆栈的使用法则自动调整，以保证后续的 PUSH 不会破坏先前 PUSH 进去的内容。

堆栈的功能就是把寄存器的数据临时备份在内存中，以便将来能恢复之——在一个任务或一段子程序执行完毕后恢复。正常情况下，PUSH 与 POP 必须成对使用，而且参与的寄存器，不论是身份还是先后顺序都必须完全一致。当 PUSH/POP 指令执行时，SP 指针的值也跟着自减/自增。

... (主程序)

; R0=X, R1=Y, R2=Z

BL Fx1

```

PUSH {R0} ; 把 R0 存入栈 & 调整 SP
PUSH {R1} ; 把 R1 存入栈 & 调整 SP
PUSH {R2} ; 把 R2 存入栈 & 调整 SP
...
; 执行 Fx1 的功能, 中途可以改变 R0-R2 的值

```

```

POP      {R2}      ;恢复 R2 早先的值 & 再次调整 SP
POP      {R1}      ;恢复 R1 早先的值 & 再次调整 SP
POP      {R0}      ;恢复 R0 早先的值 & 再次调整 SP
BX      LR       ;返回

;回到主程序
;R0=X, R1=Y, R2=Z (调用 Fx1 的前后 R0-R2 的值完好无损, 从寄存器上下文来看, 就好像什么都没发生过一样)

```

图 3.10 基本的堆栈操作：每次处理单个寄存器

堆栈操作的进一步探讨

如果参与的寄存器比较多, 这种 PUSH 和 POP 岂不是又臭又长? 放心, PUSH/POP 指令足够体贴, 支持一次操作多个寄存器。像这样:

```

PUSH      {R0-R2}      ;压入 R0-R2
PUSH      {R3-R5, R8, R12} ;压入 R3-R5, R8, 以及 R12
在 POP 时, 可以如下操作:
POP      {R3-R5, R8, R12} ;弹出 R3-R5, R8, 以及 R12
POP      {R0-R2}      ;弹出 R0-R2

```

注意: 在寄存器列表中, 不管寄存器的序号是以什么顺序给出的, 汇编器都将把它们升序排序。然后先 **push** 序号大的寄存器, 所以也就先 **pop** 序号小的寄存器。(这是译者在实验中发现的)。如果不按升序写寄存器, 也许有些汇编器会给出语法错误。

PUSH/POP 对子还有这样一种特殊形式, 形如

```

PUSH      {R0-R3, LR}
POP      {R0-R3, PC}

```

请注意: POP 的最后一个寄存器是 PC, 并不是先前 PUSH 的 LR。这其实是一个返回的小技巧。与其按部就班地把先前 LR 的值弹回 LR, 再复制给 PC 来返回; 不如干脆绕过 LR, 直接传给 PC! 那不怕 LR 的值没有被恢复吗? 不怕, 因为 LR 在子程序调用中的唯一用处, 就是在返回时提供返回地址。因此, 在返回后, 先前保存的返回地址就没有利用价值了, 所以只要 PC 得到了正确的值, 不恢复也没关系。

PUSH 指令等效于与使用 R13 作为地址指针的 STMDB 指令, 而 POP 指令则等效于使用 R13 作为地址指针的 LDMIA 指令——STMDB/LDMIA 还可以使用其它寄存器作为地址指针。至于这两个指令的细节, 第 4 章讲到指令系统时再介绍。

3.7 Cortex-M3 的堆栈实现

Cortex-M3 使用的是“向下生长的满栈”模型。堆栈指针 SP 指向最后一个被压入堆栈的 32 位数值。在下一次压栈时, SP 先自减 4, 再存入新的数值。

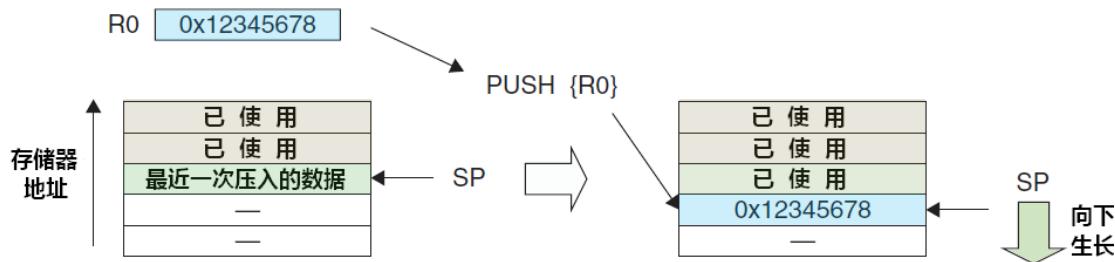


图 3.13 Cortex-M3 堆栈的 PUSH 实现方式

POP 操作刚好相反：先从 SP 指针处读出上一次被压入的值，再把 SP 指针自增 4。

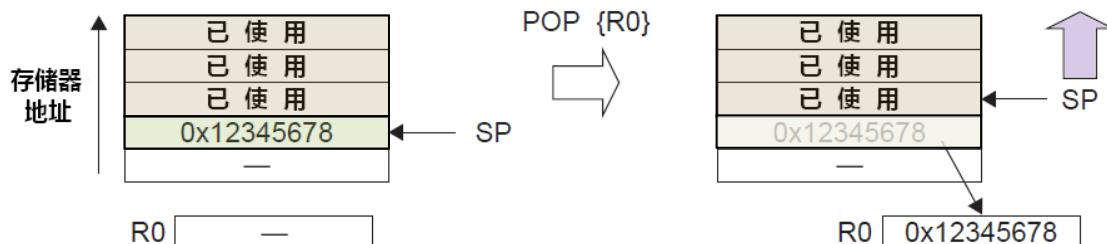


图 3.14 Cortex-M3 堆栈的 POP 实现方式

译注：虽然 POP 后被压入的数值还保存在栈中，但它已经无效了，因为下次的 PUSH 将覆盖它的值！

在进入 ESR 时，CM3 会自动把一些寄存器压栈，这里使用的是发生本异常的瞬间正在使用的 SP 指针（MSP 或者是 PSP）。离开 ESR 后，只要 ESR 没有更改过 CONTROL[1]，就依然使用发生本次异常的瞬间正在使用的 SP 指针来执行出栈操作。

3.7.1 再论 Cortex-M3 的双堆栈机制

我们已经知道了 CM3 的堆栈是分为两个：主堆栈和进程堆栈，CONTROL[1]决定如何选择。

当 CONTROL[1]=0 时，只使用 MSP，此时用户程序和异常 handler 共享同一个堆栈。这也是复位后的缺省使用方式。

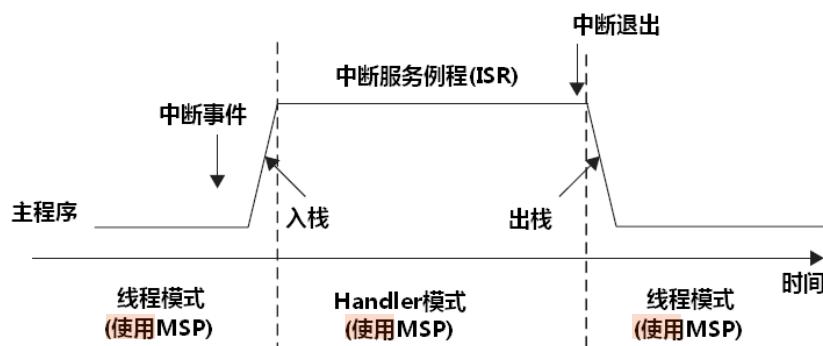


图 3.15 CONTROL[1]=0 时的堆栈使用情况

当 CONTROL[1]=1 时，线程模式将不再使用 MSP，而改用 PSP（handler 模式永远使用 MSP）。这样做好处在哪里？原来，在使用 OS 的环境下，只要 OS 内核仅在 handler 模式下执行，用户应用程序仅在用户模式下执行，这种双堆栈机制派上了用场——防止用户程序的堆栈错误破坏 OS 使用的堆栈。

译注：此时，进入异常时的自动压栈使用的是进程堆栈，进入异常 handler 后才自动改为 MSP，退出异常时切换回 PSP，并且从进程堆栈上弹出数据。

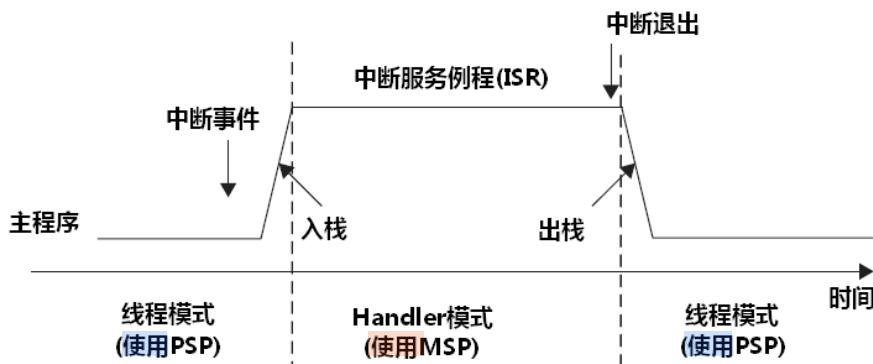


图 3.16 CONTROL[1]=1 时的堆栈切换情况

在特权级下，可以指定具体的堆栈指针，而不受当前使用堆栈的限制，示例代码如下：

```

MRS      R0,      MSP      ; 读取主堆栈指针到 R0
MSR      MSP,      R0      ; 写 R0 的值到主堆栈中
MRS      R0,      PSP      ; 读取进程堆栈指针到 R0
MSR      PSP,      R0      ; 写 R0 的值到进程堆栈中

```

通过读取 PSP 的值，OS 就能够获取用户应用程序使用的堆栈，进一步地就知道了在发生异常时，被压入寄存器的内容，而且还可以把其它寄存器进一步压栈（使用 STMDB 和 LDMIA 的书写形式）。OS 还可以修改 PSP，用于实现多任务中的任务上下文切换。

3.8 复位序列

在离开复位状态后，CM3 做的第一件事就是读取下列两个 32 位整数的值：

- 从地址 0x0000,0000 处取出 MSP 的初始值。
- 从地址 0x0000,0004 处取出 PC 的初始值——这个值是复位向量，LSB 必须是 1。然后从这个值所对应的地址处取指。

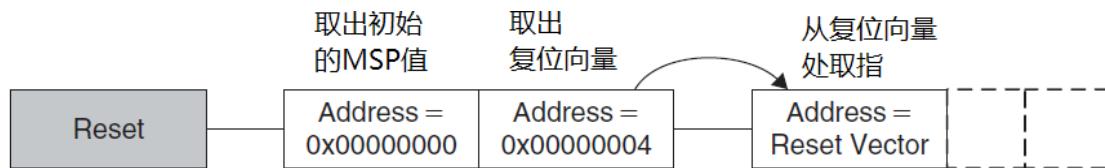


图 3.17 复位序列

请注意，这与传统的 ARM 架构不同——其实也和绝大多数的其它单片机不同。传统的 ARM 架构总是从 0 地址开始执行第一条指令。它们的 0 地址处总是一条跳转指令。在 CM3 中，在 0 地址处提供 MSP 的初始值，然后紧跟着就是向量表（向量表在以后还可以被移至其它位置——译注）。向量表中的数值是 32 位的地址，而不是跳转指令。向量表的第一个条目指向复位后应执行的第一条指令。

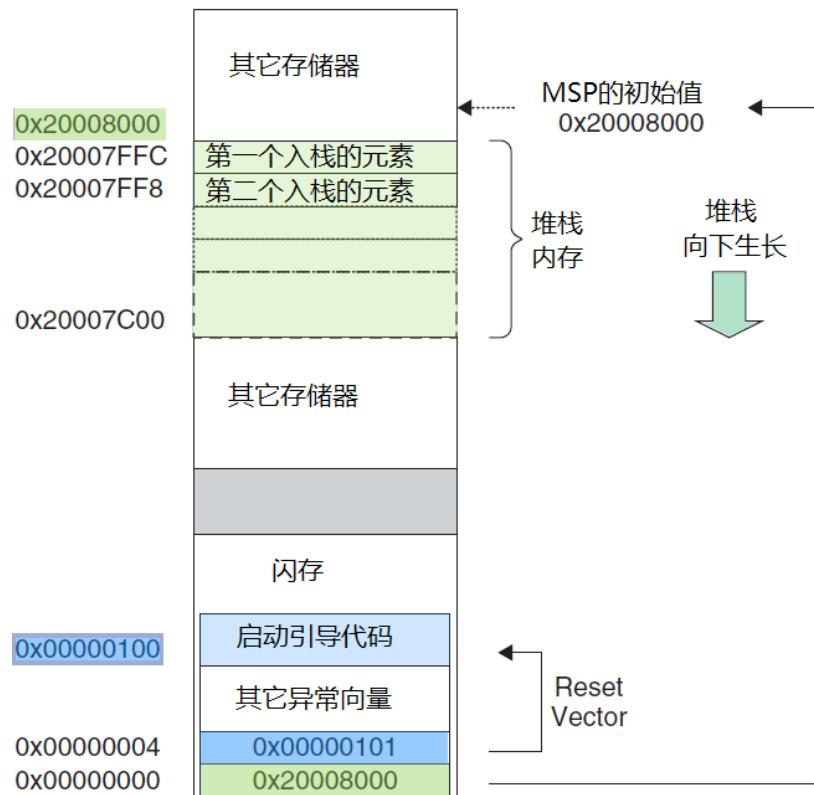


图 3.18 初始 MSP 及 PC 初始化的一个范例

因为 CM3 使用的是向下生长的满栈，所以 MSP 的初始值必须是堆栈内存的末地址加 1。举例来说，如果你的堆栈区域在 0x20007C00-0x20007FFF 之间，那么 MSP 的初始值就必须是 0x20008000。

向量表跟随在 MSP 的初始值之后——也就是第 2 个表目。要注意因为 CM3 是在 Thumb 态下执行，所以向量表中的每个数值都必须把 LSB 置 1（也就是奇数）。正是因为这个原因，图 3.18 中使用 0x101 来表达地址 0x100。当 0x100 处的指令得到执行后，就正式开始了程序的执行。在此之前初始化 MSP 是必需的，因为可能第 1 条指令还没来得及执行，就发生了 NMI 或是其它 fault。MSP 初始化好后就已经为它们的服务例程准备好了堆栈。

对于不同的开发工具，需要使用不同的格式来设置 MSP 初值和复位向量——有些则由开发工具自行计算并生成。如果想要获知细节，最快的办法就是参考开发工具提供的一个示例工程。本书的第 10 章和第 20 章介绍 ARM 提供的开发工具，第 19 章则介绍 GCC 工具链。

指令集

- 汇编语言基础
- 指令集
- 近距离地检视指令
- Cortex-M3 中的一些新好指令

终于“开荤”了，本章开始把 Cortex-M3 的指令系统展现出来，并且给出了一些简单却意味深长的例子。在本书的附录 A 中还有一个快速查阅参考。指令集的详细信息由《ARMv7-M Architecture Application Level Reference Manual》(Ref2)给出——写了两百多页呢。

如果读者以前没有写过 ARM 汇编程序，可以结合看本书的第 20 章，那里讲述了 Keil RVMDK 工具的使用，包括添加汇编源文件的方法。RVMDK 带了一个指令模拟器，对于练习汇编程序非常有帮助。那一章虽然不是很短但很简单。值得一提的是，在那一章的末尾，译者添加了少量内容，是专为学习第 4 章而添加的。

4.1 汇编语言基础

为了给本章的学习扫清障碍，这里我们先简要地介绍一下 ARM 汇编器的基本语法。本书绝大多数的汇编示例都使用 ARM 汇编器的语法，而第 19 章则使用 GCC 汇编器 AS 的语法。

4.1.1 汇编语言：基本语法

汇编指令的最典型书写模式如下所示：

标号

操作码 操作数 1, 操作数 2, ... ;注释

其中，标号是可选的，如果有，它必须顶格写。标号的作用是让汇编器来计算程序转移的地址。

操作码是指令的助记符，它的前面必须有至少一个空白符，通常使用一至二个“Tab”键来产生。操作码后面往往跟随若干个操作数，而第 1 个操作数，通常都给出本指令的执行结果存储处。不同指令需要不同数目的操作数，并且对操作数的语法要求也可以不同。举例来说，立即数必须以“#”开头，如

```
MOV R0, #0x12      ; R0 ← 0x12
MOV R1, #'A'       ; R1 ← 字母 A 的 ASCII 码
```

注释均以”;”开头，它的有无不影响汇编器工作，只是给程序员看的，能让程序更易理解。

还可以使用 EQU 指示字来定义常数，然后在代码中使用它们，例如：

```
NVIC IRQ_SETENO EQU 0xE000E100 ; 注意：常数定义必须顶格写
NVIC IRQ0_ENABLE EQU 0x1

...
LDR R0, =NVIC IRQ_SETENO ; 在这里的 LDR 是个伪指令，它会被汇编器转换成
                           ; 一条“相对 PC 的加载指令”
MOV R1, #NVIC IRQ0_ENABLE ; 把立即数传送到 R1 中
```

```
STR      R1, [R0] ; *R0=R1, 执行完此指令后 IRQ #0 被使能。
```

如果汇编器不能识别某些特殊指令的助记符, 你就要“手工汇编”——查出该指令的确切二进制机器码, 然后使用 **DCI** 编译器指示字。例如, **BKPT** 指令的机器码是 **0xBE00**, 即可以按如下格式书写:

```
DCI      0xBE00 ; 断点(BKPT), 这是一个 16 位指令
```

(在使用 **DCI** 时也必须在前面留出空白符——译注)

类似地, 你还可以使用 **DCB** 来定义一串字节常数, 字节常数还允许以字符串的形式来表达; 还可以使用 **DCD** 来定义一串 32 位整数。它们最常被用来在代码中书写表格。例如:

```
LDR      R3,      =MY_NUMBER      ; R3= MY_NUMBER
LDR      R4,      [R3]          ; R4= *R3
...
LDR      R0,      =HELLO_TEXT    ; R0= HELLO_TEXT
BL      PrintText          ; 呼叫 PrintText 以显示字符串, R0 传递参数
...
MY_NUMBER
DCD      0x12345678
HELLO_TEXT
DCB      "Hello\n",0
```

请注意: 不同汇编器的指示字和语法都可以不同。上述示例代码都是按 ARM 汇编器的语法格式写的。如果使用其它汇编器, 最好看一看它附带的示例代码。

4.1.2 汇编语言: 后缀的使用

在 ARM 处理器中, 指令可以带有后缀, 如表 4.1 所示。

后缀名	含义
S	要求更新 APSR 中的相关标志, 例如: ADDS R0, R1 ; 根据加法的结果更新 APSR 中的标志
EQ,NE,LT,GT 等	有条件地执行指令。EQ=Equal, NE= Not Equal, LT= Less Than, GT= Greater Than, 例如: BEQ <Label> ; 仅当 EQ 满足时转移 除了这 4 种, 还有若干个其它的条件。

在 Cortex-M3 中, 对条件后缀的使用有很大的限制: 只有转移指令 (B 指令) 才可随意使用。而对于其它指令, CM3 引入了 **IF-THEN** 指令块, 在这个块中才可以加后缀, 且必须加以后缀。**IF-THEN** 块由 **IT** 指令定义, 本章稍后将介绍它。另外, **S** 后缀可以和条件后缀在一起使用。共有 15 种不同的条件后缀, 稍后介绍。

4.1.3 汇编语言: 统一汇编语言书写语法

为了最有力地支持 Thumb-2, 也作为对汇编程序员的人文关怀, ARM 汇编器引了一个“统一汇编语言(UAL)”语法机制。对于 16 位指令和 32 位指令均能实现的一些操作(常见于数据处理操作), 有时虽然指令的实际操作数不同, 或者对立即数的长度有不同的限制, 但是汇编器允许开发者统一使用 32 位 Thumb-2 指令的语法格式书写(很多 Thumb-2 指令的用法也与 32 位 ARM 指令相同), 并且由汇编器来决定是使用 16 位指令, 还是使用 32 位指令。以前, Thumb 的语法和 ARM 的语法不

同，在有了 UAL 之后，两者的书写格式就统一了。

```
ADD    R0,      R1      ; 使用传统的 Thumb 语法
ADD    R0,      R0,      R1  ; 引入 UAL 后允许的等效写法 (R0=R0+R1)
```

虽然引入了 UAL，但是仍然允许使用传统的 Thumb 语法。不过有一项必须注意：如果使用传统的 Thumb 语法，有些指令会默认地更新 APSR，即使你没有加上 S 后缀。如果使用 UAL 语法，则必须指定 S 后缀才会更新。例如：

```
AND    R0,      R1      ; 传统的 Thumb 语法
ANDS   R0,      R0,      R1  ; 等值的 UAL 语法 (必须有 S 后缀)
```

在 Thumb-2 指令集中，有些操作既可以由 16 位指令完成，也可以由 32 位指令完成。例如， $R0=R0+1$ 这样的操作，16 位的与 32 位的指令都提供了助记符为“ADD”的指令。在 UAL 下，汇编器能主动决定用哪个，也可以手工指定是用 16 位的还是 32 位的：

```
ADDS   R0,      #1      ; 汇编器将为了节省空间而使用 16 位指令
ADDS.N R0,      #1      ; 指定使用 16 位指令 (N=Narrow)
ADDS.W R0,      #1      ; 指定使用 32 位指令 (W=Wide)
```

.W(Wide)后缀指定 32 位指令。如果没有给出后缀，汇编器会先试着用 16 位指令以给代码瘦身，如果不行再使用 32 位指令。因此，使用“.N”其实是多此一举，不过汇编器可能仍然允许这样的语法。

再次重申，这是 ARM 公司汇编器的语法，其它汇编器的可能略有区别，但如果没给出后缀，汇编器就总是会尽量选择更短的指令。

其实在绝大多数情况下，应用程序是用 C 写的，C 编译器也会尽可能地使用短指令。然而，当立即数超出一定范围时，或者 32 位指令能更好地适合某个操作，将使用 32 位指令。

32 位 Thumb-2 指令也可以按半字对齐（以前 ARM 32 位指令都必须按字对齐——译注），因此下例是允许的：

```
0x1000: LDR    r0, [r1]      ; 一个 16 位的指令
0x1002: RBIT.W r0          ; 一个 32 位的指令，以 0x1002 为起始地址，跨越了字的边界
```

绝大多数 16 位指令只能访问 R0-R7；32 位 Thumb-2 指令则可以随意访问 R0-R15。不过，把 R15(PC) 作为目的寄存器很容易走火入魔——用对了会有意想不到的妙处，出错时则会使程序跑飞。通常只有系统软件才会不惜冒险地做此高危行为，因此还需慎用。对 PC 的使用还有其它的戒律，如果感兴趣，可以参考《ARMv7-M 架构应用级参考手册》。

4.2 指令集

Cortex-M3 支持的指令在表 4.2 至表 4.9 列出。其中，译者添加了如下格式
边框加粗的是从 ARMv6T2 才支持的指令。

双线边框的是从 Cortex-M3 才支持的指令 (v7 的其它款式不一定支持)

译者添加

在讲指令之前，先简单地介绍一下 Cortex-M3 中支持的算术与逻辑标志。本书在后面还会展开论述。它们是：

APSR 中的 5 个标志位

- N: 负数标志(Negative)
- Z: 零结果标志(Zero)
- C: 进位/借位标志(Carry)
- V: 溢出标志(oVerflow)
- S: 饱和标志(Saturation)，它不做条件转移的依据

4.2.1 分类指令表

表 4.2 16 位数据操作指令

名字	功能
ADC	带进位加法
ADD	加法
AND	按位与 (原文为逻辑与, 有误——译注)。这里的按位与和 C 的“&”功能相同
ASR	算术右移
BIC	按位清 0 (把一个数跟另一个无符号数的反码按位与)
CMN	负向比较 (把一个数跟另一个数据的二进制补码相比较)
CMP	比较 (比较两个数并且更新标志)
CPY	把一个寄存器的值拷贝到另一个寄存器中
EOR	近位异或
LSL	逻辑左移 (如无其它说明, 所有移位操作都可以一次移动最多 31 格——译注)
LSR	逻辑右移
MOV	寄存器加载数据, 既能用于寄存器间的传输, 也能用于加载立即数
MUL	乘法
MVN	加载一个数的 NOT 值 (取到逻辑反的值)
NEG	取二进制补码
ORR	按位或 (原文为逻辑或, 有误——译注)
ROR	圆圈右移
SBC	带借位的减法
SUB	减法
TST	测试 (执行按位与操作, 并且根据结果更新 Z)
REV	在一个 32 位寄存器中反转字节序
REVH	把一个 32 位寄存器分成两个 16 位数, 在每个 16 位数中反转字节序
REVSH	把一个 32 位寄存器的低 16 位半字进行字节反转, 然后带符号扩展到 32 位
SXTB	带符号扩展一个字节到 32 位
SXTH	带符号扩展一个半字到 32 位
UXTB	无符号扩展一个字节到 32 位
UXTH	无符号扩展一个半字到 32 位

表 4.3 16 位转移指令

名字	功能
B	无条件转移
B<cond>	条件转移
BL	转移并连接。用于呼叫一个子程序, 返回地址被存储在 LR 中
BLX #im	使用立即数的 BLX 不要在 CM3 中使用
CBZ	比较, 如果结果为 0 就转移 (只能跳到后面的指令——译注)
CBNZ	比较, 如果结果非 0 就转移 (只能跳到后面的指令——译注)

IT	If-Then
----	---------

表 4.4 16 位存储器数据传送指令

名字	功能
LDR	从存储器中加载字到一个寄存器中
LDRH	从存储器中加载半字到一个寄存器中
LDRB	从存储器中加载字节到一个寄存器中
LDRSH	从存储器中加载半字，再经过带符号扩展后存储一个寄存器中
LDRSB	从存储器中加载字节，再经过带符号扩展后存储一个寄存器中
STR	把一个寄存器按字存储到存储器中
STRH	把一个寄存器存器的低半字存储到存储器中
STRB	把一个寄存器的低字节存储到存储器中
LDMIA	加载多个字，并且在加载后自增基址寄存器
STMIA	存储多个字，并且在存储后自增基址寄存器
PUSH	压入多个寄存器到栈中
POP	从栈中弹出多个值到寄存器中

16 数据传送指令没有任何新内容，因为它们是 Thumb 指令，在 v4T 时就已经定格了——译注

表 4.5 其它 16 位指令

名字	功能
SVC	系统服务调用
BKPT	断点指令。如果使能了调试，则进入调试状态（停机）。否则的话产生调试监视器异常。在调试监视器异常被使能时，调用其服务例程；如果连调试监视器异常也被禁能，则无奈下只好诉诸于一个 fault 异常
NOP	无操作
CPSIE	使能 PRIMASK(CPSIE i)/ FAULTMASK(CPSIE f)——清 0 相应的位
CPSID	除能 PRIMASK(CPSID i)/ FAULTMASK(CPSID f)——置位相应的位

表 4.6 32 位数据操作指令

名字	功能
ADC	带进位加法
ADD	加法
ADDW	宽加法（可以加 12 位立即数）
AND	按位与（原文是逻辑与，有误。对应 C 言的“ ”运算符——译注）
ASR	算术右移
BIC	位清零（把一个数按位取反后，与另一个数逻辑与）
BFC	位段清零
BFI	位段插入

CMN	负向比较 (把一个数和另一个数的二进制补码比较, 并更新标志位)
CMP	比较两个数并更新标志位
CLZ	计算前导零的数目
EOR	按位异或
LSL	逻辑左移
LSR	逻辑右移
MLA	乘加
MLS	乘减
MOVW	把 16 位立即数放到寄存器的低 16 位, 高 16 位清 0
MOV	加载 16 位立即数到寄存器 (其实汇编器会产生 MOVW——译注)
MOVT	把 16 位立即数放到寄存器的高 16 位, 低 16 位不影响
MVN	移动一个数的补码
MUL	乘法
ORR	按位或 (原文为逻辑或, 有误——译注)
ORN	把源操作数按位取反后, 再执行按位或 (原文为逻辑或, 有误——译注)
RBIT	位反转 (把一个 32 位整数用 2 进制表达后, 再旋转 180 度——译注)
REV	对一个 32 位整数按字节反转
REVH/	对一个 32 位整数的高低半字都执行字节反转
REV16	
REVSH	对一个 32 位整数的低半字执行字节反转, 再带符号扩展成 32 位数
ROR	圆圈右移
RRX	带进位位的逻辑右移一格 (最高位用 C 填充, 执行后不影响 C 的值——译注)
SFBX	从一个 32 位整数中提取任意长度和位置的位段, 并且带符号扩展成 32 位整数
SDIV	带符号除法
SMLAL	带符号长乘加 (两个带符号的 32 位整数相乘得到 64 位的带符号积, 再把积加到另一个带符号 64 位整数中)
SMULL	带符号长乘法 (两个带符号的 32 位整数相乘得到 64 位的带符号积)
SSAT	带符号的饱和运算
SBC	带借位的减法
SUB	减法
SUBW	宽减法, 可以减 12 位立即数
SXTB	字节带符号扩展到 32 位数
TEQ	测试是否相等 (对两个数执行异或, 更新标志但不存储结果)
TST	测试 (对两个数执行按位与, 更新 Z 标志但不存储结果)
UBFX	无符号位段提取
UDIV	无符号除法
UMLAL	无符号长乘加 (两个无符号的 32 位整数相乘得到 64 位的无符号积, 再把积加到另一个无符号 64 位整数中)
UMULL	无符号长乘法 (两个无符号的 32 位整数相乘得到 64 位的无符号积)
USAT	无符号饱和操作 (但是源操作数是带符号的——译注)
UXTB	字节被无符号扩展到 32 位 (高 24 位清 0——译注)

UXTH	半字被无符号扩展到 32 位（高 16 位清 0——译注）
------	-------------------------------

表 4.7 32 位存储器数据传送指令

名字	功能
LDR	加载字到寄存器
LDRB	加载字节到寄存器
LDRH	加载半字到寄存器
LDRSH	加载半字到寄存器，再带符号扩展到 32 位
LDM	从一片连续的地址空间中加载若干个字，并选中相同数目的寄存器放进去
LDRD	从连续的地址空间加载双字（64 位整数）到 2 个寄存器
STR	存储寄存器中的字
STRB	存储寄存器中的低字节
STRH	存储寄存器中的低半字
STM	存储若干寄存器中的字到一片连续的地址空间中，占用相同数目的字
STRD	存储 2 个寄存器组成的双字到连续的地址空间中
PUSH	把若干寄存器的值压入堆栈中
POP	从堆栈中弹出若干的寄存器的值

表 4.8 32 位转移指令

名字	功能
B	无条件转移
BL	转移并连接（呼叫子程序）
TBB	以字节为单位的查表转移。从一个字节数组中选一个 8 位前向跳转地址并转移
TBH	以半字为单位的查表转移。从一个半字数组中选一个 16 位前向跳转的地址并转移

表 4.9 其它 32 位指令

LDREX	加载字到寄存器，并且在内核中标明一段地址进入了互斥访问状态
LDREXH	加载半字到寄存器，并且在内核中标明一段地址进入了互斥访问状态
LDREXB	加载字节到寄存器，并且在内核中标明一段地址进入了互斥访问状态
STREX	检查将要写入的地址是否已进入了互斥访问状态，如果是则存储寄存器的字
STREXH	检查将要写入的地址是否已进入了互斥访问状态，如果是则存储寄存器的半字
STREXB	检查将要写入的地址是否已进入了互斥访问状态，如果是则存储寄存器的字节
CLREX	在本地处理器上清除互斥访问状态的标记（先前由 LDREX/LDREXH/LDREXB 做的标记）
MRS	加载特殊功能寄存器的值到通用寄存器
MSR	存储通用寄存器的值到特殊功能寄存器
NOP	无操作
SEV	发送事件
WFE	休眠并且在发生事件时被唤醒
WFI	休眠并且在发生中断时被唤醒

ISB	指令同步隔离（与流水线和 MPU 等有关——译注）
DSB	数据同步隔离（与流水线、MPU 和 cache 等有关——译注）
DMB	数据存储隔离（与流水线、MPU 和 cache 等有关——译注）

4.2.2 未支持的指令

有若干条 Thumb 指令没有得到 Cortex-M3 的支持，下表列出了未被支持的指令，以及不支持的原因。

表 4.10 因为不再是传统的架构，导致有些指令已失去意义

未支持的 指令	以前的功能
BLX #im	在使用立即数做操作数时，BLX 总是要切入 ARM 状态。因为 Cortex-M3 只在 Thumb 状态下运行，故以此指令为代表的，凡是试图切入 ARM 状态的操作，都将引发一个用法 fault。
SETEND	由 ARMv6 引入的，在运行时改变处理器端设置的指令（大端或小端）。因为 Cortex-M3 不支持动态端的功能，所以此指令也将引发 fault

CM3 也不支持有少量在 ARMv7-M 中列出的指令。比如，ARMv7M 支持 Thumb2 的协处理器指令，但是 CM3 却不能挂协处理器。表 4.11 列出了这些与协处理器相关的指令。如果试图执行它们，则将引发用法 fault（NVIC 中的 NOCP（No CoProcessor）标志置位）。

表 4.11 不支持的协处理器相关指令

未支持 的指令	以前的功能
MCR	把通用寄存器的值传送到协处理器的寄存器中
MCR2	把通用寄存器的值传送到协处理器的寄存器中
MCR2	把通用寄存器的值传送到协处理器的寄存器中，一次操作两个
MRC	把协处理器寄存器的值传送到通用寄存器中
MRC2	把协处理器寄存器的值传送到通用寄存器中
MRC2	把协处理器寄存器的值传送到通用寄存器中，一次操作两个
LDC	把某个连续地址空间中的一串数值传送到协处理器中
STC	从协处理器中传送到一段地址空间中

还有一个是改变处理器状态指令（CPS），它的一些用法也不再支持。这是因为 PSRs 的定义已经变了，以前在 ARMv6 中定义的某些位在 CM3 中并不存在。

表 4.12 不支持的 CPS 指令用法

未支持的指令	以前的功能
CPS<IE/ID>.W-A	CM3 没有“A”位

~~CPS.W #mode~~ CM3 的 PSR 中没有“mode”位

有些提示 (hint) 指令的功能不支持, 它们在 CM3 中按 “NOP” 指令对待

表 4.13 不支持的 hint 指令

未支持的指令	以前的功能
DBG	服务于跟踪系统的一条 hint 指令
PLD	预取数据。这是服务于 cache 系统的一条 hint 指令。因为在 CM3 中没有 cache, 该指令就相当于 NOP
PLT	预取指令。这是服务于 cache 系统的一条 hint 指令。因为在 CM3 中没有 cache, 该指令就相当于 NOP
SWI#0	用于多线程处理。线程使用该指令通知给硬件: 我正在做的任务可以被交换出去 (swapped out), 从而提高系统的整体性能。

4.3 近距离检视指令

从现在起, 我们将介绍一些在 ARM 汇编代码中很通用的指令及其语法。有些指令可以带有多种附加处理, 比如预移位操作。本章不会讲得面面俱到, 但理解本章后足以应付大多数大型汇编程序开发。

4.3.1 汇编语言: 数据传送

处理器的基本功能之一就是数据传送。CM3 中的数据传送类型包括

- 在两个寄存器间传送数据
- 在寄存器与存储器间传送数据
- 在寄存器与特殊功能寄存器间传送数据
- 把一个立即数加载到寄存器

用于在寄存器间传送数据的指令是 MOV。比如, 如果要把 R3 的数据传送给 R8, 则写作:

MOV R8, R3

MOV 的一个衍生物是 MVN, 它把寄存器的内容取反后再传送。

用于访问存储器的基础指令是“加载 (Load)”和“存储 (Store)”。加载指令 LDR 把存储器中的内容加载到寄存器中, 存储指令 STR 则把寄存器的内容存储至存储器中, 传送过程中数据类型也可以变通, 最常使用的格式有:

表 4.14 常用的存储器访问指令

示例	功能描述
LDRB Rd, [Rn, #offset]	从地址 Rn+offset 处读取一个字节送到 Rd
LDRH Rd, [Rn, #offset]	从地址 Rn+offset 处读取一个半字送到 Rd
LDR Rd, [Rn, #offset]	从地址 Rn+offset 处读取一个字送到 Rd
LDRD Rd1, Rd2, [Rn, #offset]	从地址 Rn+offset 处读取一个双字(64 位整数)送到 Rd1 (低 32 位) 和 Rd2 (高 32 位) 中。

STRB Rd, [Rn, #offset]	把 Rd 中的低字节存储到地址 Rn+offset 处
STRH Rd, [Rn, #offset]	把 Rd 中的低半字存储到地址 Rn+offset 处
STR Rd, [Rn, #offset]	把 Rd 中的低字存储到地址 Rn+offset 处
STRD Rd1, Rd2, [Rn, #offset]	把 Rd1 (低 32 位) 和 Rd2 (高 32 位) 表达的双字存储到地址 Rn+offset 处

如果嫌一口一口地蚕食太不过瘾, 也可以使用 LDM/STM 来鲸吞。它们相当于把若干个 LDR/STR 给合并起来了, 有利于减少代码量, 如表 4.15 所示

表 4.15 常用的多重存储器访问方式

示例	功能描述
LDMIA Rd!, {寄存器列表}	从 Rd 处读取多个字, 并依次送到寄存器列表中的寄存器。每读一个字后 Rd 自增一次, 16 位宽度
STMIA Rd!, {寄存器列表}	依次存储寄存器列表中各寄存器的值到 Rd 给出的地址。每存一个字后 Rd 自增一次, 16 位宽度
LDMIA.W Rd!, {寄存器列表}	从 Rd 处读取多个字, 并依次送到寄存器列表中的寄存器。每读一个字后 Rd 自增一次, 32 位宽度
LDMDB.W Rd!, {寄存器列表}	从 Rd 处读取多个字, 并依次送到寄存器列表中的寄存器。每读一个字前 Rd 自减一次, 32 位宽度
STMIA.W Rd!, {寄存器列表}	依次存储寄存器列表中各寄存器的值到 Rd 给出的地址。每存一个字后 Rd 自增一次, 32 位宽度
STMDB.W Rd!, {寄存器列表}	存储多个字到 Rd 处。每存一个字前 Rd 自减一次, 32 位宽度

译注: 上表中, 加粗的是符合 CM3 堆栈操作的 LDM/STM 使用方式。并且, 如果 Rd 是 R13(即 SP), 则与 POP/PUSH 指令等效。(LDMIA->POP, STMDB -> PUSH)

STMDB SP!, {R0-R3, LR} 等效于 PUSH {R0-R3, LR}
LDMIA SP!, {R0-R3, PC} 等效于 PUSH {R0-R3, PC}

Rd 后面的 “!” 是什么意思? 它表示要自增(I_ncrement)或自减(D_ecrement)基址寄存器Rd的值, 时机是在每次访问前(B_{ef}ore)或访问后(A_{ft}er)。增/减单位: 字(4字节)。例如, 记R8=0x8000, 则下面两条指令:

STMIA.W R8!, {R0-R3}; R8 值变为 0x8010, 每存一次增一次, 先存储后自增
STMDB.W R8, {R0-R3}; R8 值的“一个内部复本”先自减后再存储数据, 但 R8 的值不变

感叹号还可以用于单一加载与存储指令——LDR/STR。这也就是所谓的“带预索引”(Pre-indexing)的 LDR 和 STR。例如:

LDR.W R0, [R1, #20]! ; 预索引

该指令先把地址 R1+offset 处的值加载到 R0, 然后, R1 \leftarrow R1+ 20 (offset 也可以是负数——译注)。这里的“!”就是指在传送后更新基址寄存器 R1 的值。“!”是可选的。如果没有“!”, 则该指令就是普通的带偏移量加载指令, 不会自动调整 R0 的值。带预索引的数据传送可以用在多种数据类型上, 并且既可用于加载, 又可用于存储。

表 4.16 预索引数据传送的常见用法

示例	功能描述
LDR.W Rd, [Rn, #offset]! LDRB.W Rd, [Rn, #offset]! LDRH.W Rd, [Rn, #offset]! LDRD.W Rd1, Rd2, [Rn, #offset]!	字/字节/半字/双字的带预索引加载（不做带符号扩展，没有用到的高位全清 0——译注）
LDRSB.W Rd, [Rn, #offset]! LDRSH.W Rd, [Rn, #offset]!	字节/半字的带预索引加载，并且在加载后执行带符号扩展成 32 位整数
STR.W Rd, [Rn, #offset]! STRB.W Rd, [Rn, #offset]! STRH.W Rd, [Rn, #offset]! STRD.W Rd1, Rd2, [Rn, #offset]!	字/字节/半字/双字的带预索引存储

CM3 除了支持“预索引”外，还支持“后索引”(Post-indexing)。后索引也要使用一个立即数 offset，但与预索引不同的是，后索引忠实地使用基址寄存器 Rd 的值，把它作为传送的目的地址。待到数据传送后，再执行 $Rd \leftarrow Rd + offset$ (offset 可以是负数——译注)。如：

STR.W R0, [R1], #-12 ;后索引

该指令是把 R0 的值存储到地址 R1 处的。在存储完毕后， $R1 \leftarrow R1 + (-12)$

注意，[R1]后面是没有“!”的。可见，在后索引中，基址寄存器是无条件被更新的——也可以理解为有一个“隐藏”的“!”

表 4.17 后索引的常见用法

示例	功能描述
LDR.W Rd, [Rn], #offset LDRB.W Rd, [Rn], #offset LDRH.W Rd, [Rn], #offset LDRD.W Rd1, Rd2, [Rn], #offset	字/字节/半字/双字的带预索引加载（不做带符号扩展，没有用到的高位全清 0——译注）
LDRSB.W Rd, [Rn], #offset LDRSH.W Rd, [Rn], #offset	字节/半字的带预索引加载，并且在加载后执行带符号扩展成 32 位整数
STR.W Rd, [Rn], #offset STRB.W Rd, [Rn], #offset STRH.W Rd, [Rn], #offset STRD.W Rd1, Rd2, [Rn], #offset	字/字节/半字/双字的后预索引存储

译者添加

立即数的位数是有限制的，且不同指令的限制可以不同。这下岂不是要有的背了？其实不必！因为如果在使用中超过了限制，则汇编器会报错，所以不用担心会背成书呆子。

那能彻底消灭这种限制吗？办法是有的，只是要使用另一种形式的 LDR/STR。事实上，在 CM3 中的偏移量，除了可以使用形如#offset 的立即数，还可以使用一个寄存器。使用寄存器来提供偏移量，就可以“天南地北任我行”了。不过，如果使用寄存器提供偏移量，就不能使用“预索引”和“后索引”了——也就是说不能修改基址寄存器的值。因此下面的写法就是非法的：

```
ldr      r2,      [r0,      r3]!      ; 错误，寄存器提供偏移量时不支持预索引
ldr      r2,      [r0],    r3      ; 错误，寄存器提供偏移量时不支持后索引
```

这看起来令人扫兴吗？不过也有好消息。当使用寄存器作索引时，可以“预加工”索引寄存器的值——逻辑左移。显然，这与 C 语言数组下标的寻址方式刚好吻合，如

```
ldr      r2,      [r0, r3, lsl #2]
```

译注：PUSH/POP 作为堆栈专用操作，也属于数据传送指令类。

通常 PUSH/POP 对子的寄存器列表是严格一致的，但是 PC 与 LR 的使用方式有新意，如
;子程序入口

```
PUSH    {R0-R3, LR}
```

...

;子程序出口

```
POP     {R0-R3, PC}
```

在这个例子中，旁路了 LR，直截了当地返回。

数据传送指令还包括 MRS/MSR。还记得第 3 章讲到过 CM3 有若干个特殊功能寄存器吗？MRS/MSR 就是专门用于访问这些寄存器的。不过，这些寄存器都是维持系统正常工作的重地，因此，按理说是不能允许随意访问它们的。然而，在 CM3 上，它们是“春色满园关不住，一枝红杏出墙来”——大原则是必须在特权级下才允许访问，以免系统因误操作或恶意破坏而功能紊乱，可是 APSR 却允许在用户级下访问，想必是为了让我们有妙用非主流技巧的余地吧。CM3 的这个禁律是由硬件强制执行的，如果在用户级下以身试法，则 fault 伺候（产生 MemManage fault，若被除能则“上访”成硬 fault）〔译注〕。通常，只有系统软件（如 OS）才会操作这类寄存器，应用程序，尤其是用 C 编写的应用程序，是从来不关心这些的。

译注：当使用 MRS 访问 APSR 时，是把各个标志位按照它们在 xPSR 中占用的位序号，直接复制到寄存器中的。例如，进位标志 C 是在 xPSR.30 中，在执行了 MRS R0, APSR 指令后，则 R0.30=C；反之亦然，欲设置 C 位，必须在 R0.30 中给出新的 C 的值。另外，译者在模拟器和 STM32 单片机中尝试了在用户级下访问 PSR，并没有产生 fault，只是改写 APSR 以外的部分被忽略而已。但我觉得不必太过争论处理器的具体处理方式，因为不管怎么说这都是错误的编程行为。

下面轮到立即数上场。代码写多了我们就常常会感觉到，程序中会经常使用立即数。最典型的就是：当我们访问某个地址时，必须先把该地址加载到一个寄存器中，这就包含了一个 32 位立即数加载操作。CM3 中的 MOV/MVN 指令族负责加载立即数，族中各个成员支持的立即数位数不同。例如，16 位指令 MOV 支持 8 位立即数加载，如：

```
MOV R0, #0x12
```

32 位指令 **MOVW** 和 **MOVT** 可以支持 16 位立即数加载。

那要加载 32 位立即数怎么办呢？如果要直来直去，当前是要用两条指令来完成了。通过组合使用 **MOVW** 和 **MOVT** 就能产生 32 位立即数，但是要注意，必须先使用 **MOVW**，再使用 **MOVT**。这种顺序是不能颠倒的，因为 **MOVW** 会清零高 16 位。

不过，更流行的是另一种方法：使用汇编器提供的“**LDR Rd, =imm32**”伪指令。例如：

```
LDR, r0, =0x12345678
```

酷吧！它的名字也是 **LDR**，而且能加载 32 位立即数！但可别忘了，它是伪指令，是“妖怪变的”，而且有若干种原形。所以不要因为名字相同就混淆。

大多数情况下，当汇编器遇到 **LDR** 伪指令时，都会把它转换成一条相对于 PC 的加载指令，来产生需要的数据。。大可依赖汇编器，它会明智地使用最合适的形式来实现该伪指令。

译者添加：如果某指令需要使用 32 位立即数，可以在该指令地址的附近定义一个 32 位整数数组，把这个立即数放到该数组中。然后使用一条 **LDR Rd, [PC, #offset]** 来查表。offset 的值需要计算，它其实是 **LDR** 指令的地址与该数组元素地址的距离。手工计算 offset 是很自虐的作法，而刚才讲到的 **LDR** 伪指令则能让汇编器来自动生成这种数组，并且负责计算 offset。这种数组被广泛使用，它的学名叫“文字池”(literal pool)，通常由汇编器自动布设，汇编程序很大时可能也需要手工布设(通过 **LTORG** 指示字)。

LDR 伪指令 vs. ADR 伪指令

Both **LDR** 和 **ADR** 都有能力产生一个地址，语法和行为都有相似处，但却不尽相同。对于 **LDR**，如果汇编器发现要产生立即数是一个程序地址，它会自动地把 **LSB** 置位，例如：

```
LDR r0, =address1 ; R0= 0x4000 | 1
```

...

address1

```
0x4000: MOV R0, R1
```

在这个例子中，汇编器会认出 **address1** 是一个程序地址，所以自动置位 **LSB**。另一方面，如果汇编器发现要加载的是数据地址，则不会自作聪明，多机灵啊！看：

```
LDR R0, =address1 ; 会把 0x4000 原封不动地加载到 R0
```

...

address1

```
0x4000: DCD 0x0 ; 0x4000 处记录的是一个数据
```

ADR 指令则永远是“憨厚”的，它决不会擅自修改 **LSB**。例如：

```
ADR r0, address1 ; R0= 0x4000。注意：没有“=”号
```

...

address1

```
0x4000: MOV R0, R1
```

ADR 将如实地加载 **0x4000**。注意，语法略有不同，没有“=”号。

前面已经提到，**LDR** 通常是把要加载的数值预先定义，再使用一条 PC 相对加载指令来取出。而 **ADR** 则尝试对 PC 作算术加法或减法来取得立即数。因此 **ADR** 未必总能求出需要的立即数。其实顾名思义，**ADR** 是为了取出附近某条指令或者变量的地址，而 **LDR** 则是取出一个通用的 32 位整数。因为 **ADR** 更专一，所以得到了优化——它产生的代码效率常常比 **LDR** 的要高。

4.3.2 汇编语言：数据处理

数据处理乃是处理器的看家本领，CM3 当然要出类拔萃，它提供了丰富多彩的相关指令，每种指令的用法也是花样百出。限于篇幅，这里只列出常用的使用方式。就以加法为例，常见的有：

```
ADD    R0,      R1          ; R0 += R1
ADD    R0,      #0x12        ; R0 += 12
ADD.W  R0,      R1,      R2  ; R0 = R1+R2
```

注意：虽然助记符都是“ADD”，但是二进制机器码是不同的。

当使用 16 位加法时，会自动更新 APSR 中的标志位。然而，在使用了“.W”显式指定了 32 位指令后，就可以通过“S”后缀手工控制对 APSR 的更新，如：

```
ADD.W  R0,      R1,      R2  ; 不更新标志位
ADDS.W R0,      R1,      R2  ; 更新标志位
```

除了 ADD 指令之外，CM3 中还包含 SUB, MUL, UDIV/SDIV 等用于算术四则运算，如表 4.18 所列

表 4.18 常见的算术四则运算指令

示例	功能描述
ADD Rd, Rn, Rm ; Rd = Rn+Rm	常规加法
ADD Rd, Rm ; Rd += Rm	imm 的范围是 im8(16 位指令)或 im12(32 位指令)
ADD Rd, #imm ; Rd += imm	
ADC Rd, Rn, Rm ; Rd = Rn+Rm+C	带进位的加法
ADC Rd, Rm ; Rd += Rm+C	imm 的范围是 im8(16 位指令)或 im12(32 位指令)
ADC Rd, #imm ; Rd += imm+C	
ADDW Rd, #imm12 ; Rd += imm12	带 12 位立即数的常规加法
SUB Rd, Rn ; Rd -= Rn	常规减法
SUB Rd, Rn, #imm3 ; Rd = Rn-imm3	
SUB Rd, #imm8 ; Rd -= imm8	
SUB Rd, Rn, Rm ; Rd = Rm-Rn	
SBC Rd, Rm ; Rd -= Rm+C	带借位的减法
SBC.W Rd, Rn, #imm12 ; Rd = Rn-imm12-C	
SBC.W Rd, Rn, Rm ; Rd = Rn-Rm-C	
RSB.W Rd, Rn, #imm12 ; Rd = imm12-Rn	反向减法
RSB.W Rd, Rn, Rm ; Rd = Rm-Rn	
MUL Rd, Rm ; Rd *= Rm	常规乘法
MUL.W Rd, Rn, Rm ; Rd = Rn*Rm	
MLA Rd, Rm, Rn, Ra ; Rd = Ra+Rm*Rn	乘加与乘减
MLS Rd, Rm, Rn, Ra ; Rd = Ra-Rm*Rn	(译者添加)
UDIV Rd, Rn, Rm ; Rd = Rn/Rm (无符号除法)	硬件支持的除法
SDIV Rd, Rn, Rm ; Rd = Rn/Rm (带符号除法)	

CM3 还片载了硬件乘法器，支持乘加/乘减指令，并且能产生 64 位的积，如表 4.19 所示

表 4.19 64 位乘法指令

示例	功能描述
SMULL RL, RH, Rm, Rn ; [RH:RL]= Rm*Rn	带符号的 64 位乘法
SMLAL RL, RH, Rm, Rn ; [RH:RL]+= Rm*Rn	
UMULL RL, RH, Rm, Rn ; [RH:RL]= Rm*Rn	无符号的 64 位乘法
SMLAL RL, RH, Rm, Rn ; [RH:RL]+= Rm*Rn	

逻辑运算以及移位运算也是基本的数据操作。表 4.20 列出 CM3 在这方面的常用指令

表 4.20 常用逻辑操作指令

示例	功能描述
AND Rd, Rn ; Rd &= Rn AND.W Rd, Rn, #imm12 ; Rd = Rn & imm12 AND.W Rd, Rm, Rn ; Rd = Rm & Rn	按位与
ORR Rd, Rn ; Rd = Rn ORR.W Rd, Rn, #imm12 ; Rd = Rn imm12 ORR.W Rd, Rm, Rn ; Rd = Rm Rn	按位或
BIC Rd, Rn ; Rd &= ~Rn BIC.W Rd, Rn, #imm12 ; Rd = Rn & ~imm12 BIC.W Rd, Rm, Rn ; Rd = Rm & ~Rn	位段清零
ORN.W Rd, Rn, #imm12 ; Rd = Rn ~imm12 ORN.W Rd, Rm, Rn ; Rd = Rm ~Rn	按位或反码
EOR Rd, Rn ; Rd ^= Rn EOR.W Rd, Rn, #imm12 ; Rd = Rn ^ imm12 EOR.W Rd, Rm, Rn ; Rd = Rm ^ Rn	(按位) 异或, 异或总是按位的

译者添加

大多数涉及 3 个寄存器的 32 位数据操作指令，都可以在计算之前，对其第 3 个操作数 Rn 进行“预加工”——移位，格式为：

DataOp	Rd,	Rm,	Rn,	LSL #imm5	; 先对 Rn 逻辑左移 imm5 格
DataOp	Rd,	Rm,	Rn,	LSR #imm5	; 先对 Rn 逻辑右移 imm5 格
DataOp	Rd,	Rm,	Rn,	ASR #imm5	; 先对 Rn 算术右移 imm5 格
DataOp	Rd,	Rm,	Rn,	ROR #imm5	; 先对 Rn 圆圈右移 imm5 格
DataOp	Rd,	Rm,	Rn,	ROL #imm5	; (错误) 先对 Rn 循环左移 imm5 格
DataOp	Rd,	Rm,	Rn,	RRX	; 先对 Rn 带进位位右移一格

注意：“预加工”是对 Rn 的一个“内部复本”执行操作，不会因此而影响 Rn 的值。但如果 Rn 正巧也

Cortex-M3 还支持为数众多的移位运算。移位运算既可以与其它指令组合使用（传送指令和数据操作指令中的一些，参见文本框中的说明），也可以独立使用，如表 4.21 所示。

表 4.21 移位和循环指令

示例	功能描述
LSL Rd, Rn, #imm5 ; Rd = Rn<<imm5	逻辑左移
LSL Rd, Rn ; Rd <= Rn	
LSL.W Rd, Rm, Rn ; Rd = Rm<<Rn	
LSR Rd, Rn, #imm5 ; Rd = Rn>>imm5	逻辑右移
LSR Rd, Rn ; Rd >= Rn	
LSR.W Rd, Rm, Rn ; Rd = Rm>>Rn	
ASR Rd, Rn, #imm5 ; Rd = Rn \ggg imm5	算术右移
ASR Rd, Rn ; Rd \ggg = Rn	
ASR.W Rd, Rm, Rn ; Rd = Rm \ggg Rn	
ROR Rd, Rn ; Rd \gg = Rn	圆圈右移
ROR.W Rd, Rm, Rn ; Rd = Rm \gg Rn	
RRX.W Rd, Rn ; Rd = (Rn>>1)+(C<<31)	带进位的右移一格 亦可写作 RRX{S} Rd。此时, Rd 也要担当 Rn 的角色——译注
译者添加 (因为在 RRX 上使用 S 后缀比较特殊, 故提出来单独讲解) RRXS.W Rd, Rn ; tmpBit = Rn & 1 ; Rd = (Rn>>1)+(C<<31) ; C = tmpBit	

如果在移位和循环指令上加上“S”后缀, 这些指令会更新进位位 C。如果是 16 位 Thumb 指令, 则总是更新 C 的。图 4.1 给出了一个直观的印象

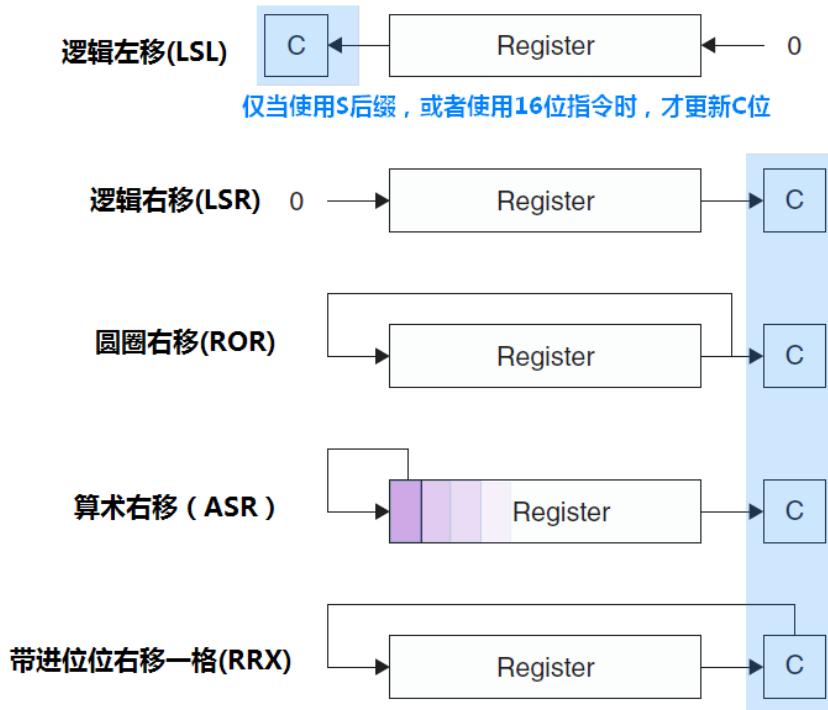


图 4.1 移位与循环指令

为什么没有圆圈左移？

在圆圈移位中，寄存器的 32 个位其实是手拉手组成一个圈的。那么这个圈向右转动 n 格，与向左转动 $32-n$ 格是等效的。这种简单的道理，玩过“丢手绢”的小朋友们都知道。因此欲圆圈左移 n 格时，只要使用圆圈右移指令，并且转动 $32-n$ 格即可。

介绍完了移位指令，接下来讲带符号扩展指令。

我们知道，在 2 进制补码表示法中，最高位是符号位，且所有负数的符号位都是 1。负数还有另一个性质，就是不管在符号位的前面再添加多少个 1，值都不变，只不过表达带符号整数的位数增多了。于是，在把一个 8 位或 16 位负数扩展成 32 位时，欲使其数值不变，就必须把所有新增的高位全填 1。至于正数或无符号数，则只需简单地把新增的高位清 0。因此，必须给带符号数开小灶，于是就有了整数扩展指令，如表 4.22 所示。

表 4.22 带符号扩展指令

示例	功能描述
SXTB Rd, Rm ; Rd = Rm 的带符号扩展	把带符号字节整数扩展到 32 位
SXTH Rd, Rm ; Rd = Rm 的带符号扩展	把带符号半字整数扩展到 32 位

我们知道，32 位整数可以被认为是由 4 个字节拼接成的，也可以被认为是由 2 个半字拼接成的。有时，需要把这些子元素倒腾倒腾，如表 4.23 所示

表 4.23 数据序翻转指令

示例	功能描述
REV.W Rd, Rn	在字中反转字节序
REV16.W Rd, Rn	在高低半字中反转字节序
REVSH.W	在低半字中反转字节序，并做带符号扩展

这些指令乍一看不太好理解，但相信看过图 4.2 后就会豁然开朗了：

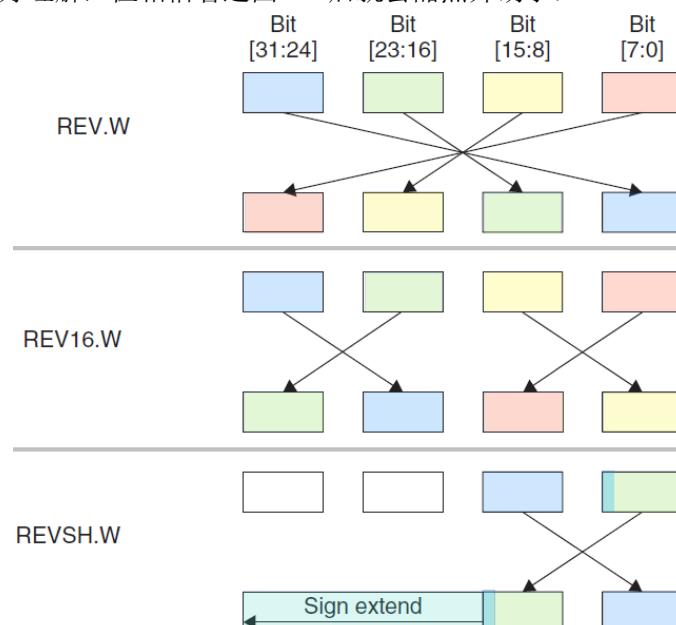


图 4.2 反序操作

数据操作指令的最后一批，是位操作指令。位操作在单片机程序中，以及在系统软件中应中常常大显身手，而且在使用这类指令时，有很多新奇但是却妙用横生的技巧。这里在表 4.24 中先列出它们，本书在后续的小节中还要展开论述。

表 4.24 位段处理及把玩指令

指令	功能描述
BFC.W Rd, Rn, #<width>	
BFI.W Rd, Rn, #<lsb>, #<width>	
CLZ.W Rd, Rn	计算前导 0 的数目
RBIT.W Rd, Rn	位反转，按位旋转 180 度
SBFX.W Rd, Rn, #<lsb>, #<width>	拷贝位段，并带符号扩展到 32 位
SBFX.W Rd, Rn, #<lsb>, #<width>	拷贝位段，并无符号扩展到 32 位

4.3.3 汇编语言：子程呼叫与无条件跳转指令

最基本的无条件跳转指令有两条：

B Label ; 跳转到 Label 处对应的地址
BX reg ; 跳转到由寄存器 reg 给出的地址

在 BX 中，reg 的最低位指示出在转移后将进入的状态：是 ARM(LSB=0)呢，还是 Thumb(LSB=1)。既然 CM3 只在 Thumb 中运行，就必须保证 reg 的 LSB=1，否则一个 fault 打过来。

呼叫子程序时，需要保存返回地址，正点的指令是：

BL Label ; 跳转到 Label 对应的地址，并且把跳转前的下条指令地址保存到 LR
BLX reg ; 跳转到由寄存器 reg 给出的地址，并根据 REG 的 LSB 切换处理器状态，
; 还要把转移前的下条指令地址保存到 LR

执行这些指令后，就把返回地址存储到 LR (R14) 中了，从而才能使用“BX LR”等形式返回。

使用 BLX 要小心，因为它还带有改变状态的功能。因此 reg 的 LSB 必须是 1，以确保不会试图进入 ARM 状态。如果忘记置位 LSB，则 fault 伺候。

对于艺高胆大的玩家来说，使用以 PC 为目的寄存器的 MOV 和 LDR 指令也可以实现转移，并且往往能借此实现很多“老实”的程序达不到的功效，常见形式有：

MOV PC, Rn ; 转移地址由 Rn 给出
LDR PC, [Rn] ; 转移地址存储在 Rn 所指向的存储器中
POP {..., PC} ; 把返回地址以弹出堆栈的风格送给 PC，
; 从而实现跳转，这也是比较平易近人的技巧
LDMIA SP!, {..., PC} ; POP 的另一种等效写法

同理，使用这些技巧，你也必须保证送给 PC 的值必须是奇数 (LSB=1)。

注意：有心的读者可能已经发现，ARM 的 BL 虽然省去了耗时的访内操作，却只能支持一级子程序调用。如果子程序再呼叫“孙程序”，则返回地址会被覆盖。因此当函数嵌套多于一级时，必须在调用“孙程序”之前先把 LR 压入堆栈——也就是所谓的“溅出”。

4.3.4 汇编语言：标志位与条件转移

在应用程序状态寄存器中有 5 个标志位，但只有 4 个被条件转移指令参考。绝大多数 ARM 的条件转移指令根据它们来决定是否转移，如表 4.25 所示

表 4.25 Cortex-M3 APSR 中可以影响条件转移的 4 个标志位

标志位	PSR 位序号	功能描述
N	31	负数（上一次操作的结果是个负数）。N=操作结果的 MSB
Z	30	零（上次操作的结果是 0）。当数据操作指令的结果为 0，或者比较/测试的结果为 0 时，Z 置位。
C	29	进位 / 借位（上次操作导致了进位或者借位）。C 用于无符号数据处理，最常见的就是当加法进位及减法借位时 C 被置位。此外，C 还充当移位指令的中介（详见 v7M 参考手册的指令介绍节）。
V	28	溢出（上次操作结果导致了数据的溢出）。该标志用于带符号的数据处理。比如，在两个正数上执行 ADD 运算后，和的 MSB 为 1（视作负数），则 V 置位。

在 ARM 中，数据操作指令可以更新这 4 个标志位。这些标志位除了可以当作条件转移的判据之外，还能在一些场合下作为指令是否执行的依据（详见 If-Then 指令块），或者在移位操作中充当各种中介角色（仅进位位 C）。

担任条件跳转及条件执行的判据时，这 4 个标志位既可单独使用，又可组合使用，以产生共 15 种跳转判据，如下表 4.26 所示

表 4.26 跳转及条件执行判据

符号	条件	关系到的标志位
EQ	相等 (Equal)	Z==1
NE	不等 (NotEqual)	Z==0
CS/HS	进位 (CarrySet) 无符号数高于或相同	C==1
CC/LO	未进位 (CarryClear) 无符号数低于	C==0
MI	负数 (Minus)	N==1
PL	非负数	N==0
VS	溢出	V==1
VC	未溢出	V==0
HI	无符号数大于	C==1 && Z==0
LS	无符号数小于等于	C==0 Z==1
GE	带符号数大于等于	N==V
LT	带符号数小于	N!=V
GT	带符号数大于	Z==0 && N==V

LE	带符号数小于等于	$Z==1 \mid\mid N!=V$
AL	总是	-

表中共有 15 个条件组合 (AL 相当于无条件——译注), 通过把它们点缀在无条件转移指令 (B) 的后面, 即可做成各式各样的条件转移指令, 例如:

BEQ label ;当 $Z=1$ 时转移

亦可以在指令后面加上“.W”, 来强制使用 Thumb-2 的 32 位指令来做更远的转移(其实没必要, 汇编器会自行判断——译注), 例如:

BEQ.W label

这些条件组合还可以用在 If-Then 语句块中, 比如:

```
CMP R0, R1 ; 比较 R0, R1
IT TET GT ; If R0>R1 Then (T代表Then, E代表Else)
MOVGT R2, R0
MOVGT R3, R1
MOVLE R2, R0
MOVGT R3, R1
```

(本章的后面有对 IT 指令和 If-Then 块进行详细说明。请留意上例中对字体和格式的严格使用——译注)

在 CM3 中, 下列指令可以更新 PSR 中的标志:

- 16 位算术逻辑指令
- 32 位带 S 后缀的算术逻辑指令
- 比较指令 (如, CMP/CMN) 和测试指令 (如 TST/TEQ)
- 直接写 PSR/APSR (MSR 指令)

大多数 16 位算术逻辑指令不由分说就会更新标志位(不是所有的 16 位指令都这样, 例如 ADD.N Rd, Rn, Rm 是 16 位指令, 但不更新标志位——译注), 32 位的都可以让你使用 S 后缀来控制。例如:

```
ADDS.W R0, R1, R2 ; 使用 32 位 Thumb-2 指令, 并更新标志
ADD.W R0, R1, R2 ; 使用 32 位 Thumb-2 指令, 但不更新标志位
ADD R0, R1 ; 使用 16 位 Thumb 指令, 无条件更新标志位
ADDS R0, #0xcd ; 使用 16 位 Thumb 指令, 无条件更新标志位
```

译注: 虽然真实指令的行为如上所述。但是在你用汇编语言写代码时, 如果使用了 UAL (统一汇编语言), 汇编器会做调整, 最终生成的指令不一定和与你在字面上写的指令相同。对于 ARM 汇编器而言, 调整的结果是: **如果没有写后缀 s, 汇编器就一定会产生不更新标志位的指令**。可见, 使用 UAL 的一大好处, 就是我们完全能控制是否更新标志位, 决不会出现标志位被意外更新的情况。

S 后缀的使用要当心。16 位 Thumb 指令可能会无条件更新标志位, 但也可能不更新标志位。为了让你的代码能在不同汇编器下有相同的行为, 当你需要更新标志, 以作为条件指令的执行判据时, 一定不要忘记加上 S 后缀。

CM3 中还有比较和测试指令, 它们存在的目的就是更新标志位, 因此是会无条件影响标志位的, 如下所述。

CMP 指令。 CMP 指令在内部做两个数的减法, 并根据差来设置标志位, 但是不把差写回。CMP 可有如下的形式:

CMP R0, R1 ; 计算 R0-R1 的差, 并且根据结果更新标志位

CMP R0, 0x12 ; 计算 R0-0x12 的差, 并且根据结果更新标志位

CMN 指令。 CMN 是 CMP 的一个孪生姊妹, 只是它在内部做两个数的加法 (相当于减去减数的

相反数), 如下所示:

```
CMN    R0,    R1      ; 计算 R0+R1 的和, 并根据结果更新标志位
CMN    R0,    0x12    ; 计算 R0+0x12 的和, 并根据结果更新标志位
```

TST 指令。**TST** 指令的内部其实就是 **AND** 指令, 只是不写回运算结果, 它也无条件更新标志位。它的用法与 **CMP** 的用法相同:

```
TST    R0,    R1      ; 计算 R0 & R1, 并根据结果更新标志位
TST    R0,    0x12    ; 计算 R0 & 0x12, 并根据结果更新标志位
```

TEQ 指令。**TEQ** 指令的内部其实就是 **EOR** 指令, 只是不写回运算结果, 它也无条件更新标志位。它的用法与 **CMP** 的用法相同:

```
TEQ    R0,    R1      ; 计算 R0 ^ R1, 并根据结果更新标志位
TEQ    R0,    0x12    ; 计算 R0 ^ 0x12, 并根据结果更新标志位
```

4.3.5 汇编语言: 指令隔离(barrier)指令和存储器隔离指令

CM3 中的另一股新鲜空气是一系列的隔离指令(亦可以译成“屏障”、“路障”, 可互换使用——译者注)。它们在一些结构比较复杂的存储器系统中是需要的(典型地用于流水线和写缓冲——译者注)。在这类系统中, 如果没有必要的隔离, 会导致系统发生紊乱危象(race condition), (相当于数电中的“竞争与冒险”——译者注)。

举例来说, 如果可以在运行时更改存储器的映射关系或者内存保护区的设置,(通过写 **MPU** 的寄存器), 就必须在更改之后立即补上一条 **DSB** 指令(数据同步指令)。因为对 **MPU** 的写操作很可能会被放到一个写缓冲中。写缓冲是为了提高存储器的总体访问效率而设的, 但它也有副作用, 其中之一, 就是会导致写内存的指令被延迟几个周期执行, 因此对存储器的设置不能即刻生效, 这会导致紧临着的下一条指令仍然使用旧的存储器设置——但程序员的本意显然是使用新的存储器设置。这种紊乱危象是后患无穷的, 常会破坏未知地址的数据, 有时也会产生非法地址访问 **fault**。紊乱危象还有其它的表现形式, 后续章节会一一介绍。CM3 提供隔离指令族, 就是要消灭这些紊乱危象(在有些讲解计算机体系结构的书中, 这类紊乱危象也被称为“存储器相关”——译注)。

CM3 中共有 3 条隔离指令, 如表 4.27 所列

表 4.27 隔离指令

指令名	功能描述
DMB	数据存储器隔离。 DMB 指令保证: 仅当所有在它前面的存储器访问操作都执行完毕后, 才提交(commit)在它后面的存储器访问操作。
DSB	数据同步隔离。比 DMB 严格: 仅当所有在它前面的存储器访问操作都执行完毕后, 才执行在它后面的指令(亦即任何指令都要等待存储器访问操作——译者注)
ISB	指令同步隔离。最严格: 它会清洗流水线, 以保证所有它前面的指令都执行完毕之后, 才执行它后面的指令。

DMB 在双口 **RAM** 以及多核架构的操作中很有用。如果 **RAM** 的访问是带缓冲的, 并且写完之后马上读, 就必须让它“喘口气”——用 **DMB** 指令来隔离, 以保证缓冲中的数据已经落实到 **RAM** 中。**DSB** 比 **DMB** 更保险(当然也是有执行代价的), 它是宁可错杀也不漏网——清空了写缓冲, 使得任何它后面的指令, 不管要不要使用先前的存储器访问结果, 通通等待访问完成。大虾们可以在有绝对信心时使用 **DMB**, 新手还是使用 **DSB** 比较保险。

同 **DMB/DSB** 相比, **ISB** 指令看起来似乎最强悍, 但是却一身都是“愣劲”, 不由分说就“动粗”。

不过它还有其它的用场——对于高级底层技巧：“自我更新”(self-modifying)代码，非常有用。举例来说，如果某个程序从下一条要执行的指令处更新了自己，但是先前的旧指令已经被预取到流水线中去了，此时就必须清洗流水线，把旧版本的指令洗出去，再预取新版本的指令。因此，必须在被更新代码段的前面使用 ISB，以保证旧的代码从流水线中被清洗出去，不再有机会执行（译者觉得这种做法太工于技巧，有点“作秀”，现实编程中应该极少会用到，因此读者不必太钻它）。

4.3.6 汇编语言：饱和运算

饱和运算可能是读者在以前不太听说的。不过其实很简单。如果读者学过模电，或者知道放大电路中所谓的“饱和削顶尖失真”，理解饱和运算就更加容易。而且饱和运算指令确实是打算用于信号处理程序的。

CM3 中的饱和运算指令分为两种：一种是“没有直流分量”的交流信号饱和——带符号饱和运算；另一种无符号饱和运算则类似于“削顶尖失真+单向导通”。

饱和运算多用于信号处理。比如，信号放大。当信号被放大后，有可能使它的幅值超出允许输出的范围。如果傻乎乎地只是清除 MSB，则常常会严重破坏信号的波形，而饱和运算则只是使信号产生削顶尖失真。如图 4.3 所示。

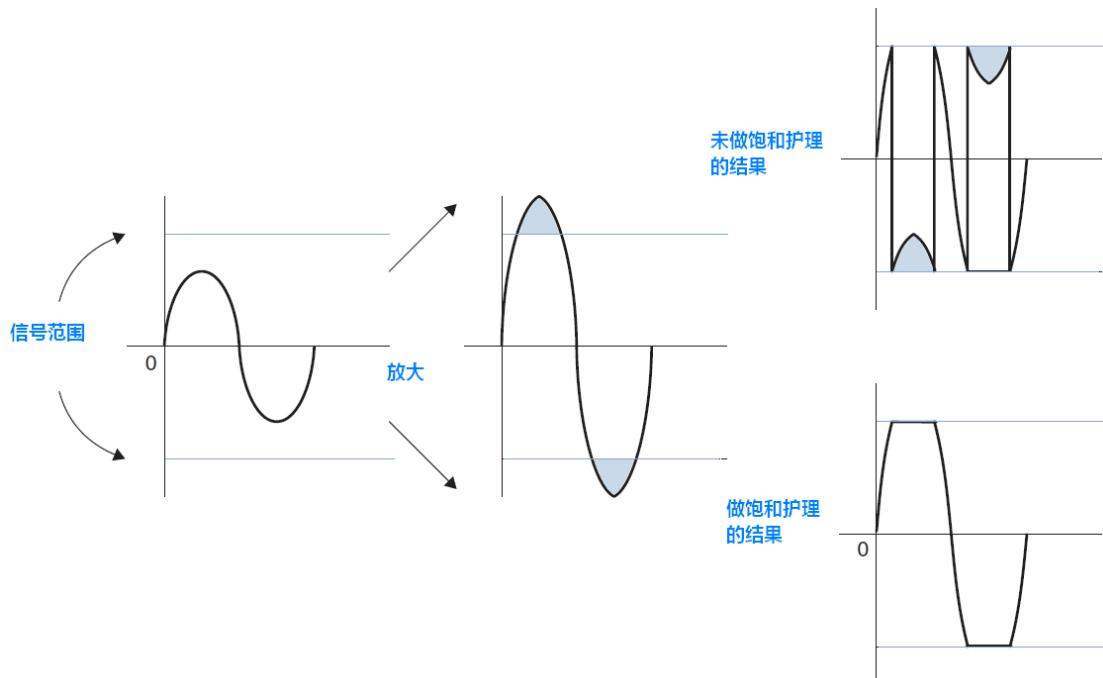


图 4.3 带符号饱和运算

可见，饱和运算的“护理”虽然不能消灭失真，但那种委琐的变形是可以消灭的。表 4.28 列出饱和运算指令。

表 4.28 饱和运算指令

指令名	功能描述
SSAT.W Rd, #imm5, Rn, {,shift}	以带符号数的边界进行饱和运算（交流）
SSAT.W Rd, #imm5, Rn, {,shift}	以无符号数的边界进行饱和运算（带纹波的直流）

饱和运算的结果可以拿去更新 Q 标志(在 APSR 中)。Q 标志在写入后可以用软件清 0——通过写 APSR——还记得 APSR 是“红杏出墙”的吗？

Rn 存储“放大后待做饱和运算的信号”，(Rn 总是 32 位带符号整数——译者注)。同很多其它数据操作指令类似，Rn 也可以使用移位来“预加工”。

Rd 存储饱和运算的结果。

#imm5 用于指定饱和边界——该由多少位的带符号整数来表达允许的范围 (奇数也可以使用)，取值范围是 1-32。举例来说，如果要把一个 32 位(带符号)整数饱和到 12 位带符号整数(-2048 至 2047)，则可以如下使用 SSAT 指令

SSAT{.W} R1, #12, R0

这条指令对于 R0 不同值的执行结果如表 4.29 所示

表 4.29 带符号饱和运算的示例运算结果

输入(R0)	输出(R1)	Q 标志位
0x2000(8192)	0x7FF(2047)	1
0x537(1335)	0x537(1335)	无变化
0x7FF(2047)	0x7FF(2047)	无变化
0	0	无变化
0xFFFFE000(-8192)	0xFFFF800(-2048)	1
0xFFFFFB32(-1230)	0xFFFFFB32(-1230)	无变化

如果需要把 32 位整数饱和到无符号的 12 位整数 (0-4095)，则可以如下使用 USAT 指令

USAT{.W} R1, #12, R0

该指令的执行情况如图 4.4 演示

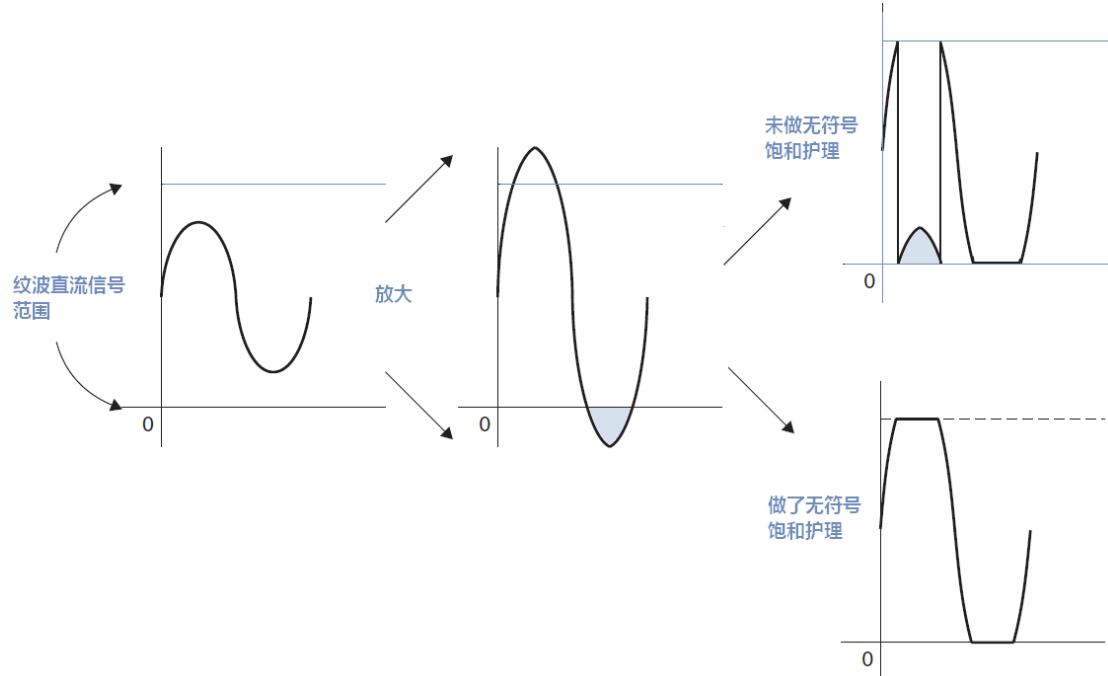


图 4.4 无符号饱和运算

表 4.30 无符号饱和运算的示例运算结

输入(R0)	输出(R1)	Q 标志位
0x2000(8192)	0FFF(4095)	1
0xFFFF(4095)	0FFF(4095)	无变化
0x1000(4096)	0FFF(4095)	1
0x800(2048)	0x800(2048)	无变化
0	0	无变化

0x80000000(-2G)	0	1
0xFFFFFB32(-1230)	0	1

4.4 CM3 中一些前卫的指令

这里列出几条从 v6 和 v7 开始才支持的最新指令。

4.4.1 MRS 和 MSR

虽然名字与以前的 ARM 相同, 但功能变了。这两条指令是访问特殊功能寄存器的“专用通道”——当然必须在特权级下使用, 除了 APSR 可以在用户级下访问外。

指令语法如下:

MRS **<Rn>, <SReg>** ; 加载特殊功能寄存器的值到 Rn
MSR **<Sreg>, <Rn>** ; 存储 Rn 的值到特殊功能寄存器

SReg 可以是下表中的一个:

表 4.31 MRS/MSR 可以使用的特殊功能寄存器

符号	功能
IPSR	当前服务中断号寄存器
EPSR	执行状态寄存器 (读回来的总是 0)。它里面含 T 位, 在 CM3 中 T 位必须是 1, 所以在更改 EPSR 时要格外小心——译注。
APSR	上条指令结果的标志
IEPSR	IPSR+EPSR
IAPSR	IPSR+APSR
EAPSR	EPSR+APSR
PSR	$xPSR = APSR+EPSR+IPSR$
MSP	主堆栈指针
PSP	进入堆栈指针
PRIMASK	常规异常屏蔽寄存器
BASEPRI	常规异常的优先级阈值寄存器
BASEPRI_MAX	等同 BASEPRI, 但是施加了写的限制: 新的优先级比较比旧的高 (更小的数)
FAULTMASK	fault 屏蔽寄存器 (除了包含 PRIMASK 的全部功能外, 还能除能硬 fault)
CONTROL	控制寄存器 (堆栈选择, 特权等级设置)

下面给出一个指定 PSP 进行更新的例子:

```
LDR    R0,      =0x20008000
MSR    PSP,      R0
BX    LR        ; 如果是从异常返回到线程状态, 则使用新的 PSP 的值作为栈顶指针
```

4.4.2 IF-THEN

IF-THEN(IT)指令围起一个块, 里面最多有 4 条指令, 它里面的指令可以条件执行。

IT 指令已经带了一个“T”, 因此还可以最多再带 3 个“T”或者“E”。并且对 T 和 E 的顺序没有要求。其中 T 对应条件成立时执行的语句, E 对应条件不成立时执行的语句。在 If-Then 块中的指令必须加上条件后缀, 且 T 对应的指令必须使用和 IT 指令中相同的条件, E 对应的指令必须使用和 IT 指令中相反的条件。

IT 的使用形式总结如下：

IT	<cond>	; 围起 1 条指令的 IF-THEN 块
IT<x>	<cond>	; 围起 2 条指令的 IF-THEN 块
IT<x><y>	<cond>	; 围起 3 条指令的 IF-THEN 块
IT<x><y><z>	<cond>	; 围起 4 条指令的 IF-THEN 块

其中<x>, <y>, <z>的取值可以是“T”或者“E”。而<cond>则是在表 4.26 中列出的条件 (AL 附录)。

[译注 17]: IT 指令使能了指令的条件执行方式，并且使 CM3 不再预取不满足条件的指令。又因为它在使用时取代了条件转移指令，还避免了在执行流转移时，对流水线的清洗和重新指令预取的开销，所以能优化 C 结构中的微小 if 块和很多“?:”运算符

IT 指令优化 C 代码的例子如下面伪代码所示：

```
if (R0==R1)
{
    R3 = R4 + R5;
    R3 = R3 / 2;
}
else
{
    R3 = R6 + R7;
    R3 = R3 / 2;
}
```

可以写作：

CMP R0, R1	; 比较 R0 和 R1
ITTEE EQ	; 如果 R0 == R1, Then-Then-Else-Else
ADDEQ R3, R4, R5	; 相等时加法
ASREQ R3, R3, #1	; 相等时算术右移
ADDNE R3, R6, R7	; 不等时加法
ASRNE R3, R3, #1	; 不等时算术右移

4.4.3 CBZ 和 CBNZ

比较并条件跳转指令专为循环结构的优化而设，它只能做前向跳转。语法格式为：

CBZ <Rn>, <label>
CBNZ <Rn>, <label>

它们的跳转范围较窄，只有 0-126。

典型范围如下所示：

```
while (R0!=0)
{
    Function1();
}
变成
Loop
    CBZ R0, LoopExit
    BL Function1
```

B Loop

LoopExit:

与其它的比较指令不同，CBZ/CBNZ 不会更新标志位。

4.4.4 SDIV 和 UDIV

突破性的 32 位硬件除法指令，如下所示：

SDIV.W Rd, Rn, Rm

UDIV.W Rd, Rn, Rm

运算结果是 $Rd = Rn/Rm$ ，余数被丢弃。例如：

```
LDR        r0,        =300
MOV        R1,        #7
UDIV.W    R2,        R0,        R1
```

则 $R2 = 300/7 = 44$

为了捕捉被零除的非法操作，你可以在 NVIC 的配置控制寄存器中置位 DIVBYZERO 位。这样，如果出现了被零除的情况，将会引发一个用法 fault 异常。如果没有任何措施，Rd 将在除数为零时被清零。

4.4.5 REV, REVH, REV16 以及 REVSH

REV 反转 32 位整数中的字节序，REVH 则以半字为单位反转，且只反转低半字。语法格式为：

REV Rd, Rm
REVH Rd, Rm
REV16 Rd, Rm
REVSH Rd, Rm

例如，记 R0=0x12345678，在执行下列两条指定后：

```
REV        R1,        R0
REVH      R2,        R0
REV16     R3,        R0
```

则 R1=0x78563412，R2=0x12347856，R3=0x34127856。这些指令专门服务于小端模式和大端模式的转换，最常用于网络应用程序中（网络字节序是大端，主机字节序常是小端）。

REVSH 在 REVH 的基础上，还把转换后的半字做带符号扩展。例如，记 R0=0x33448899，则

REVSH R1, R0

执行后，R1=0xFFFF9988

4.4.6 RBIT

RBIT 比前面的 REV 之流更精细，它是按位反转的，相当于把 32 位整数的二进制表示法水平旋转 180 度。其格式为：

RBIT.W Rd, Rn

这个指令在处理串行比特流时大有用场，而且几乎到了没它不行的地步（不信可以去写段程序完成它的功能，看看颠来倒去的能不能把你“转晕”）。

例如，记 R1=0xB4E10C23（二进制数值为 1011,0100,1110,0001,0000,1100,0010,0011），

RBIT.W R0, R1

执行后，则 $R0=0xC430872D$ （二进制数值为 1100,0100,0011,0000,1000,0111,0010,1101）

这条指令单独使用时看不出什么作用，但是与其它指令组合使用时往往有特效，高级技巧常用到它。

4.4.7 SXTB, SXTH, UXTB, UXTH

这 4 个指令是为了体贴 C 语言的强制数据类型转换而设的，把数据宽度转换成处理器喜欢的 32 位长度（处理器字长是多少，就喜欢多长的整数，其操作效率和存储效率都最高）。它们的语法如下：

```
SXTB      Rd, Rn
SXTH      Rd, Rn
SXTB      Rd, Rn
UXTH      Rd, Rn
```

对于 SXTB/SXTH，数据带符号位扩展成 32 位整数。对于 UXTB/UXTH，高位清零。例如，记 $R0=0x55aa8765$ ，则

```
SXTB      R1, R0      ; R1=0x00000065
SXTH      R1, R0      ; R1=0xffff8765
UXTB      R1, R0      ; R1=0x00000065
UXTH      R1, R0      ; R1=0x00008765
```

4.4.8 BFC/BFI, UBFX/SBFX

这些是 CM3 提供的位段操作指令，这里所讲的位段与 C 语言中的位段是一致的，它们与系统程序和单片机程序非常对口。

BFC（位段清零）指令把 32 位整数中任意一段连续的 2 进制位 s 清 0，语法格式为：

```
BFC.W      Rd,      #lsb,      #width
```

其中，lsb 为位段的末尾，width 则指定在 lsb 和它的左边（更高有效位），共有多少个位参与操作。

例如，

```
LDR      R0, =0x1234FFFF
BFC      R0, #4, #10
```

执行完后， $R0=0x1234C00F$

译注：位段不支持首尾拼接。例如， $BFC R0, #27, #9$ 将产生不可预料的结果

BFI（位段插入指令），把某个寄存器按 LSB 对齐的数值，拷贝到另一个寄存器的某个位段中，其格式为

```
BFI.W      Rd,      Rn,      #lsb,      #width
```

例如，

```
LDR      R0,      =0x12345678
LDR      R1,      =0xAABBCCDD
BFI.W      R1,      R0, #8, #16
```

则执行后， $R1=0xAA5678DD$ （总是从 Rn 的最低位提取，#lsb 只对 Rd 起作用——译注）

UBFX/SBFX 都是位段提取指令，语法格式为：

```
UBFX.W      Rd,      Rn,      #lsb,      #width
SBFX.W      Rd,      Rn,      #lsb,      #width
```

UBFX 从 Rn 中取出任一个位段，执行零扩展后放到 Rd 中（请比较与 BFI 的不同）。例如：

```
LDR      R0,      =0x5678ABCD
UBFX.W  R1,      R0, #12,#16
```

则 R0=0x0000678A

类似地，SBFX 也抽取任意的位段，但是以带符号的方式进行扩展。例如：

```
LDR      R0,      =0x5678ABCD
SBFX.W  R1,      R0, #8,#4
```

则 R0=0xFFFFFFFFFB

上述例子为了描述方便使用了 4 比特对齐的#lsb 和#width，但事实上并无此限制——译注

4.4.9 LDRD/STRD

CM3 在一定程度上支持 64 位整数。其中 LDRD/STRD 就是为 64 位整数的数据传送而设的，语法格式为：

```
LDRD.W    RL, RH, [Rn, #+/-offset] {!}; 可选预索引的 64 位整数加载
LDRD.W    RL, RH, [Rn],#+/-offset      ; 后索引的 64 位整数加载
STRD.W    RL, RH, [Rn, #+/-offset] {!}; 可选预索引的 64 位整数存储
STRD.W    RL, RH, [Rn],#+/-offset      ; 后索引的 64 位整数存储
```

例如，记 (0x1000)=0x1234_5678_ABCD_EF00：则

```
LDR      R2, =0x1000      ;
LDRD.W  R0, R1, [R2]
执行后， R0=0xABCD_EF00, R1=0x1234_5678
```

同理，我们也可以使用 STRD 来存储 64 位整数。在上面的例子执行完毕后，若执行如下代码：

```
STRD.W  R1, R0, [R2]
执行后， (0x1000)=0xABCD_EF00_1234_5678，从而实现了双字的字序反转操作。
```

4.4.10 TBB, TBH

高级语言都提供了“分类讨论”式控制结构，如 C 语言的 `switch`，Basic 语言的 `Select Case`。通常，给我们的印象是比较靠后的 `case` 执行起来效率比较低，因为要一个一个地查。有了 TBB/TBH 后，则改善了这类结构的执行效率（可以对比 51 中的 `MOVC`）

TBB（查表跳转字节范围的偏移量）指令和 TBH（查表跳转半字范围的偏移量）指令，分别用于从一个字节数组表中查找转移地址，和从半字数组表中查找转移地址。TBH 的转移范围已经足以应付任何臭长的 `switch` 结构。如果写出的 `switch` 连 TBH 都搞不定，只能说那人有严重自虐倾向。

因为 CM3 的指令至少是按半字对齐的，表中的数值都是在左移一位后才作为前向跳转的偏移量的。又因为 PC 的值为当前地址+4，故 TBB 的跳转范围可达 $255*2+4=514$ ；TBH 的跳转范围更可高达 $65535*2+4=128KB+2$ 。请注意：Both TBB 和 TBH 都只能作前向跳转，也就是说偏移量是一个无符号整数。

TBB 的语法规则为：

TBB.W [Rn, Rm] ; PC+= Rn[Rm]*2

在这里，Rn 指向跳转表的基址，Rm 则给出表中元素的下标。图 4.5 指示了这个操作

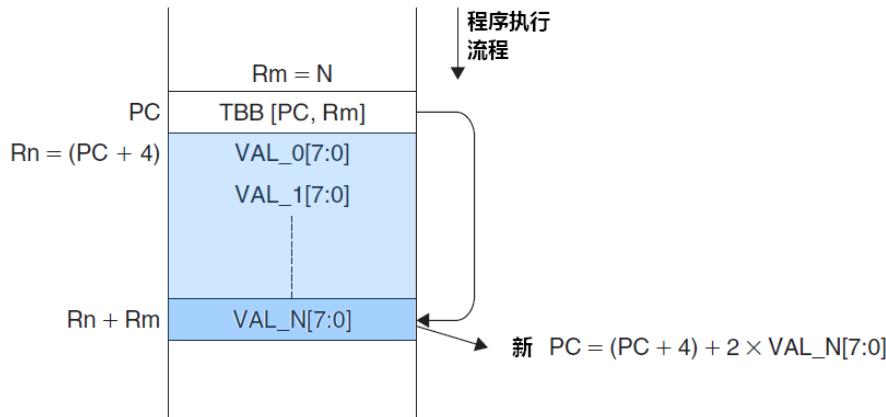


图 4.5 TBB 功能演示

如果 Rn 是 $R15$ ，则由于指令流水线的影响， Rn 的值将是 $PC+4$ 。通常很少有人会手工计算表中偏移量，因为很繁，而且程序修改后要重新计算，尤其是当跨源文件查表时（由连接器负责分配地址）。所以这种指令在汇编中很少用到，通常是 C 编译器专用的，它可以在每次编译时重建该表。不过，可以为各入口地址取个标号，而且此指令还有其它的使用方式。在系统程序的开发中，此指令可以提高程序的运行效率。为了提供一个节能高效的操作系统或者基础函数库，必须挖空心思地使用各种奇异的技巧，甚至在极端情况下，还要严重违反常规程序设计的基本原则。

另外还要注意的是，不同的汇编器可能会要求不同的语法规则。在 ARM 汇编器 (armasm.exe) 中，TBB 跳转表的创建方式如下所示：

```

TBB.W [pc, r0] ; 执行此指令时，PC 的值正好等于 branchtable
branchtable
    DCB ((dest0 - branchtable)/2) ; 注意：因为数值是 8 位的，故使用 DCB 指示字
    DCB ((dest1 - branchtable)/2)
    DCB ((dest2 - branchtable)/2)
    DCB ((dest3 - branchtable)/2)
dest0
    ... ; r0 = 0 时执行
dest1
    ... ; r0 = 1 时执行
dest2
    ... ; r0 = 2 时执行
dest3
    ... ; r0 = 3 时执行

```

TBH 的操作原理与 TBB 相同，只不过跳转表中的每个元素都是 16 位的。故而下标为 Rm 的元素要从 $Rn+2*Rm$ 处去找。如图 4.6 所演示：

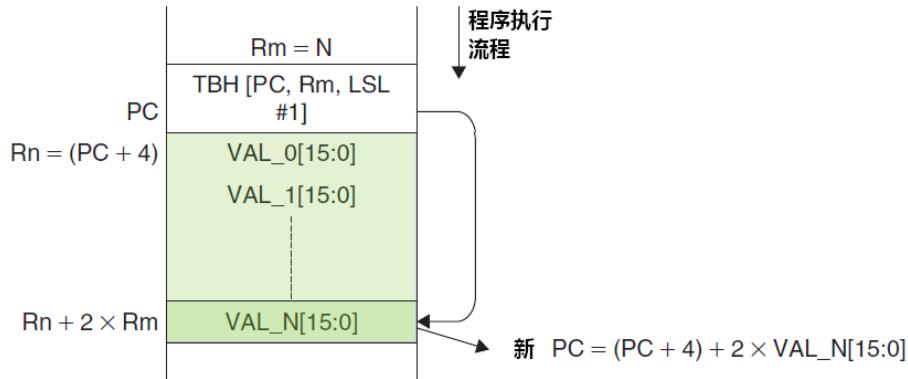


图 4.6 TBH 功能演示

TBH 跳转表的创建方式与 TBB 的类似，如下所示：

TBH.W [pc, r0, LSL #1] ; 执行此指令时，PC 的值正好等于 branchtable
branchtable

```

DCI ((dest0 - branchtable)/2) ; 注意：数值是 16 位的，故使用 DCI 指示字
DCI ((dest1 - branchtable)/2)
DCI ((dest2 - branchtable)/2)
DCI ((dest3 - branchtable)/2)

dest0
    ... ; r0 = 0 时执行
dest1
    ... ; r0 = 1 时执行
dest2
    ... ; r0 = 2 时执行
dest3
    ... ; r0 = 3 时执行

```


存储器系统

- 存储器系统的功能概览
- 存储器映射
- 存储器的各种访问属性
- 缺省的存储器访问许可
- 位带操作
- 非对齐数据传送
- 互斥访问
- 端模式

5.1 存储系统功能概览

CM3 的存储器系统与从传统 ARM 架构的相比，已经有过脱胎换骨般的改革了：

第一，它的存储器映射是预定义的，并且还规定好了哪个位置使用哪条总线。

第二，CM3 的存储器系统支持所谓的“位带”(bit-band) 操作。通过它，实现了对单一比特的原子操作。位带操作仅适用于一些特殊的存储器区域中，见本章论述。

第三，CM3 的存储器系统支持非对齐访问和互斥访问。这两个特性是直到了 v7M 时才出来的。
最后，CM3 的存储器系统支持 both 小端配置和大端配置。

5.2 存储器映射

CM3 只有一个单一固定的存储器映射。这一点极大地方便了软件在各种 CM3 单片机间的移植。举个简单的例子，各款 CM3 单片机的 NVIC 和 MPU 都在相同的位置布设寄存器，使得它们变得与具体器件无关。尽管如此，CM3 定出的条条框框是粗线条的，它依然允许芯片制造商灵活细腻地分配存储器空间，以制造出各具特色的单片机产品。

存储空间的一些位置用于调试组件等私有外设，这个地址段被称为“私有外设区”。私有外设区的组件包括：

- 闪存地址重载及断点单元(FPB)
- 数据观察点单元(DWT)
- 仪器化跟踪宏单元(ITM)
- 嵌入式跟踪宏单元(ETM)
- 跟踪端口接口单元(TPIU)
- ROM 表

在后续讨论调试特性的章节中，将详细讲述这些组件。

CM3 的地址空间是 4GB，程序可以在代码区，内部 SRAM 区以及外部 RAM 区中执行。但是因为指令总线与数据总线是分开的，最理想的是把程序放到代码区，从而使取指和数据访问各自使用自

己的总线，并行不悖。

让我们先看一看这 4GB 的粗线条划分：

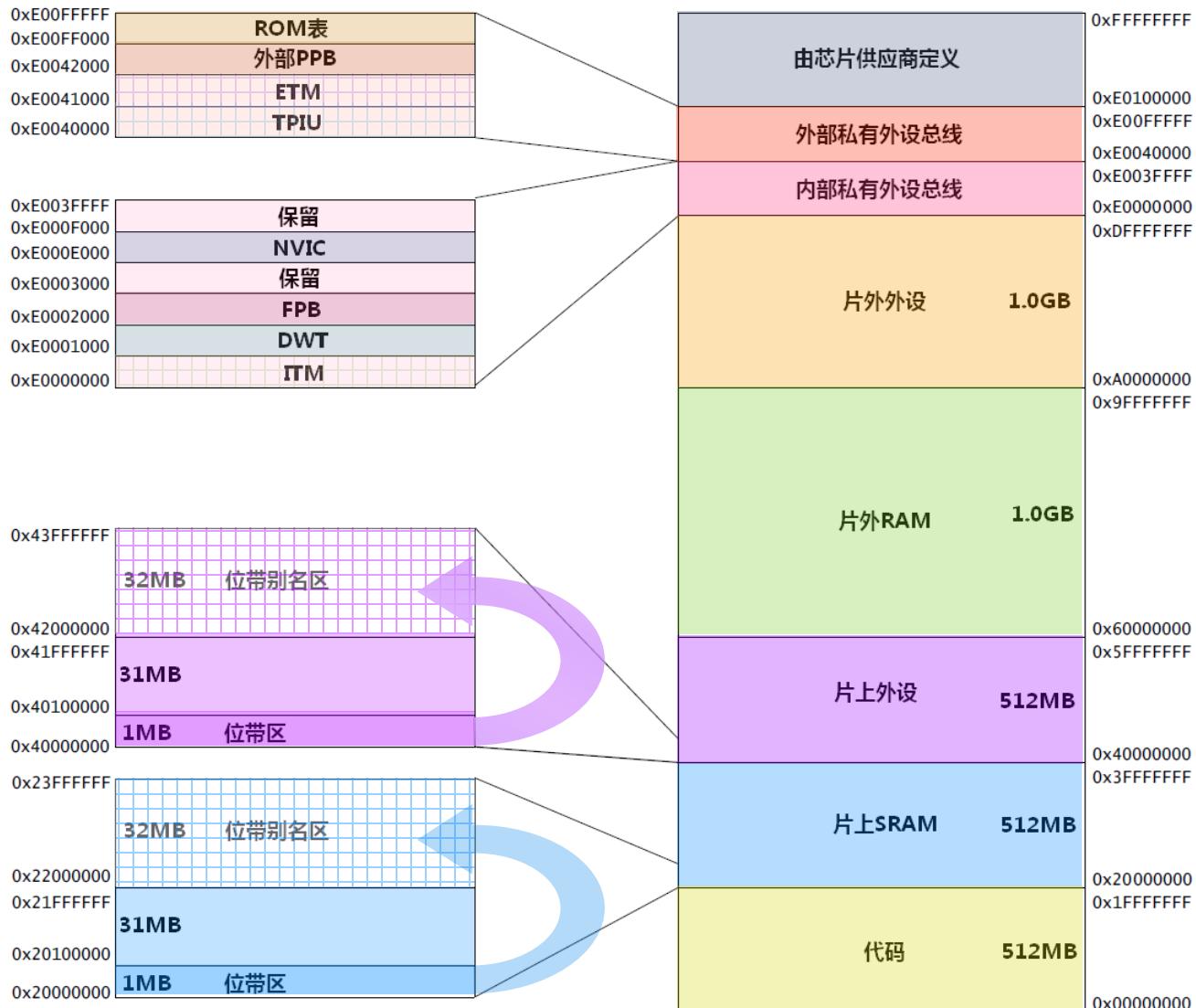


图 5.1 Cortex-M3 预定义的存储器映射

内部 SRAM 区的大小是 512MB，用于让芯片制造商连接片上的 SRAM，这个区通过系统总线来访问。在这个区的下部，有一个 1MB 的区间，被称为“位带区”。该位带区还有一个对应的、32MB 的“位带别名(alias)区”，容纳了 8M 个“位变量”(对比 8051 的只有 128 个位变量)。位带区对应的是最低的 1MB 地址范围，而位带别名区里面的每个字对应位带区的一个比特。位带操作只适用于数据访问，不适用于取指。通过位带的功能，可以把多个布尔型数据打包在单一的字中，却依然可以从位带别名区中，像访问普通内存一样地使用它们。位带别名区中的访问操作是原子的，消灭了传统的“读一改一写”三步曲。位带操作的细节待会还要讲到。

地址空间的另一个 512MB 范围由片上外设(的寄存器)使用。这个区中也有一条 32MB 的位带别名，以便于快捷地访问外设寄存器，用法与内部 SRAM 区中的位带相同。例如，可以方便地访问各种控制位和状态位。要注意的是，外设区内不允许执行指令。

还有两个 1GB 的范围，分别用于连接外部 RAM 和外部设备，它们之中没有位带。两者的区别在于外部 RAM 区允许执行指令，而外部设备区则不允许。

最后还剩下 **0.5GB** 的隐秘地带, CM3 内核的闺房就在这里面, 包括了系统级组件, 内部私有外设总线 **s**, 外部私有外设总线 **s**, 以及由提供者定义的系统外设。

私有外设总线有两条:

- AHB 私有外设总线, 只用于 CM3 内部的 AHB 外设, 它们是: NVIC, FPB, DWT 和 ITM。
- APB 私有外设总线, 既用于 CM3 内部的 APB 设备, 也用于外部设备 (这里的“外部”是对内核而言)。CM3 允许器件制造商再添加一些片上 APB 外设到 APB 私有总线上, 它们通过 APB 接口来访问。

NVIC 所处的区域叫做“系统控制空间 (SCS)”, 在 SCS 里的除了 NVIC 外, 还有 SysTick、MPU 以及代码调试控制所用的寄存器, 如图 5.2 所示:

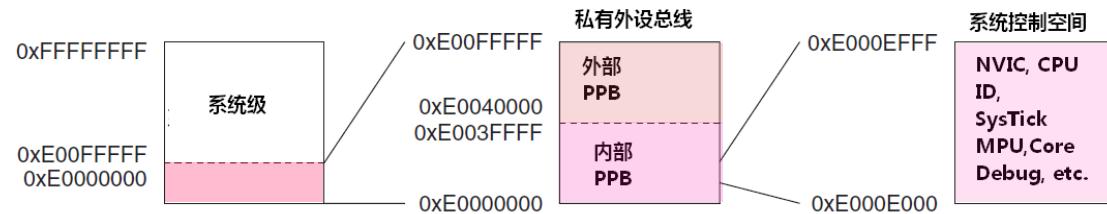


图 5.2 系统控制空间(SCS)

最后, 未用的提供商指定区也通过系统总线来访问, 但是不允许在其中执行指令。

CM3 中的 MPU 是选配的, 由芯片制造商决定是否配上去。

上述的存储器映射只是个粗线条的模板, 半导体厂家会提供更展开的图示, 来表明芯片中片上外设的具体分布, RAM 与 ROM 的容量和位置信息。

5.3 存储器的各种访问属性

CM3 在定义了存储器映射之外, 还为存储器的访问规定了 4 种属性, 分别是:

- 可否缓冲(Bufferable)
- 可否缓存(Cacheable)
- 可否执行(Executable)
- 可否共享(Sharable)

如果配了 MPU, 则可以通过它配置不同的存储区, 并且覆盖缺省的访问属性。CM3 片内没有配备缓存, 也没有缓存控制器, 但是允许在外部添加缓存。通常, 如果提供了外部内存, 芯片制造商还要附加一个内存控制器, 它可以根据可否缓存的设置, 来管理对片内和片外 RAM 的访问操作。地址空间可以通过另一种方式分为 8 个 512MB 等份:

1. 代码区 (0x0000_0000 - 0x1FFF_FFFF)。该区是可以执行指令的, 缓存属性为 WT (“写通”, Write Through), 即不可以缓存。此区亦允许布设数据存储器。在此区上的数据操作是通过数据总线接口的 (估计读数据使用 D-Code, 写数据使用 System), 且在此区上的写操作是缓冲的。
2. SRAM 区 (0x2000_0000 - 0x3FFF_FFFF)。此区用于片内 SRAM, 写操作是缓冲的, 并且可以选择 WB-WA(Write Back, Write Allocated)缓存属性。此区亦可以执行指令, 以允许把代码拷贝到内存中执行——常用于固件升级等维护工作。
3. 片上外设区 (0x4000_0000 - 0x5FFF_FFFF)。该区用于片上外设, 因此是不可缓存的, 也不可以在此区执行指令 (这也称为 eXecute Never, XN。ARM 的参考手册大量使用此术语)。
4. 外部 RAM 区的前半段 (0x6000_0000 - 0x7FFF_FFFF)。该区可用于布设片上 RAM 或片外 RAM, 可缓存 (缓存属性为 WB-WA), 并且可以执行指令。

5. 外部 RAM 区的后半段 (`0x8000_0000 - 0x9FFF_FFFF`)。除了不可缓存(WT)外, 同前半段。
6. 外部外设区的前半段(`0xA000_0000 - 0xBFFF_FFFF`)。用于片外外设的寄存器, 也用于多核系统中的共享内存 (需要严格按顺序操作, 即不可缓冲)。该区也是个不可执行区。
7. 外部外设区的后半段(`0xC000_0000 - 0xDFFF_FFFF`)。目前与前半段的功能完全一致。
8. 系统区(`0xE000_0000 - 0xFFFF_FFFF`)。此区是私有外设和供应商指定功能区。此区不可执行代码。系统区涉及到很多关键部位, 因此访问都是严格序列化的 (不可缓存, 不可缓冲)。而供应商指定功能区则是可以缓存和缓冲的。

需要注意的是, 在 CM3 的第一版中, 代码区的存储器属性是被硬件连接成可缓存可缓冲的, 无法通过 MPU 来更改。

译者添加

写通, 写回, 与写时申请

- 写回(Write Back): 写入的数据先逗留在缓存中, 待到必要时再落实到最终目的地, 这也是 cache 的最基本职能, 用于改善数据传送的效率, 减少对访问主存储器的访问操作。
- 写通(Write Through): 写操作“穿透”中途的缓存, 直接落入最终的存储器目的地址中。可见, 写通操作架空了 cache, 但它使写操作的结果立即生效。这常用于和片上外设或其它处理器共享的内存中, 如显卡的显存, 片上外设寄存器, 以及双核系统中的共享内存。写通操作和 C 中的“volatile”可以配合使用——带 volatile 属性的变量往往放到写通型地址区间中。
- 写时申请(Write Allocate): 像也不太清楚~

5.4 存储器的缺省访问许可

CM3 有一个缺省的存储访问许可, 它能防止使用户代码访问系统控制存储空间, 保护 NVIC、MPU 等关键部件。缺省访问许可在下列条件时生效:

- 没有配备 MPU
- 配备了 MPU, 但是 MPU 被禁用

如果启用了 MPU, 则 MPU 可以在地址空间中划出若干个 regions, 并为不同的 region 规定不同的访问许可权限。

缺省的存储器访问许可权限如表 5.1 所示

表 5.1 存储器的缺省访问许可

存储器区域	地址范围	用户级许可权限
代码区	<code>0000_0000 - 1FFF_FFFF</code>	无限制
片内 SRAM	<code>2000_0000 - 3FFF_FFFF</code>	无限制
片上外设	<code>4000_0000 - 5FFF_FFFF</code>	无限制
外部 RAM	<code>6000_0000 - 9FFF_FFFF</code>	无限制
外部外设	<code>A000_0000 - DFFF_FFFF</code>	无限制
ITM	<code>E000_0000 - E000_0FFF</code>	可以读。对于写操作, 除了用户级下允许时的 stimulus 端口外, 全部忽略
DWT	<code>E000_1000 - E000_1FFF</code>	阻止访问, 访问会引发一个总线 fault
FPB	<code>E000_2000 - E000_3FFF</code>	阻止访问, 访问会引发一个总线 fault

NVIC	E000_E000 - E000_EFFF	阻止访问, 访问会引发一个总线 fault 。但有个例外: 软件触发中断寄存器可以被编程为允许用户级访问。
内部 PPB	E000_F000 - E003_FFFF	阻止访问, 访问会引发一个总线 fault
TPIU	E004_0000 - E004_0FFF	阻止访问, 访问会引发一个总线 fault
ETM	E004_1000 - E004_1FFF	阻止访问, 访问会引发一个总线 fault
外部 PPB	E004_2000 - E004_2FFF	阻止访问, 访问会引发一个总线 fault
ROM 表	E00F_F000 - E00F_FFFF	阻止访问, 访问会引发一个总线 fault
供应商指定	E010_0000 - FFFF_FFFF	无限制

当一个用户级访问被阻止时, 会立即产生一个总线 **fault**。

5.5 位带操作

支持了位带操作后, 可以使用普通的加载/存储指令来对单一的比特进行读写。在 **CM3** 中, 有两个区中实现了位带。其中一个是 **SRAM** 区的最低 **1MB** 范围, 第二个则是片内外设区的最低 **1MB** 范围。这两个位带中的地址除了可以像普通的 **RAM** 一样使用外, 它们还都有自己的“位带别名区”, 位带别名区把每个比特膨胀成一个 **32** 位的字。当你通过位带别名区访问这些字时, 就可以达到访问原始比特的目的。

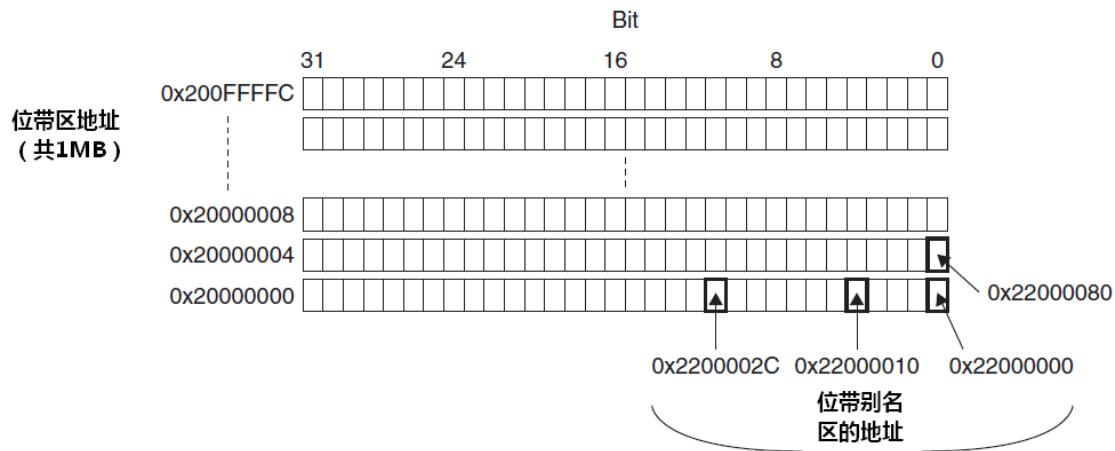


图 5.3A 位带区与位带别名区的膨胀关系图 A:

译者添加 下图从另一个侧面演示比特的膨胀对应关系

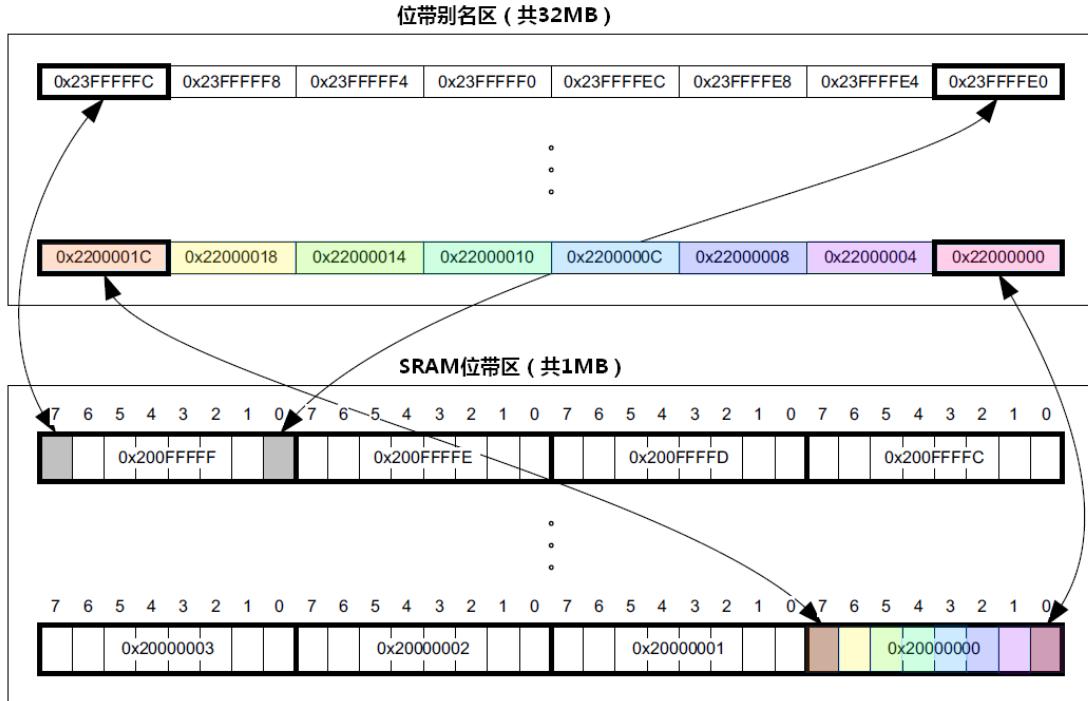


图 5.3B 位带区与位带别名区的膨胀对应关系图 B

举例：欲设置地址 0x2000_0000 中的比特 2，则使用位带操作的设置过程如下图所示：

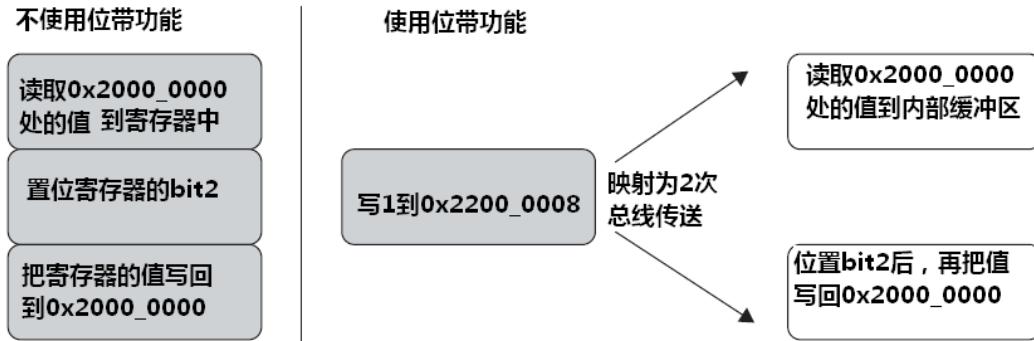


图 5.4 写数据到位带别名区

对应的汇编代码如图 5.5 所示

Without Bit-Band	With Bit-Band
LDR R0, =0x20000000 ; Setup address	LDR R0, =0x22000008 ; Setup address
LDR R1, [R0] ; Read	MOV R1, #1 ; Setup data
ORR.W R1, #0x4 ; Modify bit	STR R1, [R0] ; Write
STR R1, [R0] ; Write back result	

图 5.5 位带操作与普通操作的对比，在汇编程序的角度上

位带读操作相对简单些：



图 5.6 从位带别名区中读取比特

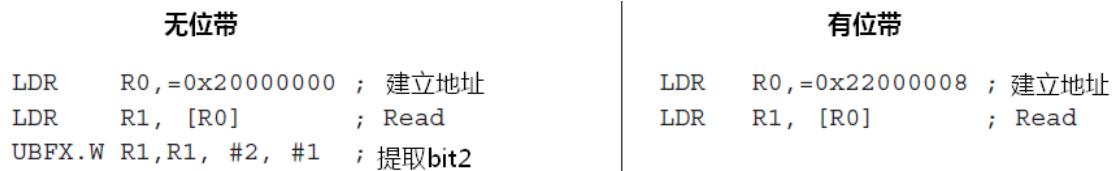


图 5.7 读取比特时传统方法与位带方法的比较

位带操作的概念其实 30 年前就有了，那还是 8051 单片机开创的先河。如今，CM3 将此能力进化，这里的位带操作是 8051 位寻址区的威力极度加强版。

CM3 使用如下术语来表示位带存储的相关地址

- 位带区：支持位带操作的地址区
- 位带别名：对别名地址的访问最终会变换为对位带区的访问（注意：这中途有一个地址映射过程）

在位带区中，每个比特都映射到别名地址区的一个字——这是个只有 LSB 才有效的字。当一个别名地址被访问时，会先把该地址变换为位带地址。对于读操作，读取位带地址中的一个字，再把需要的位右移到 LSB，并把 LSB 返回。对于写操作，把需要写的位左移至对应的位序号处，然后执行一个原子的“读一改一写”过程。

支持位带操作的两个内存区的范围是：

0x2000_0000-0x200F_FFFF (SRAM 区中的最低 1MB)

0x4000_0000-0x400F_FFFF (片上外设区中的最低 1MB)

对于 SRAM 位带区的某个比特，记它所在字节地址为 A，位序号为 n (0 <= n <= 7)，则该比特在别名区的地址为：

```
AliasAddr = 0x22000000 + ((A - 0x20000000) * 8 + n) * 4 = 0x22000000 + (A - 0x20000000) * 32 + n * 4
```

对于片上外设位带区的某个比特，记它所在字节的地址为 A，位序号为 n (0 <= n <= 7)，则该比特在别名区的地址为：

```
AliasAddr = 0x42000000 + ((A - 0x40000000) * 8 + n) * 4 = 0x42000000 + (A - 0x40000000) * 32 + n * 4
```

上式中，“*4”表示一个字为 4 个字节，“*8”表示一个字节中有 8 个比特。

对于 SRAM 内存区，位带别名的重映射如表 5.2 所示：

表 5.2 SRAM 区中的位带地址映射

位带区	等效的别名地址
0x20000000.0	0x22000000.0
0x20000000.1	0x22000004.0

0x20000000.2	0x22000008.0
...	
0x20000000.31	0x2200007C.0
0x20000004.0	0x22000080.0
0x20000004.1	0x22000084.0
0x20000004.2	0x22000088.0
...	
0x200FFFFC.31	0x23FFFFFFC.0

对于片上外设，映射关系如下表所示：

表 5.3 片上外设区中的位带地址映射

位带区	等效的别名地址
0x40000000.0	0x42000000.0
0x40000000.1	0x42000004.0
0x40000000.2	0x42000008.0
...	
0x40000000.31	0x4200007C.0
0x40000004.0	0x42000080.0
0x40000004.1	0x42000084.0
0x40000004.2	0x42000088.0
...	
0x400FFFFC.31	0x43FFFFFFC.0

这里再不嫌啰嗦地举一个例子：

1. 在地址 0x20000000 处写入 0x3355AAC
2. 读取地址 0x22000008。本次读访问将读取 0x20000000，并提取比特 2，值为 1。
3. 往地址 0x22000008 处写 0。本次操作将被映射成对地址 0x20000000 的“读一改一写”操作（原子的），把比特 2 清 0。
4. 现在再读取 0x20000000，将返回 0x3355AAC8 (bit[2]已清零)。

位带别名区的字只有 LSB 有意义。另外，在访问位带别名区时，不管使用哪一种长度的数据传送指令（字/半字/字节），都把地址对齐到字的边界上，否则会产生不可预料的结果。

5.5.1 位带操作的优越性

位带操作有什么优越性呢？最容易想到的就是通过 GPIO 的管脚来单独控制每盏 LED 的点亮与熄灭。另一方面，也对操作串行接口器件提供了很大的方便（典型如 74HC165, CD4094）。总之位带操作对于硬件 I/O 密集型的底层程序最有用处了。对于大范围使用位标志的系统程序来说，位带机制也是一大福音。

Cortex-M 中还有一个称为“bit-bang”的概念，它通常是通过“bit-band”实现的，但是它俩在学术上是两个不同的概念（不过本书中除了这里之外，就再也没有提到过 bit-bang——译注）。

位带操作还能用来化简跳转的判断。当跳转依据是某个位时，以前必须这样做：

- ◆ 读取整个寄存器
- ◆ 掩蔽不需要的位
- ◆ 比较并跳转

现在只需：

- ◆ 从位带别名区读取状态位
- ◆ 比较并跳转

使代码更简洁，这只是位带操作优越性的初等体现，位带操作还有一个重要的好处是在多任务中，用于实现共享资源在任务间的“互锁”访问。多任务的共享资源必须满足一次只有一个任务访问它——亦即所谓的“原子操作”。以前的读一改一写需要 3 条指令，导致这中间留有两个能被中断的空当。于是可能会出现如下图所示的紊乱危象：

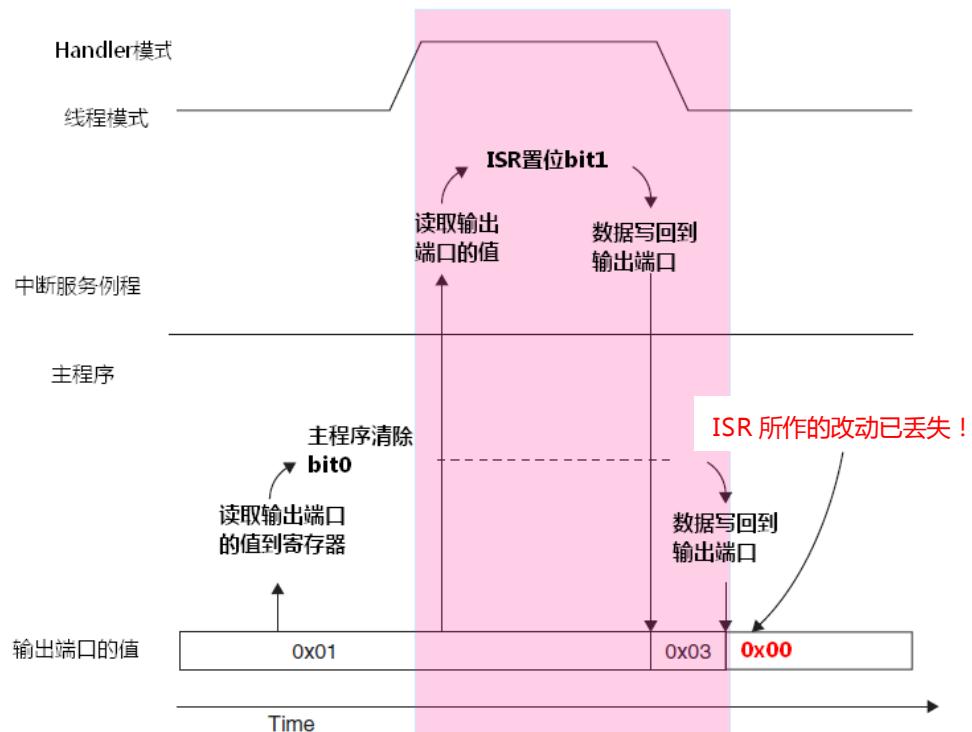


图 5.8 共享资源在紊乱危象下丢失数据演示

同样的紊乱危象可以出现在多任务的执行环境中。其实，图 5.8 所演示的情况可以看作是多任务的一个特例：主程序是一个任务，ISR 是另一个任务，这两个任务并发执行。

通过使用 CM3 的位带操作，就可以消灭上例中的紊乱危象。CM3 把这个“读一改一写”做成一个硬件级别支持的原子操作，不能被中断，如图 5.9 所演示

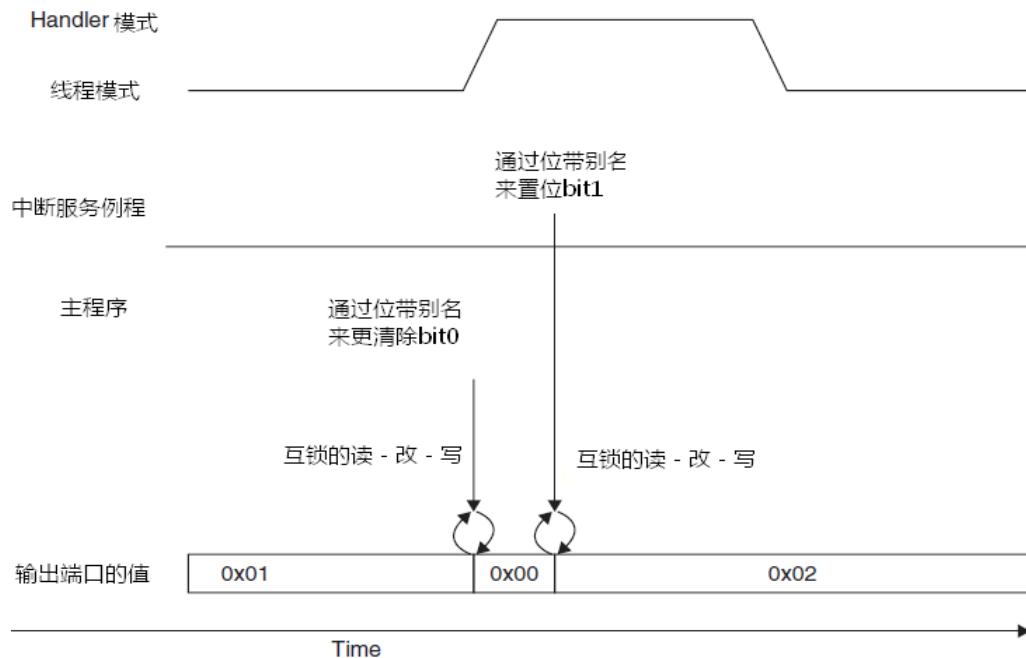


图 5.9 通过位带操作实现互锁访问，从而避免紊乱危象的演示

同样道理，多任务环境中的紊乱危象亦可以通过互锁访问来避免。

5.5.2 其它数据长度上的位带操作

位带操作并不只限于以字为单位的传送。亦可以按半字和字节为单位传送。例如，可以使用 **LDRB/STRB** 来以字节为长度单位去访问位带别名区，同理可用于 **LDRH/STRH**。但是不管用哪一个对子，都必须保证目标地址对齐到字的边界上。

5.5.3 在 C 语言中使用位带操作

不幸的是，在 C 编译器中并没有直接支持位带操作。比如，C 编译器并不知道同一块内存能够使用不同的地址来访问，也不知道对位带别名区的访问只对 LSB 有效。欲在 C 中使用位带操作，最简单的做法就是 **#define** 一个位带别名区的地址。例如：

```
#define DEVICE_REG0 ((volatile unsigned long *) (0x40000000))
#define DEVICE_REG0_BIT0 ((volatile unsigned long *) (0x42000000))
#define DEVICE_REG0_BIT1 ((volatile unsigned long *) (0x42000004))

...
*DEVICE_REG0 = 0xAB; // 使用正常地址访问寄存器
...
*DEVICE_REG0 = *DEVICE_REG0 | 0x2; // 使用传统方法设置 bit1
...
*DEVICE_REG0_BIT1 = 0x1; // 通过位带别名地址设置 bit1
```

为简化位带操作，也可以定义一些宏。比如，我们可以建立一个把“位带地址+位序号”转换成别名地址的宏，再建立一个把别名地址转换成指针类型的宏：

```
//把“位带地址+位序号”转换成别名地址的宏
#define BITBAND(addr, bitnum) ((addr & 0xF0000000)+0x2000000+((addr &
0xFFFF)<<5)+(bitnum<<2))

//把该地址转换成一个指针
#define MEM_ADDR(addr) *((volatile unsigned long *) (addr))
```

在此基础上，我们就可以如下改写代码：

```
MEM_ADDR(DEVICE_REG0) = 0xAB;           //使用正常地址访问寄存器
MEM_ADDR(DEVICE_REG0)= MEM_ADDR(DEVICE_REG0) | 0x2; //传统做法
MEM_ADDR(BITBAND(DEVICE_REG0,1)) = 0x1;           //使用位带别名地址
```

请注意：当使用位带功能时，要访问的变量必须用 **volatile** 来定义。因为 C 编译器并不知道同一个比特可以有两个地址。所以就要通过 **volatile**，使得编译器每次都如实地把新数值写入存储器，而不再会出于优化的考虑，在中途使用寄存器来操作数据的复本，直到最后才把复本写回——这会导致按不同的方式访问同一个位会得到不一致的结果（可能被优化到不同的寄存器来保存中间结果——译注）

译者添加

在 **GCC** 和 **RealView MDK** (即 **Keil**) 开发工具中，允许定义变量时手工指定其地址。如：

```
volatile unsigned long bbVarAry[7] __attribute__(( at(0x20003014) ));  
volatile unsigned long* const pbbaVar= (void*)(0x22000000+0x3014*8*4);
```

这样，就在 **0x20003014** 处分配了 7 个字，共得到了 $32*7=224$ 个比特。

在 **long*** 后面的 “**const**” 通知编译器：该指针不能再被修改而指向其它地址。

注意：**at()** 中的地址必须对齐到 4 字节边界。

再使用这些比特时，可以通过如下的形式：

```
pbbaVar[136]=1; //置位第 136 号比特
```

不过这有个局限：编译器无法检查是否下标越界。那为什么不定义成 “**bbVarAry[224]**” 的数组呢？这也是一个编译器的局限：它不知道这个数组其实就是一个 **bbVarAry[7]**，从而在计算程序对内存的占用量上，会平白无故地多计入 $224*4$ 个字节。对于指针形式的定义，可以使用宏定义，为每个需要使用的比特取一个字面值的名字，在下标中只使用字面值名字，不再写真实的数字，就可以极大程度地避免数组越界。

请注意：在定义这“两个”变量时，前面加上了“**volatile**”。如果不使用 **bbVarAry** 来访问这些比特，而仅仅使用位带别名的形式访问时，这两个 **volatile** 均不再需要。

5.6 非对齐数据传送

CM3 支持在单一的访问中使用非（地址）对齐的传送，数据存储器的访问无需对齐。在以前，ARM 处理器只允许对齐的数据传送。这种对齐是说：以字为单位的传送，其地址的最低两位必须是 0；以半字为单位的传送，其地址的 **LSB** 必须是 0；以字节为单位的传送则无所谓对不对齐。如果使用 **0x1001, 0x1002** 或 **0x1003** 这样的地址做字传送，在以前的 ARM 处理器中则会触发一个数据流产（**Data abort**）异常——与 CM3 中总线 **fault** 异常的作用相同。

那么，非对齐访问看起来是什么样子呢？图 5.12-5.16 给出了 5 个例子。对于字的传送来说，

任何一个不能被 4 整除的地址都是非对齐的。而对于半字，任何不能被 2 整除的地址（也就是奇数地址）都是非对齐的：

	Byte 3	Byte 2	Byte 1	Byte 0
Address N + 4				[31:24]
Address N	[23:16]	[15:8]	[7:0]	

图 5.12 非对齐传送示例 1

	Byte 3	Byte 2	Byte 1	Byte 0
Address N + 4			[31:24]	[23:16]
Address N	[15:8]	[7:0]		

图 5.13 非对齐传送示例 2

	Byte 3	Byte 2	Byte 1	Byte 0
Address N + 4		[31:24]	[23:16]	[15:8]
Address N	[7:0]			

图 5.14 非对齐传送示例 3

	Byte 3	Byte 2	Byte 1	Byte 0
Address N + 4				
Address N		[15:8]	[7:0]	

图 5.15 非对齐传送示例 4

	Byte 3	Byte 2	Byte 1	Byte 0
Address N + 4				[15:8]
Address N	[7:0]			

图 5.16 非对齐传送示例 5

在 CM3 中，非对齐的数据传送只发生在常规的数据传送指令中，如 LDR/LDRH/LDRSH。其它指令则不支持，包括：

- 多个数据的加载/存储 (LDM/STM)
- 堆栈操作 PUSH/POP
- 互斥访问 (LDREX/STREX)。如果非对齐会导致一个用法 **fault**
- 位带操作。因为只有 LSB 有效，非对齐的访问会导致不可预料的结果。

事实上，在内部是把非对齐的访问转换成若干个对齐的访问的，这种转换动作由处理器总线单

元来完成。这个转换过程对程序员是透明的，因此写程序时不必操心。但是，因为它通过若干个对齐的访问来实现一个非对齐的访问，会需要更多的总线周期。事实上，节省内存有很多方法，但没有一个是通过压缩数据的地址，不惜破坏对齐性的这种旁门左道。因此，应养成好习惯，总是保证地址对齐，这也是让程序可以移植到其它 ARM 芯片上的必要条件。

为此，可以编程 NVIC，使之监督地址对齐。当发现非对齐访问时触发一个 **fault**。具体的办法是设置“配置控制寄存器”中的 **UNALIGN_TRP** 位。这样，在整个调试期间就可以保证非对齐访问能当场被发现。

5.7 互斥访问

细心的读者可能会发现，CM3 中没有类似“SWP”的指令。在传统的 ARM 处理器中，SWP 指令是实现互斥体所必需的。到了 CM3，由所谓的互斥访问取代了 SWP 指令，以实现更加老练的共享资源访问保护机制。

互斥体在多任务环境中使用，也在中断服务例程和主程序之间使用，用于给任务申请共享资源（如一块共享内存）。在某个（排他型）共享资源被一个任务拥有后，直到这个任务释放它之前，其它任务是不得再访问它的。为建立一个互斥体，需要定义一个标志变量，用来指示其对应的共享资源是否已经被某任务拥有。当另一个任务欲取得此共享资源时，它要先检查这个互斥体，以获知共享资源是否无人使用。在传统的 ARM 处理器中，这种检查操作是通过 SWP 指令来实现的。SWP 保证互斥体检查是原子操作的，从而避免了一个共享资源同时被两个任务占有（这是紊乱危象的一种常见表现形式）。

在新版的 ARM 处理器中，读/写访问往往使用不同的总线，导致 SWP 无法再保证操作的原子性，因为只有在同一条总线上的读/写能实现一个互锁的传送。因此，互锁传送必须用另外的机制实现，这就引入了“互斥访问”。互斥访问的理念同 SWP 非常相似，不同点在于：在互斥访问操作下，允许互斥体所在的地址被其它总线 **master** 访问，也允许被其它运行在本机上的任务访问，但是 CM3 能够“驳回”有可能导致竞态条件的互斥写操作。

互斥访问分为加载和存储，相应的指令对子为 LDREX/STREX，LDREXH/STREXH，LDREXB/STREXB，分别对应于字/半字/字节。为了介绍方便，以 LDREX/STREX 为例讲述它们的使用方式。

LDREX/STREX 的语法格式为：

LDREX	Rxf ,	[Rn ,	#offset]	
STREX	Rd ,	Rxf ,	[Rn ,	#offset]

（本节的以下内容是译者改编的）

LDREX 的语法同 LDR 相同，这里不再赘述。而 STREX 则不同。STREX 指令的执行是可以被“驳回”的。当处理器同意执行 STREX 时，Rxf 的值被存储到(Rn+offset)处，并且把 Rd 的值更新为 0。但若处理器驳回了 STREX 的执行，则不会发生存储动作，并且把 Rd 的值更新为 1。

其实，奥妙就在于这个“驳回”的规则上。规则可宽可严，最严格的规则是：

当遇到 STREX 指令时，仅当在它之前执行过 LDREX 指令，且在最近的一条 LDREX 指令执行后，没有执行过其它的 STR/STREX 指令，才允许执行本条 STREX 指令——也就是说只有在 LDREX 执行后，从时间上与之距离最近的一条 STREX 才能成功执行。

其它情况下，驳回此 STREX。包括：

- ◆ 中途有其它的 STR 指令执行
- ◆ 中途有其它的 STREX 指令执行。

这种最严格的规则也是最容易实现的规则。在 CM3 的技术参考手册中，推荐实现者标记出一段

有限的地址，只在这段地址中适用互斥访问的规则，而不要对所有 4GB 都限制住。这段地址通常是从 LDREX 指令族给出的地址开始，长度在 16 字节至 4K 字节范围内。但芯片制造商可能更倾向严格的规则。

在使用互斥访问时，LDREX/STREX 必须成对使用。

为什么这种有条件的驳回可以避免紊乱危象呢？让我们举个简单的例子来演示。这个例子由主程序和一个中断服务例程组成。主程序尝试对(R0)自增两次，中断服务例程则把(R0).5 置位。计(R0)的初始值为 0。

MainProgram

;第一次互斥自增

TryInc1st

```
LDREX r2, [R0]
ADD r2, #1
;执行到这里时，处理器接收到外中断 3 请求，于是转到其中断服务例程 ISREx3 中
STREX R1, R2, [R0] ; STREX 被驳回，R1=1, (R0)=0x20
```

TryInc2nd

;第二次互斥自增

```
LDREX r2, [R0]
ADD r2, #1
STREX R1, R2, [R0] ; STREX 得到执行，R1=0, (R0)=0x21
```

...

ISREx3

```
;处理器已经自动把 R0-R3, R12, LR, PC, PSR 压入栈
LDR R2, [R0]
ORR R2, #0x20
STR R2, [R0] ;在 ISREx3 中设置了(r0)的 Bit2
BX LR ;返回时，处理器会自动把 R0-R3, R12, LR, PC, PSR 弹出堆栈
```

上例中，主程序在即将执行第一条 STREX 时，产生了外部中断#3。处理器打断主程序的执行，进入其服务例程 ISREx3，它对(R0)执行了一个写操作(STR)，因此在 ISREx3 返回后，STREX 不再是 LDREX 执行后的第一条存储指令，故而被驳回。从而 ISREx3 对(R0)的改动就不会遭到破坏。随后主程序再次尝试自增运算，这一次在 STREX 执行前没有其它任何形式的存储指令，所有 STREX 成功执行。

如果主程序使用普通的 STR 会怎么样呢？对于第一次自增，主程序的 R2=1，于是执行后(R0)=1，结果，中断服务程序对(R0)的改动在此丢失！

上例是为演示方便才写了第 2 次自增尝试。实际情况是用循环实现的：

TryInc

```
LDREX r2, [R0]
ADD r2, #1
STREX R1, R2, [R0]
CMP R1, #1 ;检查 STREX 是否被驳回
BEQ TryInc ;如果发现 STREX 被驳回，则重试。
```

LDREX/STREX 的工作原理其实很简单。仍然以上一段程序为例：当执行了 LDREX 后，处理器会在内部标记出一段地址。原则上，这段地址从 R0 开始，范围由芯片制造商定义。技术手册推荐的范围是在 4 字节至 4KB 之间，但是很多粗线条的实现会标记整个 4GB 的地址。在标记以后，对于

第一个执行到的 **STR/STREX** 指令，只要其存储的地址落在标记范围内，就会清除此标记（对于整个 4GB 地址都被标记的情况，则任何存储指令都会清除此标记）。如果先后执行了两次 **LDREX**，则以后一个 **LDREX** 标记的地址为准。

执行 **STREX** 时，会先检查有没有做出过标记，如果有，还要检查存储地址是否落在标记范围内。只有通过了这两个关卡，**STREX** 才会执行。否则，就驳回 **STREX**。

当使用互斥访问时，在 **CM3** 总线接口上的内部写缓冲会被旁路，即使是 **MPU** 规定此区是可以缓冲的也不行。这保证了互斥体的更新总能在第一时间内完成，从而保证数据在各个总线主机(**master**)之间是一致的。So 系统的设计师如果设计多核系统，则必须保证各核之间看到的数据也是一致的。

译者添加的选读材料——互斥访问的深入研究

互斥访问可以递归使用，且最后一次递归的 **LDREX/STREX** 对子最先完成。如下例所示：

```
LDREXTestRecursive
    ldr    r3,      =N      ; 递归次数 N，是一个预定义的常数
LoopWrapper
    push   {r0-r2, lr}
    ldr    r0,      =0x20003000
    sub    r3,      #1
TryInc
    ldrex  r1,      [r0]
    add    r1,      #1
    ldr    lr,      =DoSTREXRcsv
    cmp    r3,      #0
    bne    LoopWrapper
DoSTREXRcsv
    strex  r2,      r1,      [r0]
    cmp    r2,      #1
    beq    TryInc
    pop    {r0-r2, pc}
```

若执行前($0x20003000$)=0，则执行后($0x20003000$)=N，且函数被递归调用 N 次。这段代码的工作流程难以用文字说清，一定要用模拟器跑过才容易理解。

本例只是为了抛砖引玉。在实际的程序中，极少会这样钻牛角尖地直接递归。但是在多任务环境下，底层的函数库往往回“重入”，这也和递归的情形很相似。另外，当读者在本书后面看到“自旋锁”的解释时，说不定也会回想起这里呢！

5.8 端模式

CM3 支持 **both** 小端模式和大端模式。但是，单片机其它部分的设计，包括总线的连接，内存控制器以及外设的性质等，也共同决定可以支持的内存类型。所以在设计软件之前，一定要先在单片机的数据手册上查清楚可以使用的端。在绝大多数情况下，基于 **CM3** 的单片机都使用小端模式——为了避免不必要的麻烦，在这里推荐读者清一色地使用小端模式。

CM3 中对大端模式的定义还与 ARM7 的不同（小端的定义都是相同的）。在 ARM7 中，大端的方式被称为“字不变大端”，而在 CM3 中，使用的是“字节不变大端”。如表 5.4 所示。

表 5.4 CM3 的字节不变大端：存储器视图

地址, 长度	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x1000, 字	D[7:0]	D[15:8]	D[23:16]	D[31:24]
0x1000, 半字	D[7:0]	D[15:8]	-	-
0x1002, 半字	D[7:0]	D[15:8]		
0x1000, 字节	D[7:0]			
0x1001, 字节		D[7:0]		
0x1002, 字节			D[7:0]	
0x1003, 字节				D[7:0]

表 5.5 CM3 的字节不变大端：在 AHB 上的数据

地址, 长度	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x1000, 字	D[7:0]	D[15:8]	D[23:16]	D[31:24]
0x1000, 半字			D[7:0]	D[15:8]
0x1002, 半字	D[7:0]	D[15:8]		
0x1000, 字节				D[7:0]
0x1001, 字节			D[7:0]	
0x1002, 字节		D[7:0]		
0x1003, 字节	D[7:0]			

请注意：在 AHB 总线上的 BE-8 模式下，数据字节 lane 的传送格式是与小端模式一致的。

这是不同于 ARM7TDMI 的行为，它在大端模式下会有另一种总线车道（lane）安排，如表 5.6 所示。

表 5.6 ARM7 的字不变大端：在 AHB 上的数据

地址, 长度	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x1000, 字	D[7:0]	D[15:8]	D[23:16]	D[31:24]
0x1000, 半字	D[7:0]	D[15:8]	-	-
0x1002, 半字	D[7:0]	D[15:8]		
0x1000, 字节	D[7:0]			
0x1001, 字节		D[7:0]		
0x1002, 字节			D[7:0]	
0x1003, 字节				D[7:0]

在 CM3 中，是在复位时确定使用哪种端模式的，且运行时不得更改。指令预取永远使用小端模式，在配置控制存储空间的访问也永远使用小端模式（包括 NVIC, FPB 等）。另外，外部私有总线地址区 0xE0000000 至 0xE0FFFFF 也永远使用小端模式。

当你的 So 设计不支持大端模式，却有一些外设包含了大端模式时，可以轻易地使用 REV/REVH 指令来完成端模式的转换。

实现 Cortex-M3 的全景概貌

- 流水线
- 详细的框图
- Cortex-M3 的总线接口
- Cortex-M3 的其它接口
- 外部私有总线
- 典型的连接方式
- 复位信号源

[译注]: 本章相对篇幅较小, 但读懂本章需要一些处理器体系结构基础知识, 还需要一些 ARM 处理器特有的基础知识, 且只有设计处理器的专业人员才必须精通。如果这方面比较薄弱, 只需知道有哪些组件, 能一句话讲出它们的功用, 支持的调试方式, 以及知道有哪几条总线即可。有些重要组件后面还会细讲。如 NVIC, 融入了后面各章中。在本章中, 可能与调试有关的内容最难一下子理解。但不用担心, 在第 15 章和第 16 章中, 对调试相关的内容有展开论述。说实话, 本章也是翻译得最累心的章节之一。

6.1 流水线

Cortex-M3 处理器使用一个 3 级流水线。流水线的 3 个级分别是: 取指, 解码和执行, 如图 6.1 所示:

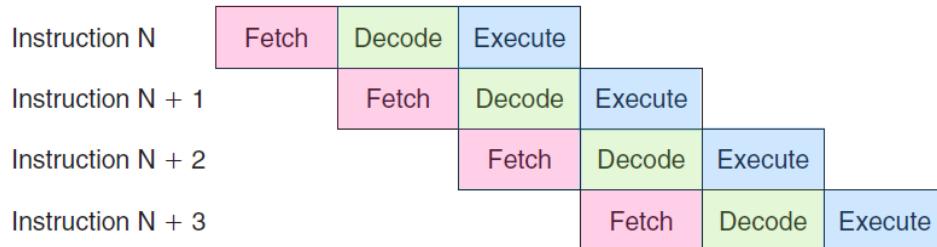


图 6.1 Cortex-M3 的三级流水线

有些人会提出质疑, 认为其实是 4 级, 理由是总线接口在访问内存时的行为。但是这一级是在处理器的外部, 故而处理器自身还是只有 3 级流水线。

当运行的指令大多数都是 16 位时, 你会发现处理器会每隔一个周期做一次取指。这是因为 CM3 有时可以一次取出两条指令来 (一次能取 32 位), 因此在第一条 16 位指令取来时, 也顺带着把第二条 16 位指令取来了。此时总线接口就可以先歇一个周期再取指。或者如果缓冲区是满的, 总线接口干脆就空闲下来了。有些指令的执行需要多个周期, 在这期间流水线就会暂停。

当执行到跳转指令时, 需要清洗流水线, 处理器会不得不从跳转目的地重新取指。为了改善这种情况, CM3 支持一定数量的 ARMv7M 新指令, 可以避免很多微型跳转, 如第 4 章讲到的 IF-THEN 语句

块。

由于流水线的存在，以及出于对Thumb代码兼容的考虑，读取PC时，会返回当前指令地址+4的值。这个偏移量总是4，不管是执行16位指令还是32位指令，这就保证了在Thumb和Thumb2之间的一致性。

在处理器内核的预取单元中也有一个指令缓冲区，它允许后续的指令在执行前先在里面排队，也能在执行未对齐的32位指令时，避免流水线“断流”。不过该缓冲区并不会在流水线中添加额外的级数，因此不会使跳转导致的性能下降（penalty）更加恶化。

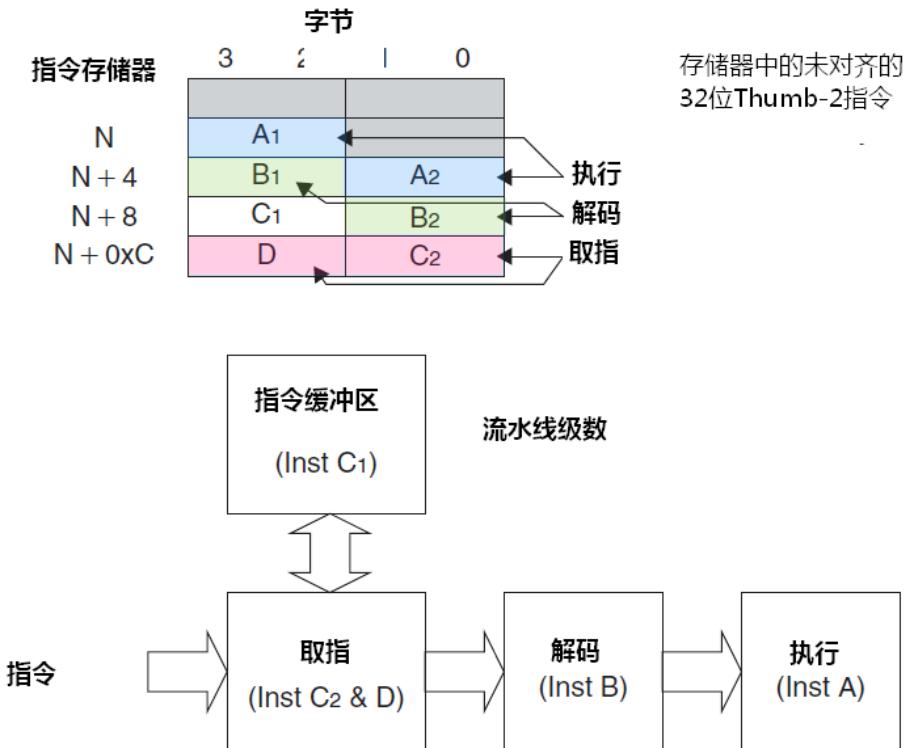


图 6.2 取指单元使用缓冲区对 32 位指令处理的性能提升

6.2 详细的框图

Cortex-M3 处理器其实是个大礼包，里面除了处理核心外，还有了好多其它组件，以用于系统管理和调试支持。

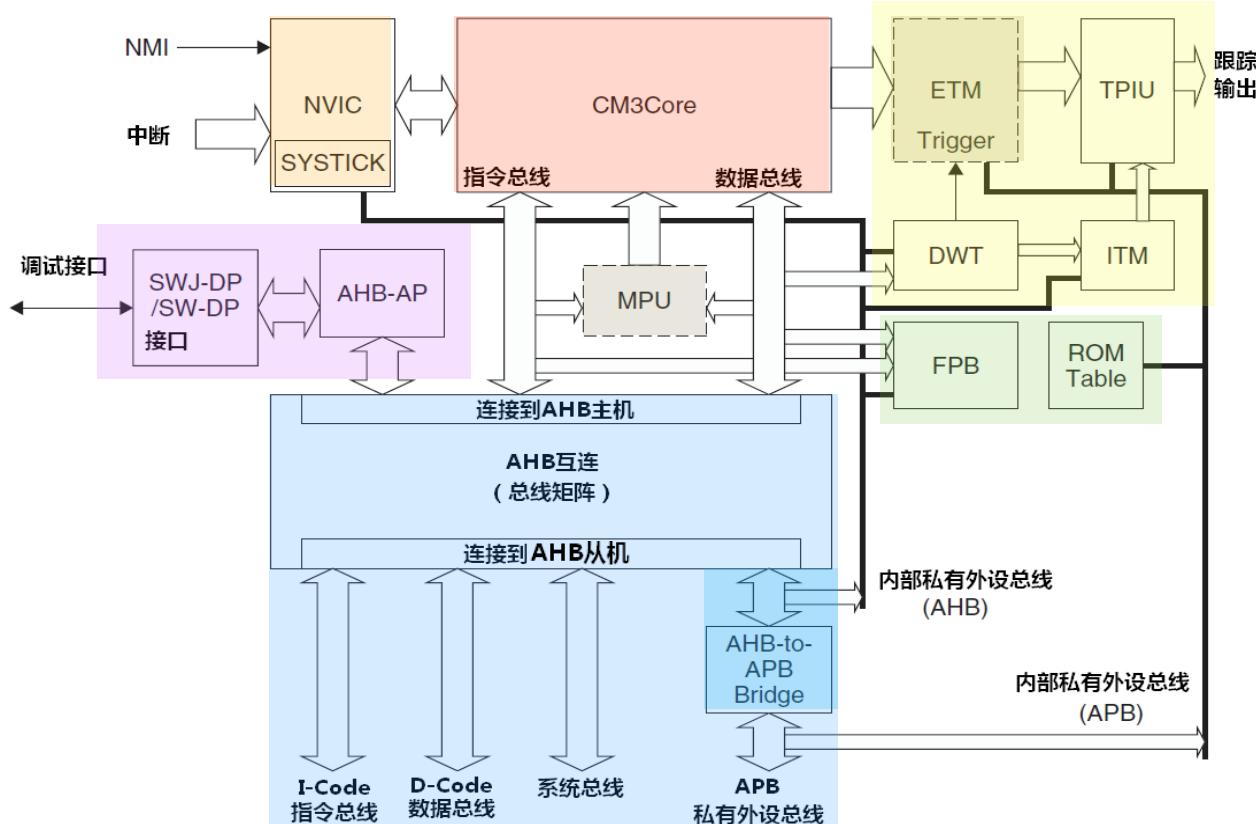


图 6.3 Cortex-M3 处理器系统方框图

请注意：虚线框住的 MPU 和 ETM 是可选组件，不一定会包含在每一个 Cortex-M3 的 MCU 中。好多新东东，图中一时看不清了，表 6.1 列出了新组件的清单。

表 6.1 方框图中的缩写及其定义

缩写	含义
NVIC	嵌套向量中断控制器
SYSTICK Timer	一个简易的周期定时器，用于提供时基，亦被操作系统所使用
MPU	存储器保护单元（可选）
CM3BusMatrix	内部的 AHB 互连
AHB to APB	把 AHB 转换为 APB 的总线桥
SW-DP/SWJ-DP	串行线调试端口/串行线 JTAG 调试端口。通过串行线调试协议或者是传统的 JTAG 协议（专用于 SWJ-DP），都可以用于实现与调试接口的连接
AHB-AP	AHB 访问端口，它把串行线/SWJ 接口的命令转换成 AHB 数据传送
ETM	嵌入式跟踪宏单元（可选组件），调试用。用于处理指令跟踪
DWT	数据观察点及跟踪单元，调试用。这是一个处理数据观察点功能的模块
ITM	仪器化跟踪宏单元
TPIU	跟踪单元的接口单元。所有跟踪单元发出的调试信息都要先送给它，它再

	转发给外部跟踪捕获硬件的。
FPB	Flash 地址重载及断点单元
ROM 表	一个小的查找表，其中存储了配置信息

可见，Cortex-M3 处理器是以一个“处理器子系统”呈现的，其 CPU 内核本身与 NVIC 和一系列调试块都亲密耦合：

- **CM3Core:** Cortex-M3 处理器的中央处理核心
- 嵌套向量中断控制器 **NVIC:** NVIC 是一个在 CM3 中内建的中断控制器。中断的具体路数由芯片厂商定义。NVIC 是与 CPU 紧耦合的，它还包含了若干个系统控制寄存器。因为 NVIC 支持中断嵌套，使得在 CM3 上处理嵌套中断时清爽而强大。NVIC 还采用了向量中断的机制。在中断发生时，它会自动取出对应的服务例程入口地址，并且直接调用，无需软件判定中断源，为缩短中断延时做出了非常重要的贡献。
- **SysTick** 定时器：系统滴答定时器是一个非常基本的倒计时定时器，用于在每隔一定的时间产生一个中断，即使是系统在睡眠模式下也能工作。它使得 OS 在各 CM3 器件之间的移植中不必修改系统定时器的代码，移植工作一下子容易多了。SysTick 定时器也是实现在 NVIC 内部的。
- 存储器保护单元：MPU 是一个选配的单元，有些 CM3 芯片可能没有配备此组件。如果有，则它可以把存储器分成一些 **regions**，并分别予以保护。例如，它可以让某些 regions 在用户级下变成只读，从而阻止了一些用户程序破坏关键数据。
- **BusMatrix:** BusMatrix 是 CM3 内部总线系统的核心。它是一个 AHB 互连的网络，通过它可以让数据在不同的总线之间并行传送——只要两个总线主机不试图访问同一块内存区域。BusMatrix 还提供了附加的数据传送管理设施，包括一个写缓冲以及一个按位操作的逻辑（位带(**bit-band**)）。
- **AHB to APB Bridge:** 它是一个总线桥，用于把若干个 APB 设备连接到 CM3 处理器的私有外设总线上（内部的和外部的）。这些 APB 设备常见于调试组件。CM3 还允许芯片厂商把附加的 APB 设备挂在这条 APB 总线上，并通过 APB 接入其外部私有外设总线。

框图中其它的组件都用于调试，通常不会在应用程序中使用它们，如下所示。

- **SW-DP/SWJ-DP:** 串行线调试端口（SW-DP）/串口线 JTAG 调试端口（SWJ-DP）都与 AHB 访问端口（AHB-AP）协同工作，以使外部调试器可以发起 AHB 上的数据传送，从而执行调试活动。在处理器核心的内部没有 JTAG 扫描链，大多数调试功能都是通过在 NVIC 控制下的 AHB 访问来实现的。SWJ-DP 支持 both 串行线协议和 JTAG 协议，而 SW-DP 只支持串行线协议。
- **AHB-AP:** AHB 访问端口通过少量的寄存器，提供了对 CM3 所有存储器的访问机能。该功能块由 SW-DP/SWJ-DP 通过一个通用调试接口（DAP^[译注]）来控制。当外部调试器需要执行动作的时候，就要通过 SW-DP/SWJ-DP 来访问 AHB-AP，再由 AHB-AP 产生所需的 AHB 数据传送。
译注：DAP 是 SW-DP/SWJ-DP 与 AHB-AP 之间的总线接口（详见第 15 章，图 15.1）
- 嵌入式跟踪宏单元 **ETM:** ETM 用于实现实时指令跟踪，但它是一个选配件，所以不是所有的 CM3 产品都具有实时指令跟踪能力。ETM 的控制寄存器是映射到主地址空间上的，因此调试器可以通过 DAP 来控制它。
- 数据观察点及跟踪单元 **DWT:** 通过 DWT，可以设置数据观察点。当一个数据地址或数据的值匹配了观察点时，就说产生了一次匹配命中事件。匹配命中事件可以用于产生一个观察点事件，后者能激活调试器以产生数据跟踪信息，或者让 ETM 联动（以跟踪在哪条指令上发生了匹配命中事件——译者注）。
- 仪器化跟踪宏单元 **ITM:** ITM 有多种用法。软件可以控制该模块直接把消息送给 TPIU（类

- 似 `printf` 风格的调试); 还可以让 DWT 匹配命中事件通过 ITM 产生数据跟踪包, 并把它输出到一个跟踪数据流中。
- 跟踪端口的接口单元 **TPIU**: TPIU 用于和外部的跟踪硬件 (如跟踪端口分析仪) 交互。在 CM3 的内部, 跟踪信息都被格式化成“高级跟踪总线 (ATB) 包”, TPIU 重新格式化这些数据, 从而让外部设备能够捕捉到它们。
 - **FPB**: FPB 提供 flash 地址重载和断点功能。Flash 地址重载是指: 当 CPU 访问某条指令时, 若该地址在 FPB 中“挂了号”, 则将把该地址重映射到另一个地址, 后者亦在编程 FPB 时指出。结果, 实际上是从映射过的地址处取指 (通常, 映射前的地址是 flash 中的地址, 映射后的地址是 SRAM 中的地址, 所以才是“Flash”地址重载——译者注)。此外, 匹配的地址还能用来触发断点事件。Flash 地址重载功能对于测试工作太有用了。例如, 通过使用 FPB 来改变程序流程, 就可以给那些不能在普通情形下使用的设备添加诊断程序代码 (such as adding diagnosis program code to a device that cannot be used in normal situations unless the FPB is used to change the program control.)。
 - **ROM 表**: 它只是一个简单的查找表。其实更像一个“注册表”: 提供了存储器的“注册”信息, 这些信息指出, 在这块 CM3 芯片中包括了哪些系统设备和调试组件, 以及它们的位置。当调试系统定位各调试组件时, 它需要找出相关寄存器在存储器中的地址, 这些信息由此表给出。在绝大多数情况下, 因为 CM3 有固定的存储器映射, 所以各组件都对号入座——拥有一致的起始地址。但是因为有些组件是可选的, 还有些组件是可以由制造商另行添加的, 各芯片制造商可能需要定制他们芯片的调试功能。以后 CM3 芯片会有越来越多的品牌和型号。而林子大了什么鸟都有, 如果确有厂商“玩另类”, 它就必须在 ROM 表中给出这些“另类”的信息, 这样调试软件才能判定正确的存储器映射, 进而可以检测可用的调试组件是何种类型。

6.3 Cortex-M3 的总线接口

这部分内容是给 SoC 设计师看的。如果你不是他们, 是不能直接访问这里讲到的到总线接口的。

通常情况下, 芯片厂商都会钩住 (hook up) 所有送往存储器和外设的总线信号。并且在少数情况下, 你会发现芯片厂商把总线连接到了总线桥上, 并且允许外部总线系统连接到芯片上。CM3 处理器的总线接口是基于 AHB-Lite 和 APB 协议的, 它们的规格在 AMBA 规格书 (第 4 版) 中给出。

6.3.1 I-Code 总线

I-Code 总线是一条基于 AHB-Lite 总线协议的 32 位总线, 负责在 0x0000_0000 – 0x1FFF_FFFF 之间的取指操作。取指以字的长度执行, 即使是对于 16 位指令也如此。因此 CPU 内核可以一次取出两条 16 位 Thumb 指令。

6.3.2 D-Code 总线

D-Code 总线也是一条基于 AHB-Lite 总线协议的 32 位总线, 负责在 0x0000_0000 – 0x1FFF_FFFF 之间的数据访问操作。尽管 CM3 支持非对齐访问, 但你绝不会在该总线上看到任何非对齐的地址, 这是因为处理器的总线接口会把非对齐的数据传送都转换成对齐的数据传送。因此, 连接到 D-Code 总线上的任何设备都只需支持 AHB-Lite 的对齐访问, 不需要支持非对齐访问。

6.3.3 系统总线

系统总线也是一条基于 AHB-Lite 总线协议的 32 位总线, 负责在 0x2000_0000 – 0xDFFF_FFFF 和

0xE010_0000 – 0xFFFF_FFFF 之间的所有数据传送，取指和数据访问都算上。和 D-Code 总线一样，所有的数据传送都是对齐的。

6.3.4 外部私有外设总线

这是一条基于 APB 总线协议的 32 位总线。此总线来负责 0xE004_0000 – 0xE00F_FFFF 之间的私有外设访问。但是，由于此 APB 存储空间的一部分已经被 TPIU、ETM 以及 ROM 表用掉了，就只留下了 0xE004_2000-E00F_F000 这个区间用于配接附加的（私有）外设。

6.3.5 调试访问端口总线

调试访问端口总线接口是一条基于“增强型 APB 规格”的 32 位总线，它专用于挂接调试接口，例如 SWJ-DP 和 SW-DP。

不要挪用此总线。第 15 章（调试架构）给出该总线的更多信息，在 ARM 的文档《CoreSight Technology System Design Guide (Ref 3)》中也有更详尽的论述。

6.4 Cortex-M3 的其它接口

除了总线接口之外，CM3 还有若干个用于其它目的的接口，这些接口的信号都不大可能会引出到引脚上，而只用于连接 SoC 不同的部分，或者干脆就没有使用。关于这些信号的详述，请参阅《Cortex-M3 Technical Reference Manual(TRM)(Ref1)》。表 6.2 中给出了它们中一些信号的简短小结。

表 6.2 杂项接口信号

信号组	功能
多处理机通信 (TXEV, RXEV)	多处理机之间的简单任务同步信号
休眠信号 (SLEEPING, SLEEPDEEP)	电源管理所用的休眠状态
中断状态信号 (ETMINTNUM, ETMINTSTATE, CURRPRI)	中断操作的状态，用于 ETM 操作和调试
复位请求(SYSRESETREQ)	来自 NVIC 的复位请求输出
锁定(Lockup) ^[译注] 和停机(Halted)状态 (LOCKUP, HALTED)	指示处理器进入了锁定状态（由在硬 fault 和 NMI 服务例程的执行错误导致），或者指示处理器被喊停（因为调试动作导致）
端输入(ENDIAN)	在内核复位时设置端模式
ETM 接口	连接到嵌入式跟踪宏单元（用于指令跟踪）
ITM 的 ATB 接口	高级跟踪总线 (ATB) 是 ARM CoreSight 调试架构下的一个总线协议，用于跟踪数据的传送。在这里，该接口负责把来自 ITM 的跟踪数据输出到 TPIU

[译注] 第 12 章讨论有关 Lockup 的更多内容

6.5 外部私有外设总线

CM3 处理器有一个外部私有外设总线(PPB)接口。外部 PPB 接口是基于高级外设总线(APB)协议构造的。用于非共享的系统设备，例如调试组件。为了支持 CoreSight 设备，该接口又包含了称为“PADDR31”的信号，给出传送的发源地。若该信号为 0，则表示是运行在 CM3 内部的软件产生了

传送操作；若为 1，则表示是调试硬件产生了传送操作。有了这个信号，外设就可以有选择地响应，比如：只响应调试硬件。也可以通融点：当软件发起数据传送时，限制一些功能。

该总线是专用的，不服务于普通的外设，这个规矩只能靠芯片设计者自觉遵守。如果设计者把通用的外设连接到该总线上，用户在使用芯片时就往往会遇到各种莫名其妙的问题——由特权访问管理造成。例如，在用户级下访问这些设备，或者在使用 MPU 时把这些设备从其它的存储 regions 中分开，都会遇到问题，势必影响芯片的销量。

外部 PPB 不支持非对齐访问。因为该总线的宽度是 32 位并且是基于 APB 的，当你在为该存储区域设计外设时，必须确保所有的寄存器地址都是按字对齐的。另外，在编写这些设备的驱动程序时，最好让所有的访问都使用字的长度。最后，PPB 访问永远是小端的。

6.6 连接方式样板

由上可见，CM3 中有若干个总线接口，初学者很容易混淆，也不太容易弄清楚它们是怎样与其它设备和存储器连接的。这里给出一个样板的连接实例，如图 6.4 所示。

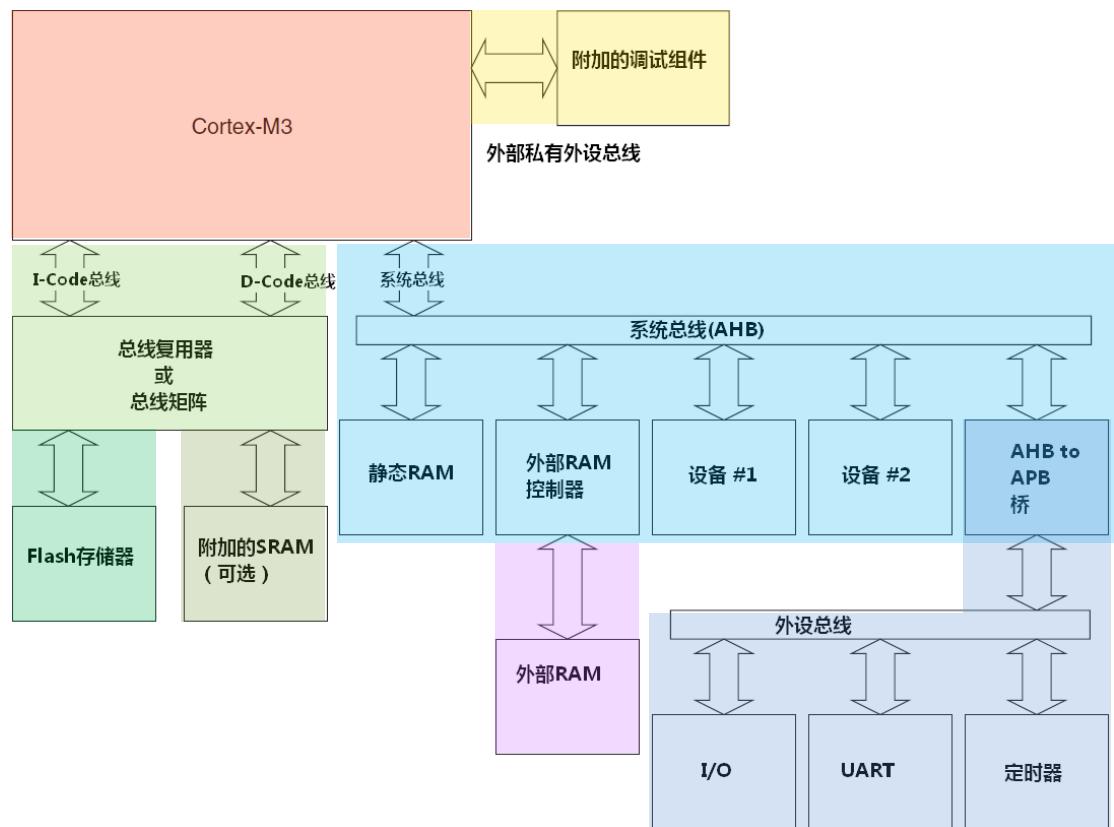


图 6.4 Cortex-M3 总线连接样板范例

因为代码存储区既可以由指令总线(I-Code)访问（当从此区取指时），也可以被数据总线(D-Code)访问（当在此区访问数据时），需要在中间插入一个总线开关，称为“总线矩阵^[译注]”；或者使用一个 AHB 总线复用器。如果使用了总线矩阵，则闪存和附加的 SRAM（如果有的话）可以被 both I-Code 和 D-Code 访问。总线矩阵可以在 ARM 的 AMBA 开发包 ADK（ADK, AMBA 组件和示例系统的集合，使用 VHDL/Verilog 编写）中提供。

[译注]：这里所讲的总线矩阵不是 CM3 内部的总线矩阵，它们是两码事。CM3 内部的总线矩阵是专门设计的，不能作为一个通用的 AHB 开关来使用。

当数据访问和取指同时尝试访问同一块区域时，可以赋予数据访问更高的优先级以提高性能。

在使用 AHB 总线矩阵把取指和数据访问分开后，如果指令总线和数据总线在同一时刻访问不同的存储器设备（例如，从 flash 中取指的同时从附加的 SRAM 中访问数据），则两者可以并行不悖。但若是只使用了总线复用器，则数据传送就不能同时发生了，然而这时电路尺寸能做得更小。不过，通常的 CM3 单片机都使用系统总线来连接 SRAM。而且主 SRAM 确实应该使用系统总线来连接。只有这样才能落到 SRAM 存储器的地址区，从而得以利用 CM3 的位带操作能力。

有些脚数比较多的单片机会带外部总线接口（EMI）。这种情况下，需要一个外部存储器控制器，因为 AHB 不接受直接把片外存储器挂在它上面，通常外部存储器控制器也连接到系统总线上。其它的 AHB 设备则可以简单地连接到系统总线上，而不需要额外的总线矩阵。

图6.4给出的只是一个很简单的典型示范，芯片设计师也可以选择其它的总线连接方案。对于软件/固件的开发，不需了解这么多细节，只需要知道详细的存储器映射就够了。

上图显示出的功能框，像总线矩阵、AHB-to-APB总线桥、存储器控制器、I/O接口、定时器以及UART等，都可以从ARM和其它IP供应商处取得。不同的CM3单片机其片上外设也不同。因此在使用时，你还需要参考器件厂家提供的参考手册。

6.7 复位信号

基于CM3的单片机对复位电路有特定的要求，具体内容在《Cortex-M3 Technical Reference Manual(Ref1)》中给出，它列出了若干个可以使用的复位信号。不过，实现成单片机后，往往只用到了1至2个。至余其它的，芯片厂商会在芯片中布设复位信号发生器，由它在内部产生剩余的复位信号。如欲获取细节，还需要参考制造商提供的数据手册，以理解如何正确复位其芯片。在CM3处理器的水平上，复位信号由表6.3列出。

表6.3 Cortex-M3中的各种复位信号

复位信号	描述
上电复位 (nPOR RESET)	在器件上电时需要把复位置为有效 (assert)，把处理器核心和调试系统一起复位
系统复位 (nSYS RESET)	只影响处理器核心、NVIC (与调试相关的除外) 以及MPU，不复位调试系统
测试复位 (nTRST)	只复位调试系统

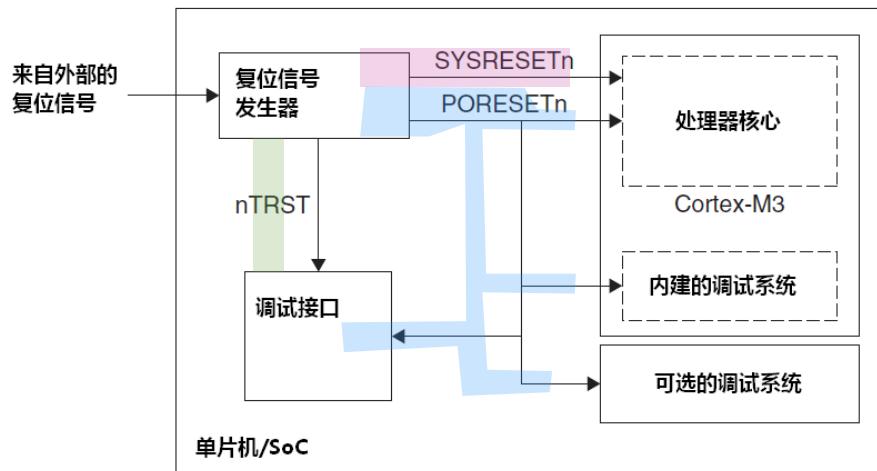


图6.5 典型的Cortex-M3芯片内部复位信号和其作用范围示意图

异常

- 异常类型
- 优先级的定义
- 向量表
- 中断输入以及悬起行为
- Fault 异常
- SVC 和 PendSV

7.1 异常类型

Cortex-M3 在内核水平上搭载了一个异常响应系统，支持为数众多的系统异常和外部中断。其中，编号为 1—15 的对应系统异常，大于等于 16 的则全是外部中断。除了个别异常的优先级被定死外，其它异常的优先级都是可编程的。

译注：所有能打断正常执行流的事件都称为异常。在本书中，经常混合使用术语“中断”与“异常”。如不加说明，则强调的都是它们对主程序所体现出来的“中断”性质，与我们以前学单片机时所讲的概念是相同的。如果非得分个丁一卯二，则中断与异常的区别在于，那 240 个中断对 CM3 核来说都是“意外突发事件”——也就是说，该请求信号来自 CM3 内核的外面，来自各种片上外设和外扩的外设，对 CM3 来说是“异步”的；而异常则是因 CM3 内核的活动产生的——在执行指令或访问存储器时产生，因此对 CM3 来说是“同步”的。

因为芯片设计者可以修改 CM3 的硬件描述源代码，所以做成芯片后，支持的中断源数目常常不到 240 个，并且优先级的位数也由芯片厂商最终决定。

类型编号为 1—15 的系统异常如表 7.1 所示（注意：没有编号为 0 的异常），从 16 开始的外部中断类型如表 7.2 所示。

表 7.1 系统异常清单

编号	类型	优先级	简介
0	N/A	N/A	没有异常在运行
1	复位	-3 (最高)	复位
2	NMI	-2	不可屏蔽中断 (来自外部 NMI 输入脚)
3	硬(hard)fault	-1	所有被除能的 fault，都将“上访”(escalation)成硬 fault。只要 FAULTMASK 没有置位，硬 fault 服务例程就被强制执行。Fault 被除能的原因包括被禁用，或者被 PRIMASK/BASEPRI 被掩蔽。若 FAULTMASK 也置位，则硬 fault 也被除能，此时彻底“关中”
4	MemManage fault	可编程	存储器管理 fault，MPU 访问违例以及访问非法位置均可引发。企图在“非执行区”取指也会引发此 fault
5	总线 fault	可编程	从总线系统收到了错误响应，原因可以是预取流产 (Abort) 或数据流产，企图访问协处理器也会引发此 fault
6	用法(usage)	可编程	由于程序错误导致的异常。通常是使用了一条无效指令，或者是

	Fault	非法的状态转换，例如尝试切换到 ARM 状态	
7-10	保留	N/A	N/A
11	SVCall	可编程	执行系统服务调用指令 (SVC) 引发的异常
12	调试监视器	可编程	调试监视器 (断点，数据观察点，或者是外部调试请求)
13	保留	N/A	N/A
14	PendSV	可编程	为系统设备而设的“可悬挂请求” (pendable request)
15	SysTick	可编程	系统滴答定时器 (也就是周期性溢出的时基定时器——译注)

表 7.2 外部中断清单

编号	类型	优先级	简介
16	IRQ #0	可编程	外中断#0
17	IRQ #1	可编程	外中断#1
...
255	IRQ #239	可编程	外中断#239

在 NVIC 的中断控制及状态寄存器中，有一个 **VECTACTIVE** 位段；另外，还有一个特殊功能寄存器 **IPSR**。在它们二者的里面，都记录了当前正服务的异常，给出了它的编号。

请注意：这里所讲的中断号，都是指 **NVIC** 所使用的中断号。另一方面，芯片一些管脚的名字也可能被取为类似“**IRQ #**”的名字，请不要混淆这两者，它们没有必然的映射关系。常见的情况是，**NVIC** 中编号最靠前的几个中断源被指定到片上外设，接下来的中断源才给外部中断引脚使用，因此还是要参阅芯片的数据手册来弄清楚。

如果一个发生的异常不能被即刻响应，就称它被“悬起” (**pending**)。不过，少数 **fault** 异常是不允许被悬起的。一个异常被悬起的原因，可能是系统当前正在执行一个更高优先级异常的服务例程，或者因相关掩蔽位的设置导致该异常被除能。对于每个异常源，在被悬起的情况下，都会有一个对应的“悬起状态寄存器”保存其异常请求。待到该异常能够响应时，执行其服务例程，这与传统的 **ARM** 是完全不同的。在以前，是由产生中断的设备保持住请求信号；**CM3** 则由 **NVIC** 的悬起状态寄存器来解决这个问题。于是，哪怕设备在后来已经释放了请求信号，曾经的中断请求也不会错失。

7.2 优先级的定义

在 **CM3** 中，优先级对于异常来说很关键的，它会决定一个异常是否能被掩蔽，以及在未掩蔽的情况下何时可以响应。优先级的数值越小，则优先级越高。**CM3** 支持中断嵌套，使得高优先级异常会抢占 (**preempt**) 低优先级异常。有 3 个系统异常：复位，**NMI** 以及硬 **fault**，它们有固定的优先级，并且它们的优先级号是负数，从而高于所有其它异常。所有其它异常的优先级则都是可编程的 (但不能被编程为负数——译者注)。

原则上，**CM3** 支持 3 个固定的高优先级和多达 256 级的可编程优先级，并且支持 128 缓存占 (128 的来历请见下文分解——译注)。但是，绝大多数 **CM3** 芯片都会精简设计，以致实际上支持的优先级数会更少，如 8 级，16 级，32 级等。它们在设计时会裁掉表达优先级的几个低端有效位，以减少优先级的级数 (可见，不管使用多少位来表达优先级，都是以 **MSB** 对齐的——译者注)。

举例来说，如果只使用了 3 个位来表达优先级，则优先级配置寄存器的结构会如图 7.1 所示：

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
用于表达优先级	没有实现，读回零						

图 7.1 使用 3 个位来表达优先级的情况

在图中，[4:0]没有被实现，所以读它们总是返回零，写它们则忽略写入的值。因此，对于 3 个位的情况，我们能够使用的 8 个优先级为：0x00（最高），0x20，0x40，0x60，0x80，0xA0，0xC0 以及 0xE0。

如果使用更多的位来表达优先级，则可以使用的值也更多，同时需要的门也更多——带来更多的成本和功耗。CM3 允许的最少使用位数为 3 个位，亦即至少要支持 8 级优先级。

下图给出使用 3 个位表达优先级 vs. 使用 4 个位表达优先级的图景：

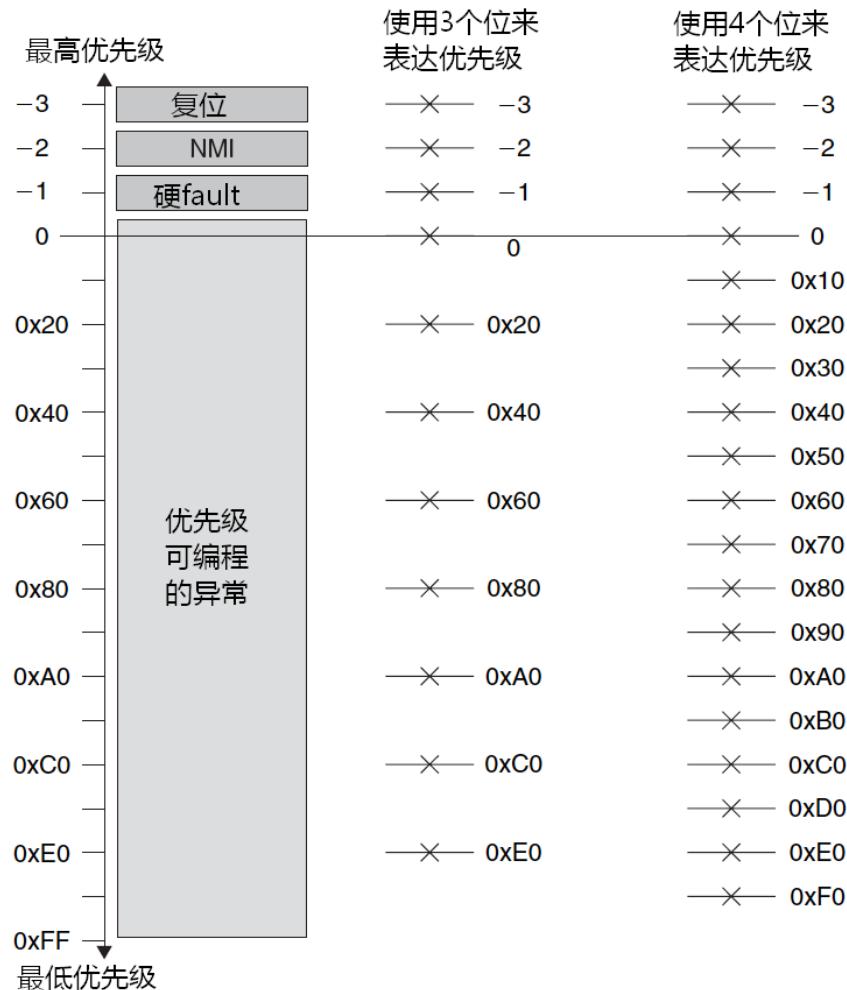


图 7.3 3 位表达的优先级 vs. 4 位表达的优先级

通过让优先级以 **MSB** 对齐，可以简化程序的跨器件移植。比如，如果一个程序早先在支持 4 位优先级的器件上运行，在移植到只支持 3 位优先级的器件后，其功能不受影响。但若是对齐到 **LSB**，则会使 **MSB** 丢失，导致数值大于 7 的低优先级一下子升高了，甚至会发生“优先级反转”：使它高于小于等于 7 的优先级。如，8 号优先级因为损失了 **MSB**，现在反而变成 0 号了；而 15 号优先级则变成 7 号优先级，它则不会影响 0-6 号优先级，使得这个问题更隐蔽。

那么当使用了 3 位、5 位及 8 位来表达优先级时，各是什么情况呢？如表 7.3 所示：

表 7.3 3 位、5 位和 8 位表达优先级时，优先级寄存器的使用情况

优先级	异常类型	3 个位表达	5 个位表达	8 个位表达
-3(最高)	复位	-3	-3	-3
-2	NMI	-2	-2	-2
-1	硬 fault	-1	-1	-1
0, 1, ... 0xFF	所有其它优先级可编程的异常	0x00, 0x20 ... 0xE0	0x00 0x08 ... 0xF8	0x00,0x01 0x02,0x03 ... 0xFE,0xFF

抢占优先级与子优先级

有钻劲儿的读者可能一直在琢磨：明明支持 256 个优先级，为啥只有 128 个抢占级，剩下一半哪儿去了？原来，为了使抢占机能变得更可控，CM3 还把 256 级优先级按位分成高低两段，分别称为抢占优先级和子优先级，如下所述。

NVIC 中有一个寄存器是“应用程序中断及复位控制寄存器”（内容见表 7.5），它里面有一个位段名为“优先级组”。该位段的值对每一个优先级可配置的异常都有影响——把其优先级分为 2 个位段：MSB 所在的位段（左边的）对应抢占优先级，而 LSB 所在的位段（右边的）对应子优先级，如表 7-4 所示。

表 7.4 抢占优先级和子优先级的表达，位数与分组位置的关系

分组位置	表达抢占优先级的位段	表达子优先级的位段
0	[7:1]	[0:0]
1	[7:2]	[1:0]
2	[7:3]	[2:0]
3	[7:4]	[3:0]
4	[7:5]	[4:0]
5	[7:6]	[5:0]
6	[7:7]	[6:0]
7	无	[7:0]（所有位）

表 7.5 应用程序中断及复位控制寄存器(AIRCR) (地址 : 0xE000_ED00)

位段	名称	类型	复位值	描述
31:16	VECTKEY	RW	-	访问钥匙：任何对该寄存器的写操作，都必须同时把 0x05FA 写入此段，否则写操作被忽略。若读取此半字，则 0xFA05
15	ENDIANESS	R	-	指示端设置。1=大端(BE8)，0=小端。此值是在复位时确定的，不能更改。
10:8	PRIGROUP	R/W	0	优先级分组
2	SYSRESETREQ	W	-	请求芯片控制逻辑产生一次复位

1	VECTCLRACTIVE	W	-	清零所有异常的活动状态信息。通常只在调试时用，或者在 OS 从错误中恢复时用。
0	VECTRESET	W	-	复位 CM3 处理器内核（调试逻辑除外），但是此复位不影响芯片上在内核以外的电路

译注：抢占优先级有时亦称为“组优先级”，或者“主优先级”。

抢占优先级决定了抢占行为：当系统正在响应某异常 L 时，如果来了抢占优先级更高的异常 H，则 H 可以抢占 L。子优先级则处理“内务”：当抢占优先级相同的异常有不止一个悬起时，就最先响应子优先级最高的异常。

这种优先级分组做出了如下规定：子优先级至少是 1 个位。因此抢占优先级最多是 7 个位，这就造成了最多只有 128 级抢占的现象。

但是 CM3 允许从比特 7 处分组，此时所有的位都表达子优先级，没有任何位表达抢占优先级，因而所有优先级可编程的异常之间就不会发生抢占——相当于在它们之中除能了 CM3 的中断嵌套机制。当然还有凌驾于法律之上的三位老大：复位，NMI 和硬 fault。它们无论何时出现，都立即无条件抢占所有优先级可编程的“平民异常”。

在计算抢占优先级和子优先级的有效位数时，必须先求出下列值：

- 芯片实际使用了多少位来表达优先级
- 优先级组是如何划分的。

举个例子，如果只使用 3 个位来表达优先级（[7:5]），并且优先级组的值是 5（从比特 5 处分组），则得到 4 级抢占优先级，且在每个抢占优先级的内部有 2 个子优先级，如图 7.4 所示。

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
抢占优先级	子优先级						

图 7.4 3 位优先级，从比特 5 处分组时，优先级位段的划分

根据图 7.4 中演示的设置，其可用优先级的具体情况如图 7.5 所示。

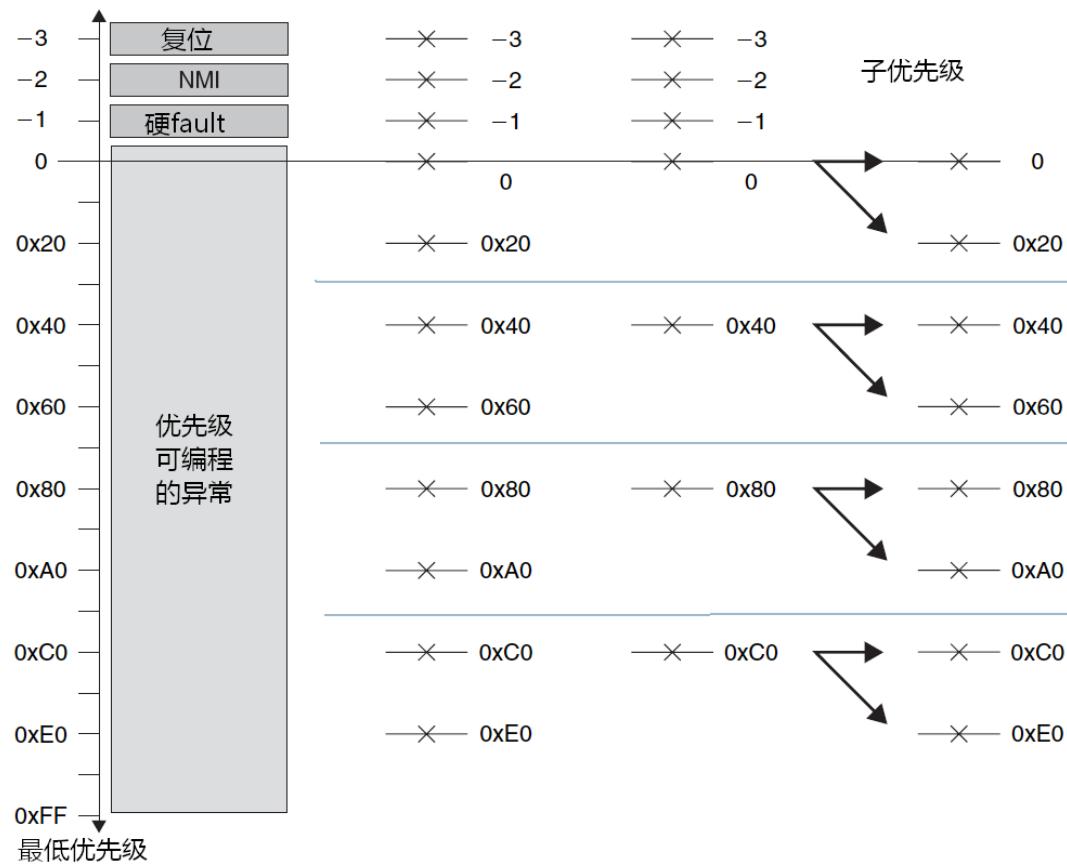


图 7.5 三位优先级，从比特 5 处分组

请注意：虽然[4:0]未使用，却允许从它们中分组。例如，如果优先级组为 1，则所有可用的 8 个优先级都是抢占优先级，如图 7.6 和图 7.7 所示。

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
抢占优先级[7:5]			抢占优先级[4:2] (未使用)			亚优先级[1:0] (未使用)	

图 7.6 3 位优先级，从比特 1 处分组

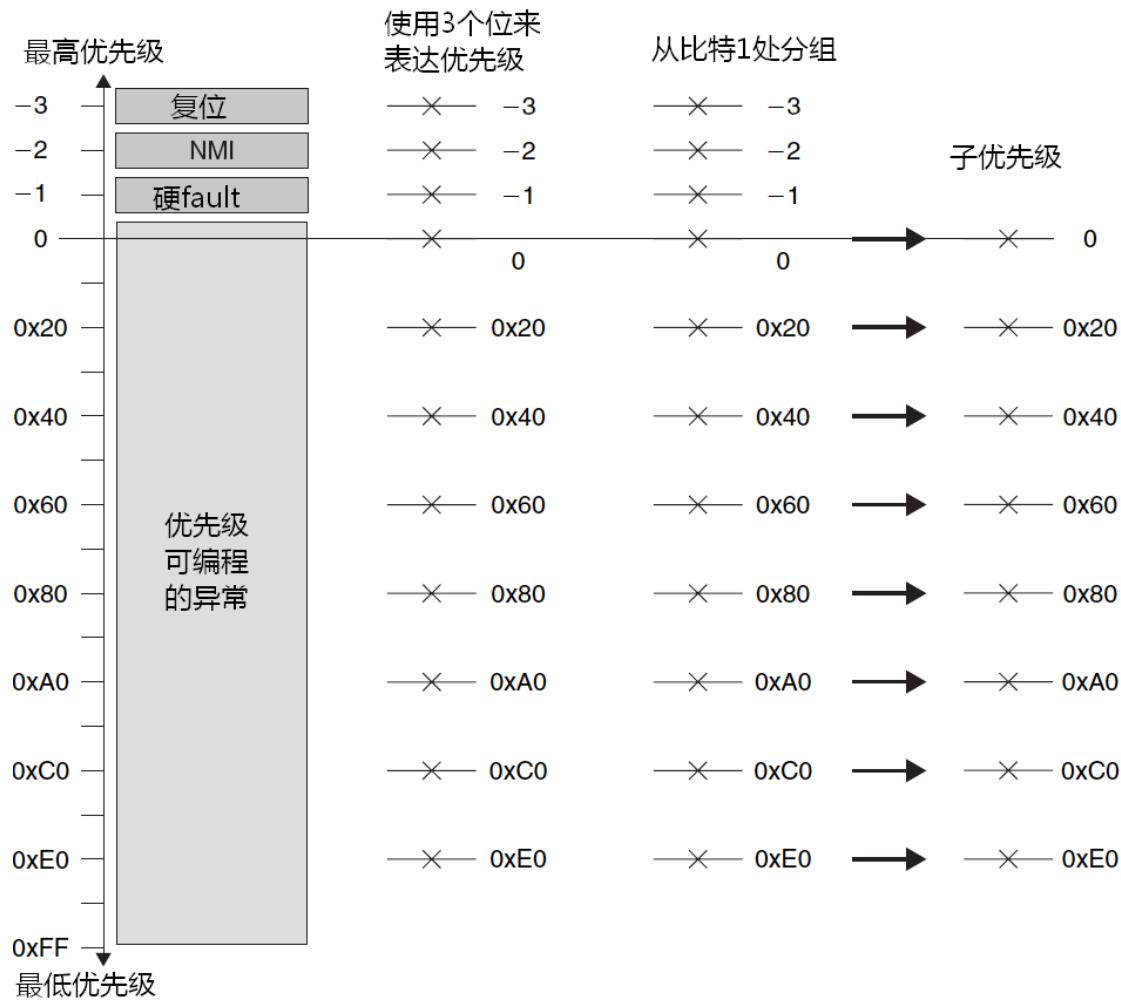


图 7.7 3 位优先级，从比特 1 处分组，详细情况

如果优先级完全相同的多个异常同时悬起，则先响应异常编号最小的那个。例如，当 IRQ #3 的优先级与 IRQ #5 的优先级相等时，IRQ #3 会比 IRQ #5 先得到响应。

虽然优先级分组的功能很强大，但是粗心地更改会使它变得很暴力，尤其是在设计硬实时系统的时候，这简直就是在玩火——常常会改变系统的响应特性。导致某些关键任务有可能得不到及时响应，凶多吉少的意外随时可能猛烈发作。其实在绝大多数情况下，优先级的分组都要预先经过计算论证，并且在开机初始化时一次性地设置好，以后就再也不动它了。只有在绝对需要且绝对有把握时，才小心地更改，并且要经过尽可能充分的测试。另外，优先级组所在的寄存器 AIRCR (回顾表 7.5) 也基本上是“一次成型”，只是需要手工产生复位时才写里面相应的位。

7.3 向量表

当发生了异常并且要响应它时，CM3 需要定位其服务例程的入口地址。这些入口地址存储在所谓的“(异常) 向量表”中。缺省情况下，CM3 认为该表位于零地址处，且各向量占用 4 字节。因此每个表项占用 4 字节，如表 7.6 所示。

表 7.6 上电后的向量表

地址	异常编号	值 (32 位整数)
0x0000_0000	-	MSP 的初始值
0x0000_0004	1	复位向量 (PC 初始值)
0x0000_0008	2	NMI 服务例程的入口地址
0x0000_000C	3	硬 fault 服务例程的入口地址
...	...	其它异常服务例程的入口地址

因为地址 0 处应该存储引导代码, 所以它通常映射到 Flash 或者是 ROM 器件, 并且它们的值不得在运行时改变。然而, 为了支持动态重分发中断, CM3 允许向量表重定位——从其它地址处开始定位各异常向量。这些地址对应的区域可以是代码区, 但更多是在 RAM 区。在 RAM 区就可以修改向量的入口地址了。为了实现这个功能, NVIC 中有一个寄存器, 称为“向量表偏移量寄存器”(在地址 0xE000_ED08 处), 通过修改它的值就能重定位向量表。但必须注意的是: 向量表的起始地址是有要求的: 必须先求出系统中共有多少个向量, 再把这个数字向上“圆整”到 2 的整次幂, 而起始地址必须对齐到后者的边界上。例如, 如果一共有 32 个中断, 则共有 $32+16$ (系统异常) = 48 个向量, 向上圆整到 2 的整次幂后值为 64, 因此向量表重定位的地址必须能被 $64*4=256$ 整除, 从而合法的起始地址可以是: 0x0, 0x100, 0x200 等。向量表偏移量寄存器的定义如表 7.7 所示。

表 7.7 向量表偏移量寄存器(VTOR) (地址: 0xE000_ED08)

位段	名称	类型	复位值	描述
7-28	TBLOFF	RW	0	向量表的起始地址
29	TBLBASE	R	-	向量表是在 Code 区 (0), 还是在 RAM 区 (1)

如果需要动态地更改向量表, 则对于任何器件来说, 向量表的起始处都必须包含以下向量:

- 主堆栈指针 (MSP) 的初始值
- 复位向量
- NMI
- 硬 fault 服务例程

后两者也是必需的, 因为有可能在引导过程中发生这两种异常。

可以在 SRAM 中开出一块空间用于存储向量表。在引导期间先填写好各向量, 然后在引导完成后, 就可以启用内存中的新向量表, 从而实现向量可动态调整的能力。

7.4 中断输入及悬起行为

本节开始讨论中断的输入和悬起行为。这也适用于 NMI, 只是对于 NMI 来说, 除了一些特殊情况之外, 将会立即无条件执行其服务例程。这些特殊情况包括: 当前已经在执行 NMI 服务例程; CPU 被调试器喊停(halted); CPU 被一些严重的系统错误锁定 (Lock up)。则新的 NMI 请求也将悬起。

译注: 并不是每种 lock up 都会除能 NMI。这是比较深入的学习, 详情参考第 12 章。

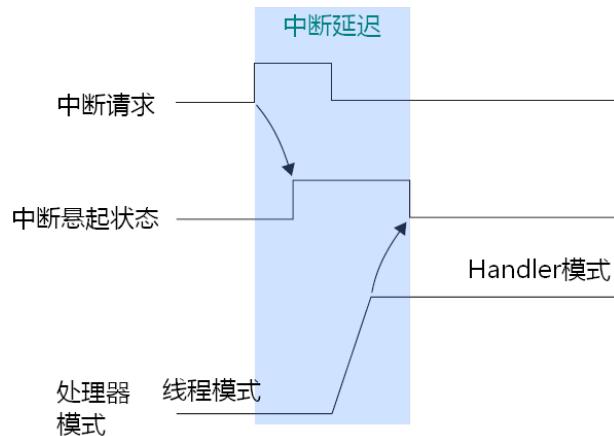


图 7.8 中断悬起示意图

当中断输入脚被置为有效 (assert) t 后，该中断就被悬起。即使后来中断源撤消了中断请求，已经被标记成悬起的中断也被记录下来。到了系统中它的优先级最高的时候，就会得到响应。

但是，如果在某个中断得到响应之前，其悬起状态被清除了（例如，在 PRIMASK 或 FAULTMASK 置位的时候软件清除了悬起状态标志），则中断被取消，如图 7.9 所示。

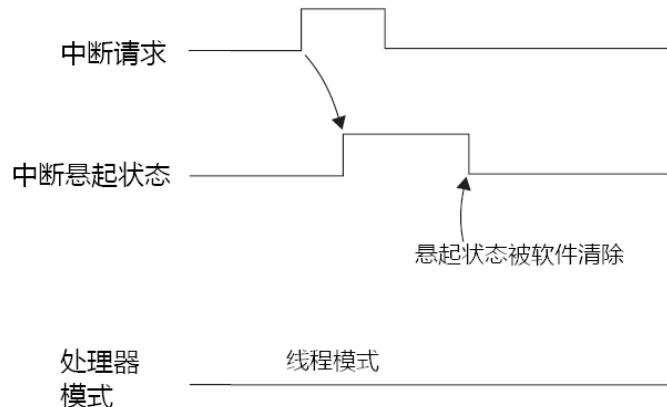


图 7.9 中断在得到处理器响应之前被清除悬起状态

当某中断的服务例程开始执行时，就称此中断进入了“活跃”状态，并且其悬起位会被硬件自动清除，如图 7.10 所示。在一个中断活跃后，直到其服务例程执行完毕，并且返回（亦称为中断退出，第九章详细讨论）后，才能对该中断的新请求予以响应（单实例）。当然，新请求在得到响应时，亦是由硬件自动清零其悬起标志位。中断服务例程也可以在执行过程中把自己对应的中断重新悬起（使用时要注意避免进入“死循环”——译注）。

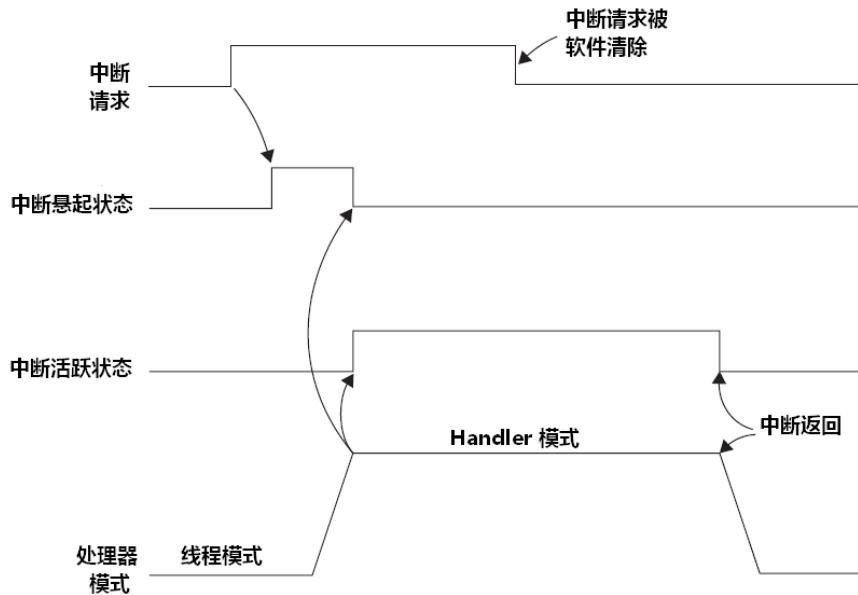


图 7.10 在处理器进入服务例程后对中断活跃状态的设置

如果中断源咬住请求信号不放，该中断就会在其上次服务例程返回后再次被置为悬起状态，如图 7.11 所示。这一点上 CM3 和传统的 ARM7TDMI 是相同的。

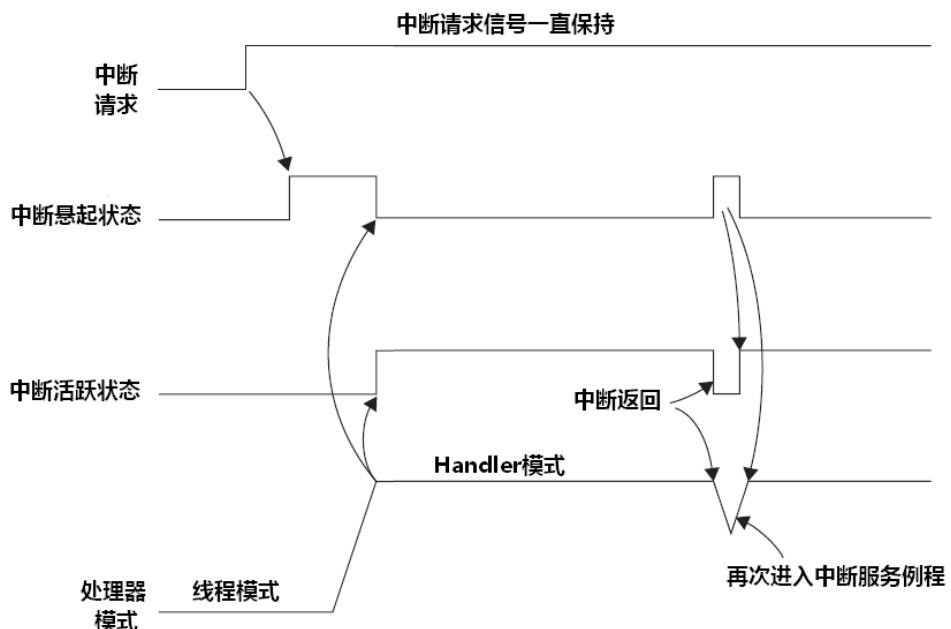


图 7.11 一直维持的中断请求导致服务例程返回后再次悬起该中断

另一方面，如果某个中断在得到响应之前，其请求信号以若干的脉冲的方式呈现，则被视为只有一次中断请求，多出的请求脉冲全部错失——这是中断请求太快，以致于超出处理器反应限度的情况。如图 7.12 所示。

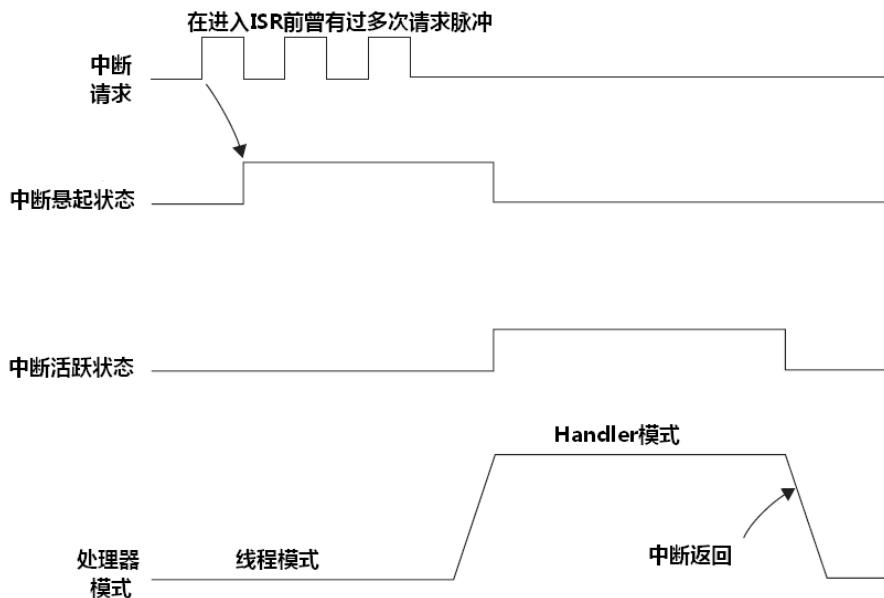


图 7.12 中断请求过快导致一部分请求错失的情况

如果在服务例程执行时，中断请求释放了，但是在服务例程返回前又重新被置为有效，则 CM3 会记住此动作，重新悬起该中断。如图 7.13 所示。

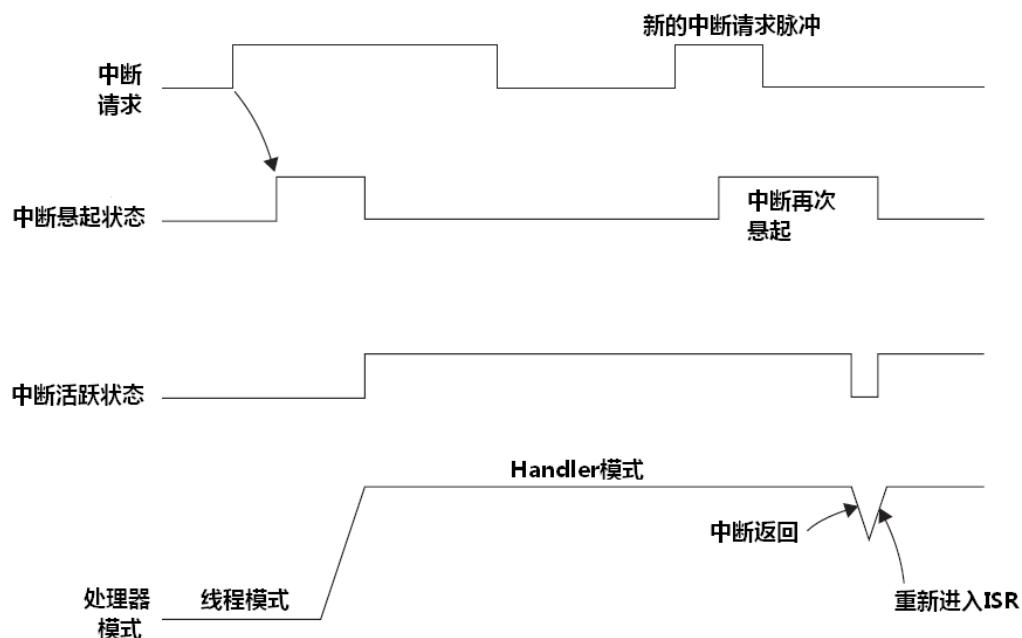


图 7.13 在执行 ISR 时中断悬起再次发生

7.5 Fault 类异常

有若干个系统异常专用于 `fault` 处理。CM3 中的 `Faults` 可分为以下几类：

- 总线 `faults`
- 存储器管理 `faults`
- 用法 `faults`

- 硬 **fault**

7.5.1 总线 **Faults**

当 AHB 接口上正在传送数据时, 如果回复了一个错误信号(**error response**), 则会产生总线 **faults**, 产生的场合可以是:

- 取指, 通常被称作“预取流产”(**prefetch abort**)
- 数据读/写, 通常被称作“数据流产”(**data abort**)

在 CM3 中, 执行如下动作时, 如果地址有误, 亦会触发总线异常:

- 中断处理起始阶段的堆栈 **PUSH** 动作。此时若发生总线 **fault**, 则称为“入栈错误”
- 中断处理收尾阶段的堆栈 **POP** 动作。此时若发生总线 **fault**, 则称为“出栈错误”
- 在处理器启动中断服务序列(**sequence**)后读取向量时。这是一种极度罕见的特殊情况, 被归类为硬 **fault**。

哪些因素会导致 AHB 回复一个错误信号?

AHB 回复的错误信号会触发总线 **fault**, 诱因可以是:

- 企图访问无效的存储器 **region**。常见于访问的地址没有相对应的存储器。
- 设备还没有作好传送数据的准备。比如, 在尚未初始化 SDRAM 控制器的时候试图访问 SDRAM。
- 在企图启动一次数据传送时, 传送的尺寸不能为目标设备所支持。例如, 某设备只接受字型数据, 却试图送给它字节型数据。
- 因为某些原因, 设备不能接受数据传送。例如, 某些设备只有在特权级下才允许访问, 可当前却是用户级。

当上述这些总线 **faults** 发生时(向量的除外), 只要没有同级或更高优先级的异常正在服务, 且没有被掩蔽, 就会执行总线 **fault** 的服务例程。如果在检测到总线 **fault** 时还检测到了更高优先级的异常, 则先处理后者, 而总线 **fault** 被标记成悬起。最后, 如果总线 **fault** 被除能, 或者总线 **fault** 是被某同级或更高优先级异常的服务例程引发的, 则总线 **fault** 被迫成为“硬伤”——上访成硬 **fault**, 使得最后执行的是硬 **fault** 的服务例程(如果当前没有执行 **NMI** 服务例程, 则立即执行硬 **fault** 服务例程——译者注)。如果在硬 **fault** 服务例程的执行中又产生了总线 **fault**(太钻牛角尖了), 内核将进入锁定状态(第 12 章详细讨论)。

欲使能总线 **fault** 服务例程, 需要在 **NVIC** 的“系统 **Handler** 控制及状态寄存器”中置位 **BUSFAULTENA** 位。要注意的是: 在使能之前, 总线 **fault** 服务例程的入口地址必须已经在向量表中配置好, 否则就成了作法自毙——程序可能跑飞。

那么, 发生了总线 **fault** 后, 我们将如何找出该 **fault** 的事故原因呢? 在这里, **NVIC** 提供了若干个 **fault** 状态寄存器, 其中一个名为“总线 **fault** 状态寄存器”(**BFSR**)的。通过它, 总线 **fault** 服务例程可以确定产生 **fault** 的场合: 是在数据访问时, 在取指时, 还是在中断的堆栈操作时。

对于精确的总线 **fault** (见下框说明), 肇事指令的地址被压在堆栈中。如果 **BFSR** 中的 **BFARVALID** 位为 1, 还可以找出是在访问哪块存储器时产生该总线 **fault** 的——该存储器的地址被放到“总线 **fault** 地址寄存器 (**BFAR**)”中。然而, 如果是不精确的总线 **fault**, 就无从定位了。因为在发生 **fault** 时, 处理器已经在执行肇事指令后, 不知又流逝了多少个周期了。

精确的总线 fault vs. 不精确的总线 fault

由数据访问产生的总线 fault，可以进一步被归类为精确总线 fault 和不精确总线 fault。在不精确的总线 faults 中，导致此 fault 的指令早已完成了。例如，缓冲区写入。启动缓冲区写入的指令不知何时已经执行了，但是写到中途时触发了总线 fault。此时，肇事指令早已“逃逸”——在若干个时钟周期就执行过了，而且不能确定是具体几个周期之前，CM3 也不会记录这期间的程序跳转动作。因此无法确认“肇事者”，故而该 fault 是不精确的。精确的总线 fault 则不同，它是被最后一个完成的操作触发的。例如，一个存储器读取导致的 fault 总是精确的，因为该指令必须等全部读完时才算执行完成。这样，任何在读取过程中发生的 fault 总能落在该指令的头上。

BFSR 寄存器的程序员模型如下所示：它是一个 8 位的寄存器，并且可以使用字传送和字节传送来读取它。如果以字方式访问，地址是 `0xE000_ED28`，并且第 2 个字节有效；如果以字节方式访问，则地址直接就是 `0xE000_ED29`，如表 7.8 所示。

表 7.8 总线 fault 状态寄存器(BFSR)，地址：`0xE000_ED29`

位段	名称	类型	复位值	描述
7	BFARVALID	-	0	=1 时表示 BFAR 有效
6:5	-	-	-	-
4	STKERR	R/Wc	0	入栈时发生错误
3	UNSTKERR	R/Wc	0	出栈时发生错误
2	IMPRECISERR	R/Wc	0	不精确的数据访问违例 (violation)
1	PRECISERR	R/Wc	0	精确的数据访问违例
0	IBUSERR	R/Wc	0	取指时的访问违例

7.5.2 存储器管理 faults

存储器管理 **faults** 多与 MPU 有关，其诱因常常是某次访问触犯了 MPU 设置的保护规范。另外，某些非法访问，例如，在不可执行的存储器区域试图取指，也会触发一个 **MemManage fault**，而且在这种场合下，即使没有 MPU 也会触发 **MemManage fault**。

MemManage faults 的常见诱因如下所示：

- 访问了所有 **MPU regions** 覆盖范围之外的地址
- 访问了没有存储器与之对应的空地址
- 往只读 **region** 写数据
- 用户级下访问了只允许在特权级下访问的地址

在 **MemManage fault** 发生后，如果其服务例程是使能的，则执行服务例程。如果同时还发生了其它高优先级异常，则优先处理这些高优先级的异常，**MemManage** 异常被悬起。如果 **MemManage fault** 是被同级或高优先级异常的服务例程引发的，或者 **MemManage fault** 被除能，则和总线 **fault** 一样：上访成硬 **fault**，最终执行的是硬 **fault** 的服务例程。如果硬 **fault** 服务例程或 **NMI** 服务例程的执行也导致了 **MemManage fault**，那就不可救要了——内核将被锁定。

可见，和总线 **fault** 一样，**MemManage fault** 必须被使能才能正常响应。**MemManage fault** 在 **NVIC** “系统 **handler** 控制及状态寄存器” 中的使能位是 **MEMFAULTENA**。如果把向量表置于 **RAM** 中，应优先建立好 **MemManage fault** 服务例程的入口地址。

为了调查 **MemManage fault** 的案发现场，**NVIC** 中有一个“存储器管理 **fault** 状态寄存器 (**MFSR**)”，它指出导致 **MemManage fault** 的原因。如果是因为一个数据访问违例 (**DACCVIOL** 位) 或是一个取指访问违例 (**IACCVIOL** 位)，则违例指令的地址已经被压入栈中。如果还有 **MMARVALID** 位被置位，则还能进一步查出引发此 **fault** 时访问的地址——读取 **NVIC** “存储器管理地址寄存器 (**MMAR**)” 的值。

MFSR 寄存器的程序员模型如下所示。它是一个 8 位的寄存器，并且可以使用字传送和字节传送来读取它。不管使用哪种访问方式，地址都是 **0xE000_ED28**。只不过如果按字访问，就只有第 1 个字节有意义。如表 7.9 所示。

表 7.9 存储器管理 **fault** 状态寄存器(**MFSR**)，地址：**0xE000_ED28**

位段	名称	类型	复位值	描述
7	MMARVALID	-	0	=1 时表示 MMAR 有效
6:5	-	-	-	-
4	MSTKERR	R/Wc	0	入栈时发生错误
3	MUNSTKERR	R/Wc	0	出栈时发生错误
2	-	-	-	-
1	DACCVIOL	R/Wc	0	数据访问违例
0	IACCVIOL	R/Wc	0	取指访问违例

7.5.3 用法 **faults**

用法 **faults** 发生的场合可以是：

- 执行了协处理器指令。**Cortex-M3** 本身并不支持协处理器，但是通过 **fault** 异常机制，可以建立一套“软件模拟”的机制，来执行一段程序模拟协处理器的功能，从而可以方便地在其它 **Cortex** 处理器间移植。
- 执行了未定义的指令。同上一点的道理，亦可以软件模拟未定义指令的功能。
- 尝试进入 **ARM** 状态。因为 **CM3** 不支持 **ARM** 状态，所以用法 **fault** 会在切换时产生。软件可以利用此机制来测试某处理器是否支持 **ARM** 状态。
- 无效的中断返回 (**LR** 中包含了无效/错误的值)
- 使用多重加载/存储指令时，地址没有对齐。

另外，如果需要严格要求程序的质量，还可以让 **CM3** 在遇到除数为零的时候，以及遇到未对齐访问的时候也产生用法 **fault**。在 **NVIC** 中有两个控制位分别与它们对应。通过设置这两个控制位，就可以激活它们。

在使能了用法 **fault** 后，如果在用法 **fault** 发生的时候，已经悬起了更高优先级的异常，则用法 **fault** 被悬起。如果某异常服务例程在执行过程中引发了用法 **fault**，并且该异常的优先级不低于用法 **fault** 的优先级；或者用法 **fault** 被除能，则和总线 **fault** 与 **MemManage fault** 一样，用法 **fault** 上访成硬 **fault**，最终执行的是硬 **fault** 的服务例程。如果硬 **fault** 服务例程或 **NMI** 服务例程的执行竟然也引发了用法 **fault**，那就不可救要了——内核又将被锁定（真不嫌唠叨啊）。

可见，和总线 **fault** 与 **MemManage fault** 一样，用法 **fault** 必须被使能才能正常响应。用法 **fault** 在 **NVIC** “系统 **handler** 控制及状态寄存器” 中的使能位是 **USGFAULTENA**。如果把向量表置于 **RAM** 中，应优先建立好用法 **fault** 服务例程的入口地址（其实作者的本意是：应先建立好 **fault**

类异常服务例程的入口地址，再建立其它异常服务例程的入口地址——译者注)。

为了调查用法 **fault** 的案发现场，NVIC 中有一个“用法 **fault** 状态寄存器 (UFSR)”，它指出导致用法 **fault** 的原因。在服务例程中，导致用法 **fault** 的指令地址被压入堆栈中。

何时会意外地试图切入 ARM 状态

导致用法 **fault** 的最常见原因就是试图切入 ARM 状态。只要在加载 PC 时使用了 LSB 为零的数 (也就是偶数)，就被视作试图切入 ARM 状态，包括：

执行“BX Rn”指令时，Rn 的 LSB=0

异常向量表中入口地址的 LSB=0

POP {...,PC} 时，弹出的数值 LSB=0，这常常是入栈的值被手工改坏造成的
在上述原因导致了用法 **fault** 后，UFSR 中的 INVSTATE 位 (INVAlid STATE) 会置位。

UFSR 的定义如图 7.10 所示。它占用了 2 个字节，可以被按半字访问或是按字访问。按字访问时的地址是 0xE000_ED28，高半字有效；按半字访问时的地址是 0xE000_ED2A。和其它的 FAULT 状态寄存器一样，它里面的位可以通过写 1 来清零。

表 7.10 用法 **fault** 状态寄存器(UFSR)，地址：0xE000_ED2A

位段	名称	类型	复位值	描述
9	DIVBYZERO	R/Wc	0	表示除法运算时除数为零(只有在 DIV_0_TRP 置位时才会发生)
8	UNALIGNED	R/Wc	0	未对齐访问导致的 fault
7:4	-	-	-	-
3	NOCP	R/Wc	0	试图执行协处理器相关指令
2	INVPC	R/Wc	0	在异常返回时试图非法地加载 EXC_RETURN 到 PC。包括非法的指令，非法的上下文以及非法的 EXC_RETURN 值。The return PC 指向的指令试图设置 PC 的值 (要理解此位的含义，还需学习后面的讨论中断级异常的章节)
1	INVSTATE	R/Wc	0	试图切入 ARM 状态
0	UNDEFINSTR	R/Wc	0	执行的指令其编码是未定义的——解码不能

7.5.4 硬 **fault**

硬 **fault** 是上文讨论的总线 **fault**、存储器管理 **fault** 以及用法 **fault** 上访的结果。如果这些 **fault** 的服务例程无法执行，它们就会成为“硬伤”——上访 (escalation) 成硬 **fault**。另外，在取向量 (异常处理时对异常向量表的读取) 时产生的总线 **fault** 也按硬 **fault** 处理。在 NVIC 中有一个硬 **fault** 状态寄存器 (HFSR)，它指出产生硬 **fault** 的原因。如果不是由于取向量造成的，则硬 **fault** 服务例程必须检查其它的 **fault** 状态寄存器，以最终决定是谁上访的。

HFSR 的定义如表 7.11 所示。

表 7.11 硬 **fault** 状态寄存器 (地址：0xE000_ED2C)

位段	名称	类型	复位值	描述
31	DEBUGEVT	R/Wc	0	硬 fault 因调试事件而产生
30	FORCED	R/Wc	0	硬 fault 是被上访的。上访者可以是总线 fault 、存储器管理 fault 或是用法 fault
29:2	-	-	-	-
1	VECTBL	R/Wc	0	硬 fault 是在取向量时发生的
0	-	-	-	-

7.5.5 应对 **faults**

在软件开发过程中，我们可以根据各种 **fault** 状态寄存器的值来判定程序错误，并且改正它们。附录 E 给出了各种 **faults** 的常见诱因，以及应对攻略。

然而，在一个实时系统中，情况则大不相同。发生了 **Faults** 后，如果不加以处理常会危及系统的运行。因此在找出了导致 **fault** 的原因后，软件必须决定下一步该怎么办。如果系统中运行了一个 RTOS，通常是终结肇事的任务。在其它情况，系统也许必须要复位。在不同的目标应用中，对 **fault** 恢复的要求也不同。总的来说，采取适当的策略有利于软件更健壮——当然最好还是防患于未然。下面就给出一些应付 **fault** 的常用方法。

复位。这也是最后一招。通过设置 NVIC “应用程序中断及复位控制寄存器” 中的 VECTRESET 位，将只复位处理器内核而不复位其它片上设施。取决于芯片的复位设计，有些 CM3 芯片可以使用该寄存器的 SYSRESETREQ 位来复位。这种只限于内核中的复位不会殃及其它的系统部件。

恢复：在一些场合下，还是有希望解决产生 **fault** 的问题的。例如，如果程序尝试访问了协处理器，可以通过一个协处理器的软件模拟器来解决此问题——当然是以牺牲性能为代价的，要不然还要硬件加速干啥。

中止相关任务：如果系统运行了一个 RTOS，则相关的任务可以被终结或者重新开始。

各个 **fault** 状态寄存器(FSRs)都保持住它们的状态，直到手工清除。Fault 服务例程在处理了相应的 **fault** 后不要忘记清除这些状态，否则如果下次又有新的 **fault** 发生，服务例程在检视 **fault** 源时，又将看到早先已经处理的 **fault** 遗留下来的状态标志。此时，将无法判断哪个 **fault** 是新发生的。FSRs 采用一个写时清除机制(写 1 时清除)。

芯片厂商也可以再添加自己的 FSR，以表示其它 **fault** 情况。

7.6 SVC 和 PendSV

注意：阅读本节的后面需要一点点多任务编程的基础知识——译者注

SVC (系统服务调用，亦简称系统调用) 和 PendSV (可悬起系统调用)，它们多用在上了操作系统的软件开发中。SVC 用于产生系统函数的调用请求。例如，操作系统通常不让用户程序直接访问硬件，而是通过提供一些系统服务函数，让用户程序使用 SVC 发出对系统服务函数的呼叫请求，以这种方法调用它们来间接访问硬件。因此，当用户程序想要控制特定的硬件时，它就要产生一个 SVC 异常，然后操作系统提供的 SVC 异常服务例程得到执行，它再调用相关的操作系统函数，后者完成用户程序请求的服务。

这种“提出要求——得到满足”的方式，很好、很强大、很方便、很灵活、很能可持续发展。首先，它使用户程序从控制硬件的繁文缛节中解脱出来，而是由 OS 负责控制具体的硬件。第二，OS 的代码可以经过充分的测试，从而能使系统更加健壮和可靠。第三，它使用户程序无需在特权级下执行，用户程序无需承担因误操作而瘫痪整个系统的风险。第四，通过 SVC 的机制，还让用户程

序变得与硬件无关，因此在开发应用程序时无需了解硬件的操作细节，从而简化了开发的难度和繁琐度，并且使应用程序跨硬件平台移植成为可能。开发应用程序唯一需要知道的就是操作系统提供的应用编程接口（API），并且在了解了各个请求代号和参数表后，就可以使用 **SVC** 来提出要求了（事实上，为使用方便，操作系统往往会提供一层封皮，以使系统调用的形式看起来和普通的函数调用一致。各封皮函数会正确使用 **SVC** 指令来执行系统调用——译者注）。其实，严格地讲，操作硬件的工作是由设备驱动程序完成的，只是对应用程序来说，它们也相当于操作系统的一部分。如图 7.14 所示

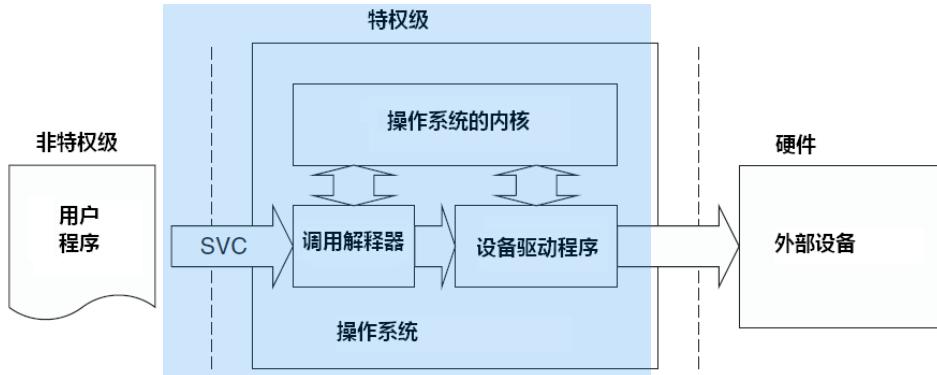


图 7.14 SVC 作为操作系统函数门户示意图

SVC 异常通过执行”**SVC**”指令来产生。该指令需要一个立即数，充当系统调用代号。**SVC** 异常服务例程稍后会提取出此代号，从而获知本次调用的具体要求，再调用相应的服务函数。例如，

SVC 0x3 ; 调用 3 号系统服务

在 **SVC** 服务例程执行后，上次执行的 **SVC** 指令地址可以根据自动入栈的返回地址计算出。找到了 **SVC** 指令后，就可以读取该 **SVC** 指令的机器码，从机器码中萃取出立即数，就获知了请求执行的功能代号。如果用户程序使用的是 **PSP**，服务例程还需要先执行 **MRS Rn, PSP** 指令来获取应用程序的堆栈指针。通过分析 **LR** 的值，可以获知在 **SVC** 指令执行时，正在使用哪个堆栈（细节在第 8 章中讨论）。

SVC vs. SWI

如果你曾使用过其它的 ARM 处理器（如 ARM7），你也许会知道那里有一个被称为“软件中断”的指令（**SWI**）。**SVC** 的地位与 **SWI** 是相同的——而且连机器码都相同。然而，因为在 **CM3** 中，异常处理模型已经“洗心革面”了，就故意把该指令也重命名，以强调它是在新生的系统中使用的。并且让程序员在把 **ARM7** 代码移植到 **CM3** 时，能充分注意到这个本质的不同（至少必须得改名，每次改名时都得到警示）。

由 **CM3** 的中断优先级模型可知，我们不能在 **SVC** 服务例程中嵌套使用 **SVC** 指令（事实上这样做也没意义），因为同优先级的异常不能抢占自身。这种作法会产生一个用法 **fault**。同理，在 **NMI** 服务例程中也不得使用 **SVC**，否则将触发硬 **fault**。

另一个相关的异常是 **PendSV**（可悬起的系统调用），它和 **SVC** 协同使用。一方面，**SVC** 异常是必须在执行 **SVC** 指令后立即得到响应的（对于 **SVC** 异常来说，若因优先级不比当前正处理的高，或是其它原因使之无法立即响应，将上访成硬 **fault**——译者注），应用程序执行 **SVC** 时都是希望所需的请求立即得到响应。另一方面，**PendSV** 则不同，它是可以像普通的中断一样被悬起的（不像 **SVC** 那样会上访）。**OS** 可以利用它“缓期执行”一个异常——直到其它重要的任务完成后才执行动作。悬起 **PendSV** 的方法是：手工往 **NVIC** 的 **PendSV** 悬起寄存器中写 **1**。悬起后，如果优先级不够高，则将缓期等待执行。

PendSV 的典型使用场合是在上下文切换时（在不同任务之间切换）。例如，一个系统中有两个

就绪的任务，上下文切换被触发的场合可以是：

- 执行一个系统调用
- 系统滴答定时器（SYSTICK）中断，（轮转调度中需要）

让我们举个简单的例子来辅助理解。假设有这么一个系统，里面有两个就绪的任务，并且通过 SysTick 异常启动上下文切换。如图 7.15 所示。

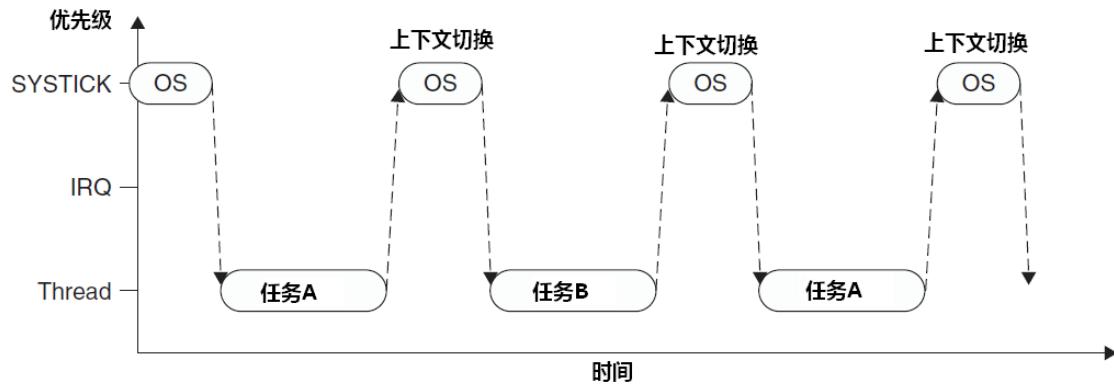


图 7.15 两个任务间通过 SysTick 进行轮转调度的简单模式

上图是两个任务轮转调度的示意图。但若在产生 SysTick 异常时正在响应一个中断，则 SysTick 异常会抢占其 ISR。在这种情况下，OS 是不能执行上下文切换的，否则将使中断请求被延迟，而且在真实系统中延迟时间还往往不可预知——任何有一丁点实时要求的系统都决不能容忍这种事。因此，在 CM3 中也是严禁没商量——如果 OS 在某中断活跃时尝试切入线程模式，将触发用法 fault 异常。

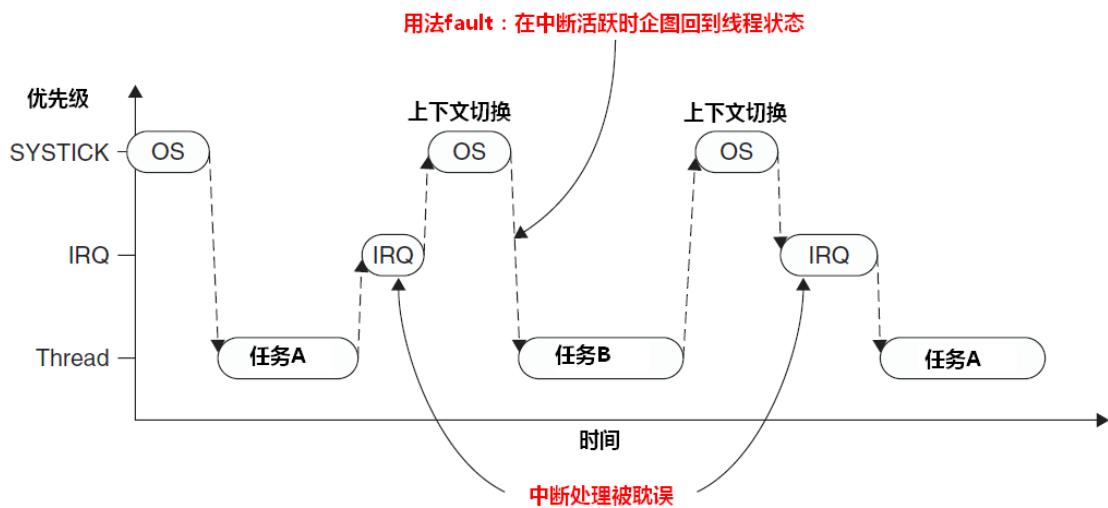


图 7.16 发生 IRQ 时上下文切换的问题

为解决此问题，早期的 OS 大多会检测当前是否有中断在活跃中，只有在无任何中断需要响应时，才执行上下文切换（切换期间无法响应中断）。然而，这种方法的弊端在于，它可以把任务切换动作拖延很久（因为如果抢占了 IRQ，则本次 SysTick 在执行后不得作上下文切换，只能等待下一次 SysTick 异常），尤其是当某中断源的频率和 SysTick 异常的频率比较接近时，会发生“共振”，使上下文切换迟迟不能进行。

现在好了，PendSV 来完美解决这个问题了。PendSV 异常会自动延迟上下文切换的请求，直到其它的 ISR 都完成了处理后才放行。为实现这个机制，需要把 PendSV 编程为最低优先级的异常。

如果 OS 检测到某 IRQ 正在活动并且被 SysTick 抢占，它将悬起一个 PendSV 异常，以便缓期执行上下文切换。如图 7.17 所示

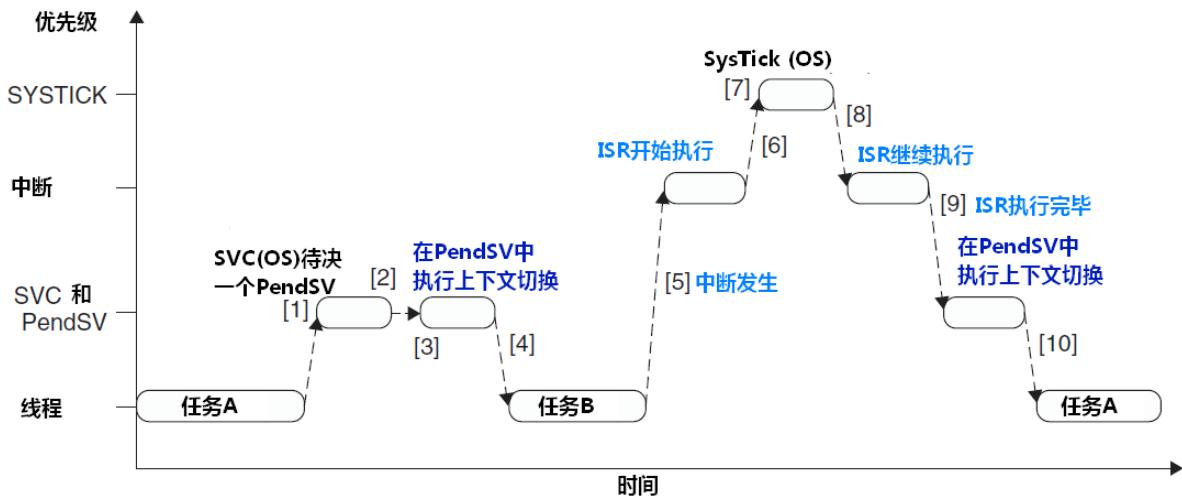


图 7.17 使用 PendSV 控制上下文切换

整个事件的流水账记录如下：

1. 任务 A 呼叫 SVC 来请求任务切换（例如，等待某些工作完成）
2. OS 接收到请求，做好上下文切换的准备，并且悬起一个 PendSV 异常。
3. 当 CPU 退出 SVC 后，它立即进入 PendSV，从而执行上下文切换。
4. 当 PendSV 执行完毕后，将返回到任务 B，同时进入线程模式。
5. 发生了一个中断，并且中断服务程序开始执行
6. 在 ISR 执行过程中，发生 SysTick 异常，并且抢占了该 ISR。
7. OS 执行必要的操作，然后悬起 PendSV 异常以作好上下文切换的准备。
8. 当 SysTick 退出后，回到先前被抢占的 ISR 中，ISR 继续执行
9. ISR 执行完毕并退出后，PendSV 服务例程开始执行，并且在里面执行上下文切换
10. 当 PendSV 执行完毕后，回到任务 A，同时系统再次进入线程模式。

NVIC 与中断控制

- NVIC 概览
- 中断配置基础
- 中断使能与除能
- 中断的悬起与解悬
- 中断建立全过程的演示
- 软件中断
- 再论 SysTick 定时器

8.1 NVIC 概览

正如前文已经多次提到的，向量中断控制器，简称 **NVIC**，是 Cortex-M3 不可分离的一部分，它与 CM3 内核的逻辑紧密耦合，有一部分甚至水乳交融在一起。**NVIC** 与 CM3 内核同声相应，同气相求，相辅相成，里应外合，共同完成对中断的响应。**NVIC** 的寄存器以存储器映射的方式访问，除了包含控制寄存器和中断处理的控制逻辑之外，**NVIC** 还包含了 MPU、SysTick 定时器以及调试控制相关的寄存器。本章中，我们将体检 **NVIC** 的中断处理控制逻辑。MPU 与调试控制逻辑在后续章节中讨论。

NVIC 共支持 1 至 240 个外部中断输入（通常外部中断写作 **IRQs**）。具体的数值由芯片厂商在设计芯片时决定。此外，**NVIC** 还支持一个“永垂不朽”的不可屏蔽中断（**NMI**）输入。**NMI** 的实际功能亦由芯片制造商决定。在某些情况下，**NMI** 无法由外部中断源控制。

NVIC 的访问地址是 **0xE000_E000**。所有 **NVIC** 的中断控制/状态寄存器都只能在特权级下访问。不过有一个例外——软件触发中断寄存器可以在用户级下访问以产生软件中断。所有的中断控制 / 状态寄存器均可按字 / 半字 / 字节的方式访问。此外，还有几个中断掩蔽寄存器也与中断控制密切相关，它们是第三章中讲到的“特殊功能寄存器”，只能通过 **MRS/MSR** 及 **CPS** 来访问。

8.2 中断配置基础

每个外部中断都在 **NVIC** 的下列寄存器中“挂号”：

- 使能与除能寄存器
- 悬起与“解悬”寄存器
- 优先级寄存器
- 活动状态寄存器

另外，下列寄存器也对中断处理有重大影响

- 异常掩蔽寄存器（**PRIMASK**, **FAULTMASK** 以及 **BASEPRI**）
- 向量表偏移量寄存器
- 软件触发中断寄存器
- 优先级分组位段

8.3 中断的使能与除能

中断的使能与除能分别使用各自的寄存器来控制——这与传统的，使用单一比特的两个状态来表达使能与除能是不同的。CM3 中可以有 240 对使能位 / 除能位 (SETENA 位/CLRENA 位)，每个中断拥有一对。这 240 个对子分布在 8 对 32 位寄存器中 (最后一对没有用完)。欲使能一个中断，我们需要写 1 到对应 SETENA 的位中；欲除能一个中断，你需要写 1 到对应的 CLRENA 位中。如果往它们中写 0，则不会有任何效果。写零无效是个很关键的设计理念：通过这种方式，使能 / 除能中断时只需把“当事位”写成 1，其它的位可以全部为零。再也不用像以前那样，害怕有些位被写入 0 而破坏其对应的中断设置 (反正现在写 0 没有效果了)，从而实现每个中断都可以自顾地设置，而互不侵犯——只需单一的写指令，不再需要读-改-写三步曲。

如上所述，SETENA 位和 CLRENA 位可以有 240 对，对应的 32 位寄存器可以有 8 对，因此使用数字后缀来区分这些寄存器，如 SETENA0, SETENA1...SETENA7，如表 8.1 所示。但是在特定的芯片中，只有该芯片实现的中断，其对应的位才有意义。因此，如果某个芯片支持 32 个中断，则只有 SETENA0/CLRENA0 才需要使用。SETENA/CLRENA 可以按字/半字/字节的方式来访问。又因为前 16 个异常已经分配给系统异常，故而中断 0 的异常号是 16，(回顾第 7 章中的表 7.2)

表 8.1 SETENA/CLRENA 寄存器族 (此表参考官方技术参考手册作了些改编——译者注)

SETENAs: xE000_E100 – 0xE000_E11C ; CLRENAAs: 0xE000_E180 – 0xE000_E19C

名称	类型	地址	复位值	描述
SETENA0	R/W	0xE000_E100	0	中断 0-31 的使能寄存器，共 32 个使能位 位[n]，中断#n 使能 (异常号 16+n)
SETENA1	R/W	0xE000_E104	0	中断 32-63 的使能寄存器，共 32 个使能位
...
SETENA7	R/W	0xE000_E11C	0	中断 224-239 的使能寄存器，共 16 个使能位
CLRENA0	R/W	0xE000_E180	0	中断 0-31 的除能寄存器，共 32 个除能位 位[n]，中断#n 除能 (异常号 16+n)
CLRENA1	R/W	0xE000_E184	0	中断 32-63 的除能寄存器，共 32 个除能位
...
CLRENA7	R/W	0xE000_E19C	0	中断 224-239 的除能寄存器，共 16 个除能位

8.4 中断的悬起与解悬

如果中断发生时，正在处理同级或高优先级异常，或者被掩蔽，则中断不能立即得到响应。此时中断被悬起。中断的悬起状态可以通过“中断设置悬起寄存器 (SETPEND)”和“中断悬起清除寄存器 (CLRPEND)”来读取，还可以写它们来手工悬起中断。

悬起寄存器和“解悬”寄存器也可以有 8 对，其用法和用量都与前面介绍的使能/除能寄存器

完全相同，见表 8.2。

表 8.2 SETPEND/CLRPEND 寄存器族 (此表参考官方技术参考手册作了些改编——译者注)

SETPENDs:0xE000_E200 – 0xE000_E21C ; CLRPENDs:0xE000_E280 - 0xE000_E29C

名称	类型	地址	复位值	描述
SETPEND0	R/W	0xE000_E200	0	中断 0-31 的悬起寄存器，共 32 个悬起位 位[n]，中断#n 悬起 (异常号 16+n)
SETPEND1	R/W	0xE000_E204	0	中断 32-63 的悬起寄存器，共 32 个悬起位
...
SETPEND7	R/W	0xE000_E21C	0	中断 224-239 的悬起寄存器，共 16 个悬起位
CLRPEND0	R/W	0xE000_E280	0	中断 0-31 的解悬寄存器，共 32 个解悬位 位[n]，中断#n 解悬 (异常号 16+n)
CLRPEND1	R/W	0xE000_E284	0	中断 32-63 的解悬寄存器，共 32 个解悬位
...
CLRPEND7	R/W	0xE000_E29C	0	中断 224-239 的解悬寄存器，共 16 个解悬位

8.4.1 优先级

每个外部中断都有一个对应的优先级寄存器，每个寄存器占用 8 位，但是 CM3 允许在最“粗线条”的情况下，只使用最高 3 位。4 个相邻的优先级寄存器拼成一个 32 位寄存器。如前所述，根据优先级组的设置，优先级可以被分为高低两个位段，分别是抢占优先级和亚优先级。优先级寄存器都可以按字节访问，当然也可以按半字/字来访问。有意义的优先级寄存器数目由芯片厂商实现的中断数目决定，优先级配置寄存器的详细信息在附录 D 中给出（表 D.18）。

表 8.3 中断优先级寄存器阵列 0xE000_E400 – 0xE000_E4EF

名称	类型	地址	复位值	描述
PRI_0	R/W	0xE000_E400	0 (8 位)	外中断#0 的优先级
PRI_1	R/W	0xE000_E401	0 (8 位)	外中断#1 的优先级
...
PRI_239	R/W	0xE000_E4EF	0 (8 位)	外中断#239 的优先级

表8.3B 系统异常优先级寄存器阵列 0xE000_ED18 - 0xE000_ED23

地址	名称	类型	复位值	描述
0xE000_ED18	PRI_4			存储器管理 <code>fault</code> 的优先级
0xE000_ED19	PRI_5			总线 <code>fault</code> 的优先级
0xE000_ED1A	PRI_6			用法 <code>fault</code> 的优先级
0xE000_ED1B	-	-	-	-
0xE000_ED1C	-	-	-	-
0xE000_ED1D	-	-	-	-
0xE000_ED1E	-	-	-	-
0xE000_ED1F	PRI_11			SVC 优先级
0xE000_ED20	PRI_12			调试监视器的优先级
0xE000_ED21	-	-	-	-
0xE000_ED22	PRI_14			PendSV 的优先级
0xE000_ED23	PRI_15			SysTick 的优先级

8.4.2 活动状态

每个外部中断都有一个活动状态位。在处理器执行了其 `ISR` 的第一条指令后，它的活动位就被置 1，并且直到 `ISR` 返回时才硬件清零。由于支持嵌套，允许高优先级异常抢占某个 `ISR`。然而，哪怕中断被抢占，其活动状态也依然为 1（请仔细琢磨前文讲到的“直到 `ISR` 返回时才清零”）。活动状态寄存器的定义，与前面讲的使能/除能和悬起/解悬寄存器相同，只是不再成对出现。它们也能按字/半字/字节访问，但他们是只读的，如表 8.4 所示。

表 8.4 ACTIVE 寄存器族 0xE000_E300-0xE000_E31C (此表参考官方技术参考手册作了些改编——译者注)

名称	类型	地址	复位值	描述
ACTIVE0	RO	0xE000_E300	0	中断 0-31 的活动状态寄存器，共 32 个状态位 位[n]，中断 #n 活动状态（异常号 16+n）
ACTIVE1	RO	0xE000_E304	0	中断 32-63 的活动状态寄存器，共 32 个状态位
...

ACTIVE7	RO	0xE000_E31C	0	中断 224-239 的活动状态寄存器，共 16 个状态位
---------	----	-------------	---	-------------------------------

8.4.3 特殊功能寄存器 PRIMASK 与 FAULTMASK

PRIMASK 用于除能在 NMI 和硬 fault 之外的所有异常，它有效地把当前优先级改为 0 (可编程优先级中的最高优先级)。该寄存器可以通过 MRS 和 MSR 以下例方式访问：

1. 关中断

```
MOV    R0,      #1
MSR    PRIMASK, R0
```

2. 开中断

```
MOV    R0,      #0
MSR    PRIMASK, R0
```

此外，还可以通过 CPS 指令快速完成上述功能：

```
CPSID  i      ;关中断
CPSIE  i      ;开中断
```

FAULTMASK 更绝，它把当前优先级改为 -1。这么一来，连硬 fault 都被掩蔽了。使用方案与 PRIMASK 的相似。但要注意的是，FAULTMASK 会在异常退出时自动清零。

掩蔽寄存器虽然能一手遮天，却都动不了 NMI，因为 NMI 是用在最危急的情况下。因此系统为它开出单行道，无需挂号只是不要迟到。当 NMI 激活时，“谁都是省略号，唯独是你不得了，第一优先谁比你重要”！试想，如果 NMI 被连接到系统的掉电报警线上，且系统是体外循环机的电源管理器……如果因为中断被除能就视而不见，则会使体外循环机因断电而失能，体外循环序列可以被意外终止，病人的生命也将丢失。

8.4.4 BASEPRI 寄存器

在更精巧的设计中，需要对中断掩蔽进行更细腻的控制——只掩蔽优先级低于某一阈值的中断——它们的优先级在数字上大于等于某个数。那么这个数存储在哪里？就存储在 BASEPRI 中。不过，如果往 BASEPRI 中写 0，则另当别论——BASEPRI 将停止掩蔽任何中断。例如，如果我们需要掩蔽所有优先级不高于 0x60 的中断，则可以如下编程：

```
MOV    R0,      #0x60
MSR    BASEPRI, R0
```

如果需要取消 BASEPRI 对中断的掩蔽，则示例代码如下：

```
MOV    R0,      #0
MSR    BASEPRI, R0
```

另外，我们还可以使用 BASEPRI_MAX 这个名字来访问 BASEPRI 寄存器，它俩其实是同一个寄存器。但是当我们使用这个名字时，会使用一个条件写操作。个中原因如下：尽管它俩在硬件水平上是同一个寄存器，但是生成的机器码不一样，从而硬件的行为也不同：使用 BASEPRI 时，可以任意设置新的优先级阈值；但是使用 BASEPRI_MAX 时则“许进不许出”——只允许新的优先级阈值比原来的那个在数值上更小，也就是说，只能一次次地扩大掩蔽范围，反之则不行。就好像绳子打了死结，只会越拉越紧。举例来说，检视下面的程序片断：

```
MOV    R0,      #0x60
MSR    BASEPRI_MAX, R0      ;掩蔽优先级不高于 0x60 的中断
MOV    R0,      #0x10
```

```

MSR    BASEPRI_MAX, R0      ;本次设置被忽略, 因为 0xf0 比 0x60 的优先级低
MOV    R0,      #0x40
MSR    BASEPRI_MAX, R0      ;Ok。扩大掩蔽范围到优先级不高于 0x40 的中断

```

为了把掩蔽阈值降低, 或者解除掩蔽, 需要使用“**BASEPRI**”这个名字。上例中, 把设置阈值为0xf0的那条指令改用**BASEPRI**, 则可以操作成功。显然, 在用户级下是不得更改**BASEPRI**寄存器的。与其它和优先级有关的寄存器一样, 系统中表达优先级的位数, 也同样影响**BASEPRI**中有意义的位数。如果系统中只使用3个位来表达优先级, 则**BASEPRI**有意义的值仅为0x00, 0x20, 0x40, 0x60, 0x80, 0xA0, 0xC0以及0xE0。

8.4.5 其它异常的配置寄存器

用法**fault**, 总线**fault**以及存储器管理**fault**都是特殊的异常, 因此给它们开了小灶。其中, 它们的使能控制是通过“系统Handler控制及状态寄存器(SHCSR)”(地址: 0xE000_ED24)来实现的。各种**faults**的悬起状态和大多数系统异常的活动状态也都在该寄存器中, 如表8.5所示。

表8.5 系统Handler控制及状态寄存器SHCSR (地址: 0xE000_ED24)

位段	名称	类型	复位值	描述
18	USGFAULTENA	R/W	0	用法 fault 服务例程使能位
17	BUSFAULTENA	R/W	0	总线 fault 服务例程使能位
16	MEMFAULTENA	R/W	0	存储器管理 fault 服务例程使能位
15	SVCALLPENDED	R/W	0	SVC 悬起中。本来已经要 SVC 服务例程, 但是却被更高优先级异常取代
14	BUSFAULTPENDED	R/W	0	总线 fault 悬起中, 细节同上。
13	MEMFAULTPENDED	R/W	0	存储器管理 fault 悬起中, 细节同上
12	USGFAULTPENDED	R/W	0	用法 fault 悬起中, 细节同上
11	SYSTICKACT	R/W	0	SysTick 异常活动中
10	PENDSVACT	R/W	0	PendSV 异常活动中
9	-	-	-	-
8	MONITORACT	R/W	0	Monitor 异常活动中
7	SVCALLACT	R/W	0	SVC 异常活动中
6:4	-	-	-	-
3	USGFAULTACT	R/W	0	用法 fault 异常活动中
2	-	-	-	-
1	BUSFAULTACT	R/W	0	总线 fault 异常活动中
0	MEMFAULTACT	R/W	0	存储器管理 fault 异常活动中

写这些寄存器时要小心, 必须确保对活动位的修改是经过深思熟虑的, 决不能粗心修改。否则, 如果某个异常的活动位被意外地清零了, 其服务例程却不知晓, 仍然执行异常返回指令, 那么CM3将视之为无理取闹——在异常服务例程以外做异常返回, 从而产生一个**fault**。

译注: 下段文字改编自《Cortex-M3 Technical Reference Manual》, pg8-29, 是给那些骨灰级玩家们看的, 因为修改这些位还有更深层次的背景和特效。译文为: 上表中的活动位虽然也是可写的, 但是改动时必须予以极度的小心, 否则这是玩火行为——设置或者清零这些位, 会改变处理器中对异常活动的记录, 却不会对应地修复堆栈中的数据(不会为了此改动而特意执行一次自动入栈或自动出栈操作), 于是埋下了破坏堆栈内容而引起程序跑飞的隐患; 另外, 其它一些重要的数据结构也得不到清除, 后患无穷。事实上, 只有操作系统在特殊场合下才会修改它们。例如: 在

任务执行系统调用的过程中执行上下文切换 (大幅提升实时性), 或者在使用软件模拟未定义指令的功能期间 (在用法fault服务例程中), 以及软件模拟协处理器的功能期间, 执行上下文切换, 同样大幅提升实时性。

下面开始讲中断控制及状态寄存器ICSR。对于NMI、SysTick定时器以及PendSV, 可以通过此寄存器手工悬起它们。另外, 在该寄存器中, 有好多位段都用于调试目的。在大多数情况下, 它们对于应用软件都没有什么用处, 只有悬起位对应用程序常常比较有参考价值, 如表8.6所示。

表8.5 中断控制及状态寄存器ICSR (地址 : 0xE000_ED04)

位段	名称	类型	复位值	描述
31	NMIPENDSET	R/W	0	写 1 以悬起 NMI。因为 NMI 的优先级最高且从不掩蔽, 在置位此位后将立即进入 NMI 服务例程。
28	PENDSVSET	R/W	0	写 1 以悬起 PendSV。读取它则返回 PendSV 的状态
27	PENDSVCLR	W	0	写 1 以清除 PendSV 悬起状态
26	PENDSTSET	R/W	0	写 1 以悬起 SysTick。读取它则返回 PendSV 的状态
25	PENDSTCLR	W	0	写 1 以清除 SysTick 悬起状态
23	ISRPREEMPT	R	0	=1 时, 则表示一个悬起的中断将在下一步时进入活动状态 (用于单步执行时的调试目的)
22	ISRPENDING	R	0	1=当前正有外部中断被悬起 (不包括 NMI)
21:12	VECTPENDING	R	0	悬起的 ISR 的编号。如果不止一个中断悬起, 则它的值是这次中断中, 优先级最高的那个。
11	RETTTOBASE	R	0	如果异常返回后将回到基级(base level), 并且没有其它异常悬起时, 此位为 1。若是在线程模式下, 在某个服务例程中, 有不止一级的异常处于活动状态, 或者在异常没有活动时执行了异常服务例程 (此时执行返回指令将产生 fault。此乃高危行为, 大虾也需慎用), 则此位为 0
9:0	VECTACTIVE	R	0	当前活动的ISR编号, 该位段指出当前运行中的ISR是哪个中断的 (提供异常序号), 包括NMI和硬fault。如果多个异常共享一个服务例程, 该例程可根据本位段的值来判定是哪一个异常的响应导致它的执行。把本位段的值减去16, 就得到了外中断的编号, 并可以用此编号来操作外中断相关的使能/除能等寄存器。

8.5 中断系统设置全过程的演示

下面给出一个简单的例子, 以演示如何建立一个外部中断。

1. 当系统启动后, 先设置优先级组寄存器。缺省情况下使用组0 (7位抢占优先级, 1位亚优先级)。
2. 如果需要重定位向量表, 先把硬fault和NMI服务例程的入口地址写到新表项所在的地址中。
3. 配置向量表偏移量寄存器, 使之指向新的向量表 (如果有重定位的话)
4. 为该中断建立中断向量。因为向量表可能已经重定位了, 保险起见需要先读取向量表偏移量寄存器的值, 再根据该中断在表中的位置, 计算出对应的表项, 再把服务例程的入口地址填写进

去。如果一直使用**ROM**中的向量表，则无需此步骤。

5. 为该中断设置优先级。

6. 使能该中断

示例汇编代码如下：

```

LDR    R0,    =0xE000ED0C    ; 应用程序中断及复位控制寄存器
LDR    R1,    =0x05FA0500    ; 使用优先级组 5 (2/6)
STR    R1,    [R0]          ; 设置优先级组
...
MOV    R4,    #8            ; ROM 中的向量表
LDR    R5,    =(NEW_VECT_TABLE+8)
LDMIA  R4!,  {R0-R1}       ; 读取 NMI 和硬 fault 的向量

STMIA R5!,  {R0-R1}       ; 拷贝它们的向量到新表中
...
LDR    R0,    =0xE000ED08    ; 向量表偏移量寄存器的地址
LDR    R1,    =NEW_VECT_TABLE
STR    R1,    [R0]          ; 把向量表重定位
...
LDR    R0,    =IRQ7_Handler ; 取得 IRQ #7 服务例程的入口地址
LDR    R1,    =0xE000ED08    ; 向量表偏移量寄存器的地址
LDR    R1,    [R1]
ADD    R1,    R1,#(4*(7+16)); 计算 IRQ #7 服务例程的入口地址

STR    R0,    [R1]          ; 在向量表中写入 IRQ #7 服务例程的入口地址
...
LDR    R0,    =0xE000E400    ; 外部中断优先级寄存器阵列的地址
MOV    R1,    #0xC0
STRB   R1,    [R0,#7]       ; 把 IRQ #7 的优先级设置为 0xC0
...
LDR    R0,    =0xE000E100    ; SETEN 寄存器的地址
MOV    R1,    #(1<<7)      ; 置位 IRQ #7 的使能位
STR    R1,    [R0]          ; 使能 IRQ #7

```

另外，如果优先级组的设置使得中断嵌套层次可以很深，则务请确认主堆栈的容量足够用。因为异常服务程序总是使用**MSP**，为安全起见，主堆栈的容量应是最大可能需求的量（嵌套最深时需要的量）。

如果应用程序储存在**ROM**中，并且不需要改变异常服务程序，则我们可以把整个向量表编码到**ROM**的起始区域（从**0**地址开始的那段）。在这种情况下，向量表的偏移量将一直为**0**，并且中断向量一直在**ROM**中，因此上例可以大大简化，只需**3步**：

1. 建立优先级组
2. 为该中断指定优先级
3. 使能该中断

如果在**I/O**密集型系统中，软件需要控制大量的硬件设备，则可能必须要考虑如下因素：

- 该芯片支持的中断数
- 该芯片中表达优先级的位数

在CM3的NVIC中，有一个名为“中断控制器类型寄存器”，它提供了该芯片中支持的中断数目，粒度是32的整数倍，（如表8.7所示）。如果你嫌它太粗枝大叶，也可以通过对每个SETENA位进行先写后读的测试，来获取支持的中断的精确数目（往各SETENA中写1，不支持的中断将永远读回0，求出第1个0的位置即可），亦可使用SETPEND等其它位来做此测试。这主要用于需要适应不同芯片的程序。如果已经确定使用固定的芯片，则无需多此一举。

表8.7 中断控制器类型寄存器ICTR (地址 : 0xE000_E004)

位段	名称	类型	复位值	描述
4:0	INTLINESUM	R	-	中断输入的数量，以 32 为粒度，如 0=1 至 32 1=33 至 64 2=65 至 96

为了判定正在使用的芯片使用了多少位来表达优先级，也可使用类似的方法：往某个优先级寄存器中写入0xFF，再读回来。则从MSB开始，有多少位是1就有多少位表达优先级。最少要使用3个位，此时你读回的是0xE0。

8.6 软件中断

软件中断，包括手工产生的普通中断，能以多种方式产生。最简单的就是使用相应的SETPEND寄存器；而更专业更快捷的作法，则是通过使用软件触发中断寄存器STIR，如表8.8所示。

表8.8 软件触发中断寄存器STIR (地址 : 0xE000_EF00)

位段	名称	类型	复位值	描述
8:0	INTID	W	-	影响编号为 INTID 的外部中断，其悬起位被置位。例如，写入 8，则悬起 IRQ #8

注意：系统异常（NMI, faults, PendSV等），不能用此法悬起。而且缺省时根本不允许用户程序改动NVIC寄存器的值。如果确实需要，必须先在NVIC的配置和控制寄存器(0xE000_ED14)中，把比特1（USERSEMPEND）置位，才能允许用户级下访问NVIC的STIR。

8.7 SysTick 定时器

SysTick定时器被捆绑在NVIC中，用于产生SysTick异常（异常号：15）。在以前，操作系统还有所有使用了时基的系统，都必须一个硬件定时器来产生需要的“滴答”中断，作为整个系统的时基。滴答中断对操作系统尤其重要。例如，操作系统可以为多个任务许以不同数目的时间片，确保没有一个任务能霸占系统；或者把每个定时器周期的某个时间范围赐予特定的任务等，还有操作系统提供的各种定时功能，都与这个滴答定时器有关。因此，需要一个定时器来产生周期性的中断，

而且最好还让用户程序不能随意访问它的寄存器，以维持操作系统“心跳”的节律。

Cortex-M3 处理器内部包含了一个简单的定时器。因为所有的 CM3 芯片都带有这个定时器，软件在不同 CM3 器件间的移植工作就得以化简。该定时器的时钟源可以是内部时钟（FCLK，CM3 上的自由运行时钟），或者是外部时钟（CM3 处理器上的 STCLK 信号）。不过，STCLK 的具体来源则由芯片设计者决定，因此不同产品之间的时钟频率可能会大不相同。因此，需要检视芯片的器件手册来决定选择什么作为时钟源。

SysTick 定时器能产生中断，CM3 为它专门开出一个异常类型，并且在向量表中有它的一席之地。它使操作系统和其它系统软件在 CM3 器件间的移植变得简单多了，因为在所有 CM3 产品间，SysTick 的处理方式都是相同的。

有 4 个寄存器控制 SysTick 定时器，如表 8.9 至表 8.12 所示。

表 8.9 SysTick 控制及状态寄存器（地址：0xE000_E010）

位段	名称	类型	复位值	描述
16	COUNTFLAG	R	0	如果在上次读取本寄存器后，SysTick 已经计到了 0，则该位为 1。如果读取该位，该位将自动清零
2	CLKSOURCE	R/W	0	0=外部时钟源(STCLK) 1=内核时钟(FCLK)
1	TICKINT	R/W	0	1=SysTick 倒数计数到 0 时产生 SysTick 异常请求 0=数到 0 时无动作
0	ENABLE	R/W	0	SysTick 定时器的使能位

表 8.10 SysTick 重装载数值寄存器（地址：0xE000_E014）

位段	名称	类型	复位值	描述
23:0	RELOAD	R/W	0	当倒数计数至零时，将被重装载的值

表 8.11 SysTick 当前数值寄存器（地址：0xE000_E018）

位段	名称	类型	复位值	描述
23:0	CURRENT	R/Wc	0	读取时返回当前倒计数的值，写它则使之清零，同时还会清除在 SysTick 控制及状态寄存器中的 COUNTFLAG 标志

表 8.10 SysTick 校准数值寄存器（地址：0xE000_E01C）

位段	名称	类型	复位值	描述
31	NOREF	R	-	1=没有外部参考时钟（STCLK 不可用） 0=外部参考时钟可用
30	SKW	R	-	1=校准值不是准确的 10ms 0=校准值是准确的 10ms
23:0	TENMS	R/W	0	在 10ms 的间隔中倒计数的格数。芯片设计者应该通过 Cortex-M3 的输入信号提供该数值。若该值读回零，则表示无法使用校准功能

校准值寄存器提供了这样一个解决方案：它使系统即使在不同的CM3产品上运行，也能产生恒定的**SysTick**中断频率。最简单的作法就是：直接把**TENMS**的值写入重装载寄存器，这样一来，只要没突破系统的“弹性极限”，就能做到每10ms来一次 **SysTick**异常。如果需要其它的**SysTick**异常周期，则可以根据**TENMS**的值加以比例计算。只不过，在少数情况下，CM3芯片可能无法准确地提供**TENMS**的值（如，CM3的校准输入信号被拉低），所以为保险起见，最好在使用**TENMS**前检查器件的参考手册。

SysTick定时器除了能服务于操作系统之外，还能用于其它目的：如作为一个闹铃，用于测量时间等。要注意的是，当处理器在调试期间被喊停（**halt**）时，则**SysTick**定时器亦将暂停运作。

中断的具体行为

- 中断/异常的响应序列
- 异常返回
- 嵌套的中断
- 咬尾中断
- 晚到（的高优先级）中断
- 异常返回值
- 中断延迟
- 异常响应期间的 `faults`

译注：在本章中，如无特殊说明，不分辨“中断”与“异常”这两个术语，可以互换使用。

9.1 中断 / 异常的响应序列

当CM3开始响应一个中断时，会在它小小的体内奔涌起三股暗流：

- 入栈：把8个寄存器的值压入栈
- 取向量：从向量表中找出对应的服务程序入口地址
- 选择堆栈指针MSP/PSP，更新堆栈指针SP，更新连接寄存器LR，更新程序计数器PC

9.1.1 入栈

响应异常的第一个行动，就是自动保存现场的必要部分：依次把xPSR, PC, LR, R12以及R3-R0由硬件自动压入适当的堆栈中：如果当响应异常时，当前的代码正在使用PSP，则压入PSP，也就是使用进程堆栈；否则就压入MSP，使用主堆栈。一旦进入了服务例程，就将一直使用主堆栈。

假设入栈开始时，SP的值为N，则在入栈后，堆栈内部的变化如表9.1表示。又因为AHB接口上的流水线操作本性，地址和数据都在经过一个流水线周期之后才进入。另外，在自动入栈的过程中，把寄存器写入堆栈内存的时间顺序，并不是与写入的空间顺序相对应的。但是机器会保证：正确的寄存器将被保存到正确的位置，如图9.1和表9.1的第3列所示。

表9.1 入栈顺序以及入栈后堆栈中的内容

地址	寄存器	被保存的顺序
旧SP (N-0)	原先已压入的内容	-
(N-4)	xPSR	2
(N-8)	PC	1
(N-12)	LR	8
(N-16)	R12	7
(N-20)	R3	6
(N-24)	R2	5
(N-28)	R1	4
新SP (N-32)	R0	3

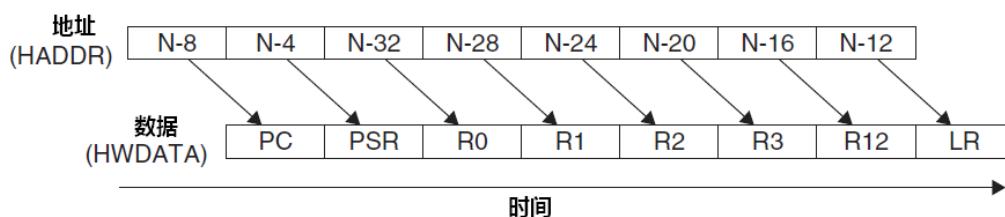


图9.1 内部入栈序列

Cortex-M3 r2p0修订版的区别

在r2p0中，自动打开了“双字对齐的堆栈工作模式”，简称双字对齐模式。在双字对齐的模式下，SP的值必须能被8整除。如果没有打开双字对齐，则与图9.1和表9.1所示的相同。但如果打开了双字对齐，可是SP却不能被8整除，则没有对齐的一个字被空出来，所有入栈寄存器的地址，依序减4。如：PC不再是N-8，而是N-12，其它寄存器亦如此。

上文所提到的这个自动压入的8字数据块，通常被称作“异常堆栈帧(exception stack frame)”。在CM3修订版2之前，缺省配置下的堆栈帧可以始于任何字对齐的地址。到了修订版2，则改为缺省配置下堆栈帧要双字对齐。之所以这样做，是为了满足AAPCS所规定的过程调用标准。这个功能其实在CM3的修订版1中就有了，只是缺省时没有打开。如欲在修订版1中开启此功能，需要手动在NVIC配置控制寄存器中置位STKALIGN位。当需要除能此特性时，也只需清除此位。关于该寄存器的更多细节，请参阅第12章（双字堆栈对齐节）。

CM3在看不见的内部“搅浑”了入栈的顺序，这是有深层次的原因的。先把PC与xPSR的值保存，就可以更早地启动服务例程指令的预取——因为这需要修改PC；同时，也做到了在早期就可以更新xPSR中IPSR位段的值。

细心的读者一定在猜测：为啥袒护R0-R3以及R12呢，R4-R11就是下等公民？原来，在ARM上，有一套的C函数调用标准约定（《C/C++ Procedure Call Standard for the ARM Architecture》，AAPCS，Ref5）。个中原因就在它上面：它使得中断服务例程能用C语言编写，编译器优先使用入栈了的寄存

器来保存中间结果（当然，如果程序过大也可能要用到R4-R11，此时编译器负责生成代码来push它们。但是，ISR应该短小精悍，不要让系统如此操心——译者注）。

如果读者再仔细看，会发现R0-R3, R12是最后被压进去的。这里也有一番良苦用心：为的是可以更容易地使用SP基址来索引寻址，（这也方便了LDM等多重加载指令。因为LDM必须加载地址连续的一串数据，而现在R0-R3, R12的存储地址连续了——译者注）。这种顺序也舒展了参数的传递过程：使之可以方便地通过读取入栈了的R0-R3取出（主要为系统软件所利用，多见于SVC与PendSV中的参数传递）。

9.1.2 取向量

当数据总线（系统总线）正在为入栈操作而忙得风风火火时，指令总线（I-Code总线）可不是凉快地坐那儿看热闹——它正在为响应中断紧张有序地执行另一项重要的任务：从向量表中找出正确的异常向量，然后在服务程序的入口处预取指。由此可以看到各自都有专用总线的好处：入栈与取指这两个工作能同时进行。

9.1.3 更新寄存器

在入栈和取向量操作完成之后，执行服务例程之前，还要更新一系列的寄存器：

- **SP**: 在入栈后会把堆栈指针（PSP或MSP）更新到新的位置。在执行服务例程时，将由MSP负责对堆栈的访问。
- **PSR**: 更新IPSR位段（地处PSR的最低部分）的值为新响应的异常编号。
- **PC**: 在取向量完成后，PC将指向服务例程的入口地址，
- **LR**: 在出入ISR的时候，LR的值将得到重新的诠释，这种特殊的值称为“**EXC_RETURN**”，在异常进入时由系统计算并赋给LR，并在异常返回时使用它。**EXC_RETURN**的二进制值除了最低4位外全为1，而其最低4位则有另外的含义（后面讲到，见表9.3和表9.4）。

以上是在响应异常时通用寄存器的变化。另一方面，在NVIC中，也会更新若干个相关有寄存器。例如，新响应异常的悬起位将被清除，同时其活动位将被置位。

9.2 异常返回

当异常服务例程执行完毕后，需要很正式地做一个“异常返回”动作序列，从而恢复先前的系统状态，才能使被中断的程序得以继续执行。从形式上看，有3种途径可以触发异常返回序列，如表9.2所示。而不管使用哪一种，都需要用到先前储到LR的EXC_RETURN。

表9.2 触发中断返回的指令

返回指令	工作原理
BX <reg>	当LR存储了EXC_RETURN时，使用BX LR即可返回
POP {PC}和 POP {...,PC}	在服务例程中，LR的值常常会被压入栈。此时即可使用POP指令把LR存储的EXC_RETURN往PC里弹，从而启动处理器的中断返回序列
LDR与LDM	把PC作为目的寄存器，亦可启动中断返回序列

有些处理器使用特殊的返回指令来标示中断返回，例如8051就使用reti。但是在CM3中，是通过把EXC_RETURN往PC里写来识别返回动作的。因此，可以使用上述的常规返回指令，从而为使用C语言编写服务例程扫清了最后的障碍（无需特殊的编译器命令，如_interrupt）。

在启动了中断返回序列后，下述的处理就将进行：

1. 出栈：先前压入栈中的寄存器在这里恢复。内部的出栈顺序与入栈时的相对应，堆栈指针的值也改回先前的值。
2. 更新NVIC寄存器：伴随着异常的返回，它的活动位也被硬件清除。对于外部中断，倘若中断输入再次被置为有效，悬起位也将再次置位，新一次的中断响应序列也可随之再次开始。

9.3 嵌套的中断

在CM3内核以及NVIC的深处，就已经内建了对中断嵌套的全力支持，根本无需使用汇编去写封皮代码(wrapper code)。事实上，我们要做的就只是为每个中断适当地建立优先级，不用再操心别的。表现在：

第一、NVIC和CM3处理器会根据优先级的设置来控制抢占与嵌套行为。因此，在某个异常正在响应时，所有优先级不高于它的异常都不能抢占之，而且它自己也不能抢占自己。

第二、有了自动入栈和出栈，就不用担心在中断发生嵌套时，会使寄存器的数据损毁，从而可以放心地执行服务例程。

然而，有一件事情却必须更加一丝不苟地处理了，否则有功能紊乱甚至死机的危险。这就是计算主堆栈容量的最小安全值。我们已经知道，所有服务例程都只使用主堆栈。所以当中断嵌套加深时，对主堆栈的压力会增大：每嵌套一级，就至少再需要8个字，即32字节的堆栈空间——而且这还没算上ISR对堆栈的额外需求，并且何时嵌套多少级也是不可预料的。如果主堆栈的容量本来就已经所剩无几了，中断嵌套又突然加深，则主堆栈有被用穿的凶险。这就好像已经表现出了高血压危象的时候，情绪又一激动，就容易中风一般。中风是一大杀手，而堆栈溢出同样是很致命的，它会使入栈数据与主堆栈前面的数据区发生混迭，使这些数据被破坏；若在服务例程返回前混迭区的数据又被更改了，则堆栈内容被破坏。这么一来在执行中断返回后，系统极可能功能紊乱，甚至当场被一击秒杀——程序跑飞/死机！

另一个要注意的，是相同的异常是不允许重入的。因为每个异常都有自己的优先级，并且在异常处理期间，同级或低优先级的异常是要阻塞的。因此对于同一个异常，只有在上次实例的服务例程执行完毕后，方可继续响应新的请求。由此可知，在SVC服务例程中，就不得再使用SVC指令，否则将fault伺候。

9.4 咬尾中断

CM3为缩短中断延迟做了很多努力，第一个要提的，就是新增的“咬尾中断”（Tail-Chaining）机制。

当处理器在响应某异常时，如果又发生其它异常，但它们优先级不够高，则被阻塞——这个我们已经知道。那么在当前的异常执行返回后，系统处理悬起的异常时，倘若还是先POP，然后又把POP出来的内容PUSH回去，这不成了砸锅炼铁再铸锅，白白浪费CPU时间吗，可知还有多少紧急的事件悬而未决呀！正因此，CM3不会傻乎乎地POP这些寄存器，而是继续使用上一个异常已经PUSH好的成果，消灭了这种铺张浪费。这么一来，看上去好像后一个异常把前一个的尾巴咬掉了，前前后后只执行了一次入栈 / 出栈操作。于是，这两个异常之间的“时间沟”变窄了很多，如图9.2所示。

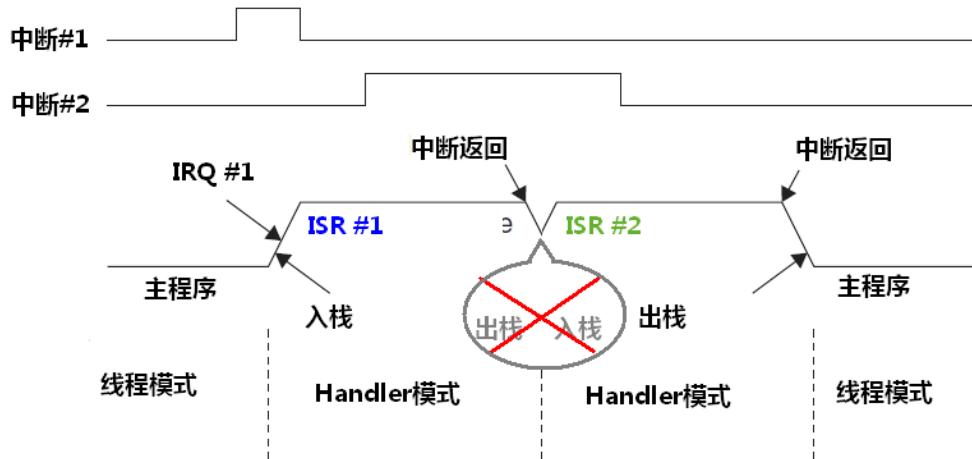


图9.2 异常咬尾示意图

为进一步帮助读者理解, 译者从另外文献上截取并改编下图:

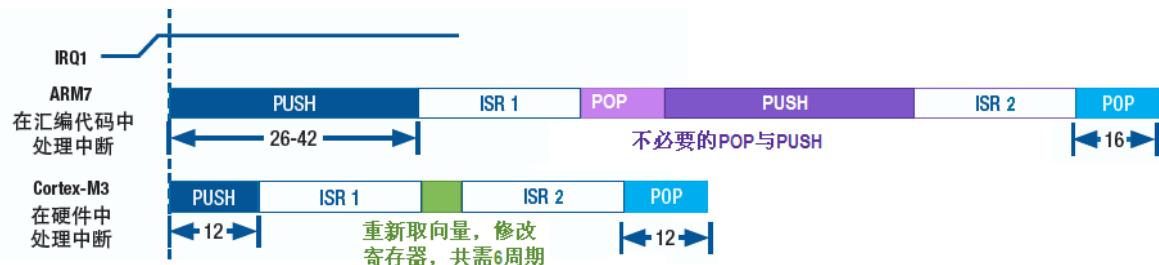


图9.2B 异常咬尾与常规处理的比较 (以ARM7TDMI为例)

9.5 晚到 (的高优先级) 异常

Cortex-M3的中断处理还有另一个机制, 它强调了优先级的作用, 这就是“晚到的异常处理”。当Cortex-M3对某异常的响应序列还处在早期: 入栈的阶段, 尚未执行其服务例程时, 如果此时收到了高优先级异常的请求, 则本次入栈就成了为高优先级中断所做的了——入栈后, 将执行高优先级异常的服务例程。可见, 它虽然来晚了, 却还是因优先级高而受到偏袒, 低优先级的异常为它“火中取栗”。

比如, 若在响应某低优先级异常#1的早期, 检测到了高优先级异常#2, 则只要#2没有太晚, 就能以“晚到中断”的方式处理——在入栈完毕后执行ISR #2, 如图9.3所示。如果异常#2来得太晚, 以至于已经执行了ISR #1的指令, 则按普通的抢占处理, 这会需要更多的处理器时间和额外32字节的堆栈空间。

在ISR #2执行完毕后, 则以刚刚讲过的“咬尾中断”方式, 来启动ISR #1的执行。

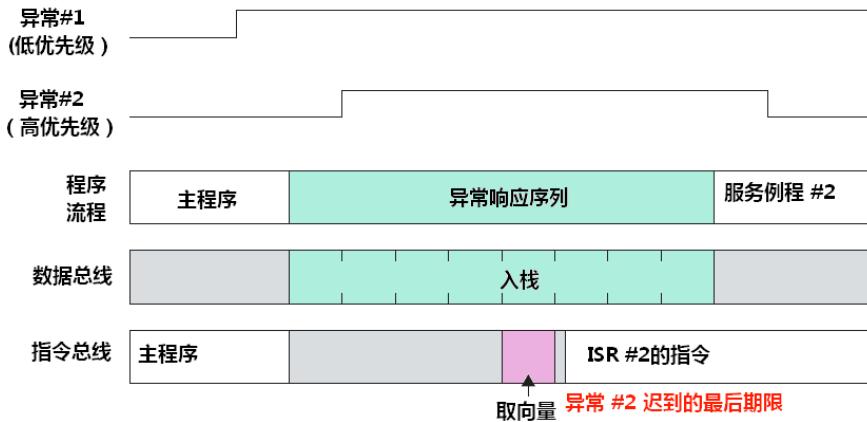


图9.3 晚到异常的处理模式图

9.6 异常返回值

前面已经讲到，在进入异常服务程序后，将自动更新LR的值为特殊的EXC_RETURN。这是一个高28位全为1的值，只有[3:0]的值有特殊含义，如表9.3所示。当异常服务例程把这个值送往PC时，就会启动处理器的中断返回序列。因为LR的值是由CM3自动设置的，所以只要没有特殊需求，就不要改动它。

表9.3 EXC_RETURN位段详解

位段	含义
[31:4]	EXC_RETURN的标识：必须全为1
3	0=返回后进入Handler模式 1=返回后进入线程模式
2	0=从主堆栈中做出栈操作，返回后使用MSP， 1=从进程堆栈中做出栈操作，返回后使用PSP
1	保留，必须为0
0	0=返回ARM状态。 1=返回Thumb状态。在CM3中必须为1

总结一下表9.3，可以得出，合法的EXC_RETURN值共3个，如表9.4所示

表9.4 合法的EXC_RETURN值及其功能

EXC_RETURN	功能
数值	
0xFFFF_FFF1	返回handler模式
0xFFFF_FFF9	返回线程模式，并使用主堆栈(SP=MSP)
0xFFFF_FFFD	返回线程模式，并使用线程堆栈(SP=PSP)

如果主程序在线程模式下运行，并且在使用MSP时被中断，则在服务例程中LR=0xFFFF_FFF9（主程序被打断前的LR已被自动入栈）。

如果主程序在线程模式下运行，并且在使用PSP时被中断，则在服务例程中LR=0xFFFF_FFFD（主

程序被打断前的LR已被自动入栈)。

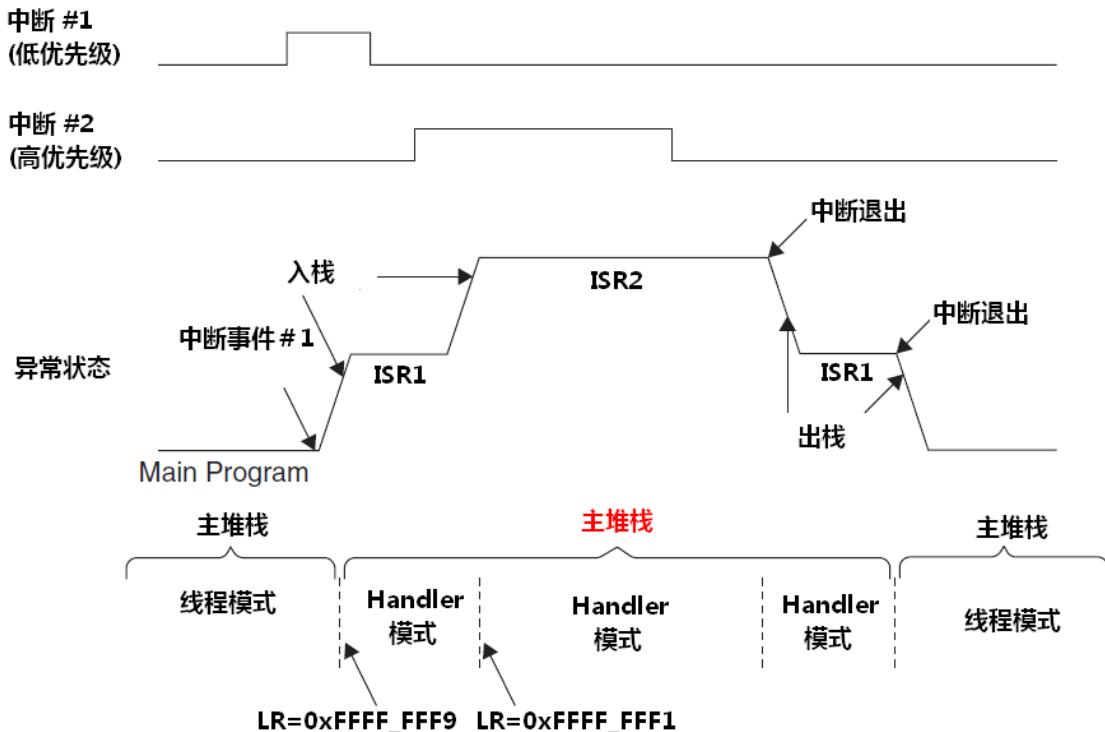


图9.4 LR的值在异常期间被设置为EXC_RETURN (线程模式使用主堆栈)

如果主程序在Handler模式下运行，则在服务例程中LR=0xFFFF_FFF1（主程序被打断前的LR已被自动入栈）。这时的所谓“主程序”，其实更可能是被抢占的服务例程。事实上，在嵌套时，更深层ISR所看到的LR总是0xFFFF_FFF1，如图9.5所示。

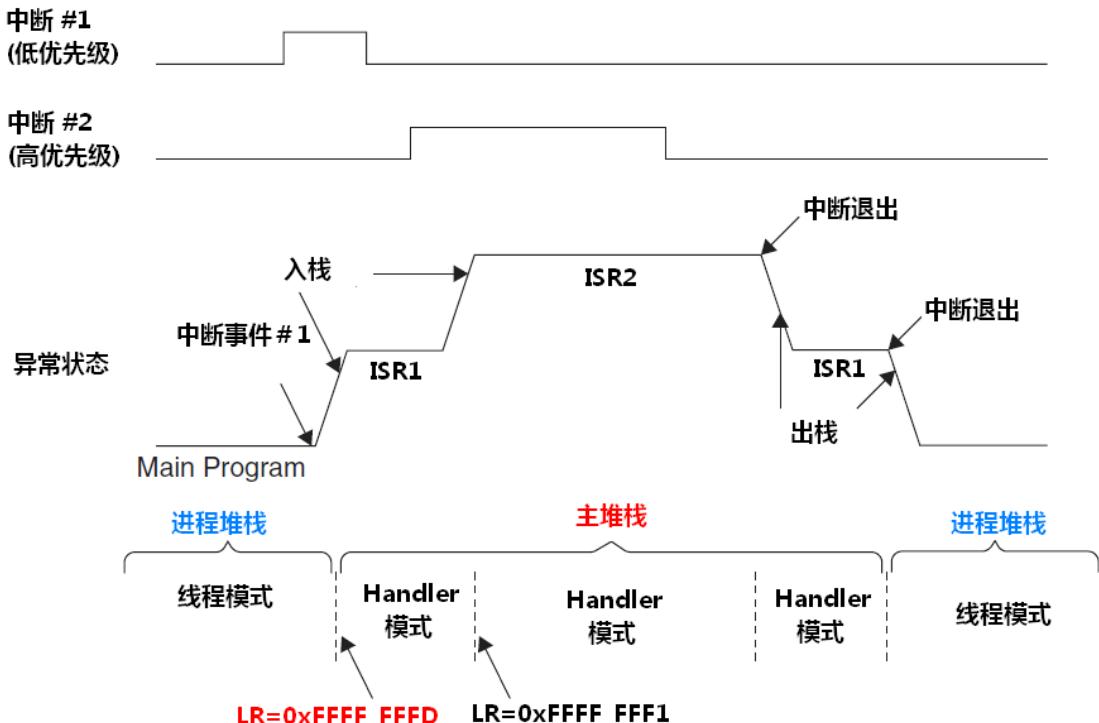


图9.5 LR的值在异常期间被设置为EXC_RETURN (线程模式使用进程堆栈)

由EXC_RETURN的格式可见，我们不能把0xFFFF_FFF0-0xFFFF_FFFF中的地址作为任何返回地址。其实也并不用担心会弄错，因为CM3已经把这个范围标记成“取指不可区”了。

9.7 中断延迟

在设计实时系统时，必须对中断延迟进行严肃和仔细地估算。在这里，中断延迟的定义是：从检测到某中断请求，到执行了其服务例程的第一条指令时，已经流逝了的时间。在CM3中，若存储器系统够快，且总线系统允许入栈与取指同时进行，同时该中断可以立即响应，则中断延迟是雷打不动的12周期（满足硬实时所要求的确定性）。在与时间赛跑的这12个周期里，处理器内部一直开足马力，进行了入栈、取向量、更新寄存器以及服务例程取指的一系列操作。但若存储器太慢以至于引入等待周期，或者还有其它因素，则会引入额外的延时——反正如果有拖后腿的，那绝不可能是CM3内核。

当处理咬尾中断时，省去了堆栈操作，因此切入新异常服务例程的耗时可以短至6周期。

有些指令需要较多的周期才能完成。它们是除法指令，双字传送指令LDRD/STRD以及多重数据传送指令(LDM/STM)。

对于前两者，CM3将为了保证中断及时响应而取消它们的执行，待返回后重新开始——这牺牲了一点性能，以及某些子程序的一点个人利益，但换来了对意外事件的更快救援。

对于LDM/STM，则有另外的处理方式。因为它们不照前两者那么浑然一体——它们其实是一串LDR/STR的速度优化版。于是，为了加速中断的响应，CM3支持LDM/STM指令的中止和继续，就好像它们只是普通的一串LDR/STR一样。为了实现“指令撕裂与粘合”的目的，需要记录中断时数据传送的进程。为此，CM3在xPSR中开出若干个“ICI位”，记录下一个即将传送的寄存器是哪一个(LDM/STM在汇编时，都把寄存器号升序排序)。在服务例程返回后，xPSR被弹出，CM3再从ICI bits中获取当时LDM/STM执行的进度，从而可以继续传送。

这个办法听起来是个好主意，只是在个别情况下还有一点限制：IF-THEN(IT)指令的执行也需要在xPSR中使用几个位，可它需要的位刚好与ICI位重合（类似C中的union）——both ICI bits和IT条件都记录在EPSR中。所以，如果在IF-THEN中使用了LDM/STM，则不再记录LDM/STM的执行进度。但尽管如此，及时响应中断依然是首要任务。此时只好把LDM/STM取消，待中断返回后继续执行
译注：仔细的读者可能会注意到，xPSR中有很多位空着没用，从而可能想不通，为啥要让“有人可怜没人爱，有人却忙不过来”。这可能是因为在其它款式中，这些位被用掉了，或者还有其它什么难言之隐。

另外，如果在总线接口上还有未完成的(outstanding)数据传送，例如有一个带缓冲的写操作未完成，处理器也只能等待此传送完成。这是迫不得已的——只有这样，才能保证在发生了总线fault时，其服务例程能够安全地抢占其它程序。

当多个中断同时请求时，也会发生中断延迟，这表现在只有优先级最高的得到立即响应，所有其它的中断将被延迟。另外，在中断嵌套时，每个中断都会阻塞同级和低优先级的中断。最后，如果中断被掩蔽（也就是俗称的关中，在多任务系统下满地都有），则在掩蔽期间也会附加中断延迟。

9.8 异常响应期间的 faults

Faults是运行时发生各种故障的表现，在中断响应期间的故障也不例外。中断响应的每一步骤都可以触发faults。

9.8.1 入栈期间

如果在入栈期间引起了总线fault，则本次入栈操作将被强行中止，并且把总线异常悬起或者在允许时立即响应。若除能了总线fault，则此次故障将成为“硬伤”——上访至硬fault。在总线fault被使能的情况下，如果它的优先级比正在响应的异常高，则抢占之，否则将悬起直到引起fault的异常执行完毕。这种情况被称为“入栈错误”(stacking error)，由总线fault状态寄存器(BFSR，地址: 0xE000_ED29)的STKERR位指示(位偏移: 4)。

如果入栈操作引起MPU访问违例，则产生存储管理fault，并且必须能立即执行MemFault服务例程，否则将无条件上访成硬fault。在发生入栈时访问违例时，存储管理fault寄存器(MFSR，地址: 0xE000_ED28)中的MSTKERR位(位偏移: 4)被置位，用于指示该fault。

入栈是自动完成的，因此不可能产生用法fault——译者。

9.8.2 出栈期间

如果在中断返回时的出栈期间引起了总线fault，则本次出栈操作将被强行中止，并且把总线异常悬起或立即响应。若除能了总线fault，则此次故障将成为“硬伤”——上访至硬fault。其它情况下，只要总线fault的优先级比当前的高(也包括比当前最深嵌套的优先级高)，则可以立即响应。这种情况称为“出栈错误”(unstacking error)，由BFSR.3指示(UNSTKERR位)。

类似地，如果是因MPU访问违例造成的MemManage fault，由MFSR.3(MUNSTKERR)指示。且MemManage fault的服务例程必须能立即执行，否则无条件硬fault。

9.8.3 取向量期间

在取向量期间发生总线fault，这是非常罕见的一种情况，这也是最严重的，因此直接上硬fault(MPU的限制则管不着取向量操作——译者注)。这种情况，由硬fault状态寄存器(HFSR，地址: 0xE000_ED2C)中的VECTTBL位(位偏移: 1)来指示。

9.8.4 无效返回时

如果LR中的EXC_RETURN不是合法的值(合法值见表9.4，包括企图返回ARM状态)，则引起用法fault。如果用法fault被除能，也上访成硬fault。此时，用法Fault状态寄存器(UFSR，地址: 0xE000_ED2A)中的INVPC位(位偏移: 2)，或者是INVSTATE位(位偏移: 1)置位。

Cortex-M3 的低层编程

- 概览
- 汇编与 C 的接口
- 典型的开发流程
- 第一步工作
- 与外界互动
- 使用数据存储器
- 使用互斥访问实现互斥锁操作
- 使用位带实现互斥锁操作
- 使用位段提取与查表跳转

10.1 概览

在 CM3 上编程，既可以使用 C 也可以使用汇编。可能还有其它语言的编译器，但是大多数人还是会在这两个世界里游弋。C 与汇编都“尺有所短，寸有所长”，不能互相取代。使用 C 能开发大型程序，而汇编则用于执行特种任务。

在使用不同的工具链和芯片时，有大量的用法和用量都随之不同。因此，本书不会深入讲解怎样精通一个具体的工具链，也不会大谈如何把程序烧到板子上。在第 19 章和第 20 章会提到一些入门知识，具体内容还需查阅相关的文献和在线帮助文档。

10.1.1 使用汇编

如果工程比较小，使用纯汇编常常是可行的，而且能使我们随心所欲地优化和控制程序。不过，这么一来的开发周期会变长。尤其是当工程变大，需要处理比较复杂的数据结构，以及要管理函数库时，汇编那狰狞的真面目就会渐显出来：各种地址和间接引用千头万绪；bug 劈头盖脸；甚至好几天都改不完，工作量激增，简直就是自虐。当然，如果你想成为系统开发的大虾，就必须以“我不下地狱谁下地狱”的决心，去勇敢面对，后天下乐而乐，百炼成钢。

不论如何，时间宝贵。我们应该以 C 来实现程序的大框架，而本着好钢用在刀刃上的原则来使用汇编，因为只有在不多的特殊场合是适合使用汇编，甚至是非使用汇编语言不可的，它们包括：

- 无法用 C 写成的函数，如操作特殊功能寄存器，以及实施互斥访问。
- 在危急关头执行处理的子程（如，NMI 服务例程）。
- 存储器极度受限，只有使用汇编才可能把程序或数据挤进去。
- 执行频率非常高的子程，如操作系统的调度程序。
- 与处理器体系结构相关的子程，如上下文切换。
- 对性能要求极高的应用，如防空炮的火控系统。

10.1.2 使用 C

用 C 写的程序可以移植，并且操作复杂数据结构时远远比汇编方便。但因为 C 是一种通用语言——

—至少是低等高级语言，它并不指定如何初始化具体的处理器（用于在 `main` 执行前准备好执行环境）。在解决这个问题时，不同的工具链都有自己的一套，因此最聪明的办法就是看一看工具链附带的示例程序。如果使用 **RealView** 开发套件（RVDS）或者 **KEIL** 的 **RealView** 微控制器开发套件（RVMDK），则编译器和汇编器是 **ARM** 提供的，而且它们中都附带了很多示例。如果使用了 **GNU** 的工具链，则第 19 章以 **CodeSourcery GNU** 工具链为例，给出一个简单的示例（其它示例可以去网上找）。

尽管在使用了 C 后，大大加速了开发，但是底层的系统控制往往还需要汇编代码。很多编译器都允许我们直接在 C 代码中插汇编，称为“内联汇编”；另外还允许我们写独立的汇编模块，与编译后的 C 模块一起连接。以往，使用内联汇编的作法比较多，但是在 ARM 编译器中，不支持对 **Thumb-2** 指令的内联汇编。取而代之的，是从 **RealView C** 编译器的 3.0 版开始，新增了所谓“嵌入式汇编”的功能，它支持 **Thumb-2** 指令。它让我们可以在 C 程序中插入使用汇编语言编写的函数，例如：

```
__asm void SetFaultMask(unsigned int new_value)
{
    //在这里使用汇编代码实现本函数
    MSR FAULTMASK, new_value // 把new_value写入FAULTMASK中
    BX LR // 返回主程序（不可省略）
}
```

RealView C 编译器对嵌入式汇编的详细论述，在《**RVCT 3.0 Compiler and Library Guide (Ref6)**》中给出。

在 **CM3** 中，嵌入式汇编还是比较需要的，因为常常会有访问特殊功能寄存器的时候。比如，在设置堆栈时，就要使用 **MRS/MSR** 指令。对于其它不能由编译器产生的指令，比如 **WFI/WFE**、互斥访问、存储器隔离等指令，也必须用汇编显式给出。

在以前的 ARM 处理器中，因为支持 ARM/Thumb 双重状态，往往需要所谓的“**interworking**”，且不同的源文件可能需要编译成不同状态下的代码。在 **CM3** 中不再有此需求，因为只使用了 **Thumb** 状态，从而工程管理清爽多了。

当使用 C 开发程序时，推荐开启 **CM3** 的双字对齐管理机制（在 **NVIC** 配置与控制寄存器中，把 **STKALIGN** 置位），代码形如：

```
#define NVIC_CCR ((volatile unsigned long *) (0xE000ED14))
*NVIC_CCR = *NVIC_CCR | 0x200; //设置STKALIGN位
```

这是用于确保系统能严格遵守 **AAPCS** 过程调用标准，个中细节请参阅第 12 章。

10.2 汇编与 C 的接口

在很多情况下，都需要让 C 程序模块与汇编程序模块互相交互，它们包括：

- 在 C 代码中使用了嵌入式汇编（或者是在 **GNU** 工具下，使用了内联汇编）
- C 程序呼叫了汇编程序，这些汇编程序是在独立的汇编源文件中实现的
- 汇编程序调用了 C 程序

在这些情况下，必须知晓参数是如何传递的，以及值是如何返回的，才能在主调函数与子程序之间协同工作。这些交互的机制在 ARM 中有明确的规定，由文档《**ARM Architecture Procedure Call Standard (AAPCS, Ref5)**》给出。

不过，在大多数场合下的情况都比较简单：当主调函数需要传递参数（实参）时，它们使用 **R0-R3**。其中 **R0** 传递第一个，**R1** 传递第 2 个……在返回时，把返回值写到 **R0** 中。在子程序中，可以随心所欲

地使用R0-R3，以及R12（回顾第9章，想想为什么会PUSH它们）。但若使用R4-R11，则必须在使用之前先PUSH它们，使用后POP回来。

可见，汇编程序使用R0-R3，R12时会很舒服。但是如果换个立场——汇编要呼叫C函数，则考虑问题的方式就有所不同：必须意识到子程序可以随心所欲地改写R0-R3，R12，却决不会改变R4-R11。因此，如果在调用后还需要使用R0-R3，R12，则在调用之前，必须先PUSH，从C函数返回后再POP它们，对R4-R11则不用操心。在本章的示例程序中，绝大多数只是调用汇编子程序，它们只影响少量寄存器，或者会在返回前恢复寄存器的内容，所以往往没有严格遵守AAPCS。这主要是为了突出其它重点，简化程序，请读者不要钻牛角尖。

10.3 典型的开发流程

在开发基于CM3的应用程序时，常常有多种源程序和库，有些是自己写的，有些是别人已经写好的（尤其是底层的软件）。上述这些开发工具生成代码的流程都差不离。对于最基本的应用，也至少需要C编译器，连接器以及二进制文件处理工具。如果使用的是ARM的工具，如RVDS或RealView编译器工具（RVCT），则它们的流程如图10.1所示。其中的“分散加载脚本”是可选的，但是当存储器映射变得比较复杂时，则需要它。

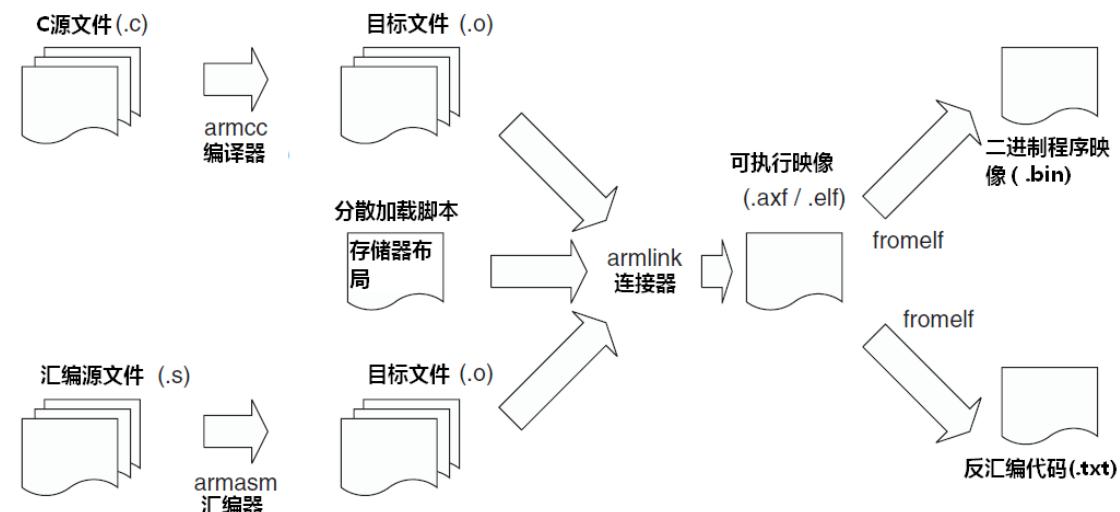


图10.1 使用ARM工具链时的典型开发流程

在上述基本工具之外，RVDS还提供了大量的其它实用程序，比如一个集成开发环境(IDE)以及调试器。欲知详情，可登录ARM网站（www.arm.com）。

10.4 第一步工作

本章为提供了若干汇编的例子，在实际应用中，这些程序都会用C写。但是以汇编的方式呈现，有助于让读者更深更好地理解CM3的工作内幕，以便在以后用C开发时，已经是过来人，心里更有底。这里给的程序都用ARM的汇编器(`armasm`)来汇编，其它工具可能对语法格式有些不同的要求。而且实际上，开发工具几乎都会把启动工作做好，让我们根本不用去想还有启动代码的事（不过，这也妨碍了我们学习得更深入）。下面，就隆重请出本书第一个完整的示例程序（请参考向量表来阅读）：

```

STACK_TOP EQU 0x20002000      ; SP初始值，常数
AREA |Header Code|, CODE
  
```

```

    DCD      STACK_TOP          ; 栈顶 (MSP的)
    DCD      Start             ; 复位向量
    ENTRY
    Start
    ; 初始化寄存器
    MOV      r0,    #10          ; 加载循环变量的初值
    MOV      r1,    #0           ; 初始化运算结果的值
    ; 计算 10+9+8+...+1

loop
    ADD      r1,    r0          ; R1 += R0
    SUBS   r0,    #1           ; R0自减，并且根据结果更新标志 (有"S"后缀)
    BNE    loop              ; if (R0!=0) goto loop;
    ; 现在，运算结果在R1中

deadloop
    B deadloop           ; 工作完成后，进入无穷循环
    END
    ; 标记文件结束

```

这个例子非常简单，它只初始化了SP以及PC，以及初始化了需要使用的寄存器，然后就执行连加循环中。

使用ARM工具来汇编该程序，命令为：

```
$> armasm --cpu cortex-m3 -o test1.o test1.s
```

命令行中的“-o”指示后面的是输出文件名——也就是**test1.o**，它也就是目标文件。接下来，我们就要使用连接器，连接各目标文件（本例中只有一个）并创建出一个可执行的映像（**ELF**），命令为：

```
$> armlink --rw_base 0x20000000 --ro_base 0x0 --map -o test1.elf test1.o
```

这里，“**--ro_base 0x0**”的意思是说，把只读区（也就是程序**ROM**）的起始地址设为0；而“**--rw_base 0x20000000**”则指定读写区（数据存储器）从0x20000000开始（在本例中，我们没有定义任何**RAM**数据）。“**--map**”选项则要求连接器给出存储器分配映射表，通过它，可以查看编译后的映像中内存的布局。

最后，我们要生成二进制烧写文件，命令行为：

```
$> fromelf --bin --output test1.bin test1.elf
```

如果想要看看生成的映像是否确实是我们想要的，还可以像这样对它做反汇编：

```
$> fromelf -c --output test1.list test1.elf
```

（其实基本上很少会做上步——译者注）

如果一切都好，就可以把**ELF**映像或者二进制代码烧写到器件中了，也可使用模拟器来测试。

10.5 与外界互动

如果能把自己的单片机与外面的世界联系起来，那该是多么令人兴奋和值得期待呀！我们常常从点亮**LED**开始，仿佛是前进路上的明灯，尽管它提供的信息非常有限，但闪烁的灯光常给人“它活着”的印象。如果要输出更多的信息，则最容易上手的方式就是往一个终端发送文本。在嵌入式产品开发中，通常是把一个**UART**接到电脑上来实现的。例如，运行**Windows**的电脑大多会有一个附送的“超级终端”程序，通过它可以很方便地让电脑扮演字符终端的角色。

Cortex-M3内核没有包含**UART**接口，但基于**Cortex-M3**的单片机都会有的，而且基本还不止一个。不同芯片的**UART**用法不同，但是本书就不讲具体的**UART**驱动了，扯得太远也跑题。在下一个例子中，我们假

设系统中有一个UART，UART中有一个“状态位”，用于指示输出缓冲是否已经准备好接收新数据。另外，还需要一个电平转换器件（如MAX3232），用于把单片机I/O口使用的电平转换成RS-232使用的电平。其实，UART并不是输出文本的唯一选择，在CM3中有很多调试组件，它们提供了一系列输出调试消息的方法：

- 半主机（**Semihosting**）：取决于调试器与代码库的支持，可以通过**NVIC**的调试寄存器来做**Semihosting**（通过调试探测设备，以**printf**的形式输出消息），第15章还要深入讨论这个主题。使用时，你要在**C**程序中使用**printf**函数，然后其输出就会显示在终端，或者显示在调试软件标准输出（**STDOUT**）上，具体细节还是请参阅第15章。
 - 硬件水平上支持的跟踪：如果使用的**CM3**单片机提供了一个跟踪接口，并且有一台外部的跟踪接口分析仪（**TPA**）的话，则可以解放出**UART**，而使用**ITM**来做形如**printf**的调试。跟踪端口就是为了这种调试而生的，它可比**UART**专业多了——速度快而且能提供多条信道。
 - 硬件水平上支持的跟踪——通过串行线查看器：作为后备方案，**CM3**的**TPIU**还提供了“串行线查看器（**SWV**）”操作模式。有了它，就可以使用远比**TPA**便宜的硬件来捕获从**ITM**发来的消息。不过，在**SWV**模式下，带宽并不富余，因此在需要输出大量数据时，本法就显得有些力不从心。

10.5.1 “Hello World” 示例程序

这次来真格的了。不过在开始前，先要指出使用何种形式把一个字符发给UART。可以把发送字符的代码做成一个子程序，由其它函数呼叫来输出数据。这样的好处在于，如果输出设备变了，则只需重写这个子程序，就可以使用不同的设备，这种修改动作也有自己的术语，叫“**retargeting**”（“目标重选”？这词还真不好翻译~）。在大型程序中，这是一个很重要的思想——软件分层，而且这也是“设备无关性”和“可移植性”的前提。

让我们看看一个简单的字符输出子程是啥模样:

```

UART0_BASE EQU      0x4000C000
UART0_FLAG EQU      UART0_BASE+0x018
UART0_DATA EQU      UART0_BASE+0x000

Putc                                     ; 该子程使用UART来发一个字符
; 入口条件: R0 = 需要发的字符
PUSH    {R1,R2, LR}                      ; 保存寄存器
LDR     R1,      =UART0_FLAG

PutcWaitLoop
LDR     R2,      [R1]                      ; 读取状态标志
TST     R2,      #0x20                     ; 检查“发送缓冲满”标志
BNE     PutcWaitLoop                      ; 若已满则重试（若UART当掉了，则可能死循环）
LDR     R1,      =UART0_DATA              ; 有空位时，就把UART发送站寄存器地址加载
STRB   R0,      [R1]                      ; 然后通过它把字符送给输出缓冲区
POP    {R1,R2, PC}                      ; 子程返回

```

在这里的UART是虚构的，其寄存器的地址和位定义都只是为了演示，抛砖引玉。如果雷同，纯属巧合。在实战时，还需要根据自己使用的UART来重塑代码，有些UART还要求更精密地检查状态位。另外，还需要一个用于初始化UART的子程——至少得设置波特率吧。我们为了突出主题，这些细节就不多谈了。在第20章中，有一个具体的例子。

现在，我们就可以通过这个基础设施一般的子程，来构造一系列的消息显示函数，它们都与输出字符的具体硬件无关了。

```

Puts ; 该子程往UART送一个字符串
; 入口条件: R0 = 待输出字符串的起始地址
; 这个字符串必须以零结尾 (C语言格式)
PUSH {R0 ,R1, LR} ; 先保存寄存器
MOV R1, R0 ; 把地址拷贝到R1, 因为待会儿调用Putc时还要用
; R0来传参数

PutsLoop
LDRB R0, [R1], #1 ; 读取一个字符, 再自增地址
CBZ R0, PutsLoopExit ; 若已到达零字符, 则执行完毕, 退出
BL Putc ; 把这个字符送往UART
B PutsLoop ; 循环, 以输出下一个字符

PutsLoopExit
POP {R0, R1, PC} ; 子程序返回

有了这个Puts, 现在终于可以正式请大牌出场了——“Hello World”主程序:

STACK_TOP EQU 0x20002000 ; SP初始值
UART0_BASE EQU 0x4000C000
UART0_FLAG EQU UART0_BASE+0x018
UART0_DATA EQU UART0_BASE+0x000

AREA | Header Code|, CODE
DCD STACK_TOP ; MSP初始值
DCD Start ; 复位向量
ENTRY

Start ; 主程序入口点
MOV r0, #0 ; 初始化各寄存器
MOV r1, #0
MOV r2, #0
MOV r3, #0
MOV r4, #0
BL Uart0Initialize ; 初始化UART0
LDR r0, =HELLO_TXT ; 让R0指向字符串的起始地址
BL Puts

deadend
B deadend ; 做完了工作, 在这里原地打转
;-----
; 各个子程序
;-----
Puts ; 该子程往UART送一个字符串
; 入口条件: R0 = 待输出字符串的起始地址
; 这个字符串必须以零结尾 (C语言格式)
PUSH {R0 ,R1, LR} ; 先保存寄存器
MOV R1, R0 ; 把地址拷贝到R1, 因为待会儿调用Putc时还要用
; R0来传参数

PutsLoop

```

```

LDRB    R0, [R1], #1          ; 读取一个字符, 再自增地址
CBZ    R0, PutsLoopExit      ; 若已到达零字符, 则执行完毕, 退出
BL     Putc                  ; 把这个字符送往UART
B      PutsLoop              ; 循环, 以输出下一个字符

PutsLoopExit
    POP {R0, R1, PC}          ; 子程序返回
;-----

Putc
; 入口条件: R0 = 需要发的字符
    PUSH {R1, R2, LR}          ; 保存寄存器
    LDR    R1, =UART0_FLAG

PutcWaitLoop
    LDR    R2, [R1]              ; 读取状态标志
    TST    R2, #0x20              ; 检查“发送缓冲满”标志
    BNE    PutcWaitLoop          ; 若已满则重试(若UART当掉了, 则可能死循环)
    LDR    R1, =UART0_DATA      ; 有空位时, 就把UART发送站寄存器地址加载
    STRB   R0, [R1]              ; 然后通过它把字符送给输出缓冲区
    POP    {R1, R2, PC}          ; 子程返回
;-----

Uart0Initialize
; 与具体硬件有关, 也不是主题, 故而略
    BX    LR                  ; 子程序返回
;-----

HELLO_TXT
    DCB "Hello world\n", 0      ; 定义零结尾的“Hello world”
    END                          ; 本文件结束

```

本示例代码在各CM3单片机之间都是高度可移植, 高度与硬件无关的。事实上, 我们只需要自己写Uart0Initialize子程, 并调整Putc。之所以日子这么好过, 是因为Putc与Puts已经完成了实质的工作。为了锦上添花, 最好再提供几个子程, 用于输出寄存器的值。首先是输出16进制数的子程。

```

PutHex
; 入口条件: R0=要显示的值
    PUSH {R0-R3, LR}
    MOV    R3, R0                ; 把R0的值拷贝到R3, 以便待会使用R0传递参数给Putc
    MOV    R0, #'0'              ; 先显示“0x”前缀
    BL    Putc
    MOV    R0, #'x'
    BL    Putc

    MOV    R1, #8                ; 初始化循环变量
    MOV    R2, #28                ; 圆圈移位偏移量

PutHexLoop
    ROR    R3, R2                ; 圆圈右移28格——相当于圆圈左移4格

```

```

AND    R0,    R3,    #0xF      ; 此时最高4位移至最低4位, 提取它们
CMP    R0,    #0xA      ; 转换成ASCII码
ITE    GE
ADDGE R0,    #55      ; 若大于等于10, 则使用字母A-F表示
ORRLT R0,    #0x30      ; 否则转换到0-9 (原文使用ADDLT, 效果相同)
BL     Putc      ; 输出一个hex字符
SUBS   R1,    #1       ; 循环变量自减
BNE    PutHexLoop      ; 检查循环变量是否已减到0, 从而循环8次
POP    {R0-R3, PC}      ; 显示完毕, 子程返回

```

使用这个子程来输出寄存器的值很方便, 如果在笔试的时候遇到这个题目, 就可以直接抄上去啦! 但如果是让你输出10进制数, 可就不像乍一看的那么好对付了, 而且它还很黄很暴力, 能放倒一大批人——要计算32位乘除法 (考官阴笑: 小样傻眼了吧)! 好在CM3下凡后, 带出来两颗大力丸——硬件乘除法指令。服下它们, 转身以后你会练成护体神功, 看见蟑螂也不怕不怕啦! 不过, 可别神经比较大, 因为考官还下了另一个小套儿等你钻呢: 在计算期间, 我们计算出的字符会是逆序的——即如果不采取措施, `0x7B(123)`会以321的顺序输出! 因此, 只好另开一个缓冲区来保存中间结果——先把所有的字符逆序放到这个缓冲区中, 来个负负得正, 最后使用`Puts`来一步到位地显示整个结果。在本例中, 使用栈空间来存储这个缓冲区, 用完即释放——在C编程中, 这叫自动局部数组变量。

```

PutDec                                ; 以10进制输出寄存器的值
; 入口条件: R0=要显示的值
; 因为是32位宽, 最大值 (0xffff_ffff) 需要10个10进制位表示, 再加上零结尾, 共需11字节
PUSH   {R0-R5, LR}      ; 保存寄存器的值
MOV    R3,    SP      ; 把当前堆栈指针拷贝到R3
SUB    SP,    SP,    #12      ; 为文本缓冲区保留出11个字节 (因为是满栈)
MOV    R1,    #0       ; NULL字符
STRB   R1,    [R3, # -1]!      ; 先把NULL字符写到字符串的结尾 (把各字符逆序输出,
; 好“负负得正” )。这里使用了更新基址的预索引

MOV    R5,    #10      ; R5保存除数
PutDecLoop
UDIV   R4,    R0,    R5      ; R4 = R0 / 10
MUL    R1,    R4,    R5      ; R1 = R4 * 10
SUB    R2,    R0,    R1      ; R2 = (R0-(R0/10)*10), 即个位
ADD    R2,    0x30      ; 转换成ASCII (因为R2只能是0-9), 亦可使用ORR
STRB   R2,    [R3, # -1]!      ; 把ascii字符送进缓冲区
MOVS   R0,    R4      ; R0 = 商, 并且根更新标志位以检查商是否为零
BNE    PutDecLoop      ; 若商为零, 则已经把所有10进制位都求出
MOV    R0,    R3      ; R0指向文本缓冲区的起始地址
BL     Puts      ; 使用Puts显示结果
ADD    SP,    SP,    #12      ; 恢复SP指针
POP    {R0-R5, PC}      ; 子程返回

```

怎么样, 这下考官得赏你一个麻花两个鸡蛋了吧! 如果读者还看过不使用乘除法指令实现该子程的代码, 再对比一下两者的执行速度 (甚至能相差数百倍), 一定会被震撼到, 留下刻骨铭心的记忆的。

10.6 使用数据存储器

重温一下我们的第一个例子：在我们做到程序连接这一步时，我们手工指定了读/写区的位置。那么我们该如何把数据放到那里呢？正点的解决方法是：在汇编源文件中定义一个相应的数据区。让连接器把数据区中的内容分派到我们指定的位置——从0x2000_0000(SRAM区的起始)处开始的内存。

回顾当时使用的连接命令：

```
$> armlink --rw_base 0x20000000 --ro_base 0x0 --map -o test1.elf test1.o
```

```
STACK_TOP EQU 0x20002000      ; SP初始值, 常数
AREA |Header Code|, CODE
    DCD    STACK_TOP      ; 栈顶 (MSP的)
    DCD    Start         ; 复位向量
    ENTRY
    Start          ; 主程序开始
    ; 初始化寄存器
    MOV    r0,    #10      ; 加载循环变量的初值
    MOV    r1,    #0       ; 初始化运算结果的值
    ; 计算 10+9+8+...+1
loop
    ADD    r1,    r0      ; R1 += R0
    SUBS   r0,    #1      ; R0自减, 并且根据结果更新标志 (有"S"后缀)
    BNE    loop          ; if (R0!=0) goto loop;
    ; 现在, 运算结果在R1中
    LDR    r0,    =MyData1
    STR    r1,    [r0]      ; 把结果存入MyData1
deadloop
    B    deadloop        ; 工作完成后, 进入无穷循环
; 定义数据区
AREA | Header Data|, DATA
    ALIGN 4
MyData1
    DCD    0 ; Destination of calculation result
MyData2
    DCD    0
END ; 文件结束标记
```

在连接阶段，连接器要把DATA区放入读/写存储器中，因此MyData1的地址就将是我们指定的0x2000_0000。

10.7 使用互斥访问实现信号量操作

互斥访问是新出来的，并且专门用于信号量的操作中。最常见的用途，就是确保需要互斥使用的共享资源只被一个任务拥有。

让我们举个例子。记**DeviceALocked**是一个位于内存中的R/W变量，用于指示设备A是否已经在使用中。任何一个任务，若欲使用设备A，都必须先检查这个变量的值。如果它的值为零，则表示设备可以使用。在任务获取到设备A后，它要把**DeviceALocked**的值改为1，表示设备A已经被占用。在设备A使用完毕后，该任务通过重新清零**DeviceALocked**来释放设备A，从而使其它任务可以使用此设备。

看起来这是个如意算盘。不过可否想过，如果两个任务都想访问设备A，是否有潜在的危险？比如，在任务1读取了**DeviceALocked**后，发现是零于是准备使用此设备，但还没来得及把它改为1，就不巧被调度器切出（比如，轮转调度），然后调度器让任务2执行，于是任务2也读到零，从而它使用设备A。但是在任务2在用完设备A之前，调度器又切回任务1。由于任务1早先读回来的是零，所以它认为设备A是空闲的，于是使用设备A，这时就违背了设备A必须互斥访问的限制，使系统出现紊乱危象！如果设备A是台打印机，则把两个文档的内容打在了一起；如果设备A是油门控制器，则可能使汽车失控或熄火，后果不堪设想。

为避免此问题，必须也保证**DeviceALocked**的互斥访问。回顾一下第5章，**STREX**指令是有返回值的，指示访问是成功还是被“驳回”。接上例，如果任务#1和任务#2都使用**STREX**，则任务#1的**STREX**将被驳回——返回1，从而任务1知道这期间已经发生了很多事，设备A已被他人占有，就避免了紊乱危象。互斥访问的模式图如图10.3所示。

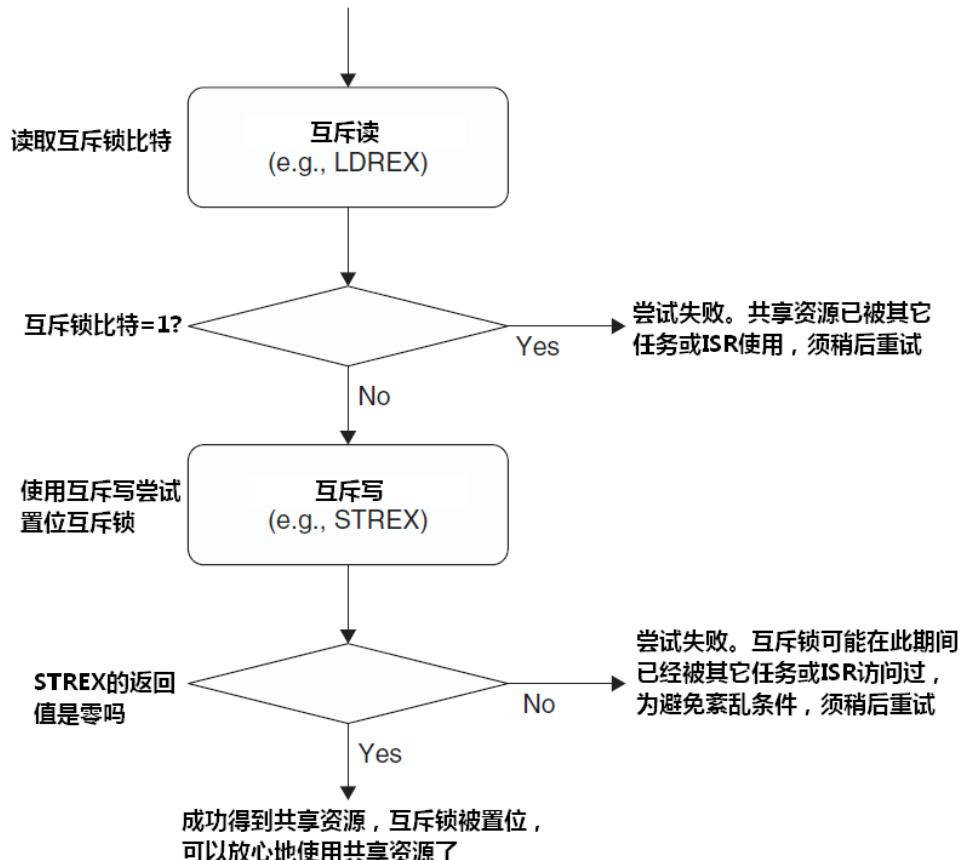


图10.3 使用互斥访问来实现信号量（互斥锁）的操作

上述操作可通过下面的示例代码实现。理解的关键在于，如果互斥访问监视器返回了失败的状态，**STREX**就被驳回。此时，不会执行写操作，从而保护了互斥锁在访问尝试失败时不被更改。

```

LockDeviceA
;
; 一个简单的函数，演示如何尝试锁住设备A
;
; 返回值: R0=0表示成功，R0=1表示失败
;
; 如果访问成功，则将把DeviceALocked的值改为1
  
```

```

PUSH {R1, R2, LR}

TryToLockDeviceA
    LDR R1, =DeviceALocked
    LDREX R2, [R1] ; 使用互斥读来标记对互斥锁的访问
    CMP R2, #0 ; 检查是否已被锁住
    BNE LockDeviceAFailed

DeviceAIsNotLocked
    MOV R0, #1 ; 准备锁住设备A
    STREX R2, R0, [R1] ; 互斥写
    CMP R2, #0
    BNE LockDeviceAFailed ; STREX失败，设备A可能已被锁

LockDeviceASucceed
    MOV R0, #0 ; 准备返回成功值
    POP {R1, R2, PC} ; 子程序返回

LockDeviceAFailed
    MOV R0, #1 ; 准备返回失败值
    POP {R1, R2, PC} ; 子程序返回

```

如果返回的是1，则为了避免紊乱危象，任务必须重试。在单处理器系统中，互斥访问主要用在ISR与主程序之间，用以保护它们共享的，并且需要互斥访问的资源（如，一块内存，一个外设）。此时，引起互斥写失败的唯一原因，就是在读写期间曾响应过中断。如果代码在特权级下运行，还可以通过设置PRIMASK，在“测试——置位”期间暂时把中断给掐了。

在多处理器系统中，情况会变得更复杂。此时，除了本机的中断，其它处理器对同一块内存的访问也可以使互斥写操作失败。为了检测到其它处理器对内存的访问，总线系统中必须加入一个“互斥访问监视”的硬件基础设施。它负责检测在互斥读写期间，总线上是否有其它主机访问了互斥锁及其临近的“高危地带”。事实上，在绝大多数低成本的CM3单片机中，都只包含了一个核，因此无需此监视器。

有了这个机制，我们就可以确信共享资源一定能互斥地使用，不会发生紊乱危象。如果一个共享资源在多次尝试时依然无法获取，则可能必须放弃对此资源的请求，有可能先前锁住该资源的任务已经崩溃了。

10.8 使用位带实现互斥锁操作

如果存储器系统支持“锁定传送”（locked transfers），或者总线上只有一个主机，还可以使用CM3的位带功能来实现互斥锁的操作。通过使用位带，则可以在C程序中实现互斥锁，但是操作过程与互斥访问是不同的。在使用位带来做资源分配的控制机制时，需要使用位带存储区的内存单元（比如，一个字），该内存单元的每个位表示资源正被特定的任务使用。

在位带别名区的读写实质上是锁定的“读-改-写”（在传送期间总线不能被其它主机占有）。因此，只要每个任务都仅修改分配给它们自己的锁定位，其它任务锁定位的值就不会丢失，即使是两个任务同时写自己的锁定位也不怕，如图10.4所示。

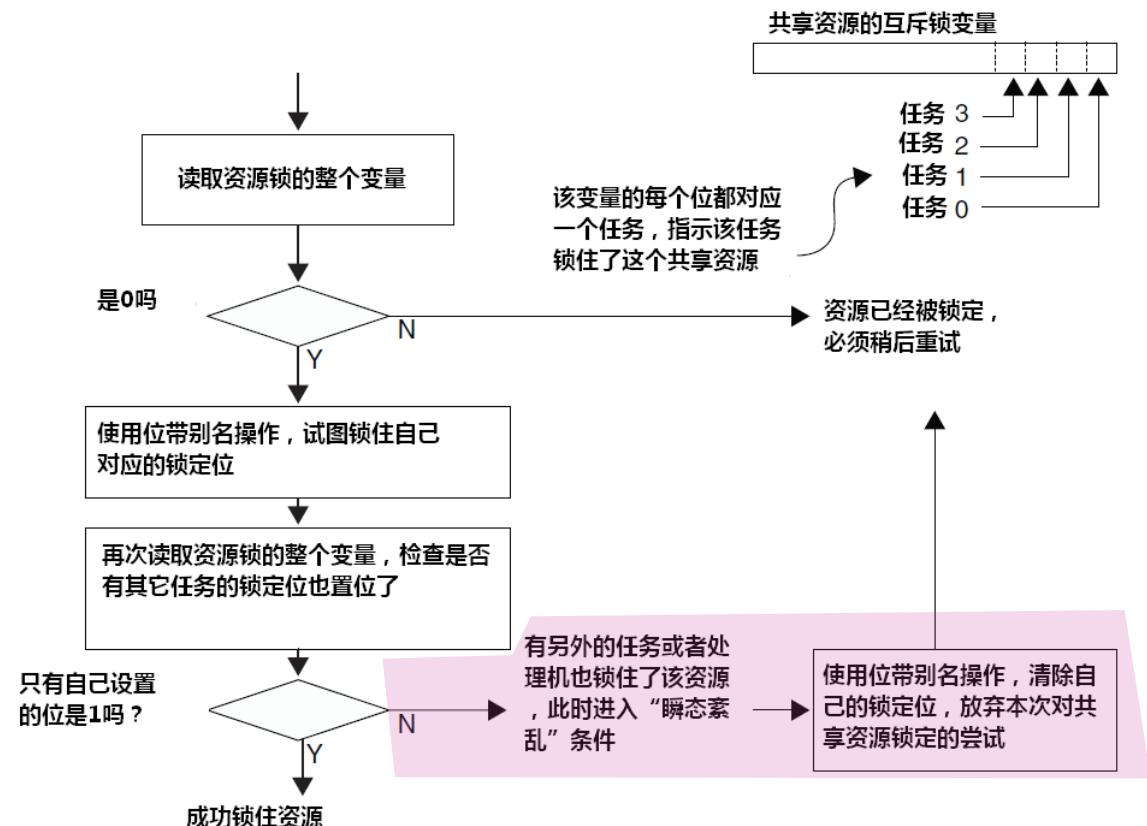


图10.4 使用位带实现互斥锁的工作流程图

从图10.4我们可以看出，位带操作有可能使共享资源在一个短期内被“多重锁定”，从而有“瞬态紊乱”。但是它不会造成危害，因为任务一定能检测到这个冲突，从而释放自己的锁。

其实，对于“测试并设置”这种互斥锁的简单操作，也可以使用“关中临界区”来保护——也就是在操作前关中断，操作后开中断。这种关中的时间是很短的，因为其它原因导致的关中通常都比这个长得多。只是有时为了无限追求实时性，有一丝希望也会尽最大的努力，就像这两种互斥锁操作那样。

10.9 使用位段提取与查表跳转

在第4章中，我们考察了位段提取指令（UBFX）和查表跳转指令（TBB/TBH）。这两条指令可以配合工作，以构建一个非常强大的“跳转树”。这对于电表及数据通信应用程序非常有意义，常使这类程序得到戏剧般地优化。这类程序在工作时，经常要判断各种各样的情况，并且“分类讨论”。有时，还需要进一步细化，作二级甚至多级的比较判断。例如，下图就演示了一个“判决树”，它根据输入量A的各位段编码，来决定启动的任务。

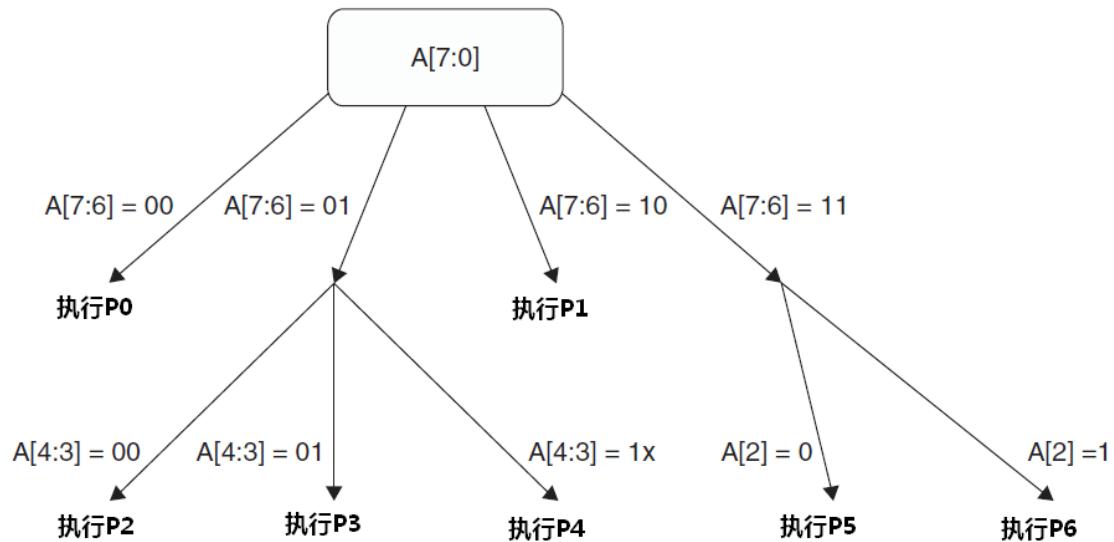


图10.5 通过各位段编码决定操作的判决树示例

```

DecodeA
    LDR    R0, =A           ; 从内存中读取A的值
    LDR    R0, [R0]
    UBFX  R1, R0, #6, #2   ; R1=R0[7:6]
    TBB    [PC, R1]

BrTable1
    DCB    ((P0 -BrTable1)/2)    ; 如果 A[7:6] = 00 则跳至P0
    DCB    ((DecodeA1-BrTable1)/2); 如果 A[7:6] = 01 则跳至DecodeA1, 继续解码
    DCB    ((P1 -BrTable1)/2)    ; 如果 A[7:6] = 10 则跳至P1
    DCB    ((DecodeA2-BrTable1)/2); 如果 A[7:6] = 11 则跳至DecodeA2

DecodeA1
    UBFX  R1, R0, #3, #2   ; R1=R0[4:3], 准备二级解码
    TBB    [PC, R1]

BrTable2
    DCB    ((P2 -BrTable2)/2)    ; 如果 A[4:3] = 00 则跳至P2
    DCB    ((P3 -BrTable2)/2)    ; 如果 A[4:3] = 01 则跳至P3
    DCB    ((P4 -BrTable2)/2)    ; 如果 A[4:3] = 10 则跳至P4
    DCB    ((P4 -BrTable2)/2)    ; 如果 A[4:3] = 11 则也跳至P4

DecodeA2
    TST   R0, #4           ; 只需检测一个位, 因此无需UBFX
    BEQ   P5
    B     P6

P0 ...
P1 ...
P2 ...
P3 ...
P4 ...
P5 ...
P6 ...
  
```

看，如果使用C来写这个这个程序，则需要使用嵌套的switch和大量的位操作；可现在却干净利落得如此爽快！如果跳转目标更远，可以使用TBH指令。

玩转异常系统

- 使用中断
- 异常/中断服务例程
- 软件中断
- 异常服务例程的示范
- 使用 SVC
- SVC 示范：用于输出数据的函数
- 在 C 中使用 SVC

NMI, Faults, SVC, PendSV, IRQ #0, IRQ #1,

自动栈操作、向量式、抢占、咬尾、晚到.....

CM3 把“中断/异常”这个概念捧到了登峰造极的境界，为实时系统的开发垫上了那么一个宽大的肩。如果在 CM3 上开发却不能善用这炙手可热的能力，那说不定都会有一种暴殄天物的感觉！

11.1 使用中断

任何一个有点型的嵌入式系统，就没有不使用中断机制的。在 CM3 中，NVIC 为我们搞定了使用中断时的很多例行任务，如优先级检查、入栈/出栈、取向量等。不过在 NVIC 能行使职能之前，还需要我们做好如下的初始化工作：

- 建立堆栈
- 建立向量表
- 分配各中断的优先级
- 使能中断

11.1.1 建立堆栈

当开发的程序比较简单时，可以从头到尾都只使用 MSP。这时，只需要保证开出一个容量够大的堆栈，再把 MSP 初始化到其顶即可——这也是单片机开发最常见的做法。

堆栈用穿是非常致命的错误，必须非常严肃地计算安全容量。在计算时，除了要计入最深函数调用时对堆栈的需求，还需要判定最多可能有多少级中断嵌套。一个笨方法（但是很保险）是假设每个中断都可以嵌套。对于每一级嵌套的中断，至少需要 8 个字（32 字节），而且如果 ISR 过于复杂，还可能有更多的堆栈需求。

因为 CM3 中的堆栈是以“向下生长的满栈”来操作 SP 的。在简单的场合中，经常可以把 SP 初始化为 SRAM 的末尾，这么一来就使所有的空闲内存都能为堆栈所用——反正不用白不用，用了也白用，如图 11.1 所示。

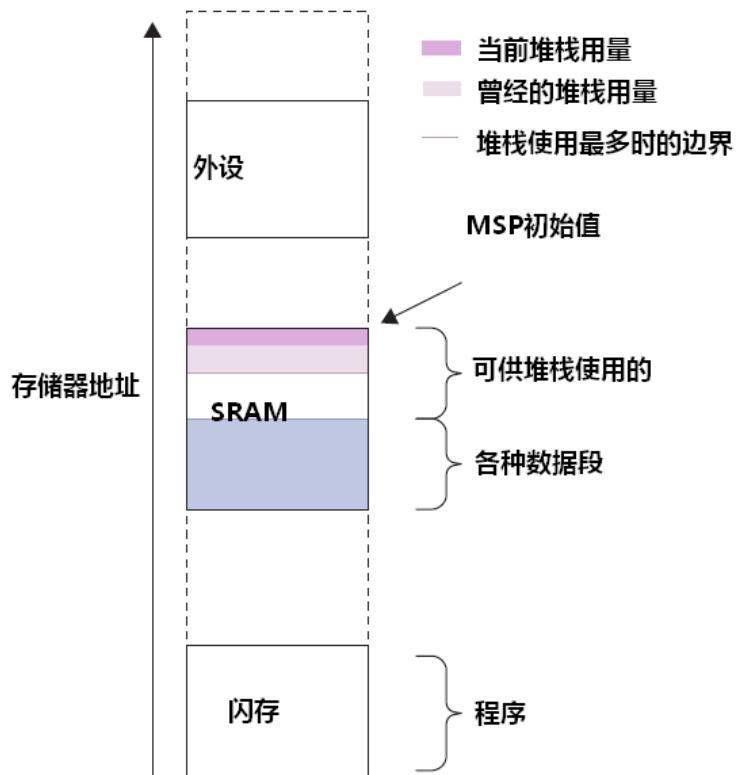


图 10.1 简单程序中典型的存储器分配

从图中我们可以看出，这种分配方式能给堆栈区留下最大的容量——所有剩余内存，而有省事又省心——省去了令人头痛的堆栈需求计算了。

然而，对于比较大型的或者是高性能指标的嵌入式系统，往往需要两个堆栈配合使用。这时，就只好勇敢地面对。必须保证各堆栈都有足够的容量，尤其是主堆栈，最容易裁在它上面。要注意的是，进程堆栈除了要满足本进程的最大需求量，还需要额外留出 8 个字，用于容纳第一级中断时被保护的寄存器。

（译者添加）事实上，准确计算主堆栈需求往往是不可能的任务，也容易过于保守而浪费宝贵的血液资源。在调试阶段时，最好先选用内存更大点的器件，然后开出足够大的内存给主堆栈。然后在调试程序时，允许随时把主堆栈曾经的最大用量输出（通过调试串口或仿真器等），这样时间长了就能估算对主堆栈的需求，正如图 10.1 中边界的作用。

11.1.2 建立向量表

如果在程序执行的从头到尾，都只给每个中断提供固定的中断服务例程（这也是目前单片机开发的绝大多数情况），则可以把向量表放到 ROM 中。在这种情况下不需要运行时重建向量表。然而，如果想让自己的设备能随机应变地对付各种复杂情况，就常常需要动态地改变中断服务例程，更新向量表就是必需的了。此时，向量表必须被转移到可读写存储器中（如内存）。

在把向量表重定位之前，往往要把现有的向量表往新的位置复制一份。需要拷贝的向量主要是系统异常的服务例程，如各种 **fault** 的、**NMI** 的以及 **SVC** 的等等。如果没有建立好这些向量就启用了新的向量表，就可能会在响应异常时把不可预料的地址取出，程序极有可能跑飞。

当我们把所有必要的向量都填好后，就可以启用了新的向量表了。然后继续往里面加入新的中断向量，例如：

; 该子程序根据异常类型建立相应的异常向量

```

; 对于IRQ, 异常号=中断号+16
; 入口条件: R0=异常类型编号
; 入口条件: R1=向量地址
PUSH {R2, LR}
LDR R2, =0xE000ED08 ; 向量表偏移量寄存器的地址
LDR R2, [R2] ; 获取向量表的首地址
STR R1, [R2, R0, LSL #2] ; 在VectTblOffset+ExcpType*4处写入向量
; ExcpType*4
POP {R2, PC} ; Return

```

11.1.3 建立中断优先级

在复位后, 对于所有优先级可编程的异常, 其优先级都被初始化为0。而对于NMI和硬fault, 由于它们要在危难之际挺身而出, 所以把它们的优先级定死为-2和-1 (高于任何其它异常)。在编程优先级寄存器时, 我们可以利用它们能按字节访问的好处, 以简化程序代码, 如:

```

; 把IRQ #4的优先级设为0xC0
LDR R0, =0xE000E400 ; 加载外部空优先级寄存器阵列的起始地址
LDR R1, =0xC0 ; 优先级
STRB R1, [R0, #4] ; 为IRQ #4设置优先级 (按字节写)

```

在CM3中, 允许使用3个位到8个位来表达优先级。为了确定具体的位数, 可以先往一个优先级寄存器中写0xFF, 再读回来, 读出多少个1, 就表示使用多少个位来表达优先级, 如下所示 (下段代码演示了RBIT配CLZ的绝技):

```

; 检测系统使用多少个位来表达优先级
LDR R0, =0xE000E400 ; 加载IRQ #0的优先级配置寄存器
LDR R1, =0xFF
STRB R1, [R0] ; 按字节写, 写入0xFF
LDRB R1, [R0] ; 读回 (如果是3位, 则应读取回0xE0)
RBIT R2, R1 ; 反转, 使之以LSB对齐
CLZ R1, R2 ; 计算前导零个数 (例如, 如果是3个1则返回5)
MOV R2, #8
SUB R2, R2, R1 ; 得到表达优先级的位数
MOV R1, #0x0
STRB R1, [R0] ; 存储结果

```

如果程序可能要跨器件移植 (常见于比较底层的基础设施函数), 那么最好只使用最高3个有效位, 对应的优先级为: 0x00, 0x20, 0x40, 0x60, 0x80, 0xA0, 0xC0以及0xE0。所有的CM3芯片都一定支持3个位表达的优先级。

还要提醒的是, 不要忘记为系统异常 (包括faults) 建立优先级。如果程序中有非常紧急的外部中断, 它们甚至需要比系统异常还紧急, 可是却因故不能连接到NMI上, 就要把系统异常的优先级调低, 才能保证紧急的中断能够抢占系统异常, 从而不被延误。

11.1.4 使能中断

在向量表与优先级都建立好后, 就到了最后一步: 开中断的时候了。

然而, 在打开中断之前, 可能还有两个步骤不能省略:

1. 如果把向量表重定位到了RAM中, 且这块RAM所在的存储器区域是写缓冲的, 向量更新就可能被

延迟。为了以防万一，必须在建立完所有向量后追加一条“数据同步隔离(DSB)”指令（见第4章），以等待缓冲写入后再继续，从而确保所有数据都已落实。

2. 开中断前可能已经有中断悬起，或者请求信号有效了，这往往是不可预料的。比如，在上电期间，信号线上有发生过毛刺，就可能会被意外地判定成一次中断请求脉冲。另外，在某些外设，如UART，在串口连接瞬间的一些噪音也可以被误判为接收到的数据，从而使中断被悬起。

在NVIC中进行中断的使能与除能时，都是使用各自的寄存器阵列(SETENA/CLRENA)来完成的：通过往适当的位写1来发出命令，而写0则不会有任何效果。这就让每个中断都可以自顾地使能和除能，而不必担心会破坏其它中断的设置。这改变了以前必须“读-改-写”的三步曲，从而在根本上消灭了在此地产生紊乱危象的可能；否则，必须使用互斥访问等机制来完成修改。下例就演示了通过置位SETENA中的位来使能中断；通过置位CLRENA中的位来除能中断：

1. 使能中断

; 根据IRQ号来使能中断的子程序

EnableIRQ

```
; 入口条件: R0=中断号
PUSH {R0-R2, LR}
AND.W R1, R0, #0x1F      ; 为该IRQ产生移位量
MOV R2, #1
LSL R2, R2, R1           ; 位旗标 = (0x1 << (N & 0x1F))
AND.W R1, R0, #0xE0       ; 若IRQ编号>31则为它生成下标偏移量
LSR R1, R1, #3            ; 地址偏移量= (N/32)*4 (每个IRQ一个位)
LDR R0, =0xE000E100       ; 加载SETENA寄存器阵列的首地址
STR R2, [R0, R1]          ; 写入该中断的位旗标，从而使能该中断
POP {R0-R2, PC}           ; 子程返回
```

2. 除能中断

几乎是照抄上一个例程，就得到了对应的除能中断的子程序：

DisableIRQ

```
; 入口条件: R0=中断号
PUSH {R0-R2, LR}
AND.W R1, R0, #0x1F      ; 为该IRQ产生移位量
MOV R2, #1
LSL R2, R2, R1           ; 位旗标 = (0x1 << (N & 0x1F))
AND.W R1, R0, #0xE0       ; 若IRQ编号>31则为它生成下标偏移量
LSR R1, R1, #3            ; 地址偏移量= (N/32)*4 (每个IRQ一个位)
LDR R0, =0xE000E180       ; 加载CLRENA寄存器阵列的首地址
STR R2, [R0, R1]          ; 写入该中断的位旗标，从而除能该中断
POP {R0-R2, PC}           ; 子程返回
```

访问NVIC寄存器的小贴士

在NVIC中，绝大多数寄存器都可以按字/半字/字节的形式访问。对于不同的场合，应灵活使用适当的形式，以简化程序的开发。比如，对优先级寄存器的按字节访问，就消除了按字/半字访问时，需要“读-改-写”的序列（为的是不影响其它中断的优先级）。

11.2 异常/中断服务例程

在CM3中，中断服务例程可以纯用C来写。与ARM7的情况相比，后者则往往需要首尾都加以汇编封皮，用以保证所有寄存器都保护了。另外，在中断嵌套时，处理器需要切换到另外的模式，以防止信息丢失。这些拖跨系统实时性和带来入门难度的繁文缛节在CM3中都被消灭了，使得编程时舒心很多。

如果用汇编来写ISR，其骨架看上去差不多如下所示：

```
irq1_handler
; 处理中断请求
...
; 消除在设备中的IRQ请求信号
...
; 中断返回
BX LR
```

如果ISR逻辑比较复杂，则常常需要更多的寄存器，这时就要启用R4-R11了。但是它们不是CM3自动入栈的，所以使用前必须手工PUSH。下一个例子演示一个保险的笨方法：保护了所有的寄存器。其实如果内存够用，使用笨方法作为起点也不失为一个不错的主意，等到日后优化程序时再去掉没有使用的寄存器。

```
irq1_handler
PUSH {R4-R11, LR} ; 保存所有可能用到的，又没有被自动入栈的寄存器
; 处理中断请求
...
; 消除在设备中的IRQ请求信号
...
; 中断返回
POP {R4-R11, PC}
```

因为POP也是启动中断返回的一条途径，所以我们把寄存器出栈与中断返回合并在一条POP中，使程序更精练。

有些外设的中断请求信号需要ISR手工清除，如：外设的中断请求是持续的电平信号——显然，对于稍纵即逝的脉冲型的请求，是无需手工清除的。若电平型中断请求没有清除，则中断返回后将再次触发已经服务过的中断。以前在ARM7中，外设必须使用这种“电平保持”的方式[译注]，直到中断被响应，因为那个时候的中断控制器没有保存悬起状态。在CM3中就解决了这个问题：只要检测到过曾经出现的中断请求，NVIC就会记住它，因此硬件只需给一个脉冲，无需再一直保持请求电平，持续的电平反而成为一种讨厌的事了。而且当其服务例程得到执行时，NVIC自动把悬起状态清除。对于这种情况，就不必在ISR中软件清除请求信号了。

译注：很多厂家都在设计ARM7芯片时添加了自己的中断控制器，这些中断控制器也常常能记住请求脉冲。

11.3 软件触发中断

触发中断有多种方法：

- 外部中断输入
- 设置NVIC的悬起寄存器中设置相关的位（第8章）
- 使用NVIC的软件触发中断寄存器（STIR）（第8章）

系统中总是会有一些中断没有用到，此时就可以当作软件中断来使用。软件中断的功用与SVC类似，两者都能用于让任务进入特权级下，以获取系统服务。不过，若要使用软件中断，必须在初始化时把NVIC配置与控制寄存器的USERSETMPEND位置位，否则是不允许用户级下访问STIR的（附录D的表D.17有该寄存器的详细说明）。

但是软件中断没有SVC专业：比如，它们是不精确的，也就是说，抢占行为不一定会立即发生，即使当时它没有被掩蔽，也没有被其它ISR阻塞，也不能保证马上响应。这也是写缓冲造成的，会影响到与操作NVIC STIR相邻的后一条指令：如果它需要根据中断服务的结果来决定如何工作（如条件跳转），则该指令可能会误动作——这也可以算是紊乱危象的一种表现形式。为解决这个问题，必须使用一条DSB指令，如下例所示：

```
MOV R0, #SOFTWARE_INTERRUPT_NUMBER
LDR R1, =0xE000EF00      ; 加载NVIC软件触发中断寄存器的地址
STR R0, [R1]      ; 触发软件中断
DSB              ; 执行数据同步隔离指令
...
```

那是否这样就万事大吉了呢？不幸的是，还不能高兴得太早，因为还有另一个隐患：如果欲触发的软件中断被除能了，或者执行软件中断的程序自己也是个异常服务程序，软件中断就有可能无法响应。因此，必须在使用前检查这个中断已经在响应中了。为达到此目的，可以让软件中断服务程序在入口处设置一个标志。

最后要注意的是，虽然是出于好心置位USERSETMPEND，但容易烧香引出鬼来：因为用户程序可能会以软件的方式触发任何一个中断，制造出各种“假象”。如果系统中包含了不受信任的用户程序，就必须全体接种疫苗——每个异常服务例程都必须检查该异常是否允许。其实，通向天堂是有路的——干嘛不用更专业的SVC来实现系统服务呢？

11.4 异常服务例程的范例

回忆第7章，我们曾提到，不管应用程序多简单，都必须在向量表中包含下列三项：复位向量、NMI向量以及硬fault向量，这是因为后两者无需使能就可以发生。在程序运行后，有时还会把向量表重定位的SRAM中。下面就演示一种重定位的情况：把向量表转移到SRAM的起始处，并且在它的后面定义数据区——存储各种全局和静态变量。程序有点长，但很多部分以前都见过了，不要怕！

```
STACK_TOP      EQU 0x20002000      ; MSP初始值
NVIC_SETEN     EQU 0xE000E100      ; SETENA寄存器阵列的起始地址
NVIC_VECTTBL   EQU 0xE000ED08      ; 向量表偏移寄存器的地址
NVIC_AIRCR     EQU 0xE000ED0C      ; 应用程序中断及复位控制寄存器的地址
NVIC_IRQPRI    EQU 0xE000E400      ; 中断优先级寄存器阵列的起始地址

AREA | Header Code|, CODE
  DCD STACK_TOP          ; MSP初始值
  DCD Start              ; 复位向量
  DCD Nmi_Handler        ; NMI服务例程
  DCD Hf_Handler          ; 硬fault服务例程
```

```

ENTRY

Start           ; 主程序开始

; 初始化各寄存器
MOV    r0,    #0
MOV    r1,    #0
...
; 把各个向量拷贝到新向量表中
LDR    r0,    =0
LDR    r1,    =VectorTableBase
LDMIA  r0!,  {r2-r5}    ; 拷贝4个字 (MSP, Reset, NMI, 硬fault)
STMIA  r1!,  {r2-r5}
DSB               ; 数据同步隔离
; 执行向量表重定位:
LDR    r0,    =NVIC_VECTTBL
LDR    r1,    =VectorTableBase
STR    r1,    [r0]
...
; 设置优先级组寄存器, 划分抢占优先级与亚优先级
LDR    r0,    =NVIC_AIRCR
LDR    r1,    =0x05FA0500 ; 从位5处划分 (共2个位表达抢占优先级)
STR    R1,    [r0]
; 建立IRQ0的向量
MOV    r0,    #0          ; IRQ#0
LDR    r1,    =Irq0_Handler
BL    SetupIrqHandler
; 建立IRQ #0的优先级
LDR    r0,    =NVIC_IRQPRI
LDR    r1,    =0xC0          ; IRQ#0的优先级
STRB   r1,    [r0,#0]    ; 写入优先级寄存器中, 用了按字节传送
DSB               ; 数据同步隔离, 保证开中断前一切都已各就各位
MOV    r0,    #0          ; 选择IRQ #0
BL    EnableIRQ
...
;-----
; 各函数
SetupIrqHandler
; 入口条件: R0 = IRQ编号
; 入口条件: R1 = IRQ服务例程的入口地址
PUSH   {R0, R2, LR}
LDR    R2,    =NVIC_VECTTBL    ; 获取向量表的地址
LDR    R2,    [R2]
ADD    R0,    #16            ; 异常号 = IRQ编号 + 16

```

```

    LSL    R0,      R0,      #2      ; 乘以4 (每个向量4字节)
    ADD    R2,      R0          ; 找出向量地址
    STR    R1,      [R2]        ; 写入服务例程
    POP    {R0, R2, PC}      ; 返回

EnableIRQ
    ; 入口条件: R0=中断号
    PUSH   {R0-R2, LR}
    AND.W R1,      R0,      #0x1F      ; 为该IRQ产生移位量
    MOV    R2,      #1
    LSL    R2,      R2,      R1      ; 位旗标 = (0x1 << (N & 0x1F))
    AND.W R1,      R0,      #0xE0      ; 若IRQ编号>31则为它生成下标偏移量
    LSR    R1,      R1,      #3      ; 地址偏移量= (N/32)*4 (每个IRQ一个位)
    LDR    R0,      =NVIC_SETEN    ; 加载SETENA寄存器阵列的首地址
    STR    R2,      [R0, R1]      ; 写入该中断的位旗标, 从而使能该中断
    POP    {R0-R2, PC}      ; 子程返回
    ;-----
    ; 异常服务程序

Hf_Handler
    ...
    BX LR          ; 在此添加硬fault的处理代码

Nmi_Handler
    ...
    BX LR          ; 在此添加NMI的响应代码

Irq0_Handler
    ...
    BX LR          ; 在此添加IRQ #0的响应代码
    BX LR ; Return

    ;-----
AREA | Header Data|, DATA
ALIGN 4
    ; 重定位的向量表

VectorTableBase    SPACE   256      ; 保留256字节作向量表
VectorTableEnd      ; (256 / 4 = 最多支持64个异常)
MyData1    DCD    0          ; 定义变量
MyData2    DCD    0
END ; 文件结尾

```

这个例子是长了点, 让我们再从后往前看。在程序的尾部, 定义了数据存储区。通过SPACE汇编指示字, 我们为向量表开出了256字节的内存空间, 从而可以容纳64个异常向量。如果把256改成别的数, 就能改变向量表的长度。在向量表的后面, 还定义了两个变量。第一个变量MyData1紧挨着向量表, 所以它的地址是0x2000_0100, 第二个MyData2是为0x2000_0104。(不过, 通常情况下, 强烈反对使用这种以计算的方式来求得变量地址。因为很容易出错, 而且只要以后再新插入新的变量定义, 则所有插入位置后面的变量地址也都要重新计算, 因为它们被“拱”到后面去了——译者注)。

再看程序的起头，在那里我们一上来就定义了若干个地址常数（NVIC寄存器的地址），由整个程序使用。通过使用一个有意义的名字取代直接抄地址，程序就更容易理解，也减少了出错。

在初始的向量表中，包含了复位向量、**NMI**向量，以及硬**fault**向量，它们是三要素。后面的代码还给出了服务例程的骨架。在开发应用程序时，必须根据程序的指标来实现这三要素的服务例程，不可省略。

这里的服务例程都是使用**BX LR**返回的，但是真到了写程序时，往往利用**POP {...,PC}** 的形式来使程序更精练（当然也可以使用**LDMIA**指令）。

进入主程序后，先初始化寄存器，然后，就通过**LDM/STM**，把向量一次多个地拷贝到新的向量表中。如果后来又添加了新的向量，则可以在**LDM/STM**中增加数量，或者再多用一对**LDM/STM**，这些都是很简单的事。

在准备好了向量表后，就可以编程**NVIC**，启用新的向量表了。但是在启用前，为了保证在向量拷贝都完成后才做下一步，我们还用了**DSB**指令来隔离。

接下来继续做与中断设置相关的工作，第一个就是建立优先级组。

这些初始化都是一劳永逸的。本例中，使用了两个子程序来完成中断的建立，从而使程序结构更清晰。其中**SetupIrqHandler**负责在向量表建立中断服务例程的入口地址，而**EnableIRQ**则用于在**NVIC**中使能一个中断。在为一个中断建立好优先级后，就可以使能它。如果还需要除能中断，则可以照葫芦画瓢地就能当场造出一个**DisableIRQ**来，只是**SETENA**改成了**CLRENA**。

11.5 使用 SVC

SVC是用于呼叫**OS**所提供**API**的正道。用户程序只需知道传递给**OS**的参数，而不必知道各**API**函数的地址。

SVC指令带一个8位的立即数，可以视为是它的参数，被封装在指令本身中，如：

SVC 3 ; 呼叫3号系统服务

则3被封装在这个**SVC**指令中。因此在**SVC**服务例程中，需要读取本次触发**SVC**异常的**SVC**指令，并提取出8位立即数所在的位段，来判断系统调用号，工作流程如11.2所示：

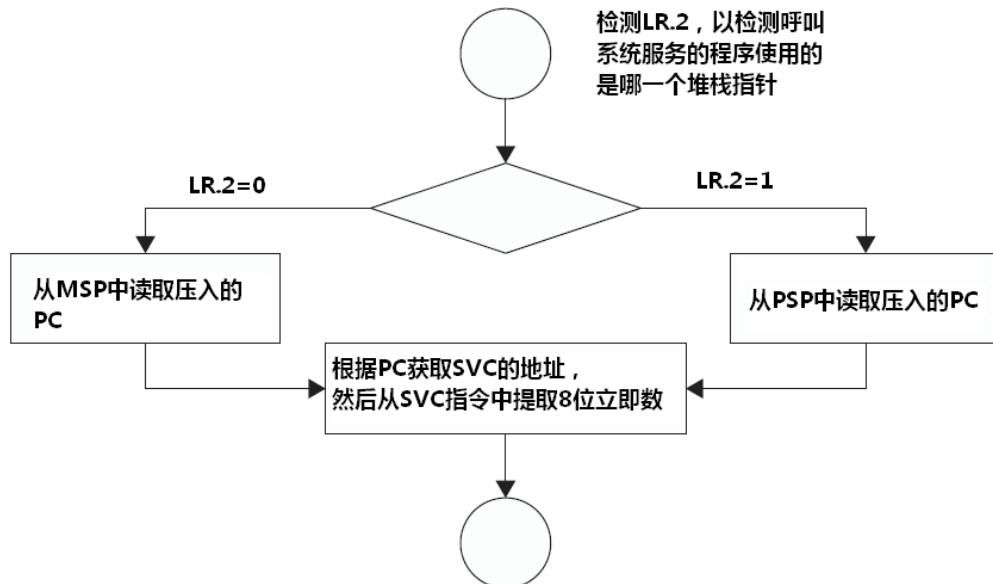


图11.2 提出SVC中立即数的一种途径

实现上图功能的代码如下所示：

```

svc_handler
    TST    LR,    #0x4          ; 测试EXC_RETURN的比特2
    ITE    EQ               ; 如果为0,
    MRSEQ  R0,    MSP          ; 则使用的是主堆栈, 故把MSP的值取出
    MRSNE  R0,    PSP          ; 否则, 使用的是进程堆栈, 故把MSP的值取出
    LDR    R1,    [R0,#24]      ; 从栈中读取PC的值
    LDRB   R0,    [R1,#-2]      ; 从SVC指令中读取立即数放到R0
    ; 准备调用系统服务函数。这需要适当调整入栈的PC的值以及LR(EXC_RETURN), 来进入OS内部
    BX    LR               ; 借异常返回的形式, 进入OS内部, 最终调用系统服务函数

```

一旦获取了调用号, 就可以用它来调用系统服务函数了。有理由相信, OS应该使用TBB/TBH查表跳转指令来加速定位正确的服务函数。然而, 如果你是设计OS的人, 必须检查这个参数的合法性, 以免因数字超出跳转表的范围而跳飞。

因为不能在SVC服务例程中嵌套使用SVC, 所以如果有需要, 就要直接调用SVC函数, 例如, 使用BL指令。

11.6 SVC 示范：用于输出函数

在前面的例子中, 我们写了若干个函数用于输出。但是有的时候, 可能有一些障碍, 使得我们不能用BL指令。例如, 需要调用的函数是在另外的目标文件中, 这就会导致有的时候我们无法定位子程序的入口地址; 另外, 如果跳转的目的地太远, 也有诸多不便; 或者, 当使用OS时, 这些输出函数已经被OS包装成系统调用了。在这些场合下, 我们就需要使用SVC来作为传送门, 如下面示例代码所示:

```

LDR    R0,    =HELLO_TXT
SVC    0          ; 请求显示字符串的系统服务。服务代号: 0
MOV    R0,    #'A'
SVC    1          ; 请求显示单一字符的系统服务。服务代号: 1
LDR    R0,    =0xC123456
SVC    2          ; 请求显示16进制数的系统服务。服务代号: 2
MOV    R0,    #1234
SVC    3 ; Display decimal value in R0

```

在使用SVC之前, 我们需要先建立SVC服务例程向量, 作法与建立IRQ的一样, 只是需要把异常号改为11。这一次, 通过巧妙地使用Thumb-2指令, 我们还可以进一步优化代码:

```

SetupExcHandler
    ; 入口条件: R0 = 异常号
    ; 入口条件: R1 = 异常服务例程
    PUSH   {R0, R2, LR}
    LDR    R2,    =NVIC_VECTTBL
    LDR    R2,    [R2]          ; 读取向量表的地址
    STR.W  R1,    [R2, R0, LSL #2] ; 表中[R2+R0<<2]的位置就是为该向量的
    POP    {R0, R2, PC}        ; 快速返回

```

对于SVC服务例程, 可以使用前面所述的方式提取服务代号。如果那些请求系统服务的程序还传递了其它参数(通过R0-R3), 则需找出正确的堆栈, 再从堆栈中, 读取进入SVC时自动压入的R0-R3值。

一个具体而微的SVC服务例程如下所示:

```

svc_handler
; 开始读取参数
TST    LR,    #0x4      ; 测试EXC_RETURN的比特2
ITE    EQ          ; 如果为0,
MRSEQ  R0,    MSP      ; 则使用的是主堆栈, 故把MSP的值取出
MRSNE  R0,    PSP      ; 否则, 使用的是进程堆栈, 故把PSP的值取出
LDR    R0,    [R1,#0]    ; 从堆栈中读取R0的值
LDR    R1,    [R1,#24]   ; 从堆栈中读取当时的PC
LDRB   R1,    [R1,#-2]   ; 提取SVC指令中的8位立即数
; 现在: R0存储了参数, R1存储了服务代号
PUSH   {LR}        ; 保护LR的值, 因为后面将使用的BL指令
CBNZ   R1,    svc_handler_1
BL     Puts        ; 调用Puts
B     svc_handler_end

svc_handler_1
CMP    R1,    #1
BNE   svc_handler_2
BL     Putc        ; 调用Putc
B     svc_handler_end

svc_handler_2
CMP    R1,    #2
BNE   svc_handler_3
BL     PutHex      ; 调用PutHex
B     svc_handler_end

svc_handler_3
CMP    R1,    #3
BNE   svc_handler_4
BL     PutDec      ; 调用PutDec
B     svc_handler_end

svc_handler_4
B     error        ; 未能识别的服务代号
...
svc_handler_end
POP   {PC} ; Return

```

译者添加: 事实上, 根据具体的系统, 不必总是教条主义, 以化简参数提取的工作。比如, 如果系统调用的参数不超过3个, 就可以把系统调用号存储到R0中, 把参数放到R1-R3中, 而省去提取服务号的操作。

上例中, 需要把`svc_handler`的代码与那些输出函数的放在一起, 以确保它们能在跳转的范围内。

细心的读者可能会问: 为什么不直接从R0-R3中读取参数, 却绕个大圈子从堆栈中读取呢, 它们不是一样的么? 原来, 这与晚到中断机制有关。仔细地想一想, 如果在入栈期间, 不巧来了另外的高优先级异常, 则会使后者的服务例程先执行。待返回后, 再以咬尾中断的方式执行SVC服务例程。我们知道, 咬尾处理时, 取消了前一个服务例程返回时的自动出栈动作。从而, 在发生了晚到+咬尾的情况下, 再执行SVC服务例程时, R0-R3已经被高优先级的服务例程用过了, 它们的值十

有八九被改过的。因此，必须从堆栈中读取。为帮助理解，在这里举一个具体的例子：

1. 用户程序把参数放到R0中，并执行SVC指令，请求系统服务
2. CM3为SVC开启了异常的响应序列，开始自动入栈，即把xPSR, PC, LR, R12, R3-R0压入堆栈
3. 入栈期间，来了一个高优先级的中断
4. 入栈完毕后，按晚到中断处理，先执行高优先级中断的服务例程。返回后，再以咬尾中断处理，此时，没有自动出栈的动作。
5. SVC服务例程以咬尾的方式开始执行。可见，此时的R0已经被高优先级服务例程用过了，不再保证是用户程序放入的参数。然而，先前入栈的R0-R3却依然保持不变（除非高优先级服务例程暗中使坏，篡改了堆栈的内容）

编程技巧：善用LDR/STR中的多种寻址方式

对比 `SetupIrqHandler` 和 `SetupExcpHandler` 的代码，我们可以看到，在 `SetupIrqHandler` 中，目标地址是用3条计算出来的，然后才使用存储指令。

而 `SetupExcpHandler` 就聪明多了，它通过对偏移寄存器做移位预处理，把计算地址巧妙地合并在存储指令的内部，使得本来3条指令做的事1条指令就搞定了。

这个小小的例子还给了我们另外的启示：学习时要求甚解，熟能生巧。CM3中有很多新指令，它们单独使用或者组合使用，能让温柔小女生的力气大增，蜕变成爱情女神。比如，CLZ与RBIT的组合使用，就快速地求得了芯片中表达优先级的位数。此外，它们还对“优先级位图调度算法”有决定性的化简意义（both时间上的和空间上的），有兴趣的读者可以拿它们去化简uC/OS-II中的调度函数，看看能不能去掉那个256字节的查找表。

11.7 在 C 中使用 SVC

如前所述，因为晚到中断的关系，SVC中不能再使用寄存器来传递参数，而是必须使用堆栈。因此，需要使用一段汇编代码来给SVC函数传参数。如果SVC服务例程的主体由C来写，则必须在前面伴随一个汇编写的封皮，用于把堆栈中的参数提取到寄存器中。下面给出一段代码来演示这个工作。这些代码是要使用ARM娘家的编译(`armcc`)和汇编(`armasm`)工具来处理的，RVDS和Keil RVMDK都使用这个工具链。

```
// 汇编封皮，用于提出堆栈帧的起始位置，并放到R0中，然后跳转至实际的SVC服务例程中
__asm void svc_handler_wrapper(void)
{
    IMPORT svc_handler
    TST    LR, #4
    ITE    EQ
    MRSEQ R0, MSP
    MRSNE R0, PSP
    B      svc_handler
}

// 不必写下BX LR来返回，而是由svc_handler来做决定
```

接下来的SVC服务例程的主体就可以由C来写了，它使用R0作为输入参数（这也是堆栈帧的起始位置），用于进一步提取服务代号，并且传递参数（通过堆栈中的R0-R3）。

恭喜呀！终于看到第一段C代码了（而且还是一段很另类的C程序哦）！

```

// 使用C写成的SVC服务例程，接受一个指针参数（pwdSF）：堆栈栈的起始地址。
// pwdSF[0] = R0 , pwdSF[1] = R1
// pwdSF[2] = R2 , pwdSF[3] = R3
// pwdSF[4] = R12, pwdSF[5] = LR
// pwdSF[6] = 返回地址（入栈的PC）
// pwdSF[7] = xPSR

unsigned long svc_handler(unsigned int* pwdSF)
{
    unsigned int svc_number;
    unsigned int svc_r0;
    unsigned int svc_r1;
    unsigned int svc_r2;
    unsigned int svc_r3;
    int retVal;          //用于存储返回值
    svc_number = ((char *) pwdSF[6])[-2]; // 没想到吧，C的数组能用得这么绝！
    svc_r0 = ((unsigned long) pwdSF[0]);
    svc_r1 = ((unsigned long) pwdSF[1]);
    svc_r2 = ((unsigned long) pwdSF[2]);
    svc_r3 = ((unsigned long) pwdSF[3]);
    printf ("SVC number = %xn", svc_number);
    printf ("SVC parameter 0 = %x\n", svc_r0);
    printf ("SVC parameter 1 = %x\n", svc_r1);
    printf ("SVC parameter 2 = %x\n", svc_r2);
    printf ("SVC parameter 3 = %x\n", svc_r3);
    //做一些工作，并且把返回值存储到retVal中
    pwdSF[0]=retVal;
    return 0;
}

```

注意，这个函数返回的其实不是0！进一步地，灰色的文字只是用于哄编译器开心的——让它认为这个函数是个有返回值的函数，而且确实返回一个数值了，于是不再吵闹着说有错或警告什么的。那返回的是啥？当然是retVal啦！有点迷糊么？那还不快往下看！

原来，SVC服务例程不能像普通的C函数那样——通过把原型声明为“**unsigned int func()**”，再在末尾来一句“**return xx;**”来返回。因为这种常规的作法在所有的ARM中其实是把返回值放到R0里。但是别忘了，这个函数可是异常服务例程，它的返回可是享受“异常返回”的待遇的——伴随着一个硬件控制的自动出栈行为，这会从堆栈中重建R0的值，从而覆盖“**return**”指定的值。因此，它必须把返回值写到堆栈中R0的位置，才能借自动出栈之机返回自己的值（**pwdSF[0]=retVal**）。

这下可真相大白了！虽然内部暗流汹涌，但是从应用程序的表面上看还是风平浪静——对于系统服务函数来说，这种独特的返回方式与普通的**return xx**效果是相同的，依然可以用普通的形式接收返回值。怎么样，这招够狠吧！其实，在写系统软件时，这根本算不上耍狠，只不过是寻常的基本功罢了，要不然怎么说C是“低级高级语言”呢。而病毒/木马所采用的“堆栈/缓冲区溢出攻击”，那才算真正的狠招呢，但是它们原理是一脉相承的。可见，对底层理解得深刻，能让我们写出更好，更强大的程序来。

在RVDS和Keil RVMDK中，为了方便我们放参数，提供了“`__svc`”编译器指示字。举例来说，如果需要在3号服务请求中传递4个参数，则可以类似下例写：

```
unsigned long __svc(0x03) CallSvc3(unsigned long svc_r0, unsigned long
svc_r1, unsigned long svc_r2, unsigned long svc_r3);
```

当C程序调用这种函数时，则编译器会自动生成SVC指令，如下所示：

```
int Func(void)
{
    unsigned long p0, p1, p2, p3; //传递给SVC服务例程的4个参数
    unsigned long svcRet; //系统服务的返回值
    ...
    svcRet=CallSvc3(p0, p1, p2, p3); // 呼叫3号系统服务，并且传递4个参数，依次为：p1,p2,p3,p4，再接收返回值到svcRet中（别忘了，这个返回值的来历不寻常）
    ...
    return;
}
```

如欲获知`__svc`的官方说明，可以查阅《RVCT 3.0 Compiler and Library Guide(Ref6)》。

如果使用的是GNU的工具链，里面没有`__svc`关键字。但是GCC支持内联汇编，可以实现此功能。例如，如果需要呼叫3号系统服务，同时传递一个参数，还接收一个返回值（两者都通过R0），则可以使用如下的内联汇编来呼叫SVC：

```
int MyDataIn = 0x123;
__asm __volatile ("mov R0, %0\n"
                  "svc 3 \n" : "" : "r" (MyDataIn) );
```

上段内联汇编码中，两个“`:`”后面分别对应输入数据——由`r(MyDataIn)`指定，以及输出数据——即上段代码中是“`%`”，语法模式如下所示：

```
__asm ( assembler_code : output_list : input_list )
```

在第19章中，给出了使用GNU工具链的更多汇编例子。如欲获取有关内联汇编的详细信息，还请参阅GNU工具链的说明文档。

编程进阶与系统行为

- 在系统中使用双堆栈
- 双字的堆栈对齐方式
- 非基级的线程模式
- 性能评估
- 当处理器被锁定时

阅读理解本章，需要有操作系统的基本概念：定义、作用及地位

12.1 在系统中使用双堆栈

CM3 的出现，让单片机业界也能出双枪李向阳。**v7-M** 架构的一个重要能力，就是提供了这个双堆栈的设计，允许把用户应用程序的堆栈与特权级/操作系统内核(**kernel**)的堆栈分开。如果再辅以 **MPU**，还能进一步地阻止用户程序访问内核的堆栈，同时也消除了内核数据被破坏的可能。

要在 CM3 中创建可靠扛打的系统，必须两手抓，两手都要硬。典型情况下，一个真正健壮的 CM3 软件系统是要使用实时操作系统内核的，通常会符合如下的要求：

- 服务例程使用 **MSP**（在“非基级线程模式”中会讲到例外情况）
- 尽管异常服务例程使用 **MSP**，但是它们在形式上返回后，内容上却可以依然继续——而且此时还能使用 **PSP**，从而实现“可抢占的系统调用”，大幅提高实时性能
- 通过 **SysTick**，实时内核的代码每隔固定时间都被调用一次，运行在特权级水平上，负责任务的调度、任务时间管理以及其它系统例行维护
- 用户应用程序以线程的形式运行，使用 **PSP**，并且在用户级下运行
- 内核在执行关键部位的代码时，使用 **MSP**，并且在辅以 **MPU** 时，**MSP** 对应的堆栈只允许特权级访问

如图 12.1 所示，假设系统内存是一块 **SRAM**，则我们可以通过 **MPU**，把它分为两个 **regions**，其中一个用于用户级，另一个用于特权级。另外别忘了 CM3 的堆栈是“向下生长的满栈”，因此需要把这两个 **SP** 初始为指向这两个 **regions** 的顶端。

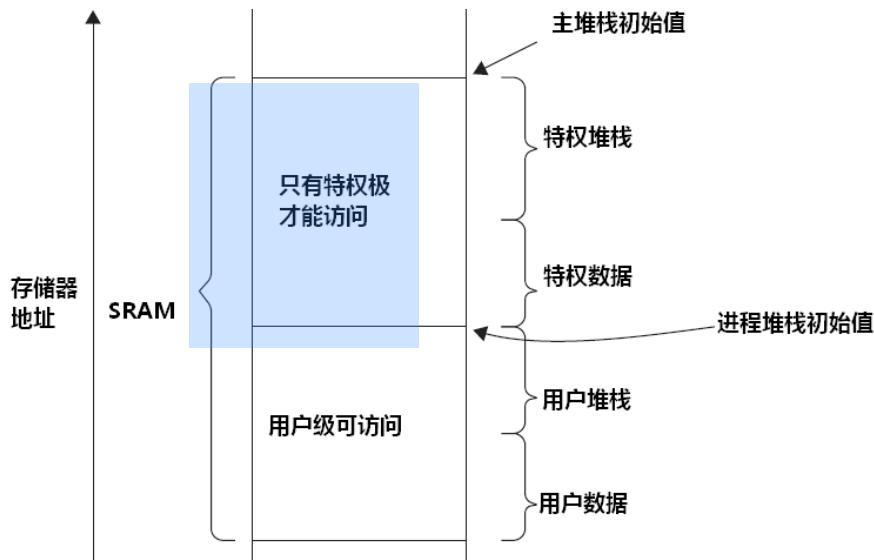


图 12.1 用户级/特权级下的内存配置示范

上电后，通过从向量表中取出 0 号向量，仅初始化了 MSP。因此，需要额外的工作来建立完整的双堆栈系统。对于使用汇编写成的代码，只需寥寥几句：

; 这段代码在用户可访问内存中，但从特权级开始执行

```

BL      MpuSetup           ; 建立MPU regions，并使能存储器保护
LDR    R0,     =PSP_TOP    ; 读取进程堆栈的栈顶
MSR    PSP,    R0          ; 并用它来初始化进程堆栈
BL      SystickSetup       ; 配置SysTick，并建立SysTick异常向量，供os日后使用
MOV    R0,    #0x3          ; 设置CONTROL寄存器，让用户程序使用PSP
MSR    CONTROL,R0         ; 并且切入用户级
B      UserAppStart        ; 到了这里已经进入了用户级，开始跳入用户程序入口

```

这个函数最好用汇编写。如果非要用 C，则会破坏 C 函数的堆栈帧：因为 C 函数常常把多出来的局部变量放到堆栈中，所以在切换堆栈指针时，函数的局部变量可能丢失。在 Cortex-M3 TRM (Ref1) 中，已经做出明确建议：使用形如 SVC 的 ISR 来调用内核，然后通过修改 EXC_RETURN 的值来切换堆栈指针。

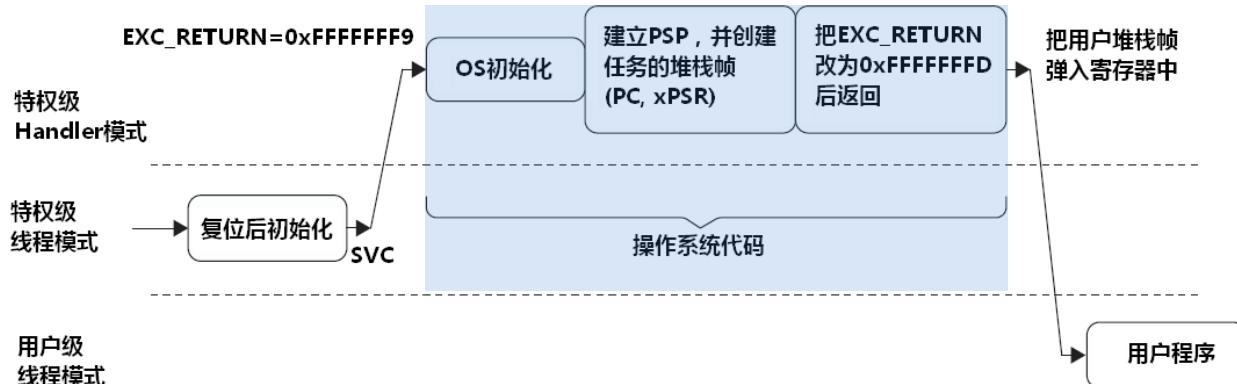


图 12.2 简单 OS 中的堆栈初始化

在操作系统中,对于 `EXC_RETURN=0xFFFFFFF` 的修改,只是再寻常不过基本需求。在开始调度用户程序后,一定还伴随着 `SysTick` 异常,它周期性地把执行权转入操作系统,从而使例行的系统管理以及必要轮转调度得以维持——差不多就是系统的心跳吧,如图 12.3 所示:



图 12.3 SysTick 异常推动简单轮转调度模式图

在这里,使用 `PendSV` (一个优先级最低的异常) 来执行上下文切换,从而消灭了在中断服务例程中出现上下文切换的可能,读者应该对此还记忆犹新吧。

然而,也有不少的程序不需要上操作系统。即便如此,使用两个栈也依然对于提升程序的可靠性大有用场。其中一个可行的方案是,以 `MSP` 启动 `CM3` 时,把 `MSP` 初始化成某进程的堆栈 (`process stack`)。这样,就可以使初始化代码使用进程堆栈中运行 (虽然还使用 `MSP`)。在正式执行应用程序前,先执行如下的初始化代码:

```
; 从特权级启动, MSP指向一个用户程序的堆栈
MpuSetup();           // 建立MPU regions并使能存储器保护
SystickSetup();       // 建立SysTick异常向量,由其服务例程作为时基的管理
SwitchStackPointer(); // 呼叫一个汇编程序来切换到PSP

/*; -----在 SwitchStackPointer 中-----
PUSH {R0, R1, LR}
MRS R0, MSP          ; 读取MSP到R0,稍后使用
LDR R1, =MSP_TOP
MSR MSP, R1          ; 让MSP指向新的MSP_TOP
MSR PSP, R0          ; 把当前的MSP存储到PSP中
MOV R0, #0x3
MSR CONTROL,R0       ; 切换到用户级,并使用PSP指向的堆栈作为当前堆栈
POP {R0, R1, PC}
; -----回到C程序中 -----
; 现在已经进入了用户级,使用PSP,并且没有弄丢局部变量
UserApplicationStart(); // 在用户级下开始执应用程序
```

12.2 双字的堆栈对齐方式

在符合 `AAPCS` 的应用程序中,对于响应异常时的堆栈操作,是有必要对齐到原始 (`primitive`) 的数据尺寸的 (1, 2, 4 或 8 字节)。这是 `CM3` 的一个可配置选项。欲使能此特性,需要把 `NVIC` 配置控制寄存器的 `STKALIGN` 置位 (在附录 D 的表 D.17 给出定义),如下面汇编代码所演示:

```

LDR    R0,      =0xE000ED14      ; R0=NVIC_CCR的地址
LDR    R1,      [R0]
ORR.W R1,      R1,      #0x200  ; 设置STKALIGN位
STR    R1,      [R0]      ; 更新NVIC_CCR

```

如果使用 C 语言，则代码如下：

```

#define NVIC_CCR ((volatile unsigned long *)(0xE000ED14))
*NVIC_CCR= *NVIC_CCR | 0x200;      /* 设置STKALIGN位*/

```

如果在入栈时 **STKALIGN** 位为 1，则 **xPSR** 的位 9 功能启用，指示在入栈时 **SP** 的值是否为了对齐而作出了调整。在出栈时，会检查入栈的 **xPSR.9**，再根据它的值把 **SP** 的值调整回去。

注意：切勿在异常服务例程中改动 **STKALIGN** 位的值，否则会使出栈时数据发生错位，彻底破坏各个寄存器的值，这常常是致命错误（跑飞，死机等）。

要注意的是，这个特性是在 **CM3** 修订版 1 开始后才引入的，早期基于版本 0 的产品则无此功能。当需要符合 **AAPCS** 时，需要启用此特性。此外，当程序的一部分是使用 C 开发，且程序中包含了对双字数据的处理时，也推荐启用此功能。

在最新的修订版 2 中，该特性不需手动使能，而是在缺省时已使能。在使用 C 开发时，如果程序包含了需要双字尺寸的数据类型（**double**, **long long** / **INT64**）时，推荐使能此特性。

12.3 非基级的线程模式

在 **CM3** 中，原则上异常服务程序要在 **handler** 模式下执行，但是也允许在服务例程中切换到线程模式。通过设置 **NVIC** 配置与控制寄存器的“非基级线程模式允许”位（**NONBASETHRDENA**，位偏移：0），可以在服务例程中把处理器切换入线程模式。为什么要这么做？如果中断服务例程是用户程序的一部分，可能需要让它在线程模式下执行，以限制它访问特权级下的资源，此时可以让此功能派上用场（对于让 **CM3** 在线程模式下赋予用户级访问权限的配置，不在本节中完成，而是在上电初始化时就一次性地做好了——译者注）。

小心地使用此功能

如果使用此功能，则需要手工调整堆栈指针，还要重建堆栈中的数据。这种乾坤大挪移可是高度危险的作业，一不小心就很容易把整个系统弄垮。所以必须格外严肃地对待。另外，在使用时，系统设计者还必须保证服务例程能正确地返回。因为在线程模式下是不允许作中断返回的，所以必须用一点手腕才行。如果放任不管，则中断无法退出，这会永远阻塞其它同级和更低优先级中断。通常，由系统软件负责完成这种工作。

在启用本功能时，必须伴随着一个“服务例程重定向”动作：中断向量指向一个运行在特权级的服务例程，但它却是应该只访问用户级内存的，因此必须先在头部切入用户级，调用真正干活的服务例程，再在最后回到特权级。演示代码如下所示：

```

redirect_handler
PUSH    {LR}
SVC    #0          ; 呼叫系统服务，用于把特权级别改为用户级
BL     User IRQ Handler
SVC    #1          ; 执行完中断处理后，回到特权级
POP    {PC}        ; 启动本次中断的返回序列

```

上例中，字体不同的中间两行是在线程模式+PSP 下执行的。在这段代码中，通过首尾的两个系统调用来完成乾坤大挪移：

使用 **SVC #0**，它先使能非基级线程模式，再拷贝主堆栈中被压入的 8 个寄存器到进程堆栈并更新 **PSP** 的值，最后修改 **EXC_RETURN**，以使返回后进入“线程模式.用户级+PSP 堆栈”

使用 **SVC #1** 来使一切归位，它除能非基级模式，恢复 **PSP** 先前的位置，并且修改 **EXC_RETURN** 以返回到特权级，继续使用主堆栈。

在最后执行到返回指令后，则终结了本次异常处理序列。虽然 **redirect_handler** 的内部有这么多的暗箱操作，但是在表面上看还是很傻很天真的，也就 5 行“安分守己”的指令而已。

听起来很神吧，那就让我们把这个内幕曝光。这可是一道大荤菜，可以尝尝系统程序大肉的味道。这个菜是很“油”的，最好边吃边看图 12.4 来帮助消化。使用上一章讲到的 **SVC** 服务例程框架，在这里搭成了真正能干活的系统服务：

```

svc_handler
; 小测试：请读者为本段代码加注释
TST    LR,    #0x4          ; 测试EXC_RETURN.2
ITE    EQ               ; 如果为零则
MRSEQ  R0,    MSP          ; 先前使用的是主堆栈，把MSP的值加载到R0
MRSNE  R0,    PSP          ; 否则，先前使用的是进程堆栈，把PSP的值加载到R0
LDR    R1,    [R0, #24]     ; 读取入栈的返回值
LDRB   R0,    [R1, #-2]     ; 提出8位立即数调用代号
CBZ    r0,    svc_service_0
CMP    r0,    #1
BEQ    svc_service_1
B.W    Unknown_SVC_Request

svc_service_0          ; 0号服务：切换到“线程模式+PSP”
MRS    R0,    PSP          ; 读取PSP
SUB   R0,    R0,    #0x20    ; 开出32字节的空间存储8个寄存器
MSR    PSP,    R0          ; 更新PSP的值
MOV    R1,    #8*4         ; R1作为拷贝堆栈帧（8个寄存器）的循环变量
svc_service_0_copy_loop
SUBS   R1,    R1,    #1*4
LDR    R2,    [SP, R1]
STR    R2,    [R0, R1]
CMP    R1,    #0
BNE    svc_service_0_copy_loop

STRB   R1,    [R0, #0x1C]   ; 在进程堆栈中清零IPSR
LDR    R0,    =0xE000ED14  ; 加载NVIC中CCR（配置与控制寄存器）的地址
LDR    r1,    [r0]
ORR    r1,    #1
STR    r1,    [r0]          ; 使能非基级线程模式（这里的地址不在位带操作区）
ORR    LR,    #0xC          ; 修改EXC_RETURN，以使得返回后进入 线程模式+PSP
BX    LR                  ; 启动异常返回序列，执行动作

svc_service_1          ; 1号服务：从线程模式+PSP返回到handler模式
MRS    R0,    PSP          ; 读取PSP到R0，以便于后续的一系列归位处理
LDR    R1,    [R0, #0x18]    ; 读取压入PSP中的返回地址（即svc #1后面的
                           ; ; POP {PC}）
STR    R1,    [SP, #0x18]    ; 因为将要返回到handler模式，所以把它转移到MSP
ADD    R0,    R0,    #0x20    ; 把PSP的值归位——刚响应外部中断时的值
MSR    PSP,    R0          ; 用归位后的值更新PSP

```

```

LDR    R0,      =0xE000ED14    ; NVIC中配置与控制寄存器（CCR）的地址
LDR    r1,      [r0]          ; 再次读取NVIC中的CCR
BIC    r1,      #1           ; 清除NONBASETHRDENA位,
STR    r1,      [r0]          ; 修改EXC_RETURN以返回handler模式, MSP亦归位
BIC    LR,      #0xC          ; 修改EXC_RETURN以返回handler模式, MSP亦归位
BX    LR

```

使用 **SVC** 是必须的，因为只有通过异常返回才能改动 **IPSR** 的值。软件触发中断也能用，但那种偏方是旁门邪道，因为它是不精确的，而且可能被阻塞（回顾前一章），带来了隐患——使得堆栈拷贝与切换操作不被立即执行。

图 12.4 给出了上述代码的工作序列图，如果吃不消的话就赶快看吧：

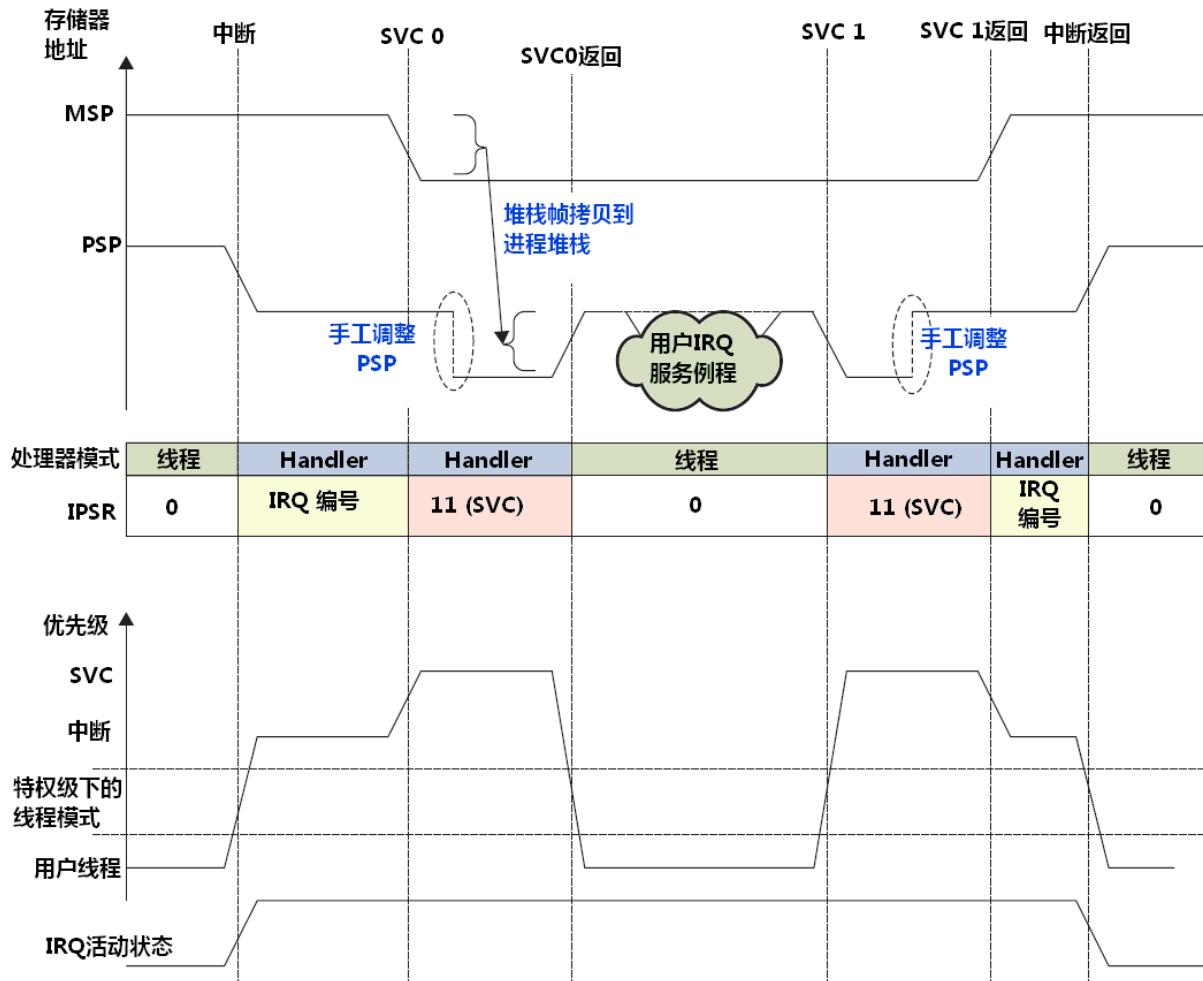


图 12.4 非基级线程模式操作模式图

手工调整PSP也是必须的。如果没有第一次调整，则在借SVC0返回的形式进入用户IRQ服务例程后，会使PSP回到进入中断前的状态。然后在执行“svc #1”时，将重新把寄存器压入栈——但此时的寄存器已经是被用户IRQ服务例程用过的了！结果，虽然PSP的值与两次调整后的还相同，但是PSP中寄存器内容已经被破坏了！

对MSP的调整也是很有魄力的，它突破了嵌套的异常在返回时，一定要从MSP出栈的教条。这段代码中对MSP和PSP的把玩，是不是很精湛？在真实的操作系统中，还有更刺激的动作。

12.4 性能评估

为了让CM3能尽情地释放她的青春能量，还需要我们清扫路上其它石子。

第一，要消灭存储器等待周期。在MCU/SoC的设计期间，就应该优化存储器系统，最起码的要求，也要允许取指和数据访问能并行不悖，这才对得起“哈佛结构”的称号。此外，应尽可能地使用32位的存储器。对于软件开发人员，还应该划清代码与数据的界线，使得程序代码从代码区执行（使用I-Code总线），而绝大部分数据都从数据区访问（使用System总线，而不要使用D-Code总线），哪怕是多浪费点内存。只有这样，才能使取指与访问数据同时进行。

第二，如果没有必要，中断向量表也放到代码区中。只有这样，才能使取向量(I-Code总线)与入栈(System总线)同时进行。如果向量表在RAM中，就会出现取向量与入栈抢总线的情况，必然导致额外的中断延迟被引入（当然在极个别情况下，如果把SRAM放到Code区，则使用D-Code总线入栈。但如果就为了放向量表而专配一个SRAM，代价未免也太大了）。

第三、限制使用非对齐访问。前面讲到，CM3总线内部其实只接受对齐访问，而由总线接口来堵窟窿：把一个非对齐的访问拆成若干个对齐的访问，来实现这种透明性。可见，一次非对齐访问可能要数次对齐访问才能完成（最坏情况下3次）。而且节省内存的正道，在于优良的程序结构和算法设计，从来不在这种见缝插针地乱挤上。除非是客观上被定死的（常见于某些早期网络协议的报文头部），否则应在心里暗下决心：决不染指非对齐访问，在设计数据结构及定义变量时，都高度自觉。在ARM汇编器中，提供了ALIGN指示字（GNU AS中也有类似的汇率器指示字），可以保证产生所需的对齐方式。

虽然我们会在绝大多数场合下使用C来开发，但是在为某个关键的功能启动“汇编级待遇”时，不要忘了使用下述的技巧，它们经常能产生意想不到的特效：

1. 使用带偏移量寻址的LDR/STR指令，进一步地，还可以对偏移量作多位预处理（LSL用得最多）。使用这种强大的寻址方式，常常能省去分立的地址增减/乘除计算操作。重温一下上一章中使能中断和使能异常子程的不同，相信会有切身的体会
2. 把上下文相关的变量放到一起——也就是说使它的地址是连续的。这样就可以创造使用LDM/STM指令的机会。只要遇到连续地址的数据传送，就使用LDM/STM。一条传送14个字的LDM指令，可远比14个LDR要快多了，而且代码也巨幅精简
3. 当遇到很小的“if then”块时，如果使用条件跳转指令，则会使流水线被清洗，花不少时间。这时，应使用IF-THEN指令（ITxxx）。IT指令在张开双臂时，最多能保护4个孩子。
4. 如果旧时需要两条Thumb指令才能完成的操作，现在可以由一条Thumb-2指令完成，则应使用Thumb-2指令。
5. 为使自己成长为大虾，要学会使用CM3的新好指令。尤其是在ARMv6后才新出来的，都是无数前人经验的结晶，常常能有戏剧般地优化（回顾RBIT与CLZ的梦幻组合）

12.5 当处理器被锁定(Lockup)时

这确实是很扎手的问题：本来就已经因为出错而进入fault服务例程了，结果fault服务例程也触犯了fault条件，升级为硬fault的。可如果硬fault服务例程也脑子进水了怎么办？一错再错，最终使CM3在万般无奈下进入锁定状态。万万要避免它，因为一旦锁定就不可救要了——几乎只能复位，这在使命-关键（mission-critical）系统中是决不允许的（像那种大型交换机、体外循环机等设备）。

12.5.1 锁定情形下的众生相

在锁定下，寄存器和存储器都被“冻结”，PC的值被强制为0xFFFF_FFFx，并且原地打转地定死在那里一直取指。与此同时，CM3的另一条名叫“LOCKUP”的输出信号线将被置为有效，芯片厂商可以检测此信号，并且在系统复位发生器上触发一个复位。

具体地说，下列场合会导致锁定：

- 在硬fault服务例程中产生faults（双重fault）
- 在NMI服务例程中产生faults
- 在复位序列（初始的MSP与PC读取）中产生总线fault

在双重fault下，NMI还能响应（再次证明了它的第一优先地位）。然而在NMI服务例程退出后，又回到锁定状态。此时，当前优先级为-1，因此可以响应NMI——NMI的优先级是-2，比当前的高。

在产品中出现锁定就等同于大限已到。但是在调试阶段也许还能让系统起死回生：如果连接了调试器，则可以喊停（halt）处理器，然后手工修改PC的值。然而这也往往是无力的：因为上下文没有了——所有的寄存器，以及中断系统，都已经物是人非，需要重新初始化，才能返回到正常的操作中。

那为什么不直接复位，好让它早点在下个轮回中转世投胎呢？原来，系统的生命是开发者赋予的，因此就要对它的生死负责。哪怕死了，也要明白死因才行。如果当场就复位了，则所有寄存器的值都归位了，不再有机会去查明当时的情况。

如果不是使命-关键系统，则可以使用一个看门狗，它可以使系统从锁定状态中复位。

还要注意的是，如果在响应NMI或硬fault的入栈/出栈阶段触发了总线fault，则不会导致锁定，只是会悬起总线fault，如图12.5所示。

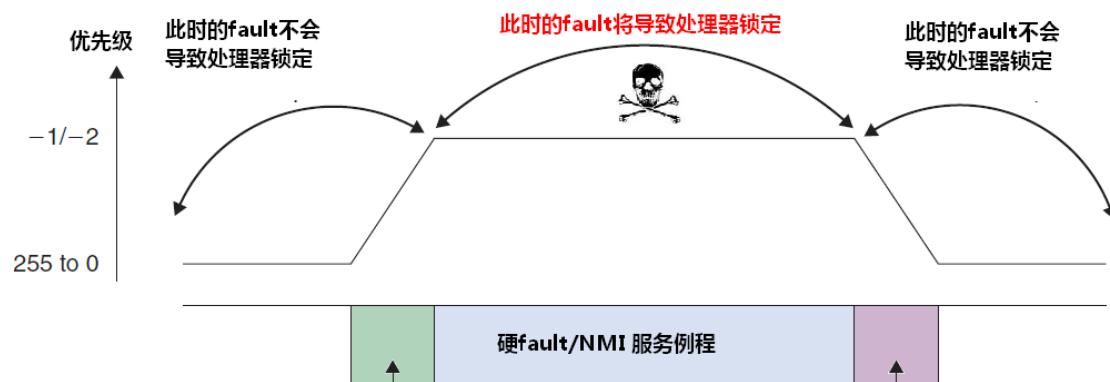


图12.5 只有在硬fault/NMI 服务例程中的fault才锁定系统

12.5.2 避免被锁定

既然被锁定就等同于死机，我们唯一能做的也只能是避免锁定状态。因为锁定只出现于NMI和硬fault的服务例程中，所以当我们在设计它们时，一定要分外地小心，就好像亲手给自己的爱人做大手术那样地一丝不苟。比如，我们应该尽量避免不必要的堆栈访问，这是有原因的。对于NMI来说，因为在进入NMI时常常是在危急关头，如：掉电，短路等硬件故障。此时，有可能存储系统已经失能了。而对于硬fault来说，有可能就是因为SP指针指飞了（干扰、堆栈溢出等），以致前面的堆栈操作触发了本次硬fault，再操作堆栈还不当场被秒杀？如下面代码所警示：

hard_fault_handler

PUSH {R4-R7,LR} ; 除非确保堆栈是安全可用的（谁能确保？），否则不要这样做

...

值此危难关头，必须沉着冷静。在我们设计硬fault，总线fault以及存储管理fault的服务例程时，

值得先花点工夫去查一查**SP**的值，看它是否在可接受的范围，然后再做后续工作。对于**NMI**服务例程来说，它做的通常是应急工作，设计系统时就应该让这种应急工作极简单（比如，只改变一个I/O脚的电平，最多也就是修改若干寄存器的值，就可以开启相关的应急硬件——译者注），因此常常可以只使用**R0-R3**以及**R12**就完全够用，无需堆栈操作。

简化硬**fault**和**NMI**的服务例程确实是个好主意：它们只做必需的，然后悬起**PendSV**，让诸如错误报告等其它工作在**PendSV**中处理，当然，软件中断兴许也能凑和着用。

除此之外，我们还必须杜绝在硬**NMI/fault**例程中使用**SVC**指令，这也是斩立决的——因为**SVC**的优先级总是没有**NMI**和硬**fault**的高，而且它又不允许悬起（悬起时触发**fault**）。这看起来很容易做到，那是饱汉不知道饿汉饥——当程序变得复杂，并且如果**NMI/硬fault**服务例程中调用了其它目标文件中的函数，就不能保证这些函数中没有使用过**SVC**。因此，在开发软件时，必须仔细地计划如何实现**SVC**。或者获取所调用函数的说明文档，确保不会出事。

Cortex-M3 的其它特性

- SysTick 定时器
- 电源管理
- 多处理器通信
- 自复位控制

到了这里，我们已经学完了 CM3 的绝大多数重要和基础的特性，再加一把劲儿，这章不难，过了以后就到了一个里程碑了。

13.1 SysTick 定时器

回顾第 8 章讲述 NVIC 时，曾走马观花地带过了 SysTick 定时器。复习一下：SysTick 是一个 24 位的倒计数定时器，当计到 0 时，将从 RELOAD 寄存器中自动重装载定时初值。只要不把它在 SysTick 控制及状态寄存器中的使能位清除，就永不停息。图 13.1 中小结了 SysTick 的相关寄存器。

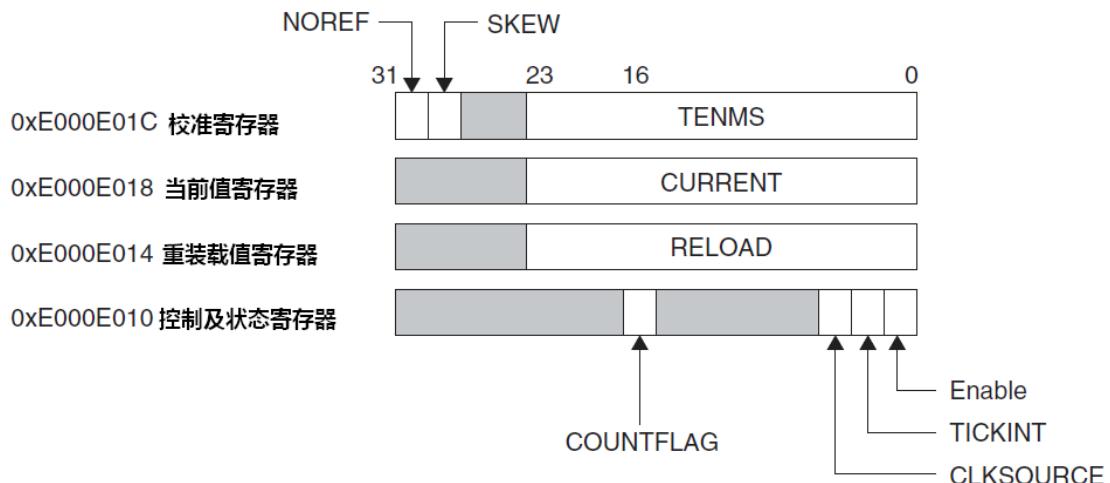


图 13.1 SysTick 相关寄存器的定义

CM3 允许为 SysTick 提供两个时钟源以供选择。第一个是内核的“自由运行时钟” FCLK。“自由”表现在它不来自系统时钟 HCLK，因此在系统时钟停止时 FCLK 也继续运行。第二个是一个外部的参考时钟。但是使用外部时钟时，因为它在内部是通过 FCLK 来采样的，因此其周期必须至少是 FCLK 的两倍（采样定理）。很多情况下芯片厂商都会忽略此外部参考时钟，因此通常不可用。通过检查校准寄存器的位[31](NOREF)，可以判定是否有可用的外部时钟源，而芯片厂商则必须把该引线连接至正确的电平。

当 SysTick 定时器从 1 计到 0 时，它将把 COUNTFLAG 位置位；而下述方法可以清零之：

读取 SysTick 控制及状态寄存器 (STCSR)

往 SysTick 当前值寄存器 (STCVR) 中写任何数据

SysTick 的最大使命，就是定期地产生异常请求，作为系统的时基。OS 都需要这种“滴答”来

推动任务和时间的管理。如欲使能 **SysTick** 异常，则把 **STCSR.TICKINT** 置位。另外，如果把向量表重定位到了 **SRAM** 中，还需要为 **SysTick** 异常建立向量，提供其服务例程的入口地址，如下段代码所演示：

; 建立**SysTick**异常服务例程

```
MOV    R0,    #0xF          ; 异常号: 15
LDR    R1,    =systick_handler ; 加载服务例程的入口地址
LDR    R2,    =0xE000ED08    ; 加载向量表偏移量寄存器的地址
LDR    R2,    [R2]          ; 读取向量表的首地址
STR    R1,    [R2, R0, LSL #2] ; 写入向量
```

下面的代码演示启用 **SysTick** 的基本程序

; 使能**SysTick**定时器，并且使能**SysTick**异常

```
LDR    R0,    =0xE000E010    ; 加载STCSR的地址
MOV    R1,    #0
STR    R1,    [R0]          ; 先停止SysTick，以防意外产生异常请求
LDR    R1,    =0x3FF        ; 让SysTick每1024周期计完一次。因为是从1023数到
                           ; 0，总共数了1024个周期，所以加载值为0x3FF
STR    R1,    [R0,#4]       ; 写入重装载的值
STR    R1,    [R0,#8]       ; 往STCVR中写任意的数，以确保清除COUNTFLAG标志
MOV    R1,    #0x7          ; 选择FCLK作为时钟源，并使能SysTick及其异常请求
STR    R1,    [R0]          ; 写入数值，开启定时器
```

除此之外，**SysTick** 定时器还提供了走完 **10ms** 所需要的格数(**TENMS** 位段)，作为时间校准的参考信息。在 **CM3** 处理器的顶层有一个 **24** 位的输入，芯片厂商可以写入一个 **10ms** 的加载值，写程序时就可以读取 **STCR** 寄存器中的 **TENMS** 位段来获取此信息。不一定每个芯片都实现了此功能，因此在使用时还需查阅芯片的数据手册。

SysTick 定时器还可以用作闹钟，作为启动一个特定任务的时间依据。例如，如果需要在 **300** 周期后执行一段代码，就可以在 **SysTick** 异常服务例程中设置执行那段代码的软件标志。使用 **SysTick** 时，清零 **CURRENT** 再编程 **RELOAD** 寄存器，以使它在 **300** 周期后产生异常，如下述代码所演示：

```
LDR    R0,    =15
LDR    R1,    =SysTickAlarm ; SysTick异常服务例程为SetupExcpHanler
BL     SetupExcpHandler   ; 调用前面章节讲到的子程来建立向量
LDR    R0,    =0xE000E010    ; SysTick寄存器组的基地址
MOV    R1,    #0          ; 编程前先除能SysTick
STR    R1,    [R0]
STR    R1,    [R0,#0x8]    ; 清零CURRENT
LDR    R1,    =(300-12)    ; 设置装载值。减去12是为了补偿中延迟
STR    R1,    [R0,#0x4]    ; 写入RELOAD
LDR    R4,    =SysTickFired ; 在RAM中的一个变量，指示是计时到期
MOV    R5,    #0          ; 初始为0
STR    R5,    [R4]
MOV    R1,    #0x7          ; 使用FCLK，使能SysTick，使能SysTick异常
STR    R1,    [R0]          ; 启动计时
LDR    R4,    =SysTickFired
```

WaitLoop

```

LDR    R5,    [R4]          ; 循环查询软件标志
CMP    R5,    #0
BEQ    WaitLoop
...
; SysTickFired在服务例程中被置位，主程序可以继续执行

```

本例中使用以前讲到的 `SetupExcpHandler` 来建立向量表，但注意：必须重定位向量表到 RAM 中才行。

SetupExcpHandler

```

; 入口条件: R0 = 异常号
; 入口条件: R1 = 异常服务例程
PUSH   {R0, R2, LR}
LDR    R2,    =NVIC_VECTTBL
LDR    R2,    [R2]          ; 读取向量表的地址
STR.W  R1,    [R2, R0, LSL #2] ; 表中[R2+R0<<2]的位置就是为该向量的
POP    {R0, R2, PC}         ; 快速返回

```

因为计数器是从 0 开始计数的，所以它会立即把 **300-12** 加载入 **CURRENT**。**12** 是中断响应的最短延时，因此减去它用以补偿。但是如果有更高优先级的异常抢占或者阻塞了它，则中断延迟还是会有的。

另外要注意的，减去 **12** 只适用于一次性(**one shot**)的闹钟操作，在这种情况下必须在 `SysTick` 服务例程中按停这个 `SysTick`。进一步地，如果其它异常把它延迟得太久，就有可能会使 `SysTick` 异常被悬起两次。因此，对于单次处理时，还需要其它一些步骤来消灭二次触发：

SysTickAlarm ; SYSTICK exception handler

```

PUSH   {LR}
LDR    R0,    =0xE000E010      ; SYSTICK寄存器组的基地址
MOV    R1,    #0
STR    R1,    [R0]          ; 除能SysTick，因为只使用一次
LDR    R0,    =0xE000ED04
LDR    R1,    =0x02000000      ; 手工清除NVIC中的SysTick悬起位
STR    R1,    [R0]
...
; 执行所需的处理工作
LDR    R2,    =SysTickFired
LDR    R1,    [R2]
ORR    R1,    #1
STR R1, [R2]                  ; 设置软件标志，与主程序同步，以执行任务
POP    {PC}                   ; 异常返回

```

在服务例程的末尾处，通过设置 `SysTickFired` 标志，通知主程序定时已经到期，可以结束循环等待了。

13.2 电源管理

不同于以往的处理器，CM3 对电源管理的重视，已经上升到处理器内核的水平上。它提供了若两种睡眠模式。在睡眠时，可以停止系统时钟，但可以让 FCLK 继续走，以允许处理器能被 `SysTick` 异常唤醒。这两种睡眠模式依次为：

睡眠：由 CM3 处理器的 `SLEEPING` 信号指示

深度睡眠：由 CM3 处理器的 SLEEPDEEP 信号指示

为了判定当前使用的是哪一种睡眠模式，以及其它睡眠时的上下文，需要检视在 NVIC 的系统控制寄存器，如表 13.1 所示。要注意，CM3 的这两条信号线是给芯片设计者看的，需要芯片设计者配合它们作一系列的处理，因此不同的芯片，响应这两种睡眠模式的方式也是不同的。粗线条的实现可能把它们两个等同处理也说不定。

表 13.1 系统控制寄存器（地址：0xE000_ED10）

位段	名称	类型	复位值	描述
4	SEVONPEND	RW	-	发生异常悬起时将发送事件。在使用 WFE 指令睡眠后，此位可以使得新悬起的中断把 CM3 从 WFE 指令处唤醒。不管这个中断的优先级是否比当前的高，都唤醒。
3	保留	-	-	-
2	SLEEPDEEP	R/W	0	当进入睡眠模式时，使能外部的 SLEEPDEEP 信号，以允许停止系统时钟
1	SLEEPONEXIT	R/W	-	激活“SleepOnExit”功能
0	保留	-	-	-

通过执行 WFI/WFE 指令，请求 CM3 进入睡眠模式，它们在 CM3 中的地位就类似于某些处理器的“sleep/slp”指令。WFI 表示 Wait-For-Interrupt，而 WFE 表示 Wait-For-Event。那么什么可以算是 event 呢？新来的中断、早先被悬起的中断，或者是通过 RXEV 信号表示的一个外部事件信号脉冲，都属于 event。在处理内部，对事件有一个锁存器，因此过去发生的事件可以用来唤醒将来才执行到的 WFE。流程如图 13.2 所示。

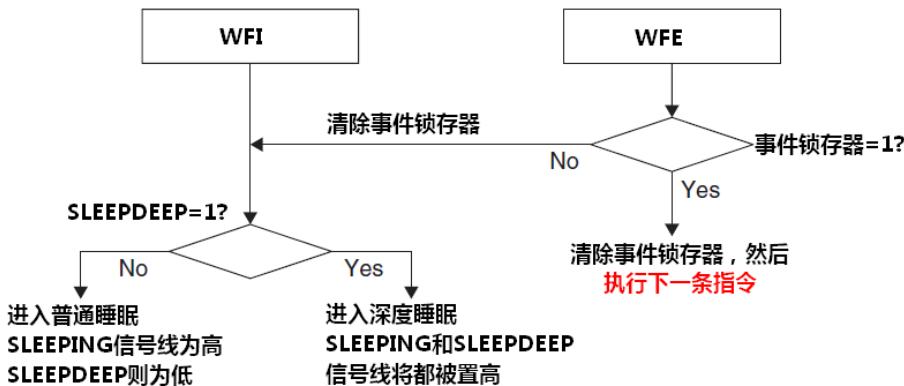


图 13.2 进入睡眠模式的序列

当处理器进入睡眠模式时，单片机作如何反应，还取决于芯片的设计。最典型的作法就是把一些外设的时钟停掉以降低功耗。当然，芯片还可以做得更有力，切断一部分功能模块的电源，甚至切断整个芯片的电源并且停止所有的时钟。这是把事情做绝了，只能通过复位来唤醒。为此，芯片厂商可以在单片机上提供一个引脚，并根据它的电平变化来产生此复位信号。另外，芯片厂商还可以在设计时加入少量的 SRAM 作为后备存储区，该区电力供应不被切断（如 STM32），以供应用程序在轮回前，先把今生离别之际的一些重要上下文存入，待到来世再报恩。

WFI/WFE 除了进入睡眠的序列不同外，它们的唤醒行为也有所不同。

当从 WFI 唤醒时，要根据异常系统的游戏规则来决定是否唤醒。只有当该中断的优先级比当前优先级要高（如果是在服务例程中使用 WFI），并且比 BASEPRI 掩蔽的高时，才唤醒处理器并执行 ISR。但如果 PRIMASK 置位，则依然唤醒处理器，然而 ISR 却不执行了。

WFE 则有点区别，不管优先级和掩蔽情况如何，只要 SETONPEND 置位，它就会错过任何一个事件，在发生事件时一定把处理器唤醒。至于是否执行 ISR，则与 WFI 的规则相同。

CM3 处理器唤醒的具体规则如表 13.2A 和表 13.2B 所示。但要注意：这是假设中断的优先级比当前优先级要高的（即没有在异常服务例程中使用 WFI/WFE，谁在这里用谁想不开）。

表 13.2AWFI 的唤醒行为（带“+”的表示执行此动作）

中断优先级	唤醒	执行 ISR
PRIMASK=0, 且 BASEPRI 不能掩蔽	+	+
PRIMASK=0, 且 BASEPRI 能够掩蔽		
PRIMASK=1, 且 BASEPRI 不能掩蔽	+	
PRIMASK=1, 且 BASEPRI 能够掩蔽		

表 13.2BWFE 的唤醒行为（带“+”的表示执行此动作）

中断优先级	唤醒	执行 ISR
PRIMASK=0, SEVONPEND=0, 且 BASEPRI 不能掩蔽	+	+
PRIMASK=0, SEVONPEND=0, 且 BASEPRI 能够掩蔽		
PRIMASK=0, SEVONPEND=1, 且 BASEPRI 不能掩蔽	+	+
PRIMASK=0, SEVONPEND=1, 且 BASEPRI 能够掩蔽	+	
PRIMASK=1, SEVONPEND=0, 且 BASEPRI 不能掩蔽	+	
PRIMASK=1, SEVONPEND=0, 且 BASEPRI 能够掩蔽		
PRIMASK=1, SEVONPEND=1, 且 BASEPRI 不能掩蔽	+	
PRIMASK=1, SEVONPEND=1, 且 BASEPRI 能够掩蔽	+	

译者小结：

1. 只有 PRIMASK=0 时，才执行 ISR
2. 对于 WFE，只要 SEVONPEND=1，则不管何时发生了什么中断，都一定会唤醒处理器
3. 不管 PRIMASK 为何值，只要优先级高到 BASEPRI 不能掩蔽，就将唤醒处理器
4. 当 PRIMASK=0 时，它不会对唤醒产生影响

CM3 还有一个“自动睡眠”的机制：SleepOnExit——它可以被编程为从中断服务例程返回后立即睡眠。这样一来，处理器的所有工作就只是响应中断了，其它时间都在睡眠。在真实的应用程序里，通常只有在程序很简单的电池供电设备中，才会用此功能。如欲使用此特性，需要把系统控制寄存器中的 SLEEPONEXIT 位置位。如图 13.3 所示。

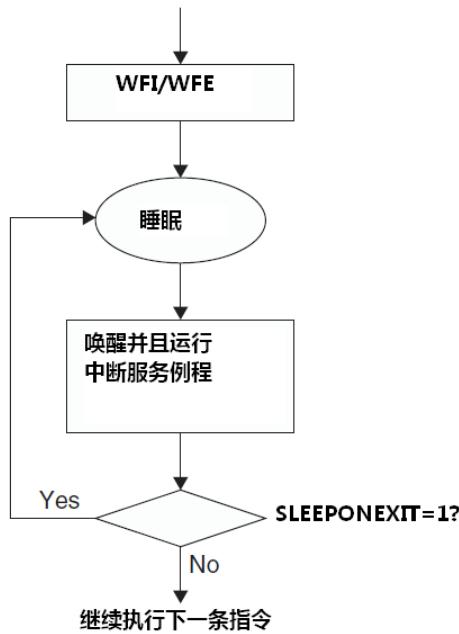


图 13.3 SleepOnExit 功能演示

13.3 多处理机通信

最让人意想不到的就是 CM3 竟然还支持简单的多核功能！它上面有一个用于处理机之间同步任务的简单通信接口。处理机有一个名为 TXEV (Transmit Event) 的输出信号，用于发送信号给其它处理机；还有一个名为 RXEV (Receive Event) 的输入信号，以接收从其它处理机发来的信号。对于一个双核系统来说，事件通信的信号的连接可以如图 13.4 所示：

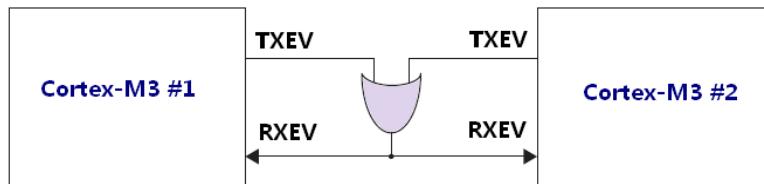


图 13.4 双核处理系统间的事件信号连接

如上一小节所述，当处理机因为 WFE 而睡眠时，可以由外部事件——即 RXEV 唤醒。CM3 提供了 SEV 指令 (Send Event)。当执行该指令时，当事处理机就会在 TXEV 上发送一个脉冲，从而可以唤醒另外的睡眠中的处理机，从而实现同步，如图 13.5 所示。

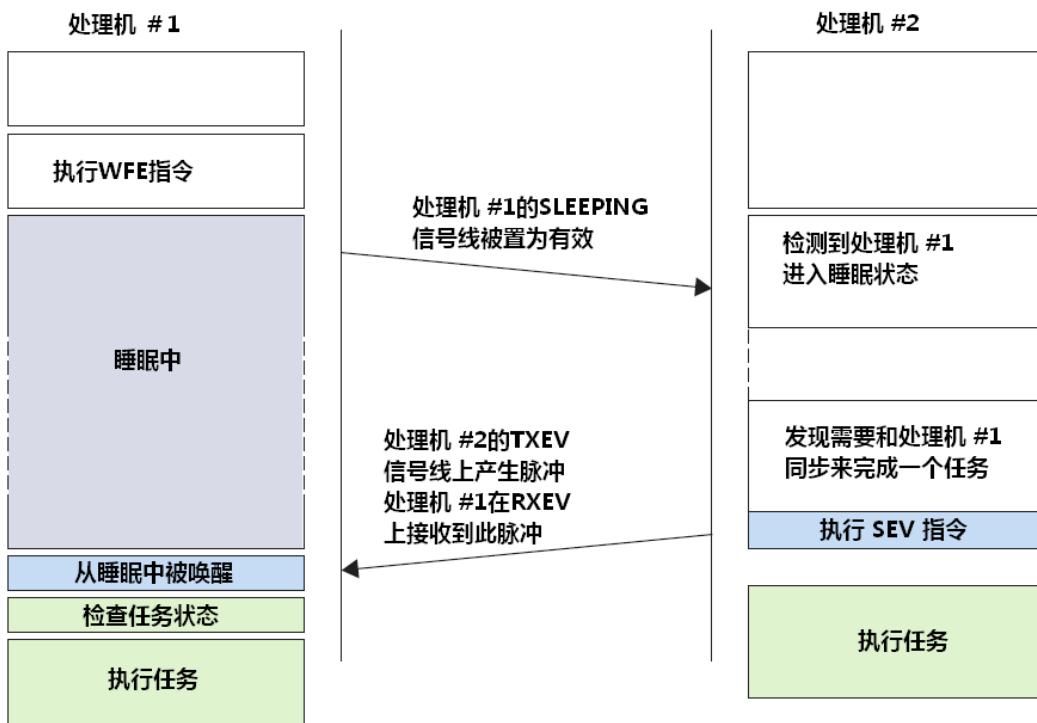


图 13.5 双核之间使用事件信号来做同步任务

在使用 WFE 同步任务时, 要明白处理器也以被其它事件唤醒, 比如中断和调试事件。所以在被唤醒时, 需要先检查是不是由同步事件信号唤醒的。使用 WFE 同步任务的流程如图 13.6 所示。

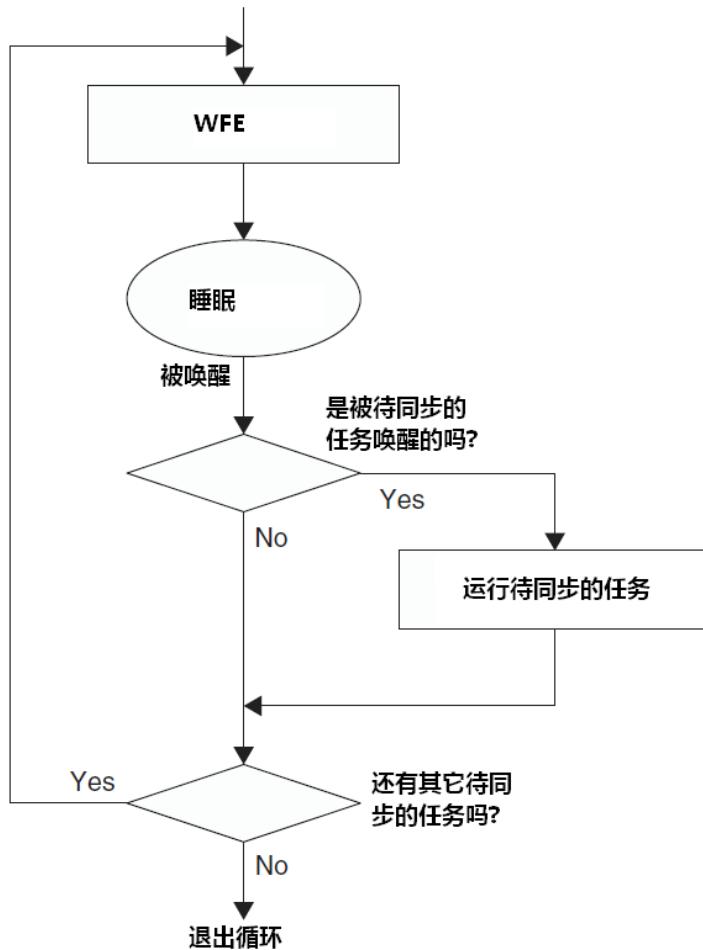


图 13.6 使用 WFE 同步任务模式图

通过使用 WFE，我们可以让两个处理器同步地配合完成一个任务（也可能会有少量时钟周期的时差，这取决于器件的实现方式）。上图演示的是两台处理器的情况，事实上处理器的数目并没有限制，但无论如何都必须有一个担当“主机”，用于发送同步事件。

当执行 WFE 时，它首先检视本地事件锁存器。如果锁存器的值为零，则使内核睡眠；如果发现锁住了先前的事件信号，则清零锁存器，并且取消此次睡眠，继续执行下一条指令。早先发生的异常、执行的 SEV 指令都可以置位锁存器。所以要注意，如果曾经执行过 SEV，则紧挨着的 WFE 不会使处理器睡眠，只是清除了锁存的值，处理器依然继续执行。

13.3.1 多机同步的深入讨论

事实上，同步问题远远要复杂得多。如果只是按图 13.6 那样单单使用 WFE，只能应付小儿科的任务同步问题。在复杂的应用程序中，为正确地同步任务还需要附加的代码。正如上文所提到的，处理器也以被其它事件唤醒，比如中断和调试事件。因此，内部的事件寄存器的当前状态常常是未知的，故而不能保证在执行 WFE 指令后就一定能进入睡眠。事实上，WFE 常常在循环中使用（用于降低系统的功耗），循环体中的代码检查状态，以判定需要同步的任务是否应该在 WFE 后执行。

这种用法最典型示例就是多核系统中的信号量。在典型的情况下，需要一个系统级的互斥访问监视器，在它的辅助下使用互斥访问指令来实现自旋锁（spin lock，熟悉 Linux 的读者请给我你们的微笑），以一轮一轮地尝试锁住共享的存储器或外设。自旋锁设施由 RTOS 提供，通常由汇编语言写成。RTOS 提供类似 `spin_lock()` 和 `spin_unlock()` 的函数（如：Linux），而任务则可以使用这两个函数来锁住所需的共享资源。

常规的自旋锁代码如下所示：

```
spin_lock ; 获取自旋锁的汇编示例代码, r0 指向自旋锁变量
    MOVS r2, #1 ; r2 待会要写入自旋锁变量, 表示资源已锁
spin_lock_loop
    LDREX r1, [r0]
    CMP r1, #0
    BNE spin_lock_loop ; 资源已被锁住, 需重试
    STREX r1, r2, [r0] ; 使用 STREX 指令尝试设置 Lock_Variable 为 1
    CMP r1, #0 ; 检查 STREX 指令的返回值
    BNE spin_lock_loop ; STREX 指令没有成功执行, 重试
    DMB ; 执行数据存储器隔离, 以确保数据已落实到物理内存中。
    BX LR ; 返回
```

在共享资源使用完毕后, 需要释放自旋锁:

```
spin_unlock ; 释放自旋锁的汇编示例代码, r0 指向自旋锁变量
    MOVS r1, #0
    DMB ; 原文 DMB 在这里使用, 疑似不妥
    STR r1, [r0] ; Clear lock
    DMB ; 执行数据存储器隔离, 以确保数据已落实到物理内存中。
    BX LR ; 返回
```

自旋锁的副作用, 就是会当 (获取锁的) 处理机空闲时使 (等待锁的) 处理机白白空转, 浪费能源。因此, 我们在上例的自旋锁中加入 **WFE/SEV** 来解决这个问题: 一方面, 在尝试上锁的函数中, 一旦发现已上锁就执行 **WFE**; 而在释放锁的函数中 (在另一个处理机中执行此函数), 释放后执行 **SEV** 指令以唤醒所有尝试上锁的处理机。

```
spin_lock_with_WFE ; 使用 WFE 配合获取自旋锁的汇编示例代码, r0 指向自旋锁变量
    MOVS r2, #1 ; r2 待会要写入自旋锁变量, 表示资源已锁
spin_lock_loop
    LDREX r1, [r0]
    CBNZ r1, lock_is_set ; 如果 r1!=0, 则表示已上锁
    STREX r1, r2, [r0] ; 使用 STREX 指令尝试设置 Lock_Variable 为 1
    CMP r1, #0 ; 检查 STREX 指令的返回值
    BNE spin_lock_loop ; STREX 指令没有成功执行, 重试
    DMB ; 执行数据存储器隔离, 以确保数据已落实到物理内存中。
    BX LR ; 返回
lock_is_set
    WFE ; 资源已锁。等待使用资源的处理机释放锁后使用 SEV 发出信号
    B spin_lock_loop ; 被唤醒, 不管是不是被 SEV 唤醒的, 先去尝试上锁
```

在共享资源使用完毕后, 需要释放自旋锁:

在解开自旋锁的函数中, 需要使用 **SEV** 指令来唤醒其它所有需要该锁的处理机。

```
spin_unlock_with_SEV ; 释放自旋锁的汇编示例代码, r0 指向自旋锁变量
    MOVS r1, #0
    DMB ; 原文 DMB 在这里使用, 疑似不妥
```

```

STR    r1, [r0]      ; Clear lock
DMB
SEV
BX    LR      ; Return

```

通过在信号量代码中配合使用事件通信接口，就可以在使用自旋锁尝试获取共享资源时消除不必要的功耗。类似的技术也可以用于创建消息队列等其它任务同步设施。

译者添加：自旋锁不是谁想用谁就能用的，必须分场合。如果是同一个处理机内的多个任务需要某共享资源，且在其它处理机上没有需要此资源的任务，就不得使用带 `WFE` 的自旋锁，因为在执行 `WFE` 后，该处理机已经睡眠了，无法再执行其它指令，更不要说调度其它任务让它调用 `spin_unlock_with_SEV` 了。此时又没有“外力”，因此就很可能要“长眠”了！进一步地，单机场合下不得使用自旋锁。因为自旋锁可能导致死循环：优先级最高的任务如果使用自旋锁未果，则在按优先级调度的 RTOS 中，如果没有反优先级倒转机制，就会使最高优先级的任务永远死循环，CPU 利用率 100%，却再也执行不了其它任务！

在大多数 CM3 系统中，会只使用一个内核。此时，常常是把 `RXEV` 脚拉低，或者连接到其它可以产生事件的外设上。

Cortex-M3 r2p0修订版新增

请注意：当使能了 `SLEEPONEXIT` 特性时，CM3 在异常退出后不经过执行 `WFI/WFE` 就会进入睡眠模式。因此当需要执行睡眠时，在正常的使用场合下，要在 `WFI/WFE` 指令（得到执行）之前就使能 `SLEEPONEXIT`。

在 Cortex-M3 修订版 2（已于 2008 年出品）中，又添加了新的特性以支持低功耗。从软件的立场上来看，`WFI/WFE` 依然故我。但在硬件上，修订版的深度睡眠模式则睡得更深：允许送往处理器内核的时钟信号停止。那这么一来怎样唤醒内核呢？原来，修订版 2 的内核新增了一个独立的单元，称作“唤醒中断控制器”。有了它，处理器内核可以在进入掉电模式时，把处理器状态信息存储到特殊的逻辑小室（`cells`）中，从而更狠地降低空闲时的功耗。

要使用新的掉电模式，还需要一个外部电源管理单元来配合，后者用于控制上电序列和掉电序列。该单元由芯片供应商提供，在使用掉电特性前可能还要编程它，因此需要参考芯片供应商提供的技术文档。关于掉电特性，还有两点要注意的。首先，是它会关掉送往 `SysTick` 定时器的时钟。第 2，当连接了一个调试器时，为了使它能够正常地访问调试寄存器，会自动除能这个掉电特性。

13.4 自复位控制

CM3 允许由软件触发复位序列，用于特殊的调试或维护目的（没事别玩啊）。在 CM3 中，有两种方法可以执行自我复位。第一种方法，是通过置位 NVIC 中应用程序中断与复位控制寄存器（AIRCR）的 `VECTRESET` 位（位偏移：0），如下所示：

```

LDR    R0,    =0xE000ED0C      ; NVIC AIRCR address
LDR    R1,    =0x05FA0001      ; 置位 VECTRESET位，前面的0x05FA是访问钥匙
STR    R1,    [R0]      ; 触发复位序列
deadloop

```

```
B      deadloop ; 该死循环保证后面的指令不可能被执行到
```

这种复位的作用范围覆盖了整个 **CM3** 处理器中，除了调试逻辑之外的所有角落，但是它不会影响到 **CM3** 处理器外部的任何电路，所以单片机上的各片上外设和其它电路都不受影响。

复位的第二种方法，是置位同一个寄存器中的 **SYSRESETREQ** 位。这种复位则会波及整个芯片上的电路：它会使 **CM3** 处理器把送往系统复位发生器的请求线置为有效。但是系统复位发生器不是 **CM3** 的一部分，而是由芯片厂商实现，因此不同的芯片对此复位的响应也不同。因此，读者需要认真参阅该芯片规格书，明白当发生片内复位时，各外设和功能模块都会回到什么样的初始状态，或者有哪些功能模块不受影响（比如，**STM32** 系列的芯片有后备存储区，该区就被特殊对待）。

SYSRESETREQ 的使用如下面代码段所演示：

```
LDR    R0,    =0xE000ED0C ; NVIC AIRCR address
LDR    R1,    =0x05FA0004 ; 置位 SYSRESETREQ, 前面的0x05FA是访问钥匙
STR    R1,    [R0]          ; 触发复位序列
```

deadloop

```
B      deadloop ; 该死循环保证后面的指令不可能被执行到
```

大多数情况下，复位发生器在响应 **SYSRESETREQ** 时，它也会同时把 **CM3** 处理器的系统复位信号(**SYSRESETn**)置为有效。通常，**SYSRESETREQ** 不应复位调试逻辑。

这里有一个要注意的问题：从 **SYSRESETREQ** 被置为有效，到复位发生器执行复位命令，往往会有一定延时。在此延时期间，处理器仍然可以响应中断请求。但我们的本意往往是要让此次执行到此为止，不要再做任何其它事情了。所以，最好在发出复位请求前，先把 **FAULTMASK** 置位。

存储保护单元 MPU

译者提示：MPU是CM3的选配件，许多CM3单片机中都没有加入。本章在翻译时，对原文改编比其它章节要强烈，且有一部分内容译自“古文观止”。如果时间不富裕，读者可以选择跳过。

- MPU 概览
- MPU 的寄存器组
- 启用 MPU
- MPU 的典型设置

14.0 译者添加的引子

MPU 进入单片机还是很新鲜的事，为了让读者预先对它更有一点认识，译者加入了引文：

引子 1：野指针与 C 语言

回顾一下，什么是指针？指针在内存中实际上是一个无符号整数 (unsigned int)，但是它的值被赋予特殊的解释：表示变量或函数的地址。所以才被形象地称为“指针”，就好像指向谁家似的。在使用指针前，都必须先让它指向有意义的，并且允许由程序使用的实体——数据和代码。而所谓“野指针”，就是指某个指针变量的值因故超出合法的范围，使其“枪口”乱指。程序逻辑错误、数组越界、堆栈溢出、指针未经初始化、对缓存与缓冲的处理不当、多任务环境中的紊乱危象，甚至是恶意地破坏等，都可以制造出野指针。如果使用野指针去读取或修改内存，则被读取或修改的位置是不可预料的。前者导致读回来的都是垃圾数据，后者则更是“血口喷人”——会破坏未知用途的数据。这常常导致系统发生莫名其妙的功能紊乱，严重时会使系统毫无征兆，没有理由地失控、死机。

野指针就像“肉里的刺，酱里的蛆”一般：一个野指针就足以崩溃整个系统，而且极其隐蔽，很难通过症状来找出是哪里存在野指针，甚至都不能判定症状是否因野指针造成（程序大了其它 bug 也很多，并且也能导致相同的症状）。野指针的发作概率越小，就越隐蔽，后患也越无穷。对于通常的单片机系统，是没有任何办法来防止野指针的破坏的，完全靠程序员的素质和自律。但智者千虑，必有一失。尤其是当程序规模变得很大时，复杂度会呈指数上升，千头万绪纠缠不清，就算是谨慎如诸葛亮，聪明如比尔·盖茨的天才，也不敢保证没有漏网之鱼。

嵌入式系统开发的首选语言是 C 语言。C 语言的指针功能非常灵活、生猛、桀骜不驯，它是电，它是光，它是 C 语言中最闪亮的“Super Star”。C 语言允许我们几乎随心所欲地把玩各种地址，离汇编语言中“放任自流”的程度也差不远了。可是，要是像汇编那样“明坏”倒也好，偏偏 C 语言中的指针还因为语言特性附加了许多十分微妙的“潜规则”，令人防不胜防；更加暧昧的是指针与数组的关系，一维数组与多维数组的关系，多维数组与“星星”点灯的关系，指针与“[]”的二重唱，指针在宏中使用时极易弄巧成拙的暗箱操作……用 C 语言的指针功能就像在玩一场勾魂的“野蛮游戏”，不知不觉其实你已“上线”，指针飞舞的世界战火连天，如果不每天因爆发了却查不出来的 bug 而以泪洗面，提高警觉快张大双眼是必要的，但年深日久还是难免有看不清楚而迟早粉身碎骨的时候。在系统程序的开发中，指针更是满天飞遍地爬，程序员在这无间世界里没有想过要逃脱，为什么要逃脱？完全是“你主宰，我崇拜，没有更好的办法，只能爱你，u r my super star”。

这里再说句题外话。如果读者不幸与被 C 语言的指针给沾上了，就要特别重视内功的修炼。除了要把《C 程序设计》以及《C 和指针》夜夜放在床头外，更要千方百计地弄到《C 陷阱与缺陷》，以及《C 专家编程》这两本书，它们两个堪比《葵花宝典》和《九阳真经》。

引子 2：使命-关键系统

这种系统往往都用于性命攸关的场合，且必须连续无故障地工作，比如，火车调度系统、生命维持系统、大型发动机驱动器、核子反应堆控制、网络/电信的数据交换中枢等。如果失能，将导致惨重的经济与损失，甚至会使无数人死于非命。因此，决不允许这类系统出现上述情况。然而，这些系统的复杂度往往都非常高，几乎不可能由开发人员保证这种可靠性。

因此，需要在硬件水平上加入一个“公安机关”。通过它设置各种类型的“禁地”，并且施加多种规章条例。一旦发现违章，则强制改变执行流和处理器的工作状态，以便可以由软件做进一步的处理。这样，就可以为不同的程序限定一个内存使用范围，从而使野指针或恶意破坏无法影响不允许访问的区域。此即存储器保护单元 (MPU)。

有时，对存储器的管理更进一步，做到可以对地址执行变换的程度，此时程序使用的地址未必是真实的存储器地址。它在 MPU 的基础上，还消灭了内存碎片和浪费，并且能进一步地让应用程序拥有方便舒适的地址空间，从而使程序规模可以扩大甚至数百倍。此即为“存储器管理单元” (MMU)。带 MMU 的系统，往往也带 cache，动态 RAM 等。这种系统对 RAM 容量的计量是以 MB 为单位的。可见，MMU 是一个对处理器定位的“分水岭”。对 MMU 的介绍已经超出了本书的范围。

(本章篇幅虽然较长，但很多内容都是在寄存器的介绍，以及示例代码的反刍上，读者请放松阅读)

14.1 MPU 概览

在 Cortex-M3 处理器中可以选配一个存储器保护单元 (MPU)，它可以实施对存储器（主要是内存和外设寄存器）的保护，从而使软件更加健壮和可靠。如果打算启用 MPU，则在使用前，必须根据需要对其编程。如果没有启用 MPU，则等同于系统中没有配 MPU。MPU 有如下的能力可以提高系统的可靠性：

- 阻止用户应用程序破坏操作系统使用的数据
- 阻止一个任务访问其它任务的数据区，从而把任务隔开。
- 可以把关键数据区设置为只读，从根本上消除了被破坏的可能。
- 检测意外的存储访问，如，堆栈溢出，数组越界。
- 此外，还可以通过 MPU 设置存储器 regions 的其它访问属性，比如，是否缓区，是否缓冲等。

MPU 在执行其功能时，是以所谓的“region”为单位的。一个 region 其实就是一段连续的地址，只是它们的位置和范围都要满足一些限制（对齐方式，最小容量等）。CM3 的 MPU 共支持 8 个 regions。怎么，嫌少？是少了点，不过，还允许把每个 region 进一步划分成更小的“子 region”。此外，还允许启用一个“背景 region”（即没有 MPU 时的全部地址空间），不过它是只能由特权级享用。在启用 MPU 后，就不得再访问定义之外的地址区间，也不得访问未经授权的 region。否则，将以“访问违例”处理，触发 MemManage fault。

MPU 定义的 regions 可以相互交迭。如果某块内存落在多个 region 中，则访问属性和权限将由编号最大的 region 来决定。比如，若 1 号 region 与 4 号 region 交迭，则交迭的部分受 4 号 region 控制。

14.2 MPU 的寄存器组

操作 MPU 是通过访问它的若干寄存器来实现的，如下表所示。

(译者注：此表摘自Cortex-M3 TRM)

名字	访问	地址	初值
MPU类型寄存器 MPUTR	RO	0xe000,ed90	A
MPU控制寄存器 MPUCR	RW	0xe000,ed94	0x0000,0000
MPU region号寄存器MPURNR	RW	0xe000,ed98	-
MPU region基址寄存器MPURBAR	RW	0xe000,ed9c	-
MPU region属性及容量寄存器(s) MPURASR	RW	0xed00,eda0	-
MPU region基址寄存器的别名1	D9C的别名	0xed00,eda4	-
MPU region属性及容量寄存器的别名1	DA0的别名	0xed00,eda8	-
MPU region基址寄存器的别名2	D9C的别名	0xed00,edac	-
MPU region属性及容量寄存器的别名2	DA0的别名	0xed00,edb0	-
MPU region基址寄存器的别名3	D9C的别名	0xed00,edb4	-
MPU region属性及容量寄存器的别名3	DA0的别名	0xed00,edb8	-

让我们来详细地介绍上述寄存器，第一个就是MPU类型寄存器（MPUTR），如表14.1所示

表 14.1 MPU 类型寄存器 MPUTR (地址 : 0xE 0 00_ED90)

位段	名称	类型	复位值	描述
23:16	IREGION	R	0	MPU 支持的指令 region 数量。因为 ARMv7-M 只使用单个统一的 MPU，此位段永远为零
15:8	DREGION	R	0	MPU 支持的数量。若系统中配了 MPU 则为 8，否则为零
0	SEPARATE	R	0	固定为零

从表中我们可以看出，通过读取DREGION的值，能够判断芯片中是否配了MPU。

接下来我们看一看MPU控制寄存器MPUCR如表14.2所示

表 14.2 MPU 控制寄存器 MPUCR (地址 : 0xE 0 00_ED94)

位段	名称	类型	复位值	描述
2	PRIVDEFENA	RW	0	是否为特权级打开缺省存储器映射（即背景

				region)。 1=特权级下打开背景 region 0=不打开背景 region。任何访问违例以及对 region 外地址区的访问都将引起 fault
1	HFNMIENA	RW	0	1=在 NMI 和硬 fault 服务例程中不强制除能 MPU 0=在 NMI 和硬 fault 服务例程中强制除能 MPU
0	ENABLE	RW	0	使能 MPU

通过把PRIVDEFENA置位，可以在没有建立任何region就使能MPU的情况下，依然允许特权级程序访问所有地址，而只有用户级程序被拒之门外。然而，如果设置了其它的region并且使能了MPU，则背景region与这些region重合的部分，就要受各region的限制。为了方便理解，让我们作一个对比，看看PRIVDEFENA在置位与清零时，系统对访问的限制有何不同，如图14.1所示。

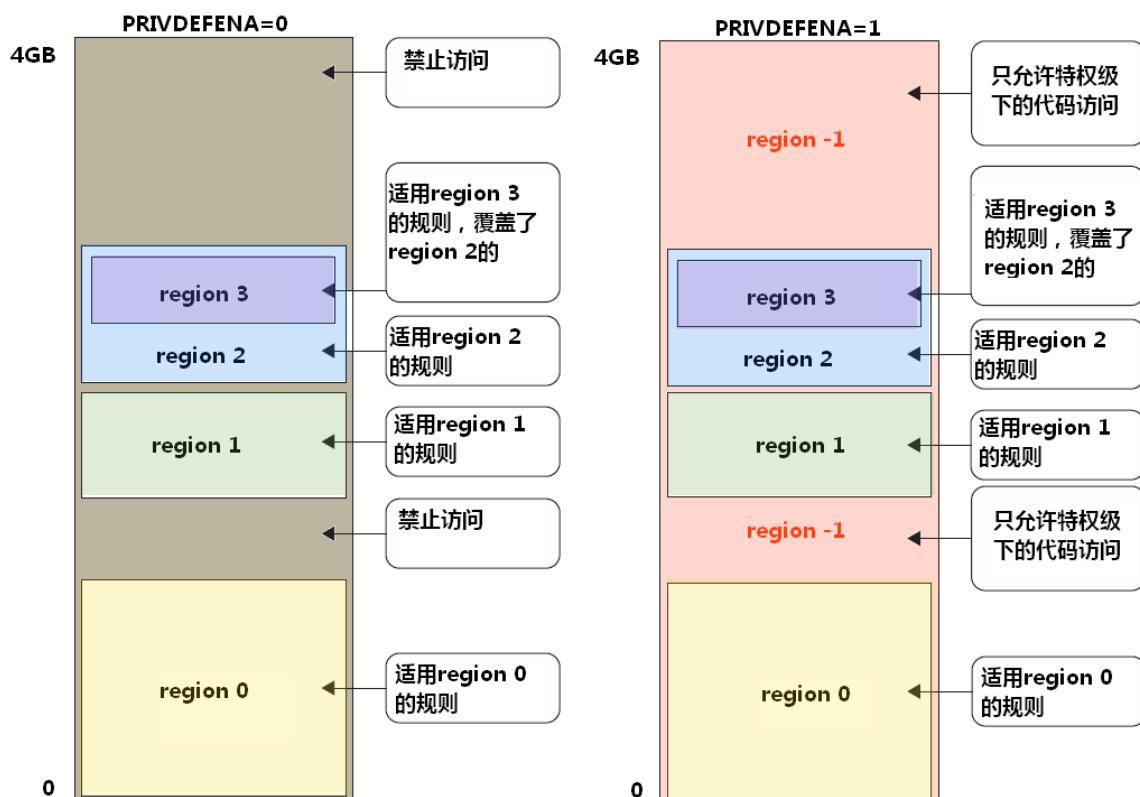


图14.1 PRIVDEFENA的影响

要注意，只要没有极另类的考虑，就要到万事就绪后，最后一步才置位ENABLE位。否则，就有可能因region没有配置好而意外地产生MemManage fault。很多条件下，为安全起见，最好在执行配置MPU的子程前先除能MPU，待执行后再重新使能MPU。

注意：这里有个例外：不管MPU如何限制，响应异常时的取向量操作，以及对系统分区(system partition)的访问总是不受影响的。

译注：这里所说的“系统分区”，作者并没有解释过。估计有可能是包含在前面提到的“SCS”区

When the MPU is enabled, only the system partition and vector table loads are always accessible.

配置任何一个region之前，都需要在MPU内选中这个region，这可以通过把region号写入MPU region号寄存器(MPURNR)来完成，其定义如表14.3所示

表 14.3 MPU region 号寄存器 MPURNR (地址 : 0xE000_ED98)

位段	名称	类型	复位值	描述
7:0	REGION	RW	-	选择下一个要配置的 region。因为只支持 8 个 region, 所以事实上只有[2:0]有意义

选好了region后, 就可以在另外两个寄存器中配置该region的所有属性了。

为了能快速地配置多个regions, 还有另一种快捷方式。在MPU region基址寄存器(MPURBAR)中有两个位段: VALID和REGION, 它们配合使用可以绕过MPURNR。MPURBAR的定义如表14.4所示

表 14.4 MPU region 基址寄存器 MPURBAR (地址 : 0xE000_ED9C)

位段	名称	类型	复位值	描述
31:N	ADDR	RW	-	Region 基址字段。N 取决于 region 容量, 以使基址在数值上能被容量整除。在 MPU region 属性及容量寄存器中有个 SZENABLE 位段, 它决定 ADDR 中有多少个位被采用。
4	VALID	RW	-	决定是否理会写入REGION字段的值 1=MPU region 号寄存器被REGION覆盖 0=MPU region 号寄存器的值保持不变
3:0	REGION	RW	-	MPU region 覆写位段

从表中我们可以看出, 基址必须对齐到region容量的边界。举例来说, 如果你定义的region容量是64KB, 那么它的基址就必须能被64KB整除。这里, 像0x0001,0000; 0x0002,0000这样的, 就是合法的基址 (低16位为0)。

如果读取REGION位段, 返回的总是当前的region号, 并且VALID总返回0。通过设置VALID=1和REGION=n, 也可以改变一个region的编号。相比于先设置MPU region号寄存器再设置本寄存器的正统做法而言, 这是一个快捷方式。

注意: 必须以字的方式来访问本寄存器, 否则结果不可预知。

配置好了基地址, 我们还需要详细定义region的其它方方面面。这需要设置MPU属性及容量寄存器。这个寄存器是全体可读可写的, 且复位值未知, 如下表14.5所示:

表14.5 MPU region属性及容量寄存器MPURASR (地址 : 0xE000_EDA0)

位段	长度	名称	功能																																				
31:29	3	-	保留																																				
28	1	XN	1=此区禁止取指 2=此区允许取指																																				
27	1	-	保留																																				
26:24	3	AP	访问许可, 如下表所示																																				
			<table border="1"> <thead> <tr> <th>值</th><th>特权级下的许可</th><th>用户级下的许可</th><th>典型用法</th></tr> </thead> <tbody> <tr> <td>0b000</td><td>禁地</td><td>禁地</td><td>该区没有存储器, 是空地址</td></tr> <tr> <td>0b001</td><td>RW</td><td>禁地</td><td>OS以及系统软件使用的数据区</td></tr> <tr> <td>0b010</td><td>RW</td><td>RO</td><td>禁止在用户级下更改的高危地带</td></tr> <tr> <td>0b011</td><td>RW</td><td>RW</td><td>共享内存, 或彻底开放的设备</td></tr> <tr> <td>0b100</td><td>n/a</td><td>n/a</td><td>n/a</td></tr> <tr> <td>0b101</td><td>RO</td><td>禁地</td><td>OS使用的常量数据</td></tr> <tr> <td>0b110</td><td>RO</td><td>RO</td><td>常量数据或只读存储器的地址区</td></tr> <tr> <td>0b111</td><td>RO</td><td>RO</td><td>常量数据或只读存储器的地址区</td></tr> </tbody> </table>	值	特权级下的许可	用户级下的许可	典型用法	0b000	禁地	禁地	该区没有存储器, 是空地址	0b001	RW	禁地	OS以及系统软件使用的数据区	0b010	RW	RO	禁止在用户级下更改的高危地带	0b011	RW	RW	共享内存, 或彻底开放的设备	0b100	n/a	n/a	n/a	0b101	RO	禁地	OS使用的常量数据	0b110	RO	RO	常量数据或只读存储器的地址区	0b111	RO	RO	常量数据或只读存储器的地址区
值	特权级下的许可	用户级下的许可	典型用法																																				
0b000	禁地	禁地	该区没有存储器, 是空地址																																				
0b001	RW	禁地	OS以及系统软件使用的数据区																																				
0b010	RW	RO	禁止在用户级下更改的高危地带																																				
0b011	RW	RW	共享内存, 或彻底开放的设备																																				
0b100	n/a	n/a	n/a																																				
0b101	RO	禁地	OS使用的常量数据																																				
0b110	RO	RO	常量数据或只读存储器的地址区																																				
0b111	RO	RO	常量数据或只读存储器的地址区																																				
23:22	2	-	保留																																				
21:19	3	TEX	类型扩展																																				
18	1	S	Sharable (可否共享) 1=共享可 0=共享不可																																				
17	1	C	Cachable (可否缓存) 1=缓存可 0=缓存不可																																				
16	1	B	Buffable (可否缓冲) 1=缓冲可 0=缓冲不可																																				
15:8	8	SRD	子region除能位段。每设置SRD的一个位, 就会除能与之对应的一个子region。容量大于128字节的region都被划分成8个容量相同的子region。容量小于等于128字节的region不能再分。更多信息, 请参见对子Region的论述。																																				
7:6	2	-	保留																																				
5:1	5	REGIONSIZEx	Region容量, 单位是字节。容量为 $1 \ll (\text{REGIONSIZE}+1)$, 但是最小容量为32字节																																				
0	1	SZENABLE	1=使能此region 0=除能此region																																				

表中提到了“子region”的概念 ([15:8])。原来, 8个region的定义过于粗枝大叶, 因而允许再精雕细琢, 把每个region的内部进一步划分成更小的块, 这就是子region。但是子region的使用有限制: 每个region必须8等分, 每份是一个子region, 而且所有子region的属性都与“父region”的是相同的。每个子region可以独立地使能或除能 (相当于可以部分地使能一个region) : SRD中的8个位, 每个位控制一个子region是否被除能。如SRD.3=0, 则3号子region被除能。如果某个子region被除能, 且其对应的地址范围又没有落在其它region中, 则对该子region覆盖范围的访问将引发fault。最后, 能被“大卸八块”的region, 最小也要有256字节。如果是对128字节或者是更小的region划分子region, 则后果是不可预料的。

再看它的AP位段，为了详细说明把它做成了一个表中表。AP位段用于限定各种访问权限，这也是加以分区保护的最重要组成部分。

位段[28]的名字是XN (eXecute Never)，它决定在本region中是否允许取指。如果不允许取指（清零），则任何指令预取都将触发MemManage fault。这有什么用？通常，可以把新得到的还不受信任的代码先存储到此区，待经过身份鉴定后，再允许它执行。

表中楷体的TEX, S, B和C（整体位于[21:16]），对应着存储系统中比较高级的概念。CM3中没有缓存(cache)，但是CM3是以v7-M的架构设计的，而v7-M支持外部缓存（差不多是L2缓存的地位）以及更先进的存储器系统。按v7-M的规格说明，通过对这些位段的编程，来支持多样的内存管理模型。从v6开始，ARM架构支持两级缓存（与x86的缓存系统是异曲同工的），分别是：内部缓存和外部缓存，它们可以有不同的缓存方针(policy)，这些位组合的详细功能如下表所示：

表14.6 TEX,C,B对存储器类型的决定

TEX	C	B	描述	存储器类型	可否共享
000	0	0	严格按顺序	严格按顺序	总是可以
000	0	1	共享的设备	设备	总是可以
000	1	0	片外或片内的“写通”型内存，没有写allocate	普通	S位决定
000	1	1	片外或片内的“写回”型内存，没有写allocate	普通	S位决定
001	0	0	片外或片内的“缓存不可”型内存	普通	S位决定
001	0	1	n/a	n/a	n/a
001	1	0	实现者您说了算	您说了算	您说了算
001	1	1	片外或片内的“写回”型，带读和写的allocate	普通	S位决定
010	1	x	共享不可的设备	设备	总是不可
010	0	1	n/a	n/a	n/a
010	1	x	n/a	n/a	n/a
1BB	A	A	带缓存的内存。BB=适用于片外内存，AA=适用于片内内存	普通	S位决定

表中最后一项越发离奇，它是TEX的MSB=1时的情况。此时，如果该region是片内存储器，则由C和B决定其缓存属性（AA）；如果是片外存储器，则由TEX的[1:0]决定其缓存属性（BB）。不管是AA还是BB，每个数值的含义都是相同的，如下表所示：

表14.7 缓存方针编码

存储器属性编码 (AA and BB)	高速缓存策略
00	缓存不可
01	写回，读写均有allocate
10	写通，写没有allocate
11	写回，写没有allocate

欲知缓存行为和缓存方针的更多详情，请参阅《ARM Architecture Application Level Reference Manual(Ref2)》。

再看本章开头的寄存器表，最后的8个其实是4对，且后3对都第1对的别名，这可真是“狡兔三窟”啊。再仔细看，你会发现它们的地址是连续的。这下是不是有看出一些端倪了？请看下段译自Cortex-M3 TRM的解释：

9.2.3 使用别名(alias)寄存器访问MPU

通过寄存器别名机制，你可以使用STM指令加速对regions的初始化——一次可以最多初始化4个。一共有3组别名寄存器。别名以完全相同的方式来访问（真实的）寄存器，它们的存在是为了让你能以“顺序写”（STM指令）来一次更新1-4个region。当无需在某些“临界”区域中以“除能region/更改region属性/使能region”的小心方式，来一个个地进行配置时，这个机制就显得特别有用。

下面举一个一次更改4个region的代码例子：

```
; R1 = 一个指针，指向某RTOS进程控制块中的4个region对子（共8个字）
MOV R0, #NVIC_BASE
ADD R0, #MPU_REG_CTRL
LDM R1, [R2-R9] ; 加载4个region的信息
STM R0, [R2-R9] ; 一句话完成4个region的配置
```

这么一来，只要事先做好一个配置表格，就可以一气呵成了。

注意

你不能使用这些别名来读取regions的内容，因为必须要先写region号。

在C/C++下通常使用memcpy()函数来完成上段汇编的功能。但是，你必须验证CRT库，在实现memcpy()时必须是按字拷贝的——也就是两个long*指针之间的拷贝，而不得是char*, short*什么的。

本章后面还有一个“一题多解”的例子，最后的解法就是使用这里讲到的思路

14.3 启用MPU

MPU寄存器看起来比较复杂，那是自然了，毕竟已经上升到存储器管理的高度。但如果我们胸有成竹——已经想好了对存储器如何划分，这就只是一些繁琐和考验细心的体力活。典型情况下，在启用MPU的系统中，都会有下列的regions。

- 特权级的程序代码（如OS内核和异常服务例程）
- 用户级的程序代码
- 特权级程序的数据存储器，位于代码区中（data_stack）
- 用户级程序的数据存储器，位于代码区中（data_stack）
- 通用的数据存储器，位于其它存储器区域中（如，SRAM）
- 系统设备区，只允许特权级访问，如NVIC和MPU的寄存器所有的地址区间
- 常规外设区，如UART, ADC等。

对于CM3来说，绝大多数region中，都有TEX=0, C=1, B=1。系统设备（如NVIC）必须“严格顺序”（strongly ordered）访问；另一方面，外设regions则可以共享（TEX=0, C=0, B=1）。如果想要在某个region中，确保所有的总线fault都是精确的，就必须把该region严格顺序化（TEX=0, C=0, B=0）。这样一来写缓冲被除能，但也因此产生性能损失的代价。

图14.2给出了MPU初始化序列的流程模式图。在使能MPU前，或者把向量表重定位到了RAM，一定不要忘记为MemManage fault建立向量，并且在NVIC的系统handler控制及状态寄存器SHCSR中使能MemManage fault。只有这样做了，才能在产生MPU违例时，让MemManage fault服务例程得以执行。

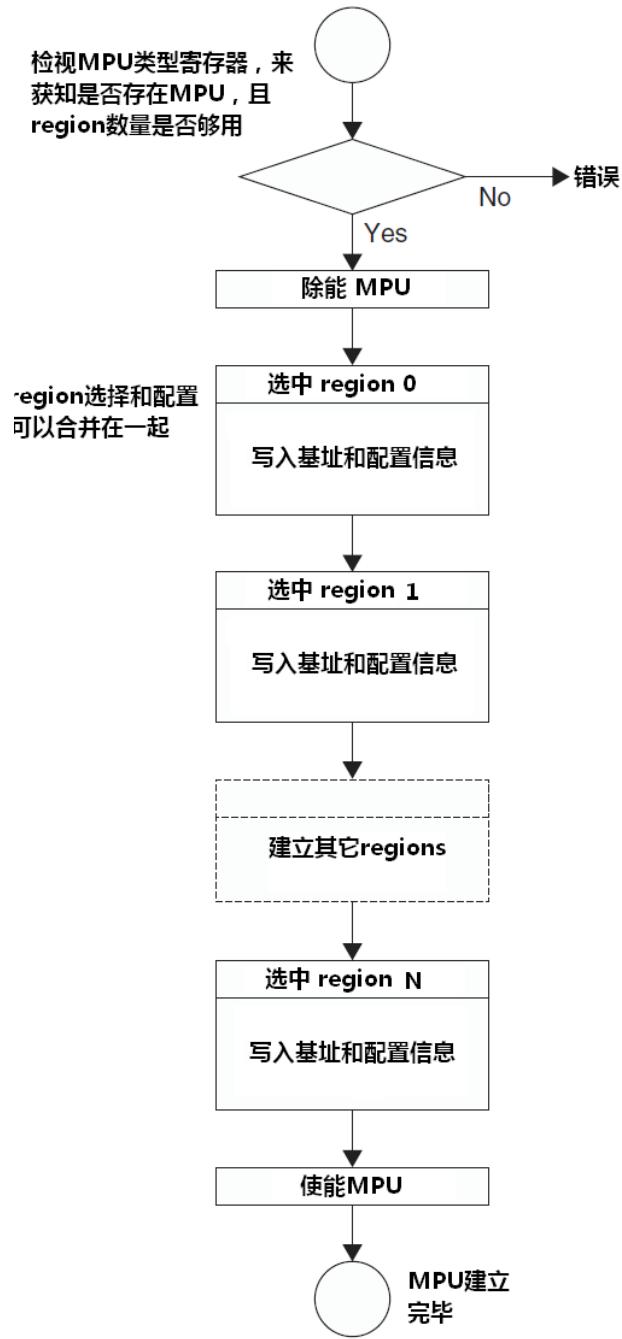


图14.2 MPU初始化序列

下面举一个简单的例子，它只有4个region，则配置代码如下所演示：

```

LDR R0, =0xE000ED98 ; Region号寄存器
MOV R1, #0 ; 选择region 0
STR R1, [R0]
LDR R1, =0x00000000 ; 基址 = 0x00000000
STR R1, [R0, #4] ; MPU Region 基址寄存器
LDR R1, =0x0307002F ; R/W, TEX=0, S=1, C=1, B=1, 16MB, Enable=1
STR R1, [R0, #8] ; MPU Region 属性及容量寄存器
MOV R1, #1 ; 选择region 1
STR R1, [R0]

```

```

LDR    R1,      =0x08000000      ; 基址 = 0x08000000
STR    R1,      [R0, #4]        ; MPU Region 基址寄存器
LDR    R1,      =0x0307002B      ; R/W, TEX=0,S=1,C=1,B=1, 4MB, Enable=1
STR    R1,      [R0, #8]        ; MPU Region 属性及容量寄存器

MOV    R1,      #2            ; 选择 region 2
STR    R1,      [R0]
LDR    R1,      =0x40000000      ; 基址 = 0x40000000
STR    R1,      [R0, #4]        ; MPU Region 基址寄存器
LDR    R1,      =0x03050039      ; R/W, TEX=0,S=1,C=0,B=1, 512MB, Enable=1
STR    R1,      [R0, #8]        ; MPU Region 属性及容量寄存器

MOV    R1,      #3            ; 选择 region 3
STR    R1,      [R0]
LDR    R1,      =0xE0000000      ; 基址 = 0xE0000000
STR    R1,      [R0, #4]        ; MPU Region 基址寄存器
LDR    R1,      =0x03040027      ; R/W, TEX=0,S=1,C=0,B=0, 1MB, Enable=1
STR    R1,      [R0, #8]        ; MPU Region 属性及容量寄存器

MOV    R1,      #1            ; 准备使能MPU
STR    R1,      [R0, #-4]       ; 使能MPU(0xE000ED98-4=0xE000ED94)

```

这段代码执行后，生成如下的4个regions:

特权级代码	0x0000_0000-0x00FF_FFFF(16MB)	全访问	缓存可
特权级数据	0x0800_0000-0x0803_FFFF(4MB)	全访问	缓存可
外设	0x4000_0000-0x5FFF_FFFF(512MB)	全访问	共享设备
系统控制	0xE000_0000-0xE00F_FFFF(1MB)	特权级访问	严格顺序, XN

通过使用基址寄存器的VALID和REGION位段，可以把region选择和基址设置的两个动作合并成一个，从而缩短代码，如下所示：

```

LDR    R0,      =0xE000ED9C      ; MPU region基址寄存器
LDR    R1,      =0x00000010      ; 基址=0x00000000, region=0, valid=1
STR    R1,      [R0, #0]        ; 设置region 0的基址
LDR    R1,      =0x0307002F      ; R/W, TEX=0,S=1,C=1,B=1, 16MB, Enable=1
STR    R1,      [R0, #4]        ; MPU Region 属性及容量寄存器

LDR    R1,      =0x08000011      ; 基址=0x08000000, region=1, valid=1
STR    R1,      [R0, #0]        ; MPU Region 基址寄存器
LDR    R1,      =0x0307002B      ; R/W, TEX=0,S=1,C=1,B=1, 4MB, Enable=1
STR    R1,      [R0, #4]        ; MPU Region 属性及容量寄存器

LDR    R1,      =0x40000012      ; 基址=0x40000000, region=2, valid=1
STR    R1,      [R0, #0]        ; MPU Region 基址寄存器
LDR    R1,      =0x03050039      ; R/W, TEX=0,S=1,C=0,B=1, 512MB, Enable=1
STR    R1,      [R0, #4]        ; MPU Region 属性及容量寄存器

```

```

LDR    R1,      =0xE0000013      ; 基址=0xE0000000, region=3, valid=1
STR    R1,      [R0, #0]        ; MPU Region 基址寄存器
LDR    R1,      =0x03040027      ; R/W, TEX=0,S=1,C=0,B=0, 1MB, Enable=1
STR    R1,      [R0, #4]        ; MPU Region 属性及容量寄存器

MOV    R1,      #1            ; 使能MPU
STR    R1,      [R0,#-8]       ; MPU控制寄存器(0xE000ED9C-8=0xE000ED94)

```

看，代码变短了吧！不过，还有比这更厉害的，让代码更短更快。这要通过使用MPU别名寄存器的地址来完成。在MPU属性及容量寄存器(MPUASR)的后面，有3组MPU基址寄存器(MPUBAR)和MPU属性及容量寄存器的别名，连同真实的MPUBAR与MPUASR，它们共有4组，分布在一个连续的8字空间中，于是就可以使用LDM/STM指令来“串烧”，如下所示：

```

LDR    R0,      =0xE000ED9C      ; MPU region基址寄存器
LDR    R1,      =MPUconfigTab    ; 预定义的MPU初始化数值表
LDMIA R1!,    {R2-R9}        ; 一气从表中读完8个字
STMIA R0!,    {R2-R9}        ; 一气初始化4个region
B     MPUconfigEnd

ALIGN 4
MPUconfigTab
; 此汇编指示字可以确保下述的字定义一定是对齐到字
; 边界的，因为在使用LDM/STM时，地址必须按字对齐

DCD    0x00000010      ; 基址=0x00000000, region=0,valid=1
DCD    0x0307002F      ; R/W, TEX=0,S=1,C=1,B=1, 16MB, Enable=1
DCD    0x08000011      ; 基址=0x08000000, region=0,valid=1
DCD    0x0307002B      ; R/W, TEX=0,S=1,C=1,B=1, 4MB, Enable=1
DCD    0x40000012      ; 基址=0x40000000, region=0,valid=1
DCD    0x03050039      ; R/W, TEX=0,S=1,C=0,B=1, 512MB, Enable=1
DCD    0xE0000013      ; 基址=0xE0000000, region=0,valid=1
DCD    0x03040027      ; R/W, TEX=0,S=1,C=0,B=0, 1MB, Enable=1
MPUconfigEnd

LDR    R0,      =0xE000ED94      ; MPU 控制寄存器
MOV    R1,      #1            ; 使能MPU
STR    R1,      [R0]

```

若用此法，显然必须保证：region配置早已安排好了，否则就只能用上面的更通用的办法。为了使软件更有模块化，可以把建立region的工作包装到一个子程序中，不妨名为MpuRegionSetup。它接受相关参数（编号，基址，容量/属性），并执行建立region的工作。主程序通过呼叫它若干次来逐一设置好每个region。

上面的小凉菜吃了三次，想必读者已经腻了吧。下面就上主菜，使用模块化的思路，代码如下所示。这段代码的后面部分还精彩地演示了新好指令BFI和UBFX的使用：

```

MpuSetup
; 入口函数，它内部呼叫若干子程序来完成MPU设置
PUSH  {R0-R6,LR}
LDR   R0,      =0xE000ED94      ; MPU 控制寄存器
MOV   R1,      #0
STR   R1,      [R0]        ; 配置前先除能MPU
; --- Region #0 ---
LDR   R0,      =0x00000000      ; Region 0: 基址 = 0x00000000

```

```

MOV    R1,    #0x0          ; Region 0: Region号 = 0
MOV    R2,    #0x17         ; Region 0: 容量 = 0x17 (16MB)
MOV    R3,    #0x3          ; Region 0: AP = 0x3 (全访问)
MOV    R4,    #0x7          ; Region 0: MemAttrib = 0x7
MOV    R5,    #0x0          ; Region 0: 子region除能=0
MOV    R6,    #0x1          ; Region 0: {XN, Enable} = 0,1
BL     MpuRegionSetup
; --- Region #1 ---
LDR    R0,    =0x08000000  ; Region 1: 基址 = 0x08000000
MOV    R1,    #0x1          ; Region 1: Region号 = 1
MOV    R2,    #0x15         ; Region 1: 容量 = 0x15 (4MB)
MOV    R3,    #0x3          ; Region 1: AP = 0x3 (全访问)
MOV    R4,    #0x7          ; Region 1: MemAttrib = 0x7
MOV    R5,    #0x0          ; Region 1: 子region除能=0
MOV    R6,    #0x1          ; Region 1: {XN, Enable} = 0,1
BL     MpuRegionSetup
...
; --- Region #4-#7 除能 ---
MOV    R0,    #4
BL     MpuRegionDisable
MOV    R0,    #5
BL     MpuRegionDisable
MOV    R0,    #6
BL     MpuRegionDisable
MOV    R0,    #7
BL     MpuRegionDisable
LDR    R0,    =0xE000ED94  ; MPU 控制寄存器
MOV    R1,    #1
STR    R1,    [R0]          ; 使能MPU
POP    {R0-R6, PC}        ; 返回

```

MpuRegionSetup

```

; MPU region 设置及启用子程
; 入口条件:
;   R0 = 基址
;   R1 = Region号
;   R2 = 容量
;   R3 = AP (访问许可)
;   R4 = MemAttrib ({TEX[2:0], S, C, B})
;   R5 = 子region除能
;   R6 = {XN,Enable}

PUSH   {R0-R1, LR}
BIC    R0,    R0,    #0x1F  ; 清零基址中铁定不会用到的位段
BFI    R0,    R1,    #0, #4  ; 把region号插入到R0[3:0]
ORR    R0,    R0,    #0x10  ; 置位VALID位
LDR    R1,    =0xE000ED9C  ; 加载MPU Region基址寄存器的地址
STR    R0,    [R1]          ; 填写之
AND    R0,    R6,    #0x01  ; 读取使能位
UBFX   R1,    R6,    #1, #1  ; 读取XN位
BFI    R0,    R1,    #28, #1 ; 把 XN 插入到 R0[28]
BFI    R0,    R2,    #1, #5  ; 把region容量(R2[4:0])插入到R0[5:1]中
BFI    R0,    R3,    #24, #3  ; 把AP(R3[2:0])插入到R0[26:24]中
BFI    R0,    R4,    #16, #6  ; 把memattrib(R4[5:0])插入到R0[21:16]中
BFI    R0,    R5,    #8, #8   ; 把子SRD(R5[7:0])插入到R0[15:8]中
LDR    R1,    =0xE000EDAO  ; 加载MPU Region属性及容量寄存器的地址
STR    R0,    [R1]          ; 填写之
POP    {R0-R1, PC}        ; 返回

```

```

MpuRegionDisable
; 该子程序用于除能一个region
; 入口条件: R0 = 待除能的region号
PUSH {R1, LR}
AND R0, R0, #0xF      ; region号只取低4位
ORR R0, R0, #0x10     ; 设置VALID位
LDR R1, =0xE000ED9C   ; 加载MPU Region 基址寄存器的地址
STR R0, [R1]           ; 填写之
MOV R0, #0
LDR R1, =0xE000EDAO   ; 加载MPU Region 属性及容量寄存器的地址
STR R0, [R1]           ; 把它归零, 这也蕴涵了除能的命令
POP {R1, PC}          ; 返回

```

在本例中, 我们还添加了一个用于除能和“复位”无用region的子程序。当你不知道某个region是否被用过时, 使用它来使其“归零”是最安全不过的了。

注意代码中位段操作的几行, 想想看, 如果用普通的移位和数据传送指令, 将会繁琐成什么样子!

14.4 MPU 的典型设置

在典型的情况下, 当需要阻止用户程序访问特权级的数据和代码时, 可以启用MPU。在设计MPU regions时, 需要考虑到下列的regions:

1. 代码region
 - a) 特权级代码, 包括初始的向量表
 - b) 用户级代码
2. SRAM region
 - a) 特权级数据, 包括主堆栈
 - b) 用户级数据, 包括进程堆栈
 - c) 特权级位带别名区
 - d) 用户级位带别名区
3. 外设
 - a) 特权级外设
 - b) 用户级外设
 - c) 特权级外设的位带别名区
 - d) 用户级外设的位带别名区
4. 系统控制空间 (NVIC以及调试组件)
 - a) 仅允许特权级访问

看, 上面列出了11个region, 已经超出了MPU支持的最多8个, 这可如何是好? 不怕, 还记得有个“背景region”吗? (忘了的话快去看图14.1)。我们可以把所有的特权级regions都归入背景region中(PRIVDEFENA=1)。这样一来, 就只需要明确定义用户级的regions——才5个。剩下3个后备的“槽”, 可以用于在外部RAM中(如果有的话)设置额外的regions, 也可以用于保护只读数据, 还能用于“没收”一部分的RAM等, 总之这是大虾们绽放智慧光芒的地方。

14.4.1 使用子region除能的示例

在上面的分析中, 外设是对用户开放的。但是如果误用某个外设可能导致严重后果的话, 我们就需要禁止用户级程序随意访问它。这样一来, 就会从外设存储器空间中割下几块肉, 使一个完整的空间变成若干个更小的。对付这种情况, 有如下的办法:

- 定义多个用户级外设regions
- 在用户级外设region中重叠地定义一个特权级的region
- 在用户级外设region中启用“子region除能机制”。

前两个办法很容易耗尽宝贵的8个“region槽”。芯片在设计时应为每个外设都开出相同容量的外设空间（用不完的就空着），这样才能让开发者容易使用第3种方法，即除能子region。通过除能子region，就很容易地从用户级region中擦掉一部分，让它回到背景region中了。一个例子如图14.3所示

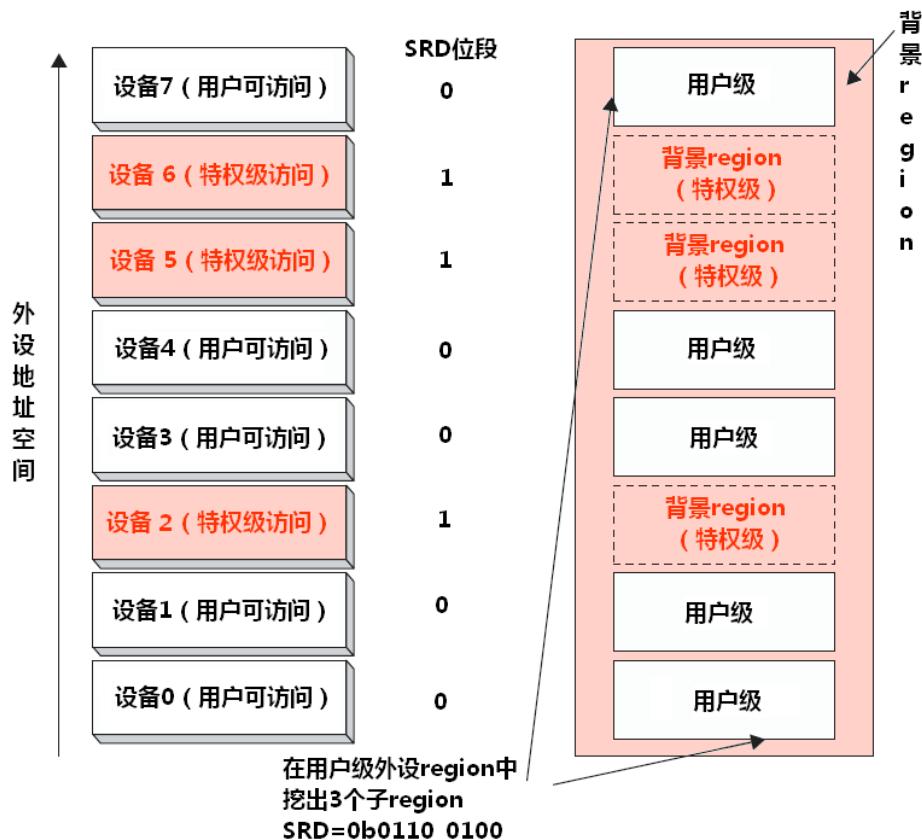


图14.3 “前景”的用户级region被SRD挖出子regions

该技巧也可以用在普通的存储器regions中，但这会使程序更加复杂，所以最好不要玩飘——通常是外设才需要此功能。在使用时，只要把上例中的子region除能参数改为非0即可，如

```
MOV      R5,      #0x64          ; Region 1: 子region 2, 5, 6被除能
```

最后，再根据上一个例子的框架，举一个可能在实际的单片机上会出现的例子：

```
MpuSetup          ; 入口函数，它内部呼叫若干子程序来完成MPU设置
PUSH    {R0-R6,LR}
LDR     R0,      =0xE000ED94    ; MPU 控制寄存器
MOV     R1,      #0
STR     R1,      [R0]          ; 配置前先除能MPU
; --- Region #0 --- 用户级程序
LDR     R0,      =0x00004000    ; Region 0: 基址 = 0x00004000
```

```

MOV    R1,    #0x0          ; Region 0: Region号 = 0
MOV    R2,    #0x0D         ; Region 0: 容量 = 0x0D (16KB)
MOV    R3,    #0x3          ; Region 0: AP = 0x3 (全访问)
MOV    R4,    #0x2          ; Region 0: TEX=0,S=0,C=1,B=0
MOV    R5,    #0x0          ; Region 0: 子region除能=0
MOV    R6,    #0x1          ; Region 0: {XN, Enable} = 0,1
BL     MpuRegionSetup

; --- Region #1 --- 用户级数据
LDR    R0,    =0x200000000 ; Region 1: 基址 = 0x20000000
MOV    R1,    #0x1          ; Region 1: Region号 = 1
MOV    R2,    #0x0B         ; Region 1: 容量 = 0x0B (4KB)
MOV    R3,    #0x3          ; Region 1: AP = 0x3 (全访问)
MOV    R4,    #0xB          ; Region 1: TEX=1,S=0,C=1,B=0
MOV    R5,    #0x0          ; Region 1: 子region除能= 0
MOV    R6,    #0x1          ; Region 1: {XN, Enable} =0,1
BL     MpuRegionSetup

; --- Region #2 --- 用户级位带别名区
LDR    R0,    =0x220000000 ; Region 2: 基址 = 0x22000000
MOV    R1,    #0x2          ; Region 2: Region号 = 2
MOV    R2,    #0x10         ; Region 2: 容量 = 0x010 (128KB)
MOV    R3,    #0x3          ; Region 2: AP = 0x3 (全访问)
MOV    R4,    #0xB          ; Region 2: TEX=1,S=0,C=1,B=0
MOV    R5,    #0x0          ; Region 2: 子region除能= 0
MOV    R6,    #0x1          ; Region 2: {XN, Enable} =0,1
BL     MpuRegionSetup

; --- Region #3 --- 用户级外设
LDR    R0,    =0x400000000 ; Region 3: 基址 = 0x40000000
MOV    R1,    #0x3          ; Region 3: Region号 = 3
MOV    R2,    #0x13         ; Region 3: 容量 = 0x013 (1MB)
MOV    R3,    #0x3          ; Region 3: AP = 0x3 (全访问)
MOV    R4,    #0x1          ; Region 3: TEX=1,S=0,C=1,B=0
MOV    R5,    #0x64         ; Region 3: 子region 2,5,6除能
MOV    R6,    #0x3          ; Region 3: {XN, Enable} =1,1
BL     MpuRegionSetup

; --- Region #4 --- 用户级外设的位带别名区
LDR    R0,    =0x420000000 ; Region 4: 基址 = 0x42000000
MOV    R1,    #0x4          ; Region 4: Region号 = 4
MOV    R2,    #0x18         ; Region 4: 容量 = 0x018 (32MB)
MOV    R3,    #0x3          ; Region 4: AP = 0x3 (全访问)
MOV    R4,    #0x1          ; Region 4: TEX=1,S=0,C=1,B=0
MOV    R5,    #0x64         ; Region 4: 子region 2,5,6除能
MOV    R6,    #0x3          ; Region 4: {XN, Enable} =1,1
BL     MpuRegionSetup

; --- Region #5 --- 外部RAM

```

```
LDR    R0,      =0x60000000    ; Region 5: 基址 = 0x60000000
MOV    R1,      #0x5          ; Region 5: Region号 = 5
MOV    R2,      #0x17         ; Region 5: 容量 = 0x010 (16MB)
MOV    R3,      #0x3          ; Region 5: AP = 0x3 (全访问)
MOV    R4,      #0xB          ; Region 5: TEX=0,S=0,C=1,B=1
MOV    R5,      #0x0          ; Region 5: 子region除能= 0
MOV    R6,      #0x1          ; Region 5: {XN, Enable} =0,1
BL     MpuRegionSetup

; --- Region #6 ---      未使用, 把它归零
MOV    R0,      #6
BL     MpuRegionDisable

; --- Region #7 ---      未使用, 把它归零
MOV    R0,      #7
BL     MpuRegionDisable
```

（原文中，上例加灰的指令把SRD设置为0x9B，即~0x64，看起来似乎是SRD的某个位为零时才除能对应的子region，与图14.3中给出的二进制数值相反。在译者查阅其它资料后仍然不能确定时，就求助于ARM了。感谢ARM的姜宁先生为译者肯定了正确的答案！——译者注）。

上例的代码执行后，建立的regions如下表所示（假设单片机有32KB flash, 8KB RAM）：

表14.8 上例代码执行后建立的各regions

地址范围	容量	类型	存储器属性 C, B, A, S, XN	MPU region	说明
0000_0000至 0000_3FFF	16KB	RO	C, -, A, -, -	背景	特权级程序
0000_4000至 0000_7FFF	16KB	RO	C, -, A, -, -	Region #0	用户级程序
2000_0000至 2000_0FFF	4KB	RW	C, B, A, -, -	Region #1	用户级数据
2000_1000至 2000_1FFF	4KB	特权级 RW	C, B, A, -, -	背景	特权级数据
2200_0000至 2001_FFFF	128KB	RW	C, B, A, -, -	Region #2	用户级数据的位带别名区
2202_0000至 2203_FFFF	128KB	特权级 RW	C, B, A, -, -	背景	特权级数据的位带别名区
4000_0000至 400F_FFFF	1MB	RW	-, B, -, -, XN	Region #3	用户级外设
4004_0000至 4005_FFFF	128KB	特权级 RW	-, B, -, -, XN	背景 Region #3中被删除的子region 2	在用户级外设地址范围中的特权级外设
400A_0000至 400B_FFFF	128KB	特权级 RW	-, B, -, -, XN	背景 Region #3中被删除的子region 5	在用户级外设地址范围中的特权级外设
400C_0000至 400D_FFFF	128KB	特权级 RW	-, B, -, -, XN	背景 Region #3中被删除的子region 6	在用户级外设地址范围中的特权级外设
4200_0000至 43FF_FFFF	32MB	RW	-, B, -, -, XN	Region #4	用户级外设的位带别名区
4280_0000至 42BF_FFFF	4MB	特权级 RW	-, B, -, -, XN	背景 Region #4中被删除的子region 2	在用户级外设位带别名区地址范围中的特权级外设
4340_0000至 437F_FFFF	4MB	特权级 RW	-, B, -, -, XN	背景 Region #4中被删除的子region 5	在用户级外设位带别名区地址范围中的特权级外设
4380_0000至 43BF_FFFF	4MB	特权级 RW	-, B, -, -, XN	背景 Region #4中被删除的子region 6	在用户级外设位带别名区地址范围中的特权级外设
6000_0000至 60FF_FFFF	16MB	RW	C, B, A, -, -	Region #5	外部RAM
E000_0000至 E00F_FFFF	1MB	特权级	-, -, -, -, XN	背景	NVIC, 调试组件, 以及私有外设总线

调试系统架构

- 调试特性概览
- CoreSight 技术概览
- 调试模式
- 调试事件
- Cortex-M3 中的断点
- 调试时访问寄存器
- 内核的其它调试特性

15.1 调试特性概览

一直以来，单片机的调试不是很突出的主题，很多山寨点的程序在开发中，甚至都没有调试的概念，而只是把生成的映像直接烧入片子，再根据错误症状来判断问题，然后修改程序重新烧，周而复始，直到问题解决或放弃为止。能够格算得上调试的活动，至少也是设置断点、观察寄存器和内存、监视变量等。使用仿真头和 JTAG（如 AVR），可以方便地实现这些基本的调试要求。在开发比较大的应用程序时，强劲的调试手段是非常重要的。当 bug 复杂到无法分析时，只能用调试来追踪它。如果没有调试手段，简直就束手无策。

正因为此，在 CM3 中，调试机能突然在一夜之间，就从丑小鸭变成了白天鹅，得到了登峰造极般的，令人非常惊艳的强化。CM3 提供了多种多样的调试模型和调试组件，很多想到的和没想到的调试方式这里都有，让人惊叹“原来调试还可以做到这种程度”。为了方便进一步学习，我们把 CM3 丰满的调试功能分为两类，每类中都有更具体的调试项目，如下所列：

侵入式调试（这也是基本的调试机能）

- a) 停机以及单步执行程序
- b) 硬件断点
- c) 断点指令（BKPT）
- d) 数据观察点，作用于单一地址、一个范围的地址，以及数据的值。
- e) 访问寄存器的值（既包括读，也包括写）
- f) 调试监视器异常
- g) 基于 ROM 的调试（闪存地址重载(flash patching)）

非侵入式调试（大多数人更少接触到的，高级的调试机能）

- h) 在内核运行的时候访问存储器
- i) 指令跟踪，需要通过可选的嵌入式跟踪宏单元（ETM）
- j) 数据跟踪
- k) 软件跟踪（通过 ITM（指令跟踪单元））
- l) 性能速写（profiling）（通过数据观察点以及跟踪模块）

可见，我们以前最常用的调试都属于侵入式调试。所谓“侵入式”，主要是强调这种调试会打

破程序的全速运行。非侵入式调试则是锦上添花的一类，当调试大型软件和多任务环境下的软件系统时，非侵入式调试有不可替代之强大功效。

在 CM3 处理器的内部，包含了一系列的调试组件。CM3 的调试系统基于 ARM 亲手打造且吐血推荐的“CoreSight（内核景象）”调试架构。该架构是一个专业设计的体系，它允许使用标准的方案来访问调试组件，收集跟踪信息，以及检测调试系统的配置。

15.2 CoreSight 技术概览

CoreSight 调试架构的定义简直包罗万象，包括调试接口协议、调试总线协议、对调试组件的控制、安全特性、跟踪接口等。在《CoreSight Technology System Design Guide(Ref3)》中，对 CoreSight 有详细的讲述，此外，在 Cortex-M3 TRM 中也开出了若干章，专门叙述 CM3 中调试组件的设计。但是这些内容通常只是给设计调试软件的人看的，我们软硬件开发者不要陷得太深。不过，懂一点调试系统的组成结构和基本工作原理，还是很有助于让我们善加利用这强大无比的调试系统，大幅加速程序的开发的。

15.2.1 处理器的调试接口

CM3 的调试系统已经与 ARM7/ARM9 的大相径庭了，基于新好 CoreSight 架构，它从头到脚都是新的。以前的 ARM 处理器都提供 JTAG 接口，通过它来控制对寄存器和存储器的访问。在 CM3 中全变了——对处理器上总线逻辑的控制使用另外的总线接口，即通过所谓的“调试访问端口(DAP)”。DAP 与 AMBA 中的 APB 很相似。在 CM3 中，把 JTAG 或串行线协议都转换成 DAP 总线接口协议，再控制 DAP 来执行调试动作。

CM3 内部的调试总线 DAP 是 APB 的近亲，所以很容易在它上面挂上很多调试组件，从而使得调试系统可大可小，伸缩性极强。此外，把调试接口和调试硬件分开，也是颇具匠心的：芯片中实际使用的调试接口类型变得透明化。从而不管使用了什么样的调试接口，相同的调试任务都可以按照同一个方式执行。

在 CM3 处理器内核中，实际的调试功能由 NVIC 和若干调试组件来协作完成。调试组件包括 FPB，DWT，ITM 等。NVIC 中有一些寄存器，用于控制内核的调试动作，如停机、单步；其它的一些功能块则控制观察点、断点，以及调试消息的输出等。

就目前来看，CM3 支持两种调试主机接口（`debug host interface`）：第一个是广为使用的 JTAG 接口，另一个则是新的“串行线(Serial Wire, SW)调试接口”。新出的 SW 接口对信号线的需求只有两条。ARM 公司还提供了若干种调试主机接口模块（称为“调试接口”（DP））。DP 充当处理器与调试器的中介：它的一端连接到调试器上，另一端则连接到 CM3 的 DAP 接口上。

选择串行线的理由

CM3 主要针对低成本的单片机市场。单片机往往没有很富裕的管脚资源。而 JTAG 协议需要使用 4 根脚，而 SW 则只需要两根。

15.2.2 DP 模块，AP 模块和 DAP

从外部调试器到 CM3 调试接口的连接，需要多级互联才能完成，如图 15.1 所示。

第一步，是通过 DP 接口模块（通常是 SWJ-DP 或 SW-DP），先把外部信号转换成一个通用的 32 位调试总线信号（图表中的 DAP 总线）。SWJ-DP 支持 SW 与 JTAG 两种协议，而 SW-DP 则只支持 SW。另外，在 CoreSight 产品中还可以使用一种 JTAG-DP，它只支持 JTAG 协议。DAP 总线上的地址是

32 位的，其中高 8 位用于选择访问哪一个设备，由此可见，最多可以在 DAP 总线上面挂 256 个设备。在 CM3 处理器的内部，只用掉了一个设备的地址，还剩下的 255 个都可以用于连接访问端口(AP)到 DAP 总线上。

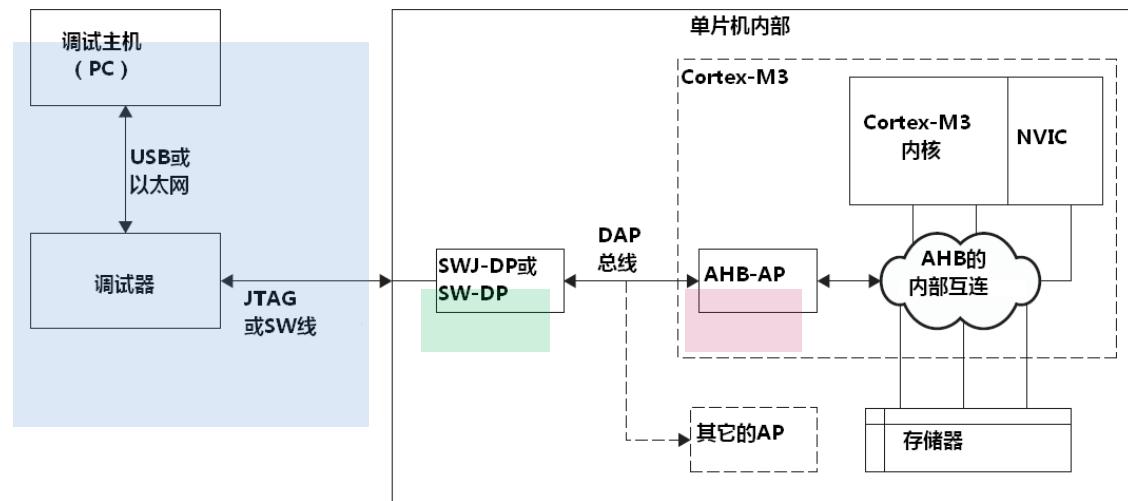


图 15.1 高度主机到 Cortex-M3 的连接

在把数据从 DAP 接口传递给 CM3 处理器后，下一步就连接到了一个称为“AHB-AP”的 AP 设备上，它相当于一个总线桥，用于把 DAP 总线的命令转换为 AHB 总线上的数据传送，再插入到 CM3 内部的总线网络中。这么一来，CM3 的整个寻址空间就都在覆盖范围之内了，连 NVIC 中的调试控制寄存器组也包括在内。在 CoreSight 系列产品中，AP 设备可以有好几种类型，包括 APB-AP 和 JTAG-AP。APB-AP 顾名思义，是用于产生 APB 总线数据传送动作的，而 JTAG-AP 则用于控制传统的、基于 JTAG 的测试接口，例如 ARM7 上的调试接口。

15.2.3 跟踪接口

CoreSight 架构的另一个部分用于跟踪。在 CM3 中有 3 种跟踪源：

1. **指令跟踪**：由 ETM（嵌入式跟踪宏单元）产生
2. **数据跟踪**：由 DWT 产生
3. **调试消息**：由 ITM 产生，提供形如 `printf` 的消息输入，送到调试器的 GUI 中

在跟踪过程中，先把跟踪源产生的数据裹成数据包，然后把数据包送到“高级跟踪总线(ATB)”上进行传送。在 CoreSight 的架构中，如果某 SoC 含有多个跟踪源（例如，多核系统），则需要一种硬件水平的 ATB 归并器（merger），把 ATB 数据流归并成一条（在 CoreSight 架构中，这种硬件被命名为 ATB funnel）。归并后的数据流都送往 TPIU（跟踪端口接口单元），TPIU 再把数据导出到片外的跟踪硬件设备。在数据送到了调试主机（PC）后，再由 PC 端的调试软件还原为先前的多条数据流。

尽管在 CM3 中拥有多个跟踪源，但 CM3 内建了一个归并硬件，因此不需要再添加 ATB funnel 模块了。跟踪输出接口可以直接连接到专为 CM3 设计的 TPIU 上，然后就可以供 PC 控制的外部硬件捕捉仪来跟踪数据。

15.2.4 CoreSight 的性质

基于 CoreSight 的调试设计有很多优势：

- 即使在处理器运行时，也可以查看存储器和外设的寄存器的内容
- 使用单一调试器，就可以控制多核系统的调试接口。例如，如果使用 JTAG，则只需要一个 TAP 控制器，不管芯片中有几个处理机都一样。
- 内部的调试接口是基于单总线的方式设计的，因此非常有弹性，也简化了为芯片的其它部分设计附加的测试逻辑。
- 它使得多条“跟踪数据流”可以由单一的“跟踪捕获设备”来收集，送到 PC 机上之后再还原出先前的各条数据流。

CM3 中的调试系统是基于 **CoreSight** 的，但是又有一些“变异”：

- CM3 的跟踪组件是重新设计的，有些在 CM3 中的 ATB 接口是 8 位的，而纯种的 **CoreSight** 的都是 32 位的。
- CM3 的调试系统没有实现 **TrustZone**——ARM 提供的一种技术，用于在嵌入式产品中提供安全特性。
- 调试组件所需的空间挤到了系统的存储器映射中。而在标准的 **CoreSight** 系统中，是为调试总线另开了一个地址空间的。例如，在 **CoreSight** 系统中，系统连接的概念图如图 15.2 所示：

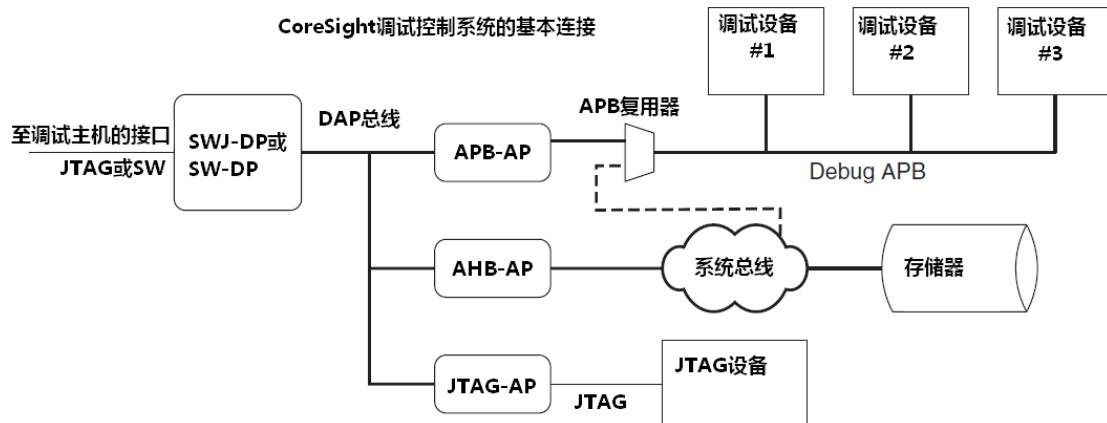


图 15.2 CoreSight 系统设计概念图

而在 CM3 中，调试设备共享同一个同一个存储器映射，如图 15.3 所示

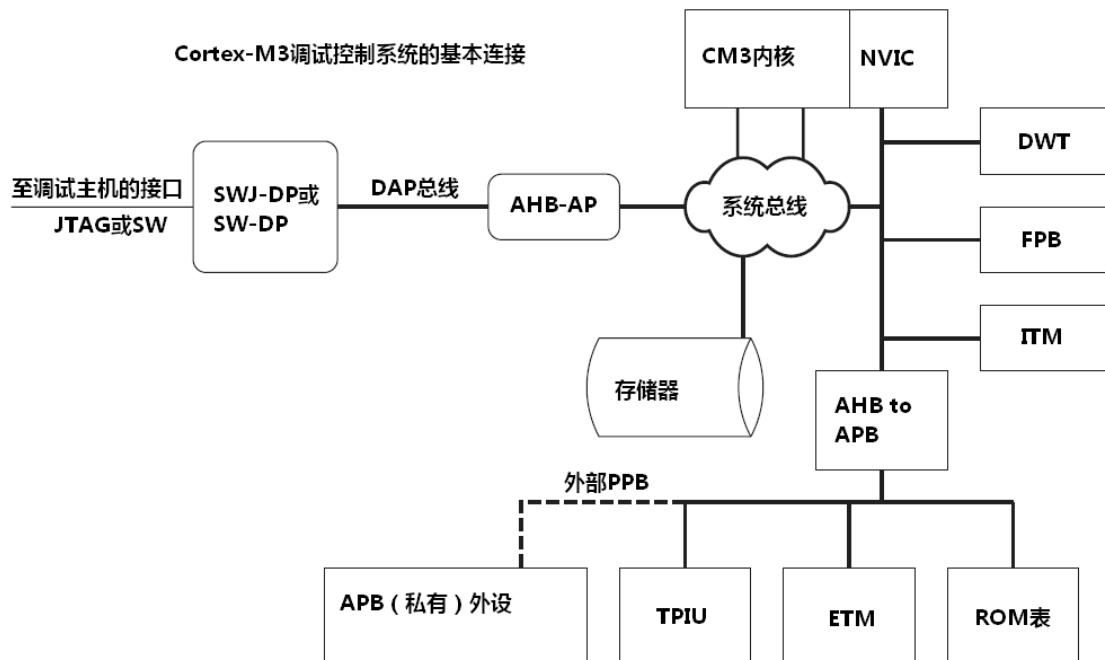


图 15.3 Cortex-M3 的调试系统

尽管 CM3 的调试组件在实现上与标准 CoreSight 系统的有些出入，但是通信接口与协议是与 CoreSight 架构兼容的，并且可以直接挂接到 CoreSight 系统上，标准 CoreSight 的调试组件也可以挂接到 CM3 上。例如，(标准) CoreSight 调试组件，诸如 TPIU，调试端口以及跟踪基础设施等，可以供 CM3 使用，并且以此来把调试能力延伸到多核调试系统中。

关于 CoreSight 架构的更多内容，请参阅《CoreSight Technology System Design Guide(Ref3)》。

15.3 调试模式

在 CM3 中的调试操作模式分为两种。第一种称为“halt”（停机模式），在进入此模式时，处理器完全停止程序的执行。第二种则称为“debug monitor exception”（调试监视器模式），此时处理器执行相应的调试监视器异常服务例程，由它来执行调试任务，并且依然允许更高优先级的异常抢占它。调试监视器的异常号为 12，优先级可编程。除了调试事件可以触发异常外，手工设置其悬起位也可以触发本异常。

1. 停机模式

- 指令执行被停止
- SysTick 定时器停止
- 支持单步操作
- 中断可以在这期间悬起，并且可以在单步执行时响应。也可以掩蔽它们，使得单步时不受干扰

2. 调试监视器模式

- 处理器执行调试监视器异常的服务例程（异常号：12）
- SysTick 定时器继续运行
- 新来的中断按普通执行时的原则来抢占
- 执行单步操作
- 存储器的内容（如堆栈内存）会在调试监视器的响应始末得到更新，因为有自动入栈和出栈的动作

之所以加入调试监视器模式，是考虑到了在某些电子系统运行的过程中，是不可以停机的。例如，对于汽车引擎控制器以及电机控制器，就必须在处理调试动作的同时让处理器继续运行下去，这样才能保证被测试的设备不会意外损坏（例如，不需要在调试过程中让电机停转——译者注）。有了调试监视器，就可以停止并调试线程级的应用程序，也可以调试低优先级的中断服务例程。在这同时，高优先级的中断和异常能够响应。

如果要进入停机模式，需要把 NVIC 调试停机控制及状态寄存器（DHCSR）的 C_DEBUGEN 位置位。这个位只能由调试器来设置，没有调试器是不能把 CM3 停机的。在 C_DEBUGEN 置位后，就可以设置 DHCSR.C_HALTI 位来喊停处理器。此 C_HALTI 位可以由软件置位。

DHCSR 的位段定义比较特殊：读时是一种定义，写时又是另外一种定义。对于写操作，必须先往[31:16]中写入一个“访问钥匙”值。而对于读操作，则无此钥匙，并且读回来的高半字包含了状态位，如表 15.1 所示。

表 15.1 调试停机控制及状态寄存器 DHCSR（地址：0xE000_EDF0）

位段	名称	类型	复位值	描述
----	----	----	-----	----

31:15	KEY	W	-	调试钥匙。必须在任何写操作中把该位段写入 A05F, 否则忽略写操作
25	S_RESET_ST	R	-	内核已经或即将复位, 读后清零
24	S_RETIRE_ST	R	-	在上次读取以后指令已执行完成, 读后清零
19	S_LOCKUP	R	-	1=内核进入了锁定状态
18	S_SLEEP	R	-	1=内核睡眠中
17	S_HALT	R	-	1=内核已停机
16	S_REGRDY	R	-	1=寄存器的访问已经完成
15:6	保留	-	-	
5	C_SNAPSTALL	RW	0*	打断一个 stalled 存储器访问
4	保留	-	-	
3	C_MASKINTS	RW	0*	调试期间关中断, 只有在停机后方可设置
2	C_STEP	RW	0*	让处理器单步执行, 在 C_DEBUGEN=1 时有效
1	C_HALT	RW	0*	喊停处理器, 在 C_DEBUGEN=1 时有效
0	C_DEBUGEN	RW	0*	使能停机模式的调试

*: DHCSR 中的控制位是在上电复位时得到复位的。系统复位(例如, 往 NVIC 应用程序中断及复位寄存器中写命令)不会影响到它们

在正常情况下, 只有调试器会操作 DHCSR, 应用程序不要乱动它, 以免使调试工具出现问题。

当使用调试监视器模式时, 由另一个 NVIC 中的寄存器来负责控制调试活动, 它是 NVIC 调试异常及监视器控制寄存器 (DEMCR), 其定义如表 15.2 所示。

表 15.2 调试及监视器控制寄存器 DEMCR (地址: 0xE000_EDFC)

位段	名称	类型	复位值	描述
24	TRCENA	RW	0*	跟踪系统使能位。在使用 DWT, ETM, ITM 和 TPIU 前, 必须先设置此位
23:20	保留			
19	MON_REQ	RW	0	1=调试监视器异常不是由硬件调试事件触发, 而是由软件手工悬起的
18	MON_STEP	RW	0	让处理器单步执行, 在 MON_EN=1 时有效
17	MON_PEND	RW	0	悬起监视器异常请求, 内核将在优先级允许时响应
16	MON_EN	RW	0	使能调试监视器异常
15:11	保留			
10	VC_HARDERR	RW	0*	发生硬 fault 时停机调试
9	VC_INERR	RW	0*	指令/异常服务错误时停机调试
8	VC_BUSERR	RW	0*	发生总线 fault 时停机调试
7	VC_STATERR	RW	0*	发生用法 fault 时停机调试
6	VC_CHKERR	RW	0*	发生用法 fault 使能的检查错误时停机调试 (如未对齐, 除数为零)
5	VC_NOCPERR	RW	0*	发生用法 fault 之无处理器错误时停机调试
4	VC_MMERR	RW	0*	发生存储器管理 fault 时停机调试
3:1	保留			

0	VC_CORERESET	RW	0*	发生内核复位时停机调试
---	--------------	----	----	-------------

*: DEMCR 中的控制位是在上电复位时得到复位的。系统复位 (例如, 往 NVIC 应用程序中断及复位寄存器中写命令) 不会影响到它们

该寄存器不仅包含了调试监视器的控制位, 还包含了跟踪系统的使能位 (TRCENA) 以及若干向量捕获 (Vector Catch, VC) 控制位。VC 功能只有在停机模式下才能使用。如果某个异常 (或者内核复位) 发生了, 并且对应的 VC 位置位, 则将自行产生一个停机请求, 并且在执行完当前指令后立即把处理器喊停。

虽然 TRCENA 和 VC 控制相关的位只有上电时才复位, 但是其它用于控制监视器模式的位, 则也会因系统复位而复位。

15.4 调试事件

Cortex-M3 可以由很多种理由进入调试模式 (both 停机模式和调试监视器模式)。对于停机模式, 满足图 15.4 所示的条件可以喊停处理器。但即使是停机后, 也可由上电复位和系统复位来复位处理器。

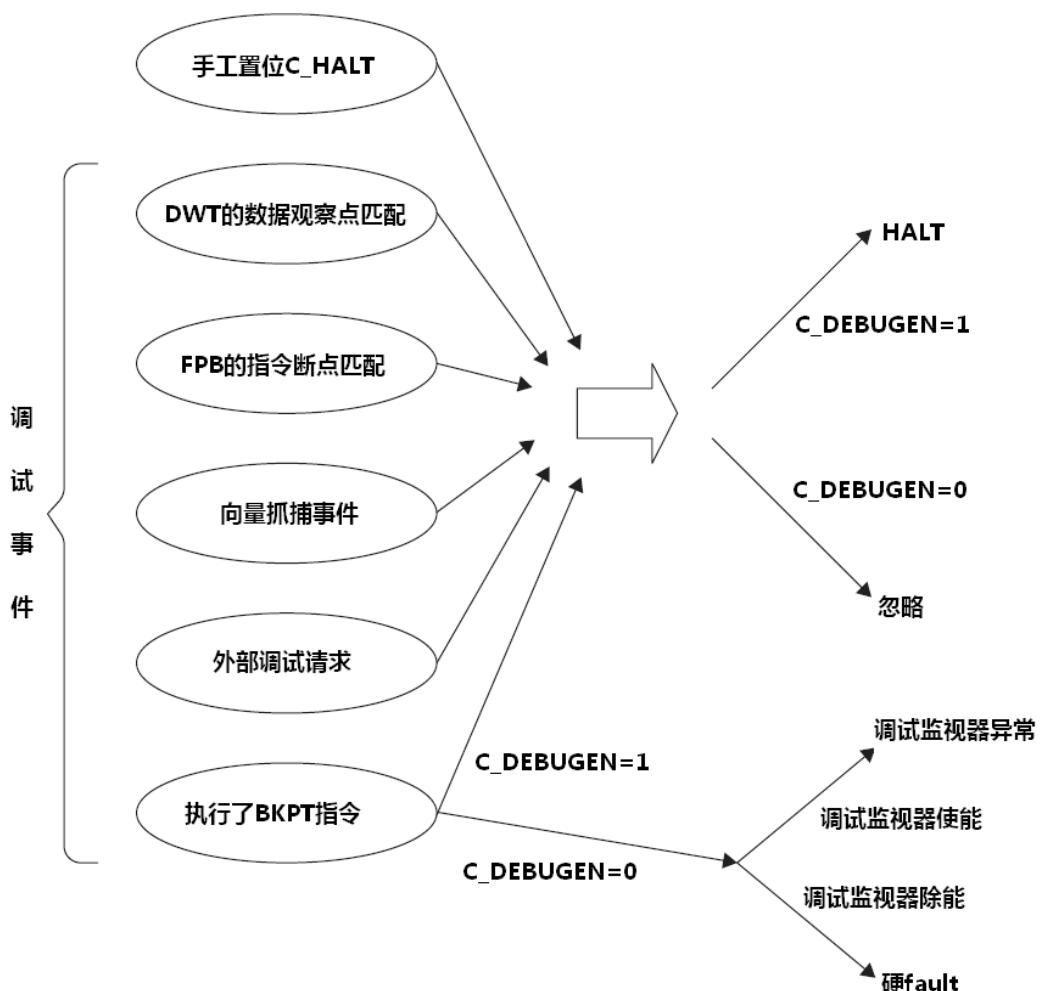


图 15.4 停机模式下对调试事件的响应

图中，外部调试请求信号是通过 CM3 上的一个称为“EDBGRQ”的信号线传来的，该信号线的实际连接方式取决于单片机/SoC 的设计。在有些场合下可以把该信号硬线连至低电平，从而使外部调试请求永远无法送达；也可以把它连接到附加的调试组件上（芯片厂商可以添加额外的调试组件）；或者在多核系统中，可以用来连接其它处理机的调试事件。

在调试活动完成后，通过清除 C_HALT 位，可以让程序继续执行。

类似地，在调试监视器模式下，也可以由一系列的调试事件来进入调试模式，如图 15.5 所示。

从图中可见，在调试监视器模式下，与在停机模式下的动作方式还是有一点区别的。这是因为调试监视器异常仅仅是异常的一种，它可以影响当前的优先级，但是不能使处理器停下来。

在调试活动完成后，通过该异常的返回，即可回到正常的程序执行中。

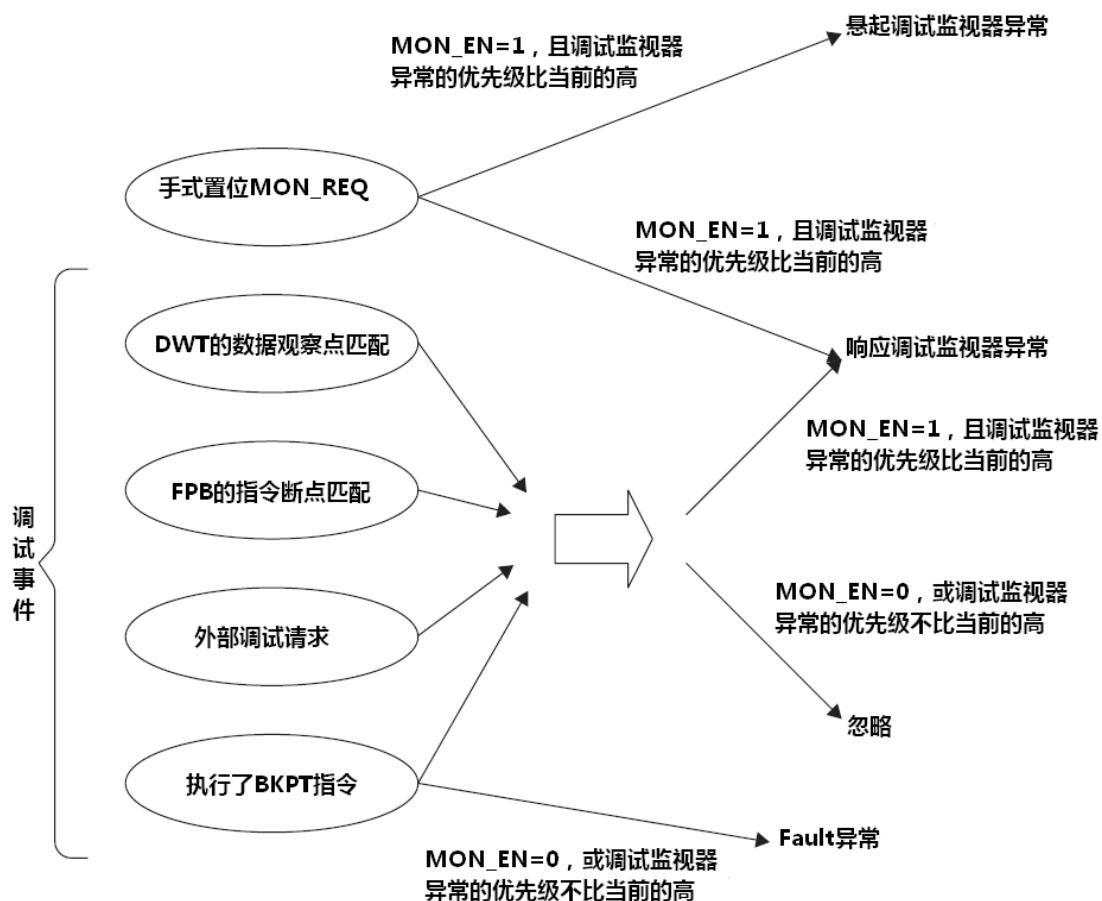


图 15.5 调试监视器模式下对调试事件的响应

15.5 Cortex-M3 中的断点

在大多数单片机中，用得最多的可能就是断点了。在 CM3 中，有两种断点机制：

- 断点指令
- 基于 FPB 地址比较器的断点

断点指令的格式为 BKPT #im8，它是一个 16 位的 Thumb 指令，编码为 0xBExx——其低 8 位就是指令中 #im8 的值。当该指令执行时，会产生一个调试事件。当 C_DBGEN 置位时可以用于喊停处理器内核；或者当调试监视器使能时，触发调试监视器异常。对于后者，因为调试监视器异常也是一种优先级可编程的普通异常，所以也可以因为其优先级不够高而不能立即响应。可见，因为 NMI 和硬

fault的优先级总是比它的高，所以不能在它们的服务例程中使用**BKPT**指令来启动调试——只有在它们返回时才能响应调试监视器异常。

使用**BKPT**时另一个要注意的是，当调试监视器异常返回后，它返回到的是**BKPT**指令的地址，而不是返回**BKPT**后面一条指令的地址。这与常规的异常返回是不同的，原因在于，在正常情况下使用**BKPT**指令时，**BKPT**用于取代一条正常的指令，并且当命中了该断点而执行了调试动作后，把该**BKPT**指令所占用的内存恢复为先前被**BKPT**取代的指令，并且让该指令是下一条即将执行的指令，而其它的部分不受影响(这其实也是软件断点的实现方式)。

如果在**BKPT**指令执行时却发现**C_DEBUGEN**和**MON_EN**都为0，则会因为无法进入调试而上访成硬**fault**，并且把硬**fault**状态寄存器(**HFSR**)的**DEBUGEVT**位给置1，同时在调试**fault**状态寄存器(**DFSR**)中的**BKPT**位也置1。

如果程序存储器的值不能更改，则可以通过编程**FPB**来产生硬件断点。但是，只支持6个指令地址和两个文字地址。下一章将展开叙述**FPB**。

使用**BKPT**指令取代正常指令，以及对**FPB**的编程，通常都是在我们设置断点时，由调试器负责做的事。

15.6 调试时访问寄存器

在**NVIC**中，还有两个寄存器与与调试功能有关。它们分别是：调试内核寄存器选择者寄存器(**DCRSR**)，以及调试内核寄存器数据寄存器(**DCRDR**)，如表15.3和表15.4所示。调试器需要通过这两个寄存器来访问处理器的寄存器，并且只有在处理器停机时，才能使用这里的寄存器传送功能。

表 15.3 调试内核寄存器选择者寄存器 DCRSR (地址 : 0xE000_EDF4)

位段	名称	类型	复位值	描述
16	REGWnR	W	-	1=写寄存器 0=读寄存器
15:5	保留	-	-	-
4:0	REGSEL	W	-	00000= R0 00001=R1 ... 01111=R15 10000=xPSR 10001=MSP 10010=PSP 10100=特殊功能寄存器组 [31:24]: CONTROL [23:16]: FAULTMASK [15:8]: BASEPRI [7:0]: PRIMASK

表 15.4 调试内核寄存器数据寄存器 DCRDR (地址 : 0xE000_EDF8)

位段	名称	类型	复位值	描述
31:0	DATA	R/W	-	读回来的寄存器的值，或欲写入寄存器的值，寄

存器由 DCRSR 选择

欲使用这两个寄存器来读取内核的寄存器的内容，则必须按如下的顺序做：

1. 确定处理器已停机
2. 往DCRSR写数据，其中位16要为0，表示这是要读数据
3. 查询，直到DHCSR.S_REGRDY=1
4. 读取DCRDR以获取寄存器的内容

寄存器写操作的顺序与上面的类似：

1. 确定处理器已停机
2. 往DCRDR中写数据
3. 往DCRSR写数据，其中位16要为1，表示这是要写数据
4. 查询，直到DHCSR.S_REGRDY=1

使用DCRSR和DCRDR来访问寄存器，只适用于停机模式。如果选择了调试监视器模式，则对于自动入栈的寄存器，可以从堆栈中读写它们；对于其它寄存器，就可以直接在服务例程中访问。

如果有合适的函数库和调试器的支持，还可以使用DCRDR来做半主机(semihosting)。比如说，当应用程序执行了printf语句时，文字的输出可以通过一系列的putc()调用来完成。在实现putc()时，可以让它把输出的字符和状态写到DCRDR中，然后触发调试模式。接下来，调试器可以检测到内核停机状态，并且读取被输出的字符。然而，这种形式的半主机需要喊停内核。更正点的半主机是使用ITM，它则没有此限制。

15.7 内核的其它调试特性

在NVIC中，还有其它一些与调试有关的特性，它们包括：

- **外部调试请求信号**：NVIC提供了一个外部调试请求信号，通过它可以让CM3处理器由外部调试事件触发而进入调试模式。举一个外部调试事件的例子：在多核系统中，可以是其它处理器的调试状态，这对于调试多核系统的意义决非等闲。如果是单核的单片机，则基本上是把该信号拉低。
- **调试fault状态寄存器**：因为在CM3上有多种调试事件，故而设置了一个DFSR，以资调试器来判断是发生了哪种调试事件。
- **复位控制**：在调试期间，可以使用VECTRESET控制位来重启处理器内核（位于NVIC应用程序中断及复位控制寄存器中（地址：0xE000_ED0C））。通过使用这种方式，可以让处理器的复位波及到调试系统。
- **中断掩蔽**：在单步时这个功能是非常体贴的。因为在单步时，往往是为了集中精力分析某段代码的逻辑，此时不希望受到任何骚扰，哪怕是响应中断也是很讨厌的事。通过置位C_MASKINTS位（在调试停机控制及状态寄存器中，（地址：0xE000_EDF0）），就可以在单步期间掩蔽中断。
- **终止Stalled总线传送**：如果一个总线传送被stall了一个很长的时间，就可以强制终结它。在调试停机及状态寄存器中有一个C_SNAPSTALL位，把它置位即可。但是这个功能只有在停机模式下才能由调试器使用。

调试组件

- 简介
- 跟踪组件：数据观察点与跟踪(DWT)
- 跟踪组件：仪器化跟踪宏单元(ITM)
- 跟踪组件：嵌入式跟踪宏单元(ETM)
- 跟踪组件：跟踪端口接口单元(TPIU)
- 闪存地址重载与断点单元(FPB)
- AHB 访问端口
- ROM 表

16.1 简介

在 CM3 的大礼包中有很多调试组件，使用它们可以执行各种调试功能：断点、数据观察点、闪存地址重载以及各种跟踪等。如果您是一位软件开发人员，则也许永远无需了解调试组件的细节，因为它们通常只是由调试器及其周边工具使用的。

本章对每种调试组件做一个基本的介绍，如果需要了解它们的更详细信息，如编程模型，则请参阅《Cortex-M3 Technical Reference Manual(Ref1)》。

所有的调试及跟踪组件，以及 FPB，都可以经由 CM3 的私有外设总线来编程。在大多数情况下，只有调试主机才会编程这些组件。强烈反对应用程序尝试访问调试组件（除了对 ITM 中 stimulus 端口寄存器的访问），这样做很容易与调试器发生冲突。

16.1.1 Cortex-M3 的跟踪系统

如前所述，CM3 的跟踪系统是基于 CoreSight 架构的，跟踪数据被打成数据包，并且它们的长度可变。跟踪组件使用高级跟踪总线(ATB)来发送这些数据包给 TPIU，TPIU 则把它们格式化，转换成符合“跟踪总线接口协议”的数据包。格式化后的数据包发到片外，可以使用跟踪端口分析仪(TPA)之类的设备捕获它们。整个数据流动的路线如图 16.1 所示：

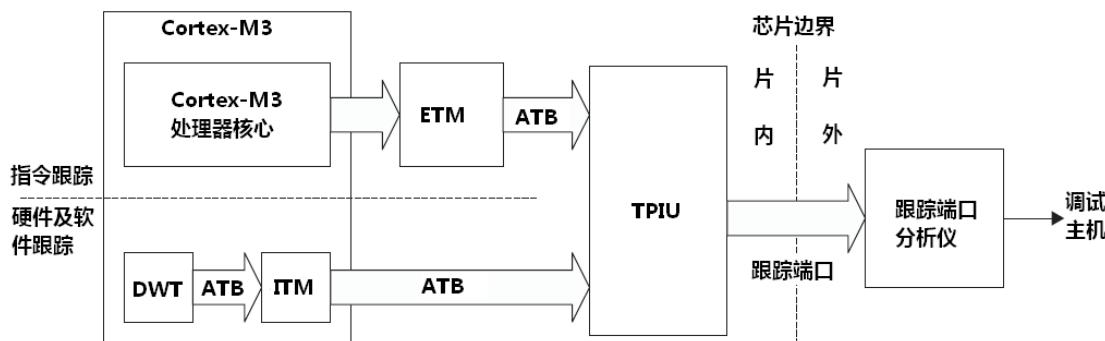


图 16.1 Cortex-M3 的消化系统模式图

从上图可见，在 CM3 中可以有 3 种跟踪源：ETM, ITM 和 DWT。其中，ETM 是一个可选组件，因此有些 CM3 芯片中没有配。在操作中，每个跟踪源都被赋予一个 7 位的 ID 号 (ATID)，随着它

所发出的数据包一起送出。这样，在从归并的数据流中还原各原始的数据流时，就可以使用 **ATID** 来作为识别的手段。与其它标准的 **CoreSight** 组件不同的是，CM3 的调试组件内建了归并 **ATB** 数据流的逻辑；而在标准的 **CoreSight** 系统中，**ATB** 数据包归并器是一个独立的功能块，并且被称为“**ATB funnel**”。

在使用跟踪系统之前，必须把 **DEMCR.TRCENA** 置位（回顾表 15.2，或者参阅表 D.37）。在这之前，跟踪系统是处于除能状态的。在正常的操作中，如果不需要跟踪，则通过清零 **TRCENA** 来除能一些与跟踪有关的逻辑，可以降低系统的功耗。

16.2 跟踪组件：数据观察点与跟踪(DWT)

本节的主角是 **DWT**，它提供的调试功能包括：

1. 它包含了 4 个比较器，可以配置成在发生比较匹配时，执行如下动作：
 - a) 硬件观察点（产生一个观察点调试事件，并用它来调用调试模式，包括停机模式和调试监视器模式）
 - b) **ETM** 触发，可以触发 **ETM** 发出一个数据包，并汇入指令跟踪数据流中
 - c) 程序计数器（**PC**）采样器事件触发
 - d) 数据地址采样器触发
 - e) 第一个比较器还能用于比较时钟周期计数器（**CYCCNT**），用于取代对数据地址的比较
2. 作为计数器，**DWT** 可以对下列项目进行计数：
 - a) 时钟周期（**CYCCNT**）
 - b) 被折叠（**Folded**）的指令
 - c) 对加载/存储单元（**LSU**）的操作
 - d) 睡眠的时钟周期
 - e) 每指令周期数（**CPI**）
 - f) 中断的额外开销（**overhead**）
3. 以固定的周期采样 **PC** 的值
4. 中断事件跟踪

当用于硬件观察点或 **ETM** 触发时，比较器既可以比较数据地址，也可以比较程序计数器 **PC**。当用于其它功能时，比较器则只能比较数据地址。

每一个比较器都有 3 个寄存器

- **COMP** 寄存器
- **MASK** 寄存器
- **FUNCTION** 控制寄存器

其中，**COMP** 寄存器是一个 32 位寄存器，用于存储要比较的值。**MASK** 寄存器可以用于掩蔽数据地址的一些位，被掩蔽的位不参与比较。如表 16.1 所示：

表 16.1 **MASK** 寄存器定义

MASK	被忽略的位段
0	忽略所有的位
1	忽略[0]

2	忽略[1:0]
3	忽略[2:0]
...	
15	忽略[14:0]

比较器的 **FUNCTION** 寄存器用于决定该比较器的功能。为了避免潜在的不可预料的行为，必须先编程 **MASK** 和 **COMP**，最后再编程 **RUNCTION**。如果要更改某个比较器的功能，必须先把 **FUNCTION** 清零——除能该比较器，再重新配置一回，依然是最后配置 **FUNCTION**。

DWT 中有剩余的计数器，它们典型地用于程序代码的“性能速写”(**profiling**)。通过编程它们，就可以让它们在计数器溢出时发出事件(以跟踪数据包的形式)。最典型地，就是使用 **CYCCNT** 寄存器来测量执行某个任务所花的周期数，这也可以用作时间基准相关的目的(操作系统中统计 CPU 使用率可以用到它)。

16.3 跟踪组件：仪器化跟踪宏单元 (ITM)

ITM 有如下的功能：

- 软件可以直接把控制台消息写到 **ITM stimulus** 端口，从而把它们输出成跟踪数据。
- **DWT** 可以产生跟踪数据包，并通过 **ITM** 把它们输出。
- **ITM** 可以产生时间戳数据包并插入到跟踪数据流中，用于帮助调试器求出各事件的发生时间。

因为 **ITM** 要使用跟踪端口来输出数据，所以芯片上必须有 **TPIU** 单元，否则无法输出——在使用 **ITM** 前要确认此事。如果不幸地没有 **TPIU**，也还可以使用 **NVIC** 调试寄存器，或者使用最后一招——求助于 **UART** 来输出控制台消息。

欲使用 **ITM**，必须把 **DEMCR.TRCENA** 位置位，否则 **ITM** 处于除能状态，无法使用。

另外，在 **ITM** 寄存器中还有一个锁。在编程 **ITM** 之前，必须写入一个访问钥匙值 **0xC5AC_CE55** (**CoreSight** 的 **ACCESS**) 到这个解锁寄存器。否则，所有对 **ITM** 寄存器的写操作都被忽略。

最后，**ITM** 自己也是另一个控制寄存器(可能是说控制寄存器的名字也是“**ITM**”吧)，用于控制对各功能的独立使能。

控制寄存器中含了 **ATID** 位段，作为 **ITM** 在 **ATB** 中的 **ID** 值。这个 **ID** 必须是唯一的——每个跟踪源都必须有唯一的 **ID** 值，从而使调试主机能从接收到的跟踪数据包中分离出各跟踪源的数据。

16.3.1 基于 **ITM** 的软件跟踪

ITM 的一个主要用途，就是支持调试消息的输出(例如，**printf** 格式的输出)。**ITM** 包含了 32 个刺激(**stimulus**)端口，允许不同的软件把数据输出到不同的端口，从而让调试主机可以把它们的消息分离开。通过编程“跟踪使能寄存器”，每个端口都可以独立地使能/除能，还可以允许或禁止用户进程对它执行写操作。

与基于 **UART** 的文字输出不同，使用 **ITM** 输出不会对应用程序造成很大的延迟。在 **ITM** 内部有一个 **FIFO**，它使写入的输出消息得到缓冲。不过，为了安全起见，最好还是在写入前检查该 **FIFO** 被填满的程度。

输出的消息被送往 **TPIU**，然后可以通过“跟踪端口接口”或者“串行线接口”来收集它们。在最终的代码中也无需移除产生调试消息的代码，而是可以把 **TRCENA** 位清零，这样 **ITM** 就被除能，

调试消息也不会输出，你也可以在一个“live”系统中开启消息输出。另外，通过设置跟踪使能寄存器，可以限定允许使用的端口。

16.3.2 基于 ITM 和 DWT 的硬件跟踪

ITM 也能用于输出硬件跟踪数据，这些数据由 DWT 产生，ITM 则担任跟踪数据包的归并单元，如图 16.2 所示。欲使用 DWT 跟踪，需要在 ITM 控制寄存器中置位 DWTEN 位，剩下的 DWT 跟踪设置在 DWT 中完成。

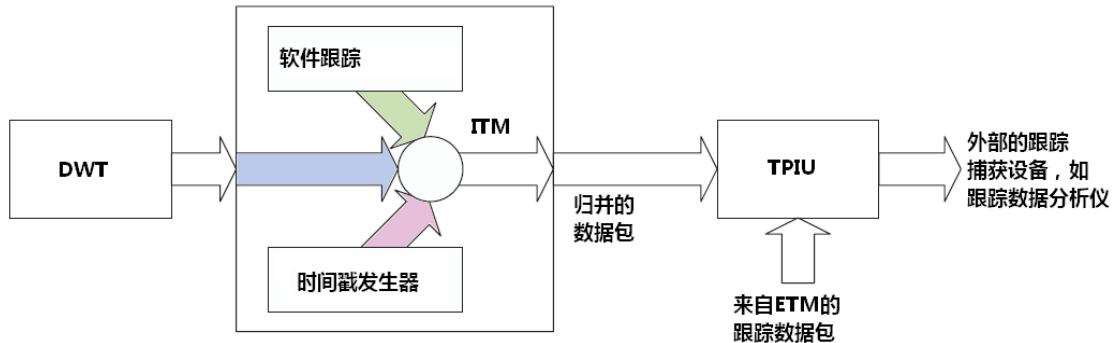


图 16.2 在 ITM 和 TPIU 上的数据包归并模式图

16.3.3 ITM 时间戳

ITM 还附带了一个时间戳的功能：当一个新的跟踪数据包进入了 ITM 的 FIFO 时，ITM 就会把一个差分的时间戳数据包插入到跟踪数据流中。跟踪捕获设备在得到了这些时间戳后，就可以找出各跟踪数据之间的时间相关信息。另外，在时间戳计数器溢出时也会发生时间戳数据包。

16.4 跟踪组件：嵌入式跟踪宏单元

ETM 功能块用于提供指令跟踪（即指令执行的历史记录），它是个选配件，不一定出现在所有的 CM3 产品上。当它使能后，并且在跟踪操作开始后，它会产生指令跟踪数据包。ETM 中也有一个 FIFO 缓冲区，为跟踪数据流的捕捉提供够用的时间。

为了减少产生的数据量，ETM 并不会一直忙不迭地输出处理器当前正在执行的地址。通常它只输出有关程序执行流的信息，并且只有在需要时才输出完整的地址（例如，当一个跳转发生时）。因为调试主机也有一份二进制映像的拷贝，它可以使用此拷贝来重建指令的执行序列。

ETM 也与其它的调试组件互相交互。例如，它与 DWT 的比较器就有关系：DWT 的比较器可用于产生 ETM 的触发信号，或者控制跟踪的启动与停止。

与传统 ARM 处理器的 ETM 不同的是，CM3 的 ETM 没有自己的地址比较器，而是由 DWT 的比较器代为完成。事实上，CM3 的 ETM 与传统 ARM 的 ETM 有很大的区别。

欲使用 ETM，必须执行下述的建立步骤（由调试器及其周边工具完成）

1. 把 DEMCR.TRCENA 位置位（DEMCR 寄存器的定义参见表 15.2 或 D.37）。
2. 解锁 ETM 以编程它的寄存器：往 ETMLOCK_ACCESS 寄存器中写 0xC5AC_CE55。
3. 编程 ATBID 寄存器 (ATID)，赋予 ETM 一个唯一的标识，以便把它的跟踪数据包与其它跟踪源的跟踪数据包分开。
4. ETM 的 NIDEN 输入信号必须为高电平。该信号的实现是取决于具体的器件的，还需要参考该器件的数据手册。

5. 编程 ETM 控制寄存器组以产生跟踪数据。

16.5 跟踪组件：跟踪端口接口单元 (TPIU)

如前所述, **ITM**, **DWT** 和 **ETM** 的跟踪数据都在 **TPIU** 处汇聚。**TPIU** 用于把这些跟踪数据格式化并输出到片外, 以供跟踪端口分析仪之类的设备接收使用。**CM3** 的 **TPIU** 支持两种输出模式:

- 带时钟模式(**Clocked mode**), 使用最多4位的并行数据输出端口
- 串行线观察器(**SWV**)模式, 使用单一位的**SWV**输出(不适用于早期版本的**CM3**)

在带时钟模式下, 数据输出端口实际使用的位数是可编程的。这取决于两点。其一, 是芯片的封装; 其二, 是在应用中, 提供了多少个信号引脚给跟踪输出使用。在具体的芯片中, 通过检查**TPIU** 的寄存器, 可以判断跟踪端口的最大尺寸。此外, 跟踪数据输出的速度也是可编程的。

在**SWV**模式下, 则使用**SWV**协议。它减少了所需的输出信号数, 但是跟踪输出的最大的带宽也减少了。

欲使用**TPIU**, 需要先把**DECMR.TRCENA**置位, 还要编程“协议选择寄存器”和“跟踪端口尺寸寄存器”, 这个工作由跟踪捕捉软件完成。

Cortex-M3 r2p0修订版

在 **SWV** 模式下, 会使用 **SWV** 协议。这时, 输出信号就只需要 1 个比特了, 但是跟踪输出的最高带宽也会下降。另外, 在使用串行线调试协议时, **SWV** 模式的输出可以和 **TDO** 共享信号线。这样一来, 哪怕使用只带有标准 **JTAG** 接口的入门级调试器, 也可以通过 **DWT** 和 **ITM** 来捕捉跟踪信息。

16.6 闪存地址重载及断点单元 (FPB)

FPB有两项功能:

- 硬件断点支持。产生一个断点事件, 从而使处理器进入调试模式(停机或调试监视器异常)
- 把代码地址空间中对指令或字面值(**literal data**)的加载, 重载到**SRAM**的地址空间中。

FPB有8个比较器, 分别是:

- 6个指令比较器
- 2个字面值比较器

什么是“字面值加载”？

当我们使用汇编写程序时，常常需要往寄存器中加载立即数据。当立即数的值很大时，加载操作就无法用单一指令完成，例如：

```
LDR R0, =0xE000E400
```

因为没有任何指令能接收32位立即数，我们需要把这个立即数预先安置到另一个存储器空间中，通常放到程序代码区的后面，然后就可以使用一条相对PC的加载指令，来读取这个立即数到对应的寄存器中。因此，上条代码的汇编结果可以如下所示：

```
LDR R0, [PC, #<immed_8>*4]
; immed_8 = (字面值地址 - PC)/4
...
; 文字池
...
DCD 0xE000E400
...
```

上面的LDR也可以是Thumb-2提供的32位版本：

```
LDR.W R0, [PC, #+/-<offset_12>]
; offset_12 = 字面值地址-PC
...
; 文字池
...
DCD 0xE000E400
...
```

在实际使用中我们经常需要在代码中使用多个字面值，汇编器或编译器就会在代码区中开出一块地址范围，来集合字面值，这个块就是所谓的“文字池”。在CM3中，从文字池的数据加载通常使用D-Code总线，但比较另类的实现也可以把文字池放到RAM区中，从而使用系统总线加载。

在FPB中有一个闪存地址重载控制寄存器，它包含了FPB的使能位。此外，每个比较器在它自己的控制寄存器中，都还有各自的使能位——前者是总开关。两种使能位必须都为1时才能启用比较器。

可以通过编程比较器，把指令空间的地址重载（重映射）到SRAM地址空间中。当使用此功能时，需要编程REMAP寄存器，以提供需要重映射内容的基址。REMAP寄存器的最高3位[31:29]被硬线连接成0b001，因此限定了重映射后的地址范围在0x2000_0000-0x3FFF_FF80之间，这段地址正好落在SRAM地址空间中。

当指令地址或字面值地址与比较器中的数值发生匹配命中时，读访问就会根据REMAP的设置被重映射。

使用这个重映射功能，可以创建一些“如果...将会...”(what if)形式的测试——通过把原始指令或字面值取代成另一个来实现。并且即使是在ROM或flash中运行的代码，也能够参与此种测试。另一种用法在本质上与这种用法相同，但被取代的是跳转指令，因此行为很像“狸猫换太子”：对于某个位于flash中的子程序，在SRAM中提供一个冒充它的。通过闪存地址重载，使得在执行到调用该子程序的指令(BL)时，实际上执行的是被“调包”过的，位于SRAM中的BL，后者则跳转到

“狸猫”中。这种机制使得基于ROM的设备也可以调试（修改过的子程序暂时放到SRAM中）。

下图演示了重映射的效果

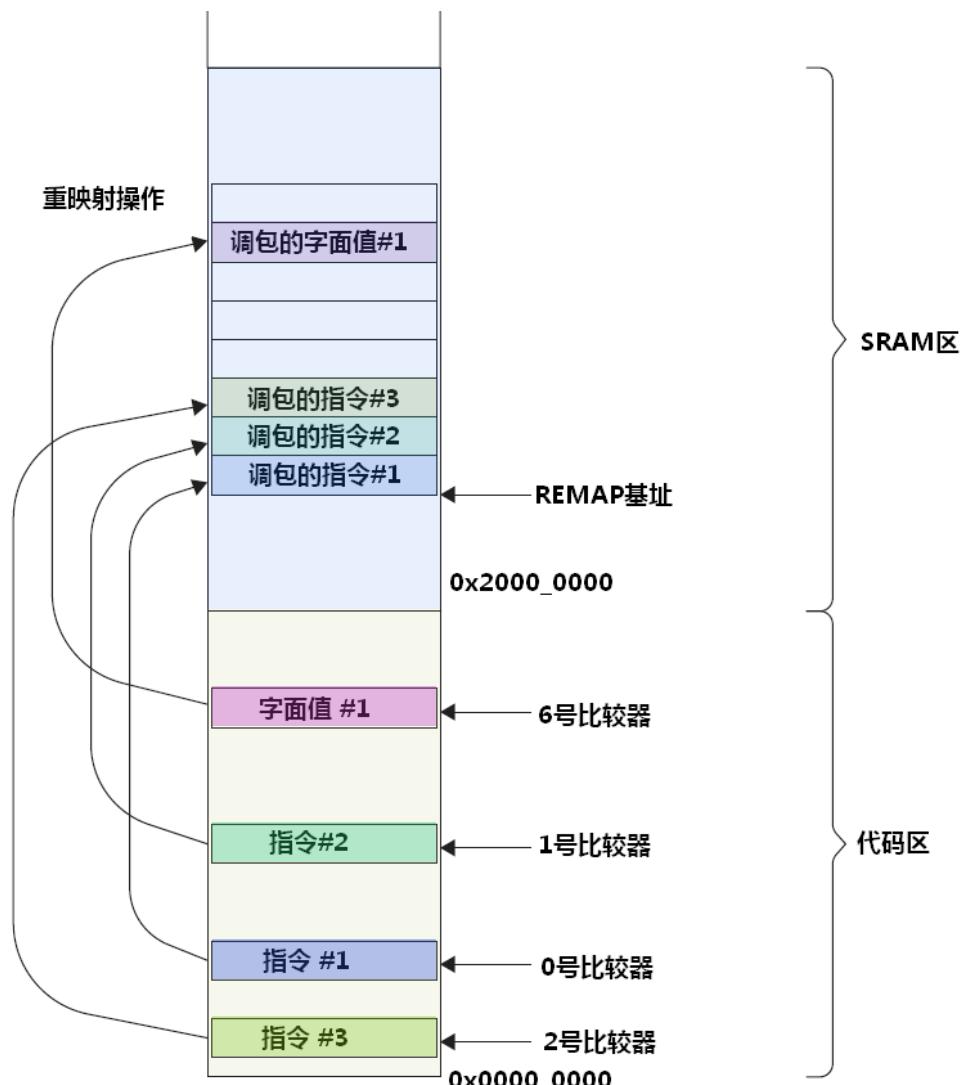


图16.3 闪存地址重载：对指令及字面值的重映射

除了地址重载，指令地址比较器的另一项功能，就是用于产生硬件断点（共6个），当地址匹配时使处理器进入调试模式。

16.7 AHB 访问端口

AHB-AP位于CM3的存储器系统和调试接口模块（SWJ-DP/SW_DP）之间，充当一个总线桥的角色。对于大多数基本的在调试主机和CM3系统之间的数据传输，只需要使用AHB-AP中的3个寄存器，它们是：

- 控制及状态字（CSW）
- 传输地址寄存器（TAR）
- 数据读/写（DRW）

AHB-AP的连接方法如图16.4所示：

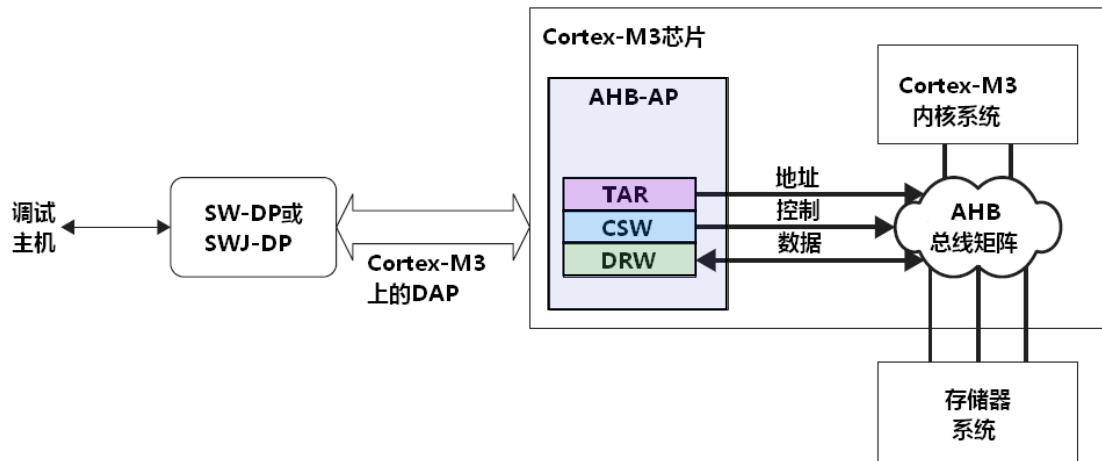


图16.4 在Cortex-M3中AHB-AP的连接

CSW寄存器可以控制传送方向(读/写)、传送大小以及传送类型等。TAR寄存器则指令传送地址,而DRW寄存器则容纳了被传送的数据(在访问该寄存器时就启动了传送)。DRW中的数据与总线上实际显示的是一致的,所以对于半字和字节传送,必须由调试硬件把得到的数据适当移位,以对齐到LSB。例如,若欲在地址0x1002上执行一次半字传送,则需要把数据放到DRW的[31:16]上。AHB-AP可以产生非对齐传送,但是它不会根据地址偏移来自动对目标数据做圆圈移位,必须由调试软件堵上这个窟窿:要么手工圆圈移位,要么把未对齐访问分解为若干个对齐的访问。

在AHB-AP中还有其它的寄存器,它们提供附加的功能。例如, AH B-AP中提供了4个banked寄存器和地址自动增量的功能,用于加快在小范围连续地址中数据访问的速度。

在CSW寄存器中,还有一个名为MasterType的位。通常需要把它置1,以此告知参与AH B-AP数据传送的硬件:该数据传送是调试器发起的。但是,调试器也可以清零此位来伪装成处理器内核。这样,在AH B上接收数据的硬件就会以为是内核发起的数据传送,从而正常地动作。这个功能可以用于测试目的,尤其是对于带有FIFO的外设,用于获知当它被调试器访问时,行为有什么不同。

16.8 ROM 表

CM3的调试系统还包含了ROM表,用于自动检测在某CM3芯片中包含了哪些调试组件。尽管作为v7-M的第一个践行者,CM3拥有一个预定义的存储器映射并且包含了标准的调试组件,但是新的Cortex-M器件可以包含不同的调试组件,并且芯片厂商在实现CM3时也可以对调试组件加以修改。为使调试工具能检测到调试系统中具体包含的组件,就提供了这张ROM表,它记录了NVIC和各个调试功能块的地址。

ROM表位于0xE00F_F000。通过分析ROM表中的内容,可以计算出系统和调试组件在存储器系统中的位置。在检测到了调试组件后,调试器可以接下来查看它们的ID寄存器,从而判定系统中哪些组件是可用的。

在CM3的ROM表中,第一条目的内容应当是: NVIC的入口地址相对于ROM表入口地址的偏移量。ROM表首条目的缺省值是0xFFFF0F003,其中位段[1:0]的作用比较特殊:它指示本条目对应的设备是存在的,并且在本条目的后面还有后续的条目(也就是说本条目不是最后一个条目)。这样,通过第一个条目,我们就知道系统中有NVIC,并且还有第2个条目,而且还能计算出NVIC的地址为0xE00F_F000+0xFFFF0_F000=0xE000_E000。

缺省的ROM表如图16.2所示。但是因为芯片厂商可以添加、移除以及把某些可选的组件替换成其它的CoreSight调试组件,这时该芯片的ROM表就会与缺省的有所不同,以反映出相应的变化。

表16.2 Cortex-M3缺省的ROM表

地址	数值	名称	功能
0xE00F_F000	0xFFFF0_F003	NVIC	指向NVIC的基址: 0xE000_E000
0xE00F_F004	0xFFFF0_2003	DWT	指向DWT的基址: 0xE000_1000
0xE00F_F008	0xFFFF0_3003	FPB	指向FPB的基址: 0xE000_2000
0xE00F_F00C	0xFFFF0_1003	ITM	指向ITM的基础: 0xE000_0000
0xE00F_F010	0xFFFF4_1003 / 0xFFFF4_1002	TPIU	指向TPIU的基址: 0xE004_0000
0xE00F_F014	0xFFFF4_2003 / 0xFFFF4_2002	ETM	指向ETM的基址: 0xE004_1000
0xE00F_F018	0	End	End-Of-Table标记
0xE00F_F0CC	1	MEMTYPE	表示在此存储器映射中, 可以访问系统存储器
0xE00F_F0D0	0	PID4	外设ID空间, 保留
0xE00F_F0D4	0	PID5	外设ID空间, 保留
0xE00F_F0D8	0	PID6	外设ID空间, 保留
0xE00F_F0DC	0	PID7	外设ID空间, 保留
0xE00F_F0E0	0	PID0	外设ID空间, 保留
0xE00F_F0E4	0	PID1	外设ID空间, 保留
0xE00F_F0E8	0	PID2	外设ID空间, 保留
0xE00F_F0EC	0	PID3	外设ID空间, 保留
0xE00F_F0F0	0	CID0	组件ID空间, 保留
0xE00F_F0F4	0	CID1	组件ID空间, 保留
0xE00F_F0F8	0	CID2	组件ID空间, 保留
0xE00F_F0FC	0	CID3	组件ID空间, 保留

数值的最低两个位用于指示该设备是否存在(bit[1])以及后面还有没有其它的表项(bit[0])。在正常情况下, NVIC, DWT和FPB总是必须存在的, 因此最后两位永远是1。然而, TPIU和ETM则可以被裁掉, 并且可能被CoreSight家庭中其它的调试组件所取代。

数值的高位部分用给出对应组件的入口地址相对于ROM表入口地址的偏移量。例如,

NVIC入口地址= 0xE00F_F000 + 0xFFFF0_F000 = 0xE000_E000 (进位位被忽略)

在开发调试工具时, 有必要从ROM表中一一查兑各调试组件, 因为难免会有些另类的CM3芯片会自定义调试组件, 并且修改ROM表, 而通过计算ROM表得到的地址是可以拿去拍板的。

开始 Cortex-M3 开发

- 选择一款 Cortex-M3 产品
- Cortex-M3 修订版 0 与修订版 1 的区别
- Cortex-M3 修订版 1 与修订版 2 的区别
- 开发工具

17.1 选择一款 Cortex-M3 产品

在根据自己的应用选择具体的 CM3 芯片时，除了要考虑存储器、外设配置以及最高主频之外，其它一些因素也会使一款 CM3 芯片与众不同，CM3 的设计允许下列参数是可以配置的，它们是：

- 外中断的数目
- 表达优先级的位数（优先级寄存器的有效宽度）
- 是否配备了 MPU
- 是否配备了 ETM
- 对调试接口的选择（SW, JTAG 或两者兼有）

对于大多数项目而言，单片机的功能和规格我们在选择时的首要考虑因素，例如：

1. 外设。对于大多数的项目，片载的外设是最重要的选择依据。外设也并非多多益善，因为它会影响到功耗和价格。
2. 存储器：CM3 单片机的闪存可以少到几 KB，多至几 MB。此外，片内 RAM 的容量也是很重要的。这些参数往往对价格有重大的冲击。
3. 时钟速度：CM3 的设计可以在 0.18um 的粗线条工艺上，也轻松上到 100MHz。然而，因为存储器访问速度的限制，芯片厂商会降低最大主频。
4. 脚印：CM3 单片机的封装也多种多样。很多 CM3 单片机的脚数都比较少，以使之更适合于低成本的应用中。

17.2 Cortex-M3 修订版 0 与修订版 1 的区别

早期的 Cortex-M3 产品是基于 Cortex-M3 处理器修订版 0 的。在 2006 年第 3 季度之后的 CM3 产品可以使用修订版 1。在本书出版之时，所有的新 CM3 器件应该都是基于修订版 1 的。了解自己使用的芯片基于哪个修订版是很重要的，因为在修订版 1 中作出了许多重要的改变和改进。在本书前面章节中，都是按新的修订版 1 来叙述的。

在翻译本书时，有两个 CM3 芯片生产商，分别是 Luminary 和 ST。译者查看了它们的资料，判定它们都是使用修订版 1 的处理器的。后续会有更多的 CM3 芯片生产商，但它们肯定不会使用老的修订版 0 了。

在编程模型中可以看见的改变包括如下内容：

- 从修订版 1 开始，响应异常时的寄存器操作可以被配置成强制对齐到双字边界，这可以通过置位 NVIC_CCR.STKALIGN 来启用。
- 因为刚才的理由，NVIC_CCR 中加入了 STKALIGN 位
- 修订版 1 的修订版中引入了新的 AUXFAULT（辅助 fault）状态寄存器（可选）

- DWT 中添加了诸如数值匹配的新功能
- ID 寄存器的值因修订版号位段而改变

在编程模式中看不见的改变更多，它们是：

代码存储空间的存储器属性被硬线连接到可缓存，已分配(**allocated**)，不可缓冲，不可共享。这会影响 **I-Code AHB** 和 **D-Code AHB**，但是不会影响系统总线接口。

支持在 **I-Code AHB** 和 **D-Code AHB** 间的总线复用操作。在此操作模式下，可以使用一个简单的总线复用器来把 **I-Code** 和 **D-Code** 归并(**merge**)，这可以降低总门数，旧修订版的则必须使用 **ADK** 总线矩阵组件。

新添加了用于连接 **AHB** 跟踪单元(**HTM**)的输出端口。**AHB** 是一个 **CoreSight** 中定义的调试组件，服务于复杂的数据跟踪操作。

调试组件或调试寄存器可以在系统复位期间访问，只有在上电复位时才无法访问。

在修订版 1 中，**NVIC_ICSR.VECTPENDING** 位段可以受 **NVIC_DHCSR.C_MASKINTS** 位的影响：当 **C_MASKINTS** 置位时，如果掩蔽了一个悬起的中断，会使 **VECTPENDING** 的值为零。

JTAG-DP 调试接口被 **SWJ-DP** 模块取代。但是仍然允许芯片厂商使用 **JTAG-DP**，因为它也是 **CoreSight** 家庭中的成员。

因为修订版 0 的 **CM3** 在响应异常时没有双字对齐堆栈的功能，有些编译器，如 **ARM** 的 **RVDS** 和 **Keil** 的 **RVMDK**，都提供了特殊的编译选项以决定是否允许软件调整入栈，以使开发出来的产品是 **EABI** 兼容的，当软件需要与其它 **EABI**-兼容开发工具时，这还是相当重要的。

为了判定使用的单片机使用了哪个修订版的 **CM3** 内核，可以使用 **NVIC** 中的 **CPUID** 寄存器，**revision** 和变种位段指出了具体使用的 **CM3** 修订版。如表 17.1 所示：

表 17.1 CPUID 基寄存器

表 17.1 CPUID 基寄存器 (地址：0xE000_ED00)

	实现者 [31:24]	变种 [23:20]	常数 [19:16]	PartNo [15:4]	Revision [3:0]
修订版 0(r0p0)	0x41	0x0	0xF	0xC23	0
修订版 1(r1p0)	0x41	0x0	0xF	0xC23	1
修订版 1 (r1p1)	0x41	0x1	0xF	0xC23	1
修订版 2(r2p0)	0x41	0x2	0xF	0xC32	0

译者查看了 ST 的 STM32 系列使用的内核，得到的结果是 r1p1。

17.2.1 修订版 1：从 JTAG-DP 到 SWJ-DP

串行线 **JTAG** 调试端口 (**SWJ-DP**) 把 **SW-DP** 和 **JTAG-DP** 的功能合二为一，并且支持自动协议检测。使用这个组件，**CM3** 设备可以支持 **both SW** 和 **JTAG** 接口。(目前可以使用的，由 **LM** 和 **ST** 所提供的芯片都是使用了 **SWJ-DP**——译者注)。

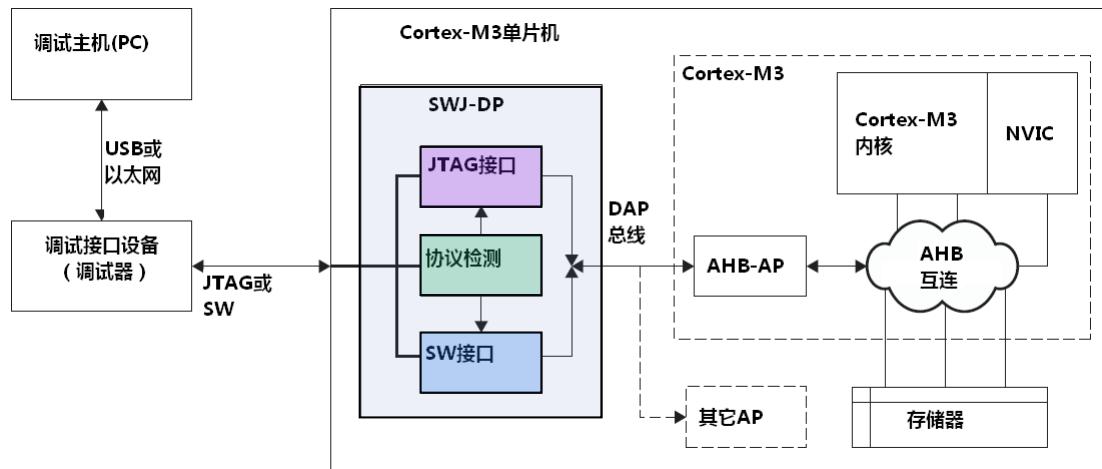


图 17.1 SWJ-DP：合并了 JTAG-DP 和 SW-DP 的功能

17.3 Cortex-M3 修订版 1 与修订版 2 的区别

在 2008 年中期, Cortex-M3 的修订版 2 发布了。估计到 2008 年底, 在市场上就能见到基于修订版 2 的芯片了。修订版 2 新增了很多特性, 它们大多数都致力于降低功耗以及提高调试的灵活性。

在修订版 2 中, 程序员模式也跟着有以下的更新。

17.3.1 双字堆栈对齐方式成为缺省值

影响异常入栈顺序和内存使用的双字堆栈对齐方式, 在修订版 2 中成为缺省使用的方式(注意: 芯片厂商可能会选择使用修订版 1 的方式)。使用此方式, 会给大多数 C 程序减少启动代码的额外开销(无需再在 NVIC 配置控制寄存器中置位 STKALIGN 比特)。

17.3.2 新增辅助控制寄存器 (Auxiliary Control Register)

为了更细腻地调校处理器的行为方式, 新增了辅助控制寄存器。比如, 为了调试方便, 通过设置此寄存器, 可以关闭 Cortex-M3 的写缓冲, 从而使总线 **faults** 总是能与存储器访问指令同步——也就是说使总线 **faults** 总是精确的。这样, 就可以每次都能从入栈的返回地址中精确地揪出肇事指令了。

辅助控制寄存器的细节如下表所示:

辅助控制寄存器 (0xE000_E008)

比特号	字段名	类型	初值	功能描述
2	DISFOLD	R/W	0	除能 IT 折叠(folding), 使 IT 指令与下一条指令(在流水线中)的执行级(execution phase)不会交迭
1	DISDEFWBUF	R/W	0	在缺省的存储器映射中除能写缓冲(对由 MPU 映射的 regions 不起作用)

0	DISMCYCINT	R/W	0	除能“指令可中断”功能。也就是不再打断 LDM, STM, 64 位乘法, 以及除法指令。
---	------------	-----	---	---

17.3.3 ID 寄存器的更新

在 NVIC 以及调试组件中的很多 ID 寄存器都更新了。例如，在 NVIC 中的 CPUID 寄存器变成了 CPUID 寄存器 (0xE000_ED00)

	实现者 [31:24]	变种 [23:20]	常数 [19:16]	PartNo [15:4]	Revision [3:0]
修订版 2 (r2p0)	0x41	0x2	0xF	0xC23	0x0

17.3.4 调试功能

修订版 2 对调试功能有了好几处改进

- DWT 中的观察点数据跟踪现在支持两种新的跟踪方式：仅跟踪读传送，以及仅跟踪写传送。这样就可以仅在数据被改变或被读时才产生跟踪数据流，于是降低了数据跟踪所需的带宽。
- 在实现调试特性时提供了更高的灵活性。比如，允许裁减可用的断点和观察点数，这样就降低了所设计产品的尺寸，这对于超低功耗的设计非常有帮助。
- 对多核系统的调试，支持力度更大。为了实现多核同时重启和单步，新增了一个接口（注意，在程序员眼中看不到这种改变）。

17.3.5 睡眠特性

在系统级设计层上，现有的睡眠特性也得到了改进。在 r2p0 中，对处理器的唤醒可以延迟，从而使得在芯片中可以更大面积地“停电”，并且在系统中所有其它部件都就绪后才继续执行程序。这个改进主要是为了照顾下面一些应用：在它们里面，有一些硬件在低功耗模式下需要关闭，但是重新打开这些硬件需要的时间比较久。

在睡眠功能的扩展之外，为降低功耗还有新招。在旧版的 CM3 中，为了让内核能醒来，在睡眠期间，依然不能停止送往内核的“自由运行时钟”。尽管该时钟消耗的能量很低，但总归还是关了更省电。

为解决这个问题，可以在处理器外面布设一个简单的中断控制器。这个控制器，取名为“唤醒中断控制器 (WIC)”。在深度睡眠期间，它要提供在 NVIC 中的，“中断掩蔽功能”的镜像，并且负责告知电源管理系统何时需要唤醒。这样，就可以在深度睡眠期间关断所有送往 CM3 处理器的时钟了。

除了可以停止时钟外，修订版 2 还可以使处理器的大多数部分都掉电，把它们的状态存储在若干特殊的逻辑小室中。在中断到达时，WIC 往电源管理单元 (PMU) 发送一个唤醒请求。在处理器重新上电后，先前的状态从特殊的逻辑小室中恢复，然后就可以响应这个中断了。

可见，有了修订版 2 中这个新的掉电能力，CM3 可以在深度睡眠期间进一步降低功耗。不过，这个特性还需要内核外的单元配合，因此不一定在所有修订版 2 的产品中都支持。

17.3.6 使用修订版 2 带来的好处和注意事项

那么，上文所讲的这些新特性，又对嵌入式产品开发带来了什么呢？

首先，是更低的功耗和更久的电池寿命。在进入了有 **WIC** 支持的深度睡眠后，整个电路就只有很小一部分还在活动中。此外，在要求极低功耗的应用中（如体内植入式医疗设备），芯片厂商可以通过减少断点和观察点的数量，来裁减芯片的尺寸。

第二，在调试和解决疑难问题的过程中，它提供了更好的灵活性。不仅体现在使用调试器的数据跟踪特性上，还新增了一个辅助控制寄存器。通过它我们可以给写缓冲做个旁路手术，从而使总线 **faults** 总是精确的。我们还可以使需要较多周期才能执行的指令不被打断，如 **LDM/STM** 指令。这样一来，在分析存储器的内容时就可以放心了。最后，对于使用多个 **Cortex-M3** 内核的系统，修订版 2 带来的“同时重启”和“多核单步执行”的功能正好雪中送炭。除此之外，在修订版 2 中还有若干个内部优化，以提高性能和改善接口特性。这样，芯片供应商就可以设计出更快的 **CM3** 产品。

然而，在享受温柔的同时，也请嵌入式程序员们留意下面的问题：

双字堆栈对齐方式与异常堆栈帧

在缺省情况下，异常堆栈帧会自动对齐到双字存储器位置。早期为修订版 0/1 写的汇编程序，如果要通过堆栈来把数据传送给异常服务程序，可能会受到影响。为了准确判定堆栈帧的起始位置是否往下挪移了一个字，异常服务例程要先读取入栈 **PSR** 的比特 9。如果不想改动旧的程序，也可以手工把 **STKALIGN** 比特清除，这样就与以前的一样了。与 **EABI** 标准兼容的应用程序不会受影响。这些程序通常是 **C** 程序，并且使用与 **EABI** 兼容的编译器编译。

SysTick 定时器也许会在深度睡眠期间停止。

如果使用的 **CM3** 单片机确实包含了掉电功能，或者是其它原因使得送往内核的时钟全体都在深度睡眠中停止，则 **SysTick** 定时器在深度睡眠期间就无法再运行。这样一来，使用了 **RTOS** 的嵌入式应用程序就需要一个外部时钟，用它来在唤醒时提供调度所需的滴答信号。

当处理器连接到一个调试器时，会自动除能新的掉电功能。

这是因为调试器需要访问处理器的调试相关寄存器。在调试会话中，能够控制内核停机或进入睡眠模式，但哪怕使能了掉电功能，也不会触发掉电序列。为了准确地测试掉电操作时的功耗，必须解除被测设备与调试器的连接。

17.4 开发工具

在开始使用 **Cortex-M3** 之前，需要准备好一些开发工具，典型的如：

- 编译器/汇编器：把 **C** 和汇编源程序转换成目标文件。几乎所有的 **C** 编译器套件都包含了对应的汇编器。
- 指令系统模拟器：模拟指令的执行，用于在软件开发早期的调试。
- 在线仿真器（**ICE**）或者调试探测器（**probe**）：连接到电脑和目标板上的调试硬件，与目标板的接口通常是 **JTAG** 或 **SW**。
- 一块开发板。
- 跟踪捕捉仪：可选的硬件设备和周边软件，可以用它来捕捉来自 **DWT** 以及 **ITM** 的输出，并且以可读的形式显示出来。
- 嵌入式操作系统：在单片机上运行的操作系统。这也是一个可选件，许多简单的应用程序不需要操作系统。但是在开发复杂度较高或者有高性能指标的系统时，常常需要使用。

17.4.1 C 编译器

截止到目前，已经有若干个 C 编译器套件可以使用了，如表 17.3 所列。

表 17.3 支持 Cortex-M3 的开发工具

公司	产品
ARM www.arm.com	Cortex-M3 在 RealView 开发套件 3.0(RVDS)中得到支持。在 RealView-ICE 1.5 可以用于连接调试硬件和调试环境。更早的 ADS1.2 和 SDT 不支持 Cortex-M3
KEIL(an ARM company) www.keil.com	大名鼎鼎的 KEIL，一度在 8051 的开发中享有盛誉。在其最新的 Realview MDK 开发工具中，支持了 Cortex-M3，其配套的仿真器是 ULINK 和 ULINK2。
CodeSourcery www.codesourcery.com	支持 Cortex-M3 的 GNU 工具链现在已经可用，下载地址是 www.codesourcery.com/gnu_toolchains/arm 。它基于 GNU 4.0 版本
Rowley Associates www.rowley.co.uk	这个工具也源自 GNU C 编译器 www.rowley.co.uk/arm/index.htm
IAR Systems www.iar.com	IAR Embedded Workbench for ARM and Cortex，它提供了 C/C++ 编译器和调试环境（从 4.40 版本开始）。IAR 在早先 AVR 单片机的开发中就是出类拔萃的。与 IAR 配套的仿真器是 JLINK
Lauterbach www.lauterbach.com	提供了 JTAG 访真器和跟踪设备

17.4.2 嵌入式操作系统支持

上档次应用程序常常需要 OS，尤其是 RTOS。许多 OS 已经被开发出来用于嵌入式产品，目前，支持 Cortex-M3 的 OS 如表 17.4 所列：

表 17.4 支持 Cortex-M3 的嵌入式操作系统

公司	产品
FreeRTOS	FreeRTOS
www.freertos.org	
Express Logic	ThreadX™ RTOS
www.expresslogic.com	
Micrium	uC/OS-II
www.micrium.com	
Accelerated Technology	Nucleus
www.acceleratedtechnology.com	
Pumpkin Inc.	Salvo RTOS
www.pumpkininc.com	
CMX Systems	CMX-RTX
www.cmx.com	
KEIL	ARTX-ARM
www.keil.com	
Segger	embOS
www.segger.com	
IAR Systems	IAR PowerPac for ARM
www.iar.com	
T-Engine 论坛	uT-Kernel
www.t-engine.org	

ARM7 应用程序移植到 Cortex-M3

- 简介
- 系统个性
- 汇编源程序
- C 源程序
- 预编译的目标文件
- 优化

18.1 简介

如果非要找出 CM3 的降临可以带来的痛苦，也许就是把运行在 ARM7TDMI 上的代码升级过来所要做的工作了，这种成长的阵痛也是在所难免的。为了降低升级难度专门开出本章，把升级过程中的重点明确地总结一下。

在计划把代码从 ARM7 移植到 CM3 时，需要考虑以下的方面：

- 系统性质
- 汇编源程序
- C 源程序
- 优化

总体来说，越是底层的代码，受到的冲击越大。像最底层的硬件控制、任务管理以及异常服务例程，它们与架构的关系最密切。另一方面，因为底层的代码往往大面积地使用汇编，因此面临改写甚至重写的工作量最大。普通的应用程序需要的改动则比较小，而且这时优良的编程习惯经常会大幅度，甚至戏剧般地降低修改工作量（最简单的就是多使用宏定义）。对于与架构无关的纯算法类应用程序，则都无需改动，只要简单地重新编译即可。

18.2 系统的个性

想必大家也已经总结出来了，CM3 与 ARM7 相比，还是有很多新的个性的。像固定的存储器映射，中断处理机制，操作模式，系统控制，以及新引入了 MPU 等。下面我们就一小结。

18.2.1 存储器映射

在不同处理器架构间的差异中，存储器映射算得上是最“外向”型的了。在 ARM7 中，是由器件厂商自由划分 4GB 的寻址空间的，再加上厂商还可能玩各种“二次映射”技术，各 ARM 芯片之间的存储器映射可以是大相径庭的。到了 CM3 中，把存储器映射被粗线条地标准化了——把 4GB 空间分成了若干个不同类型的区域，对应的存储器必须对号入座。一般地，通过设置编译和连接选项，可以轻易地适应新的 ROM 和 RAM 的映射图。但对于设备驱动程序，则情况比较复杂。如果是不同厂家的芯片，外设寄存器的用法基本上是完全不同的，此时驱动程序必须重写；如果是在同一厂家的 ARM7 和 CM3 芯片间移植，则外设寄存器有望相对一致，驱动程序只需部分改动，甚至简单到只修

改基地址即可。

许多 ARM7 芯片会提供存储器的“二次映射”功能，其中一个重要的用途，就是使向量表可以被重映射到 SRAM 中。而在 CM3 中，可以通过编程 NVIC 的寄存器来实现此功能，因此不再需要这些二次映射功能，从而许多芯片可能也去掉了完备的二次映射支持（但是可能会提供一种“硬件控制”的二次映射——上电时，由某些管脚的电平决定把哪里的存储器映射到零地址上，以支持多种引导方式。如 STM32 就采用了此法，以支持从 Flash/SRAM/原配 BootLoader 引导——译者注）。

CM3 对大端模式的支持方式也与 ARM7 的不一样。程序代码只需重新编译，但是事先做好的查找表则需要重新编码。（大端编码是多事之地，建议读者少碰它——译者注）。

从 ARM720T，以及 ARM9 等那个年代开始的处理器，为了支持像 WinCE 这样的操作系统，引入了所谓的“高端向量”功能——允许把向量表重定位到 0xFFFF_0000。CM3 并没有打算支持 WinCE（实际上最重要的原因是没有配 MMU），因此去掉了“高端向量”的支持。

18.2.2 中断/异常系统

可能 NVIC 都快引起大家的审美疲劳了。没错，在 CM3 中的中断处理已经被彻底改造，因此所有与控制中断有关的代码都需要大面积更新。而且还需要为建立中断优先级和向量表添加全新的代码。

中断返回机制也变了。这影响到了汇编代码。而且如果编译器使用指示字(*directive*)来支持 C 程序中断服务程序的话，还需要调整指示字。

过去，对中断的使能和除能是通过修改 CPSR 的，在 CM3 中没有 CPSR，而是使用 PRIMASK 或 FAULTMASK 来实现全局中断的开关。

CM3 在响应中断时，启用了一个自动栈操作的机制，因此可以把旧时的入栈和出栈指令化简。然而，旧时的 ARM 还有所谓的 FIQ，并且为 FIQ 服务例程专开了小灶——独立的 4 个寄存器(R8-R11)，专为 FIQ 服务例程使用，无需 push/pop。FIQ 其实极少利用，成了“彩色糖衣包装却没营养的药药”。在 CM3 中并没有 FIQ 的概念，因此在移植以前的 FIQ 服务例程时，在代码上必须把它当作普通的中断服务例程处理——其实因为 CM3 有自动堆栈操作，普通中断也相当于享有 FIQ 的小灶待遇。另一方面，通过提升其优先级到最高，可以使它在时间上得到 FIQ 的待遇。

实现嵌套中断的代码现在可以去掉了，因为 CM3 的 NVIC 已经内部实现了中断嵌套。

错误处理机制也大有不同。旧时的 ARM 只有 DAbort, IAbort, Undef 这 3 种异常模式对应错误处理，而到了 CM3 中，提供了很多 fault 状态寄存器来确定各种 faults，而且还定义了许多新的 fault 类型，其中最有新意的就是堆栈操作 faults、存储器管理 faults 以及硬 fault 了。因此，fault 服务例程需要重新设计。

18.2.3 MPU

MPU 是 CM3 中的新鲜血液，因此需要新的程序代码来使用它。另一方面，因为在 ARM7TDMI 中没有 MPU，因此这方面没有“代码移植”的概念。不过，在 ARM720T 上是配有 MMU 的，它的功能与 CM3 的 MPU 不一样——事实上，如果代码需要 MMU 来支持虚拟内存，根本就不能使用 CM3。

18.2.4 系统控制

系统控制也是移植程序时必须充分重视的关键内容。CM3 内建了进入睡眠模式的指令。另一方面，在 CM3 芯片中的系统控制器也有特殊的设计要求，基本上它们不会与 ARM7 芯片中的有什么相似之处。因此，要做好思想准备，来重写系统控制相关的代码。

18.2.5 操作模式

以前的 ARM 架构有 7 种操作模式，在 CM3 中，它们可以用对应的异常来取代，如表 18.1 所示：

表 8.1 把 ARM7TDMI 中的操作模式和异常映射到 CM3

在 ARM7 中的操作模式和异常	在 CM3 中与之等价的操作模式和异常
监察者 (supervisor) (复位后自动进入)	特权级的线程模式+MSP
监察者 (因 SWI 而进入)	SVC 异常
FIQ	优先级最高的外部中断
IRQ	外部中断
指令流产 (IAbt)	总线 fault
数据流产 (DAbt)	总线 fault
未定义指令	用法 fault
系统模式	特权级的线程模式+PSP
用户	用户级的线程模式+PSP

虽然在 CM3 中，可以把 ARM7 的 FIQ 对应到优先级最高的外中断，从而实现 FIQ 的时间地位。但是 ARM7 的“专用寄存器”是 R8-R11，而 CM3 自动入栈的寄存器是 R0-R3, R12。因此，旧时 FIQ 服务例程需要改用 R0-R3, R12；如果依然要使用 R8-R11，就必须先把它们手工入栈。

NMI vs. FIQ

可能有不少人曾想到过用 NMI 来取代 FIQ。的确，在一些场合中，这是可行的。但是，NMI 与 FIQ 有本质的区别，使得很多情况下它们不能互换，这也是我们必须清醒地认识到的。

第一，NMI 正如其名，是不能被除能的。而 ARM7 的 FIQ 则可以通过把 CPSR.F 置位来除能。因此，在 CM3 中何时进入 NMI 完全不可控——有可能在引导期间就进入 NMI。而在 ARM7 中，复位后 FIQ 是除能的，因此不会意外地进入。

第二，CM3 的 NMI 服务例程不得使用 SVC，而 ARM7 的 FIQ 服务例程则可以使用 SWI。另外，在 ARM7 下，即使是在 FIQ 服务例程的执行过程中，也可以转而响应其它异常 (IRQ 除外)。而在 CM3 下，如果 NMI 服务例程执行过程中发生 fault，则处理器当即被锁定。

18.3 汇编源程序

对汇编源程序的移植取决于使用的是 ARM 状态还是 Thumb 状态。

18.3.1 Thumb 状态

如果使用的是 thumb 汇编源文件，则是幸运的，在大多数情况下代码可以直接拿来用。只有个别的 thumb 指令在 CM3 中不可用：

- 任何试图转入 ARM 状态的指令（典型就是 BLX）
- 不再支持 SWI，而是要使用 SVC，而且用法上也有区别

最后，一定要只使用向下生长的满栈，CM3 的 push 和 pop 就是使用这种模型的——总有程序

喜欢玩另类，结果不但移植工作量加重了，也无法使用 C 语言了。因此可别在这里秀“叛逆精神”啊，否则会自找苦吃的，而且向下生长的满栈本来就是更合理的。

18.3.2 ARM 状态

如果不幸在汇编源文件中使用了 ARM 状态，也不要慌，仔细核对下列情况：

- 向量表：在 ARM7 中，向量表从 0 地址开始，并且由一系列的跳转指令组成。在 CM3 中，跳转表给出了 MSP 的初值以及复位向量地址，接下来的是各异常服务例程的入口地址。因此这些区别是本质上的不同，向量表必须重写。
- 寄存器初始化：在 ARM7 中，经常需要把每个模式下的寄存器分别初始化。比如，每个模式（除系统模式外）都有自己的 SP、LR 和 SPSR。CM3 去掉了这些繁文缛节，而且也不再需要把处理器的模式换来换去。
- 模式切换与状态切换：在 CM3 不再保留 ARM7 中的那些操作模式，也没有 Thumb 状态，因此相关的代码都可以移除。
- 中断的使能与除能：在 ARM7 中，中断的使能与除能是通过对 CPSR.I 来控制的。在 CM3 中则改用 PRIMASK 或 FAULTMASK。更进一步地，CM3 中没有 FIQ 的概念，因此也没有 F 位。
- 协处理器访问：CM3 不支持协处理器，因此相关的代码无法移植。但是可以通过软件模拟的办法来缓解
- 中断服务例程和中断返回：在 ARM7 中，中断服务例程的首条指令在向量表中。这条指令，除了 FIQ 服务例程的外，都必须是一种无条件跳转指令，而 CM3 中则是直接在向量表中给出 ISR 的入口地址。中断返回时，ARM7 是通过带 S 后缀的指令手工地调整 PC 的值来实现；而 CM3 则把需要返回的地址压入堆栈中，并且通过把某个 EXC_RETURN 写入 PC 来触发中断返回序列。因此，在 CM3 中，不得使用诸如 MOVS 或 SUBS 之类的指令来启动中断返回。由于这些原因，中断服务例程和中断返回的代码需要加以改动。
- 当需要启用中断嵌套时，ARM7 的作法通常是先进入系统模式再重新使能 IRQ，在 CM3 中则没有这些操作。
- FIQ 服务例程：因为在 ARM7 中，FIQ 有专用的 R8-R12；而 CM3 则自动保存了 R0-R3, R12。所以如果必须要移植 FIQ 服务例程，则需要手工保存 R8-R11。或者把本来对 R8-R11 的使用，改为以 R0-R3 的使用。
- 软件中断(SWI)服务例程：SWI 由 SVC 取代。不过，定位软件中断指令并萃取系统调用号的作法不同。在 CM3 中，通过压入栈的返回地址来计算出 SVC 指令的地址；而在 ARM7 中，则是通过 LR 来计算。
- 交换指令(SWP)：在 CM3 中没有交换指令。如果以前使用 SWP 来实现信号量，则要改为使用互斥访问来实现，因此需要改动信号量相关的代码。如果以前使用 SWP 只是为了纯粹地传送数据，则需要使用若干存储器访问指令来实现。
- 对 CPSR 和 SPSR 的访问：ARM7 中的 CPSR 在 CM3 变成了 xPSR，而 SPSR 则被去掉了。对于访问标志的应用程序代码，可以改为对 APSR 的访问。如果异常服务例程想要访问异常发生之前的 xPSR，则要读取压入堆栈中的值——这取代了 ARM7 中 SPSR 的功能，因此 CM3 中不再需要 SPSR。
- 条件执行：在 ARM7 中，大量指令都可以条件执行；而 Thumb-2 的指令则几乎都不能条件执行。在移植这些代码到 CM3 中时，对于短小的条件执行段，可以用 IF-THEN 指令封装；而比较大的则需要使用跳转指令来改建。当使用 IT 指令时要注意一些小问题。主要就是会增加代码量，有可能使得某些加载/存储指令超出最大可操作的地址范围。
- 使用 PC 计算当前代码的地址：在 ARM7 中，读取的 PC 值“读 PC 指令的地址+8”。这是由

ARM7 的三级流水线造成的一——当读取 PC 的指令处于执行阶段时, PC 已经自增了两次。同样的事情也发生在 CM3 中, 但是在代码移植到 CM3 后, 因为这些代码将在 Thumb 下执行, 所以 PC 被加的值变成 4。

- 对 R13 的使用: R13 总是 32 位的。但是在 CM3 中, 末 2 位被强制为 0。因此, 如果偶尔遇到使用 R13 作为基址的场合 (强烈反对使用), 必须更改代码, 因为末 2 位的信息已经丢失了。

18.4 C 源程序

好歹也是高级语言, 移植 C 源程序要比移植汇编的轻松很多。在许多情况下, 只消重新编译即可。但是对于使用了非主流技巧的 C 程序 (常见于系统程序中), 则可能要考虑如下的方面。

- 内联汇编: 如果使用 RVDS, 则不支持内联汇编, 因此使用了内联汇编的 C 程序需要做出修改。对于 RVDS 3.0 及更高版本, 可以使用嵌入式汇编来替代内联汇编。
- 中断服务例程: 对于使用 “`__irq`” 来创建的 ARM7 中断服务例程, 因为 CM3 使用了新的中断模型, 往往可以去掉 “`__irq`” 指示字 (不过, 如果使用 RVDS 3.0 和 RVCT 3.0, 则 `__irq` 也支持 CM3, 此时可以保留 “`__irq`”, 以强调程序的类型, 提高了可读性)。

18.5 预编译的目标文件

许多编译器都为函数库和启动代码预先编译出了目标文件。但是因为操作模式和状态模型的不同, 它们往往不能用在 CM3 上——尤其是启动代码。此时, 就必须得到它们的源代码, 并且移植到 CM3 上, 请参阅你所使用工具链的联机帮助来获取详细信息 (事实上, 推荐使用的开发工具 (KeilMDK/GCC) 都已经做好了这些事情——译者注)。

18.6 优化

CM3 中有许多新特性, 加以利用的话常常可以大大提高程序的性能, 或者降低对存储器的使用。对于积极向上的我们, 一定要挖掘这些特性:

- **使用 32 位 Thumb-2 指令:** 对于下列的场合: 先使用一条 16 位 thumb 指令把数据从一个寄存器传送到另一个, 再对该数据执行数据处理。有时能使用一条 Thumb-2 指令来完成 (这主要是因为 16 位 Thumb 指令不能使用 “高寄存器” ——译者注), 从而使所需的处理时间缩短。
- **位带操作:** 如果外设寄存器位于位带区, 则可以通过对位带别名区的访问, 大大地化简对寄存器位的操作。
- **乘法与除法:** CM3 的一个重大革新就是支持除法指令和部分支持 64 位乘法指令。请善用它们 (尤其是除法), 可以成十上百倍地提高程序的执行速度。
- **立即数:** 有些 Thumb-2 指令支持长达 12 位的立即数, 因此可以把以前 Thumb 指令无法加载的立即数使用一条 Thumb-2 来加载。
- **跳转:** 过去单条 Thumb 指令无法执行的远程跳转, 现在可以使用 Thumb-2 指令实现了。

- **布尔数据**: 对于“BOOL”型的变量, 可以强制把它们定址到内存的位带别名区。相比于过去使用字来实现 BOOL 变量, 现在只需使用以前 1/32 的内存空间。
- **IT 指令块**: 有些短距跳转可以使用 IT 指令取代, 这样做消灭了因流水线清洗而引入的等待周期, 从而提高了性能。
- **ARM/Thumb 状态切换**: 在大多情况下, 可以把大部分代码以 Thumb 指令编码, 一小部分以 ARM 指令编码。这主要是为了在平时提高代码密度, 而在紧急关头下提高性能。在 CM3 下有了 Thumb-2 代码, 可以在同一模式下解决时间与空间的权衡。这就可以去掉这些状态转换及其所带来的额外负担 (overhead), 也简化了对工程的管理。

使用 GNU 工具链开始 Cortex-M3 开发

- 背景
- 获取 GNU 工具链
- 开发流程
- 示例程序
- 访问特殊功能寄存器
- 使用未支持的指令
- GNU C 编译器的内联汇编

19.1 背景

GNU 工具链在 ARM 产品开发中使用得很广泛，并且有些为 ARM 打造的开发工具也是基于 GNU 工具链的。在目前，支持 CM3 的 GNU 工具链可以由 [CodeSourcery](http://www.codesourcery.com) 处免费下载到 (www.codesourcery.com)。而 GNU 的主打 C 编译器则在以后支持 CM3 (在 2008 年 3 月 31 日以后，主流的 GNU 工具链已经支持 Cortex-M3，对应的开发工具为 WinARM——译者注)。

本章只介绍使用 GNU 工具链的基础知识，更详细的信息还需要参阅联机帮助文档。值得一提的是，GNU 的汇编语法 (GNU 工具链中的 AS 程序) 与 ARM 的汇编语法是有些不同的。这些不同点包括变量定义、编译指示字、以及 `the like`。因此，使用 ARM RealView 工具的汇编代码在使用 GNU 工具前，还需要一些 (很枯燥的) 修改工作。

19.2 获取 GNU 工具链

编译好的 GNU 工具链可以从 www.codesourcery.com/gnu_toolchains/arm/ 处下载。有一系列的二进制构建版本。对于最简单的使用，可以使用 EABI^[注]，并且不带嵌入式 OS 支持的版本。这个工具链既有在 Windows 上使用的版本，也有在 Linux 上使用的版本。本章给出的示例程序可以用于任何一个版本上。

注 (EABI 表示嵌入式应用程序二进制接口。可执行目标文件必须符合该规格，从而可以跨开发工具集使用)

19.2.1 开发流程

和 ARM 开发工具的相似，GNU 工具链也包含了编译器、汇编器和连接器，从而使得源代码既可以使用 C，也可以使用汇编写成，如图 19.1 所示。

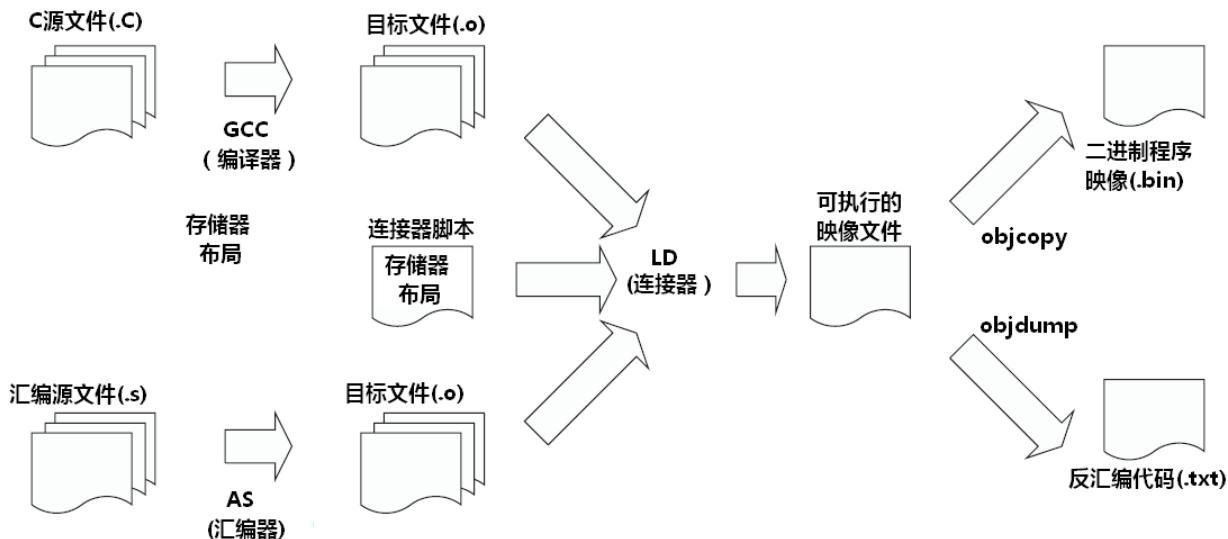


图 19.1 基于 GNU 工具链的开发流程模式图

不同的应用程序环境中也有不同版本的工具链 (Symbian, Linux, EABI 等)。取决于工具链的目标平台，相应的可执行文件通常有一个前缀。例如，如果使用了 EABI 环境，则 GCC 命令为 `arm-xxxx-eabi-gcc`。本章的目标代码使用 CodeSourcery 的 GNU ARM 工具链，如表 19.1 所示。

表 19.1 winARM20080331 GNU 工具链的命令名称

功能	命令
汇编器	<code>arm-none-eabi-as</code>
编译器	<code>arm-none-eabi-gcc</code>
连接器	<code>arm-none-eabi-ld</code>
二进制映像产生器	<code>arm-none-eabi-objcopy</code>
反汇编器	<code>arm-none-eabi-objdump</code>

在开发流程图中，连接脚本是可选的。但是当存储器映射比较复杂时，常常是必需的。

19.3 示例程序

让我们开开眼，看一看 GNU 工具链下的源代码的众生相。

19.3.1 例 1：第一个程序

作为启蒙，让我们把在第 10 章引入的简单程序使用 GCC 重写一遍。这个程序计算 $10+9+8+\dots+1$ 的值，如下所示：

```
===== example1.s =====
/* 定义常数 */
.equ STACK_TOP, 0x20000800
.text
.global _start
.code 16
.syntax unified
/* .thumbfunc */
/* .thumbfunc 仅仅在 2006Q3-26 之前的 CodeSourcery 工具中需要 */
_start:
```

```

.word STACK_TOP, start
.type start, function
/* 主程序入口点 */
start:
    movs    r0,    #10
    movs    r1,    #0
    /* 计算 10+9+8...+1 */
loop:
    adds    r1,    r0
    subs    r0,    #1
    bne    loop
/* Result is now in R1 */
deadloop:
    b      deadloop
.end
===== end of file =====

```

- **.word** 指示字定义 MSP 起始值为 **0x2000_0800**，并且把”**start**”作为复位向量。
- **.text** 也是一个预定义的指示字，表示从这以后是一个代码区，需要予以汇编。
- **.global** 使 **_start** 标号可以由其它目标文件使用。
- **.code 16** 指示程序代码使用 **thumb** 写成。
- **.syntax unified** 指示使用了统一汇编语言语法。
- **_start** 是一个标号，指示出程序区的入口点
- **start** 是另一个标号，它指示复位向量。
- **.type start, function** 宣告了 **start** 是一个函数。对于所有处于向量表中的异常向量，这种宣告都是必要的，否则汇编器会把向量的 LSB 清零——这在 **thumb** 中是不允许的。
- **.end** 指示程序文件的结束。

与 ARM 汇编器不同的是，GNU 汇编器中的标号要以 “:” 结尾；注释可以使用/*和*/，并且指示字要以一个.”作为前缀。

要注意：在 **thumb** 代码（**.code 16**）里面，复位向量（**start**）被定义成了一个函数（**.type start, function**）。这是为了使复位向量的 LSB 被强制为 1，从而表示这是以 **Thumb** 状态开始执行。否则，处理器就会尝试以 **ARM** 态开始，从而引起一个硬 **fault**。

程序写好后，使用 **as** 来汇编这个源程序，命令格式为：

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
```

执行了这个命令，就产生了目标文件 **example1.o**。命令行中的 **-mcpu** 和 **-mthumb** 决定使用的指令集。接下来执行连接，命令如下

```
$> arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o
```

然后，使用目标拷贝命令（**objcopy**）来产生二进制文件：

```
$> arm-none-eabi-objcopy -Obinary example1.out example1.bin
```

我们还可以使用目标倾倒（**dump**）命令（**objdump**）来创建一个反汇编代码来检查生成的目标文件：

```
$> arm-none-eabi-objdump -S example1.out > example1.list
```

生成的反汇编应如下所示：

```
example1.out: file format elf32-littlearm
```

```
Disassembly of section .text:
```

```
00000000 <_start>:  
0: 0800 lsrs r0, r0, #32  
2: 2000 movs r0, #0  
4: 0009 lsls r1, r1, #0  
...  
00000008 <start>:  
8: 200a movs r0, #10  
a: 2100 movs r1, #0  
0000000c <loop>:  
c: 1809 adds r1, r1, r0  
e: 3801 subs r0, #1  
10: d1fc bne.n c <loop>  
00000012 <deadloop>:  
12: e7fe b.n 12 <deadloop>
```

19.3.2 例 2：连接多个文件

如前所述，我们可以创建多个目标文件，并且把它们连接到一起。在这个例子里，我们有两个汇编程序文件，分别是 `example2a.s` 和 `example2b.s`。前者只包含向量表，而后者包含了正常的程序代码。这里，`.global` 指示字就派上用场了——在文件之前传递全局符号。

```
===== example2a.s =====  
/* 定义常数 */  
.equ STACK_TOP, 0x20000800  
.global vectors_table  
.global start  
.global nmi_handler  
.code 16  
.syntax unified  
vectors_table:  
.word STACK_TOP, start, nmi_handler, 0x00000000  
.end  
===== end of file =====  
  
===== example2b.s =====  
/* 主程序 */  
.text  
.global _start  
.global start  
.global nmi_handler  
.code 16  
.syntax unified  
.type start, function  
.type nmi_handler, function  
_  
_start:  
/* 主程序入口点 */
```

```

start:
    movs    r0,    #10
    movs    r1,    #0
    /* 计算 10+9+8...+1 */
loop:
    adds    r1,    r0
    subs    r0,    #1
    bne    loop
    /* 结果存储在R1中 */
deadloop:
    b      deadloop
    /* 为演示而设置的空NMI服务例程 */
nmi_handler:
    bx    lr
    .end
===== end of file =====

```

创建可执行映像的步骤为：

1. 汇编 `example2a.s`

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example2a.s -o example2a.o
```

2. 汇编 `example2b.s`

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example2b.s -o example2b.o
```

3. 把 2 个目标文件连接成单一的映像。要注意的是，目标文件在命令行中的顺序是重要的，它会影响在最终的目标文件中，把这两个目标文件的代码编排的顺序。

```
$> arm-none-eabi-ld -Ttext 0x0 -o example2.out example2a.o example2b.o
```

4. 产生二进制文件

```
$> arm-none-eabi-objcopy -Obinary example2.out example2.bin
```

5. 如上例，可以创建一个反汇编文件来检查所产生目标文件的内容。

```
$> arm-none-eabi-objdump -S example2.out > example2.list
```

当目标文件增多时，为简化处理过程，我们可以使用 `make` 来管理工程。另外，开发套件也常常有各自内建的功能来简化编译过程。

19.3.3 例 3：一个简单的“Hello World”程序

前两个例子算是热身，现在该动真格的了。让我们试一个“hello world”程序。但是在这里为了突出主题，我们省去了 `UART` 初始化代码。第 20 章给出了一个 C 语言写成的 `UART` 示例代码。

```
===== example3a.s =====
```

```

/* 定义常数 */
.equ STACK_TOP, 0x20000800
.global vectors_table
.global _start
.code 16
.syntax unified
vectors_table:
.word STACK_TOP, _start
.end

```

```
===== end of file =====

===== example3b.s =====

.text
.global _start
.code 16
.syntax unifi ed
.type _start, function

_start:
/* 主程序入口点 */
    movs    r0,    #0
    movs    r1,    #0
    movs    r2,    #0
    movs    r3,    #0
    movs    r4,    #0
    movs    r5,    #0
    ldr     r0,    =hello
    bl     puts
    movs    r0,    #0x4
    bl     putc

deadloop:
    b      deadloop

hello:
.ascii  "Hello\n"
.byte   0
.align

puts:
/* 该子程序向UART发送字符串 */
/* 入口条件: r0 = 字符串的起始地址 */
/* 字符串要以零结尾 */
    push    {r0, r1, lr}      /* 保存寄存器 */
    mov     r1,    r0          /* 把地址拷贝到R1, 因为 */
                               /* R0 还要用于作putc的参数 */

putsloop:
    ldrb.w  r0,    [r1],    #1  /* 读取一个字符并且自增地址 */
    cbz     r0,    putsloopexit /* 如果字符为NULL, 则跳转到结束 */
    bl     putsloop
    b      putsloop

putsloopexit:
    pop    {r0, r1, pc}      /* 返回 */
    .equ  UART0_DATA, 0x4000C000
    .equ  UART0_FLAG, 0x4000C018

putc:
/* 该子程序通过UART发送一个字符 */
```

```

/* 入口条件: R0 = 要发送的字符 */
push    {r1, r2, r3, lr}      /* 保存寄存器 */
ldr     r1,      =UART0_FLAG
putcwaitloop:
    ldr     r2,      [r1]      /* 获取状态位 */
    tst.w  r2,      #0x20      /* 检查发送缓冲区满标志 */
    bne   putcwaitloop      /* 如果已满则循环等待 */
    ldr     r1,      =UART0_DATA /* 否则继续往发送缓冲区里送数据 */
    str     r0,      [r1]
    pop    {r1, r2, r3, pc}      /* 返回 */
.end
===== end of file =====

```

在这个例子里，我们使用了`.ascii` 和`.byte` 指示字来创建一个零结尾的字符串。在定义了字符串之后，我们又使用了`.align` 来确保下一条指令会以正确的位置开始。如果不使用`.align`，汇编器则可能把下一条指令放到未对齐的地址。

创建目标代码的步骤如下所示，读者应理解下述命令的含义和作用。

```

$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example3a.s -o example3a.o
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example3b.s -o example3b.o
$> arm-none-eabi-ld -Ttext 0x0 -o example3.out example3a.o example3b.o
$> arm-none-eabi-objcopy -Obinary example3.out example3.bin
$> arm-none-eabi-objdump -S example3.out > example3.list

```

19.3.4 例 4：把数据放到 RAM 中

RW 数据需要放到 RAM 中，本例就演示在 RAM 中定义变量的方法。

```

===== example4.s =====
.equ STACK_TOP, 0x20000800
.text
.global _start
.code 16
.syntax unified
_start:
.word STACK_TOP, start
.type start, function
start:
    movs r0, #10
    movs r1, #0
    /* 计算10+9...+1 */
loop:
    adds r1, r0
    subs r0, #1
    bne loop
    /* 结果现在存储到R1中了 */
    ldr r0, =result
    str r1, [r0]

```

```

deadloop:
    b deadloop
    /* 数据区 */
    .data
result:
    .word 0
    .end
===== end of file =====

```

本例的核心就是粗体的 **.data** 指示字。使用它创建一个数据区。在该区中，使用一个 **.word** 指示字来保留一个 4 字节的空间，并且取名为 **Result**（其实 **result** 就相当于 C 中的变量名）。欲连接本程序，需要告诉连接器 **RAM** 在何处，这可以使用 **-Tdata** 选项来实现，它把数据段设置到所需的位置上：

```

$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example4.s -o example4.o
$> arm-none-eabi-ld -Ttext 0x0 -Tdata 0x20000000 -o example4.out
example4.o
$> arm-none-eabi-objcopy -Obinary -R .data example4.out example4.bin
$> arm-none-eabi-objdump -S example4.out > example4.list

```

还要注意的是，在 **objcopy** 中对 **-R .data** 选项的使用。它避免在二进制输出文件中把数据存储区也包含进去。

19.3.5 例 5：纯 C 程序

想必大家已经受够了在汇编下过日子了吧！在 GNU 工具链中的一个主要组件就是 C 编译器。在本例中，整个可执行程序——甚至是复位向量和 MSP 初值都由 C 写成。此外，还添加了一个连接器脚本，用来把各段放到正确的位置。那么，先让我们看一看 C 程序文件。

```

===== example5.c =====
#define STACK_TOP 0x20000800
#define NVIC_CCR ((volatile unsigned long *) (0xE000ED14))
// 声明函数原型
void myputs(char *string1);
void myputc(char mychar);
int main(void);
void nmi_handler(void);
void hardfault_handler(void);
// 定义向量表
__attribute__ (( section("vectors") )) void (* const VectorArray[])(void) =
{
    STACK_TOP,
    main,
    nmi_handler,
    hardfault_handler
};

// 主程序入口点
int main(void)

```

```
{  
    const char *helloworld[]="Hello world\n";  
    *NVIC_CCR = *NVIC_CCR | 0x200; /* 设置 NVIC 的 STKALIGN */  
    myputs(*helloworld);  
    while(1);  
    return(0);  
}  
  
// 函数  
void myputs(char *string1)  
{  
    char mychar;  
    int j;  
    j=0;  
    do  
    {  
        mychar = string1[j];  
        if (mychar!=0)  
        {  
            myputc(mychar);  
            j++;  
        }  
    } while (mychar != 0);  
    return;  
}  
void myputc(char mychar)  
{  
    #define UART0_DATA ((volatile unsigned long *)(0x4000C000))  
    #define UART0_FLAG ((volatile unsigned long *) (0x4000C018))  
    // Wait until busy flag is clear  
    while ((*UART0_FLAG & 0x20) != 0);  
    // Output character to UART  
    *UART0_DATA = mychar;  
    return;  
}  
  
// 空的服务例程  
void nmi_handler(void)  
{  
    return;  
}  
  
void hardfault_handler(void)  
{
```

```

    return;
}

===== end of file =====

```

注意粗体字显示的部分，它使用 `__attribute__(())`（注意，是双小括号）来指定特殊的属性。在这里则指出那个函数指针数组是放到 `vectors` 段中的。然而，这个 C 程序并没有指定 `vectors` 段在何处。那么在哪里指定 `vectors` 段的位置呢？现在该请出我们的连接器脚本文件了，工作就在这里完成。本例的连接器脚本文件为 `simple.ld`，内容如下：

```

===== simple.ld =====

/* MEMORY 命令：定义允许的存储器区域 */
/* 本部分定义了连接器允许放入数据的各存储器区域，这是 */
/* 一个可选的功能，但是对于开发很有益，它使连接器在在 */
/* 程序太大时能给你警告 */

MEMORY
{
    /* ROM 是可读的(r)和可执行的(x) */
    rom (rx) : ORIGIN = 0, LENGTH = 2M
    /* RAM 是可读的(r)，可写的(w)，可执行的(x) */
    ram (rwx) : ORIGIN = 0x20000000, LENGTH = 4M
}

/* SECTIONS 命令：定义各输入段到输出段的映射 */

SECTIONS
{
    . = 0x0;                      /* 从 0x00000000 开始 */
    .text : {
        *(vectors)           /* 向量表 */
        *(.text)              /* 程序代码 */
        *(.rodata)            /* 只读数据 */
    }
    . = 0x20000000;              /* 从 0x20000000 开始 */
    .data : {
        *(.data)              /* 数据存储器 */
    }
    .bss : {
        *(.bss)              /* 预留的数据存储器，必须初始化为零 */
    }
}
===== end of file =====

```

为使用连接脚本，需要在编译阶段把 `simple.ld` 传给编译器。

```
$> arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb example5.c -nostartfiles
-T simple.ld -o example5.o
```

然后在连接时，需要再次使用 `simple.ld`。

```
$> arm-none-eabi-ld -T simple.ld -o example5.out example5.o
```

本例中我们只有一个源文件，因此连接过程其实是可以省略的。最后再创建二进制目标文件和

反汇编文件。

```
$> arm-none-eabi-objcopy -Obinary example5.out example5.bin
$> arm-none-eabi-objdump -S example5.out > example5.list
```

读者可能还注意到了，在本例中我们使用了另一个称为 **-nostartfiles** 的编译器开关。使用它，就可以让编译器不再往可执行映像中插入启动代码（**crt**），这样做的目的之一就是减少程序映像的尺寸。不过，使用该选项的主要原因，其实是在于 **GNU** 工具链的启动代码是与发布包的提供者相关的，而有些人提供的启动代码不适合 **Cortex-M3**——它们往往是用于传统的 **ARM** 处理器的——如 **ARM7**（典型地这些启动代码使用了 **ARM** 代码，而没有使用 **Thumb** 代码）。

但是，在许多情况下，取决于应用程序和使用的库，都必须使用启动代码来执行初始化的过程，最主要的就是对数据的初始化（例如，把 **bss** 区的存储单元全部清零）。在最后一个例子中，我们将演示这个过程。

19.3.6 例 6：纯 C 程序，带有标准 C 启动代码

在正常情况下，当编译 C 程序时，会自动地把标准 C 库的启动代码包含在目标文件中，它保证运行时库得以正确地初始化。标准 C 运行时库的启动代码由 **GNU** 工具链提供，但是不同提供者提供的工具链可能有不同的启动代码。下例是基于 **CodeSourcery GNU ARM 工具链 2006q3-26** 版本的。因此，最好检查一下从工具链中的启动代码，或者从供应商处获取最新的启动代码。对于这个版本的 **CodeSourcery** 提供的工具链，其启动代码目标文件为 **armv7m-crt0.o**。但是这个版本提供的启动代码是错误的——使用了 **ARM** 代码来编写。到了 **2006q3-27** 及更晚的版本中才修正了这个 **bug**。不同提供者的 **GNU** 工具链会有不同的启动代码，而且文件名也常常不同。此时，就需要检查你所使用的 **GNU** 工具链之帮助文档来获取准确信息了。

在编译 C 源代码之前，例 5 中的 C 程序需要一些小改动。缺省情况下，**armv7m-crt0** 已经包含了一张向量表，并且在它里面，**NMI** 服务例程和硬 **fault** 服务例程分别取名为 **_nmi_isr** 和 **_fault_isr**。因此，需要移除例 5 中的向量表，并且重命名 **NMI** 和硬 **Fault** 的服务例程，如下所示：

```
// 声明函数原型
void myputs(char *string1);
void myputc(char mychar);
int main(void);
void _nmi_isr(void);
void _fault_isr(void);

// 主程序入口点
int main(void)
{
    const char *helloworld[]="Hello world\n";
    myputs(*helloworld);
    while(1);
    return(0);
}

// 函数
void myputs(char *string1)
{
```

```
char mychar;
int j;
j=0;
do
{
    mychar = string1[j];
    if (mychar!=0)
    {
        myputc(mychar);
        j++;
    }
} while (mychar != 0);
return;
}
void myputc(char mychar)
{
#define UART0_DATA ((volatile unsigned long *)(0x4000C000))
#define UART0_FLAG ((volatile unsigned long *) (0x4000C018))
// Wait until busy flag is clear
while ((*UART0_FLAG & 0x20) != 0);
// Output character to UART
*UART0_DATA = mychar;
return;
}

//空的服务例程
void _nmi_isr(void)
{
    return;
}

void _fault_isr(void)
{
    return;
}
```

在安装了 **CodeSourcery** 后, 已经包含了一系列的连接脚本, 可以从 **codesourcery/sourcery g++/arm-none-eabi/lib** 目录下找到。在下例中, 我们就使用了 **lm3s8xx-rom.ld** 文件。这个连接器脚本顾名思义, 是用于 **LM3S8XX** 系列芯片的。

在当前目录之外, 当C程序代码定位后, 一个名为“**lib**”的库子目录也在在当前目录下创建, (Aside from the current directory, when the C program code is located, a library subdirectory called **lib** is also created in the current directory) 这使得库搜索路径的设置更加简单——所需的目标文件 **armvrm-crt0.o** 以及连接器脚本都被拷贝到这个“**lib**”目录下。在下一个例子中, 我们就使用**-L lib** 选项来把“**lib**”添加到库的搜索路径中。

现在我们可以编译这个C程序了：

```
$> arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb example6.c -L lib -T
lm3s8xx-rom.ld -o example6.out
```

执行了上条命令后，就创建并且连接了目标文件**example6.out**。因为只有一个目标文件，二进制文件可以直接由它来生成：

```
$> arm-none-eabi-objcopy -O binary example6.out example6.bin
```

产生反汇编的方式则与上例相同：

```
$> arm-none-eabi-objdump -S example6.out > example6.list
```

19.4 访问特殊功能寄存器

在**CodeSourcery**的GNU ARM工具链中，可以直接使用小写的名字来访问特殊功能寄存器（注意，必须是小写的），如下所示：

```
msr      control,    r1
mrs      r1,        control
msr      apsr,      r1
mrs      r0,        psr
```

19.5 使用未支持的指令

如果使用了另外的GNU工具链，有可能那个GNU汇编器不支持一些指令。在这种情况下，则可以直接使用.**word**来插入不支持指令的二进制机器码，如下所示：

```
.equ DW_MSR_CONTROL_R0, 0x8814F380
...
MOV R0, #0x1
.word DW_MSR_CONTROL_R0 /* 相当于执行 MSR CONTROL, R0 指令 */
...
```

19.6 GNU C 编译器的内联汇编

GNU的ARM C编译器是支持内联汇编的，但此时的汇编语法看起来有点怪：

```
__asm (" inst1 op1, op2... \n"
" inst2 op1, op2... \n"
...
" inst op1, op2... \n"
: 输出操作数s /* 可选 */
: 输入操作数s /* 可选 */
```

先举一个简单的例子，进入睡眠模式的代码如下所示：

```
void Sleep(void)
{
    // 使用Wait-For-Interrupt进入睡眠模式
    __asm (
        "WFI\n"
    );
}
```

```
}
```

如果汇编代码需要一个输入变量和一个输出变量，例如，把一个变量除以5，则格式如下：

```
__asm ( "mov r0, %0\n"
        "mov r3, #5\n"
        "udiv r0, r0, r3\n"
        "mov %1, r0\n"
        : "=r" (DataOut) : "r" (DataIn) : "cc", "r3" );
```

在这个代码中，输入参数是一个C变量，名为**DataIn**（%0代表第一个参数），该代码把结果返回到另外一个C变量**DataOut**中（%1表示第2个参数）。内联汇编的代码还手工修改了寄存器r3，并且修改了条件标志cc，因此它们被列在被破坏的（**clobbered**）寄存器列表中。

更详细的内联汇编信息在[GCC-Inline-Assembly-HOWTO](#)文档中。

KEIL RealView Microcontroller Development Kit (RVMDK) 使用入门

- 简介
- uVision 使用入门
- 使用 UART 输出 “Hello World”
- 测试示例程序
- 使用调试器
- 指令模拟器
- 修改向量表
- 使用中断实现的秒表示例程序

20.1 简介

有许多商业的开发平台可以用在 CM3 上，其中最流行之一就是 KEIL 的 RealView Microcontroller Development Kit (简称 RealView MDK 或 RVMDK)。RVMDK 的前身就是曾一度在 8051 开发业界享有盛誉的 KEIL 套件。RVMDK 包含了很丰盛的组件：

- uVision
- 集成开发环境
- 调试器
- 模拟器
- 由 ARM 提供的 RealView 工具链
 - ◆ C/C++ 编译器
 - ◆ 汇编器
 - ◆ 连接器
- RTX 实时内核
- 为各单片机而设的详细启动代码 (包含源代码)
- 各种 Flash 的编程算法
- 程序示例

(英蓓特还把 RVMDK 的帮助文件翻译成了中文，并包装成“中国版”的 RVMDK——译者注)。

使用 RVMDK 来学习 CM3，甚至不需要拥有 CM3 硬件——uVison 环境包含了指令模拟器，使用它可以测试“纯粹”的 CM3 程序代码，对于学习和开发基于内核的系统软件都很有好处。

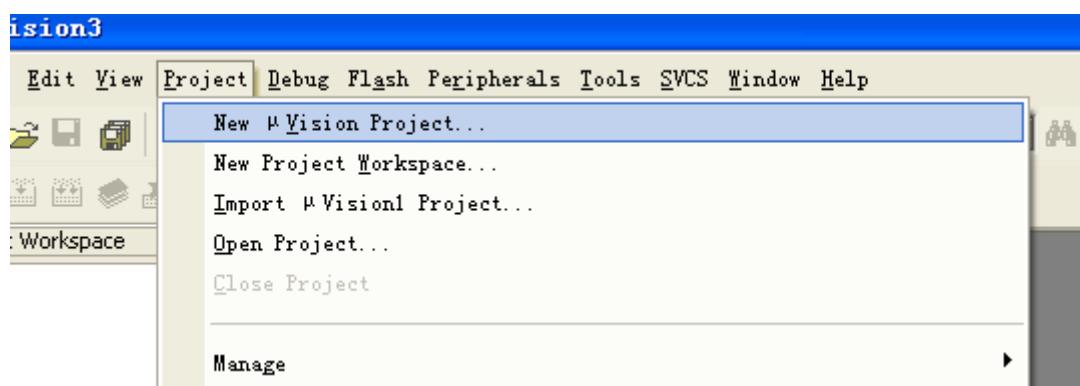
RVMDK 还可以与 GNU 工具链一起使用。

可以从 KEIL 网站上获取免费的 KEIL tool 之演示版，也可以从 <http://www.realview.com.cn/> 处下载中文的相关资源。

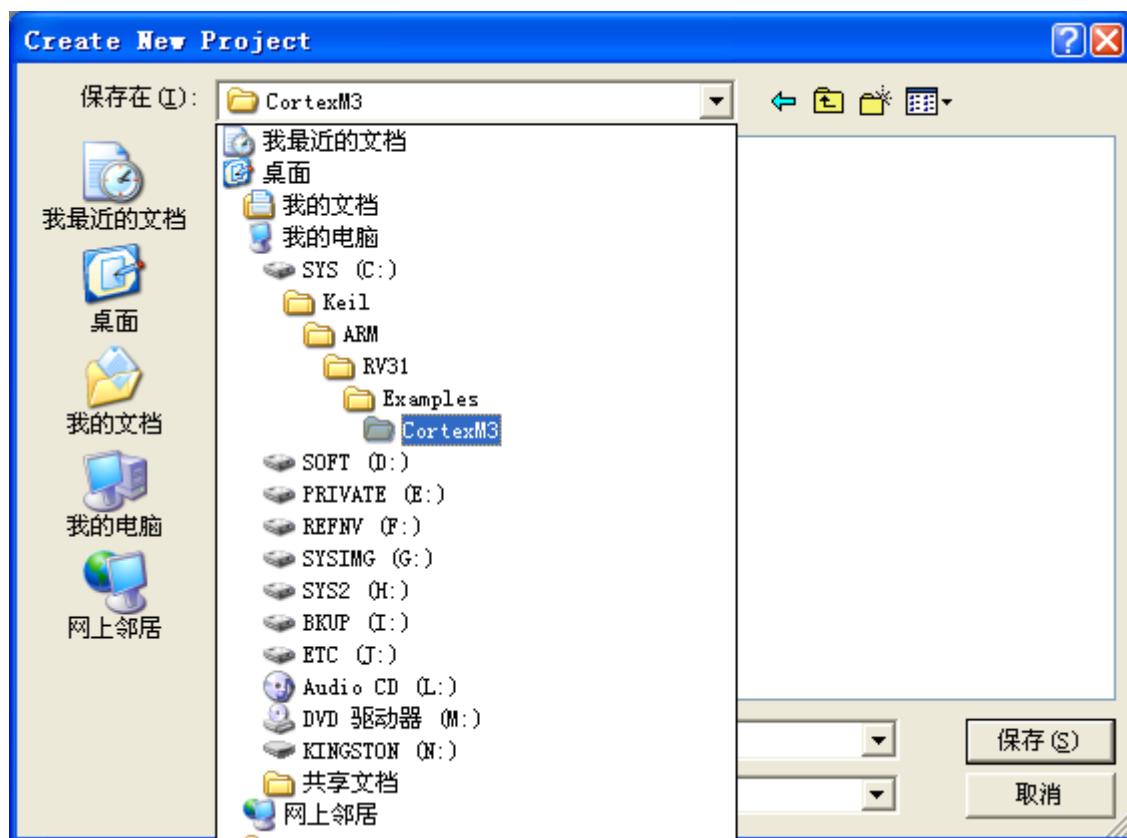
20.2 uVision 使用入门

在 RVMDK 中附带了很多示例程序，包括 Luminous Micro 的 Stellaris 系列的单片机产品，也包括了 ST 的 STM32 系列的单片机产品。这些示例都使用了厂家提供的驱动程序库（固件库）。使用固件库可以免去写代码操作外设寄存器的任务。很容易通过修改示例程序来开发自己的应用程序，也可以自己从头设计工程，再摘抄一部分示例程序的代码。本章的示例基于 RVMDK v3.03 版，并且以 Luminous Micro 的 LM3S811 器件为蓝本（目前 RVMDK 已经出了 3.20 版，且作者写本书时 Luminous Micro 是唯一的 CM3 芯片供应商。目前 ST 也出品了 STM32 系列的 CM3 芯片，预计以后 Atmel, TI, NXP 也要提供 CM3 芯片——译者注）。

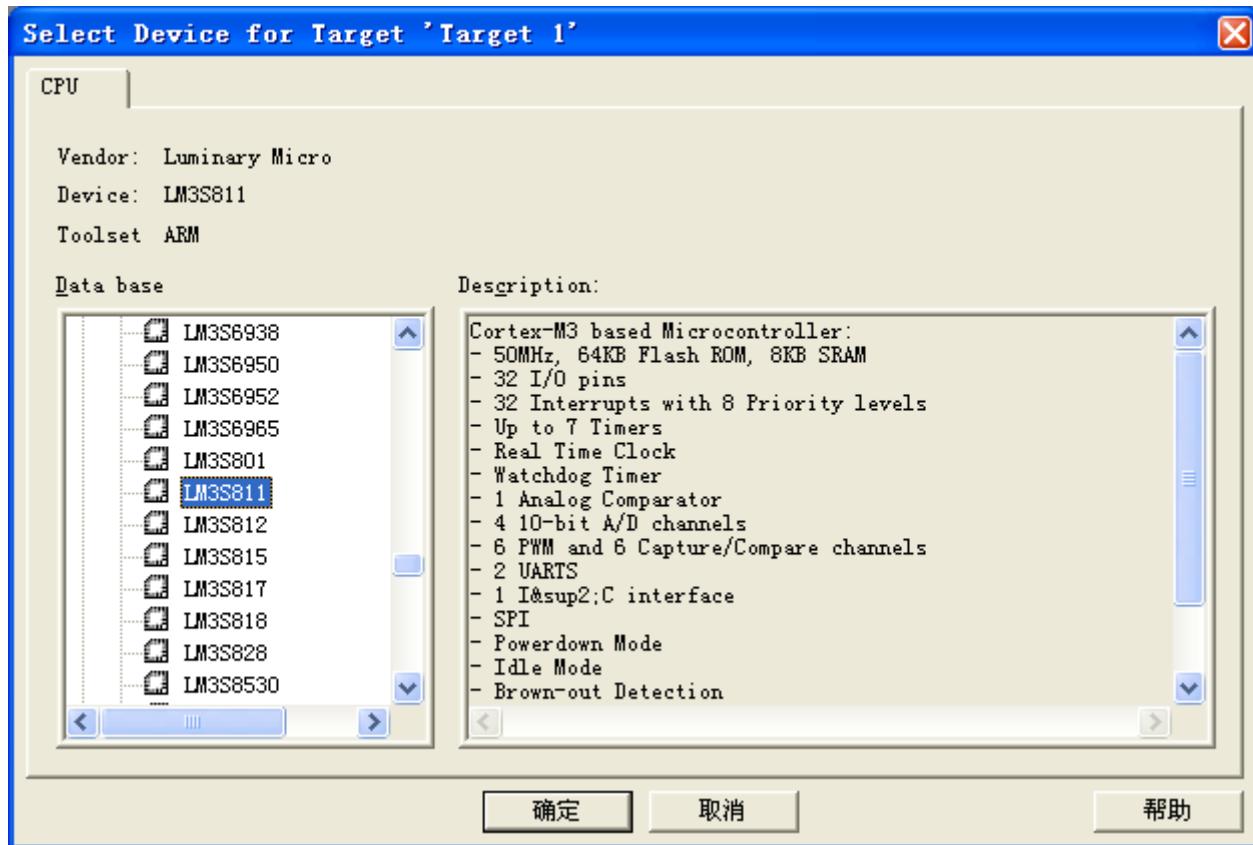
在安装了 RVMDK 后，就可以从开始菜单中启动 uVision 集成开发环境，并且它会打开一个为传统 ARM 处理器而写成的示例。我们可以关掉这个工程，并且通过选择“New Project”下拉菜单来新建一个工程：



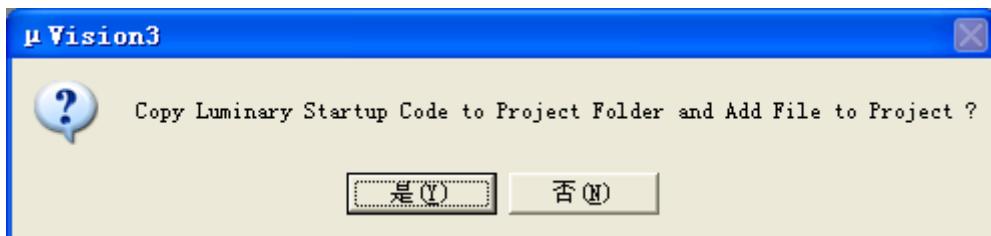
这里创建了一个名为 CortexM3 的文件夹：



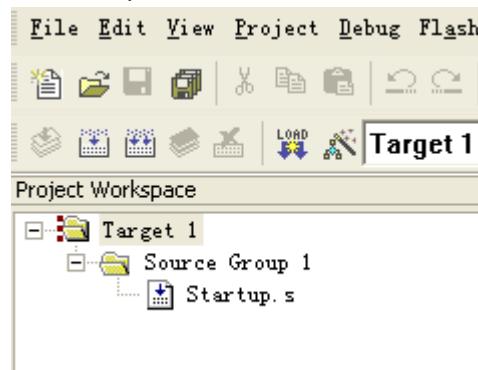
接下来为这个工程选择目标器件，在这里选择 LM3S811



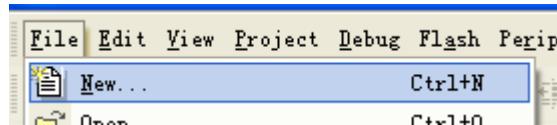
RVMDK 会询问是否使用缺省的启动代码。这里我们选择 Yes。



现在我们有了只含有一个源文件 (Startup.s) 的工程了：



接下来，我们要创建一个含有 main 函数的 C 源程序。



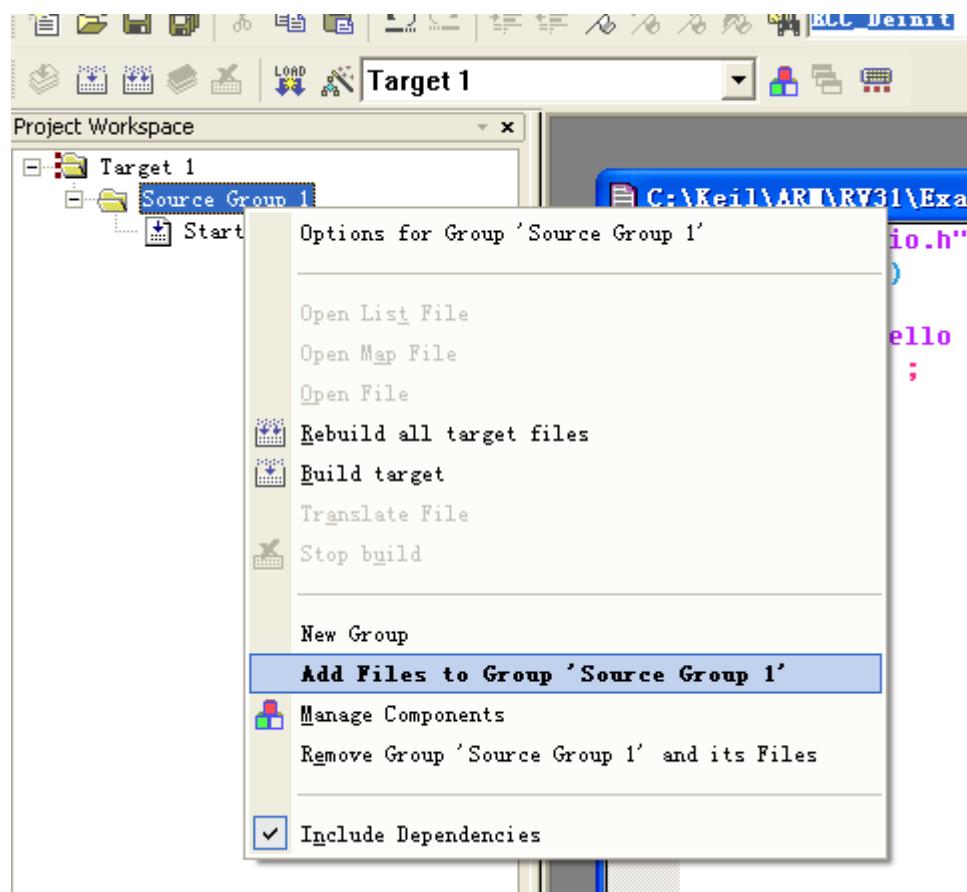
这样就创建了一个文本文件。编辑它的内容，并且存储为 hello.c：

```

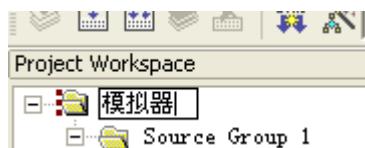
1 #include "stdio.h"
2 int main(void)
3 {
4     printf("Hello world!\n");
5     while (1);
6 }
7

```

现在，我们要把这个文件添加到 Source Group 1 中（右击 Source Group 1）



可以修改目标名“Target 1”和文件组名“Source Group 1”，以使它们更有意义。通过单击工程的工作区选中它们，过一会儿再单击即可修改（超过双击判定的最短间隔时间）：

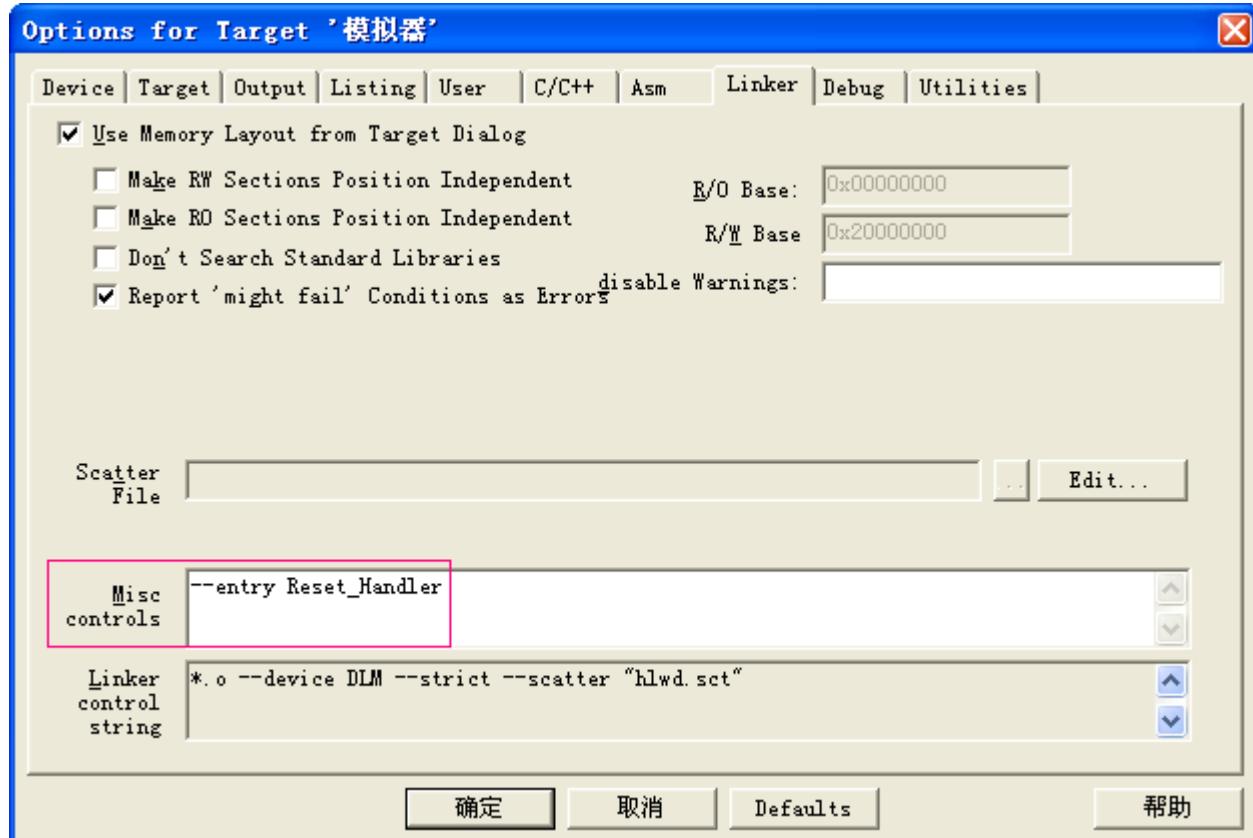


选择我们刚刚创建的 `hello.c` 并添加，则工程中就包含了 2 个源文件了：

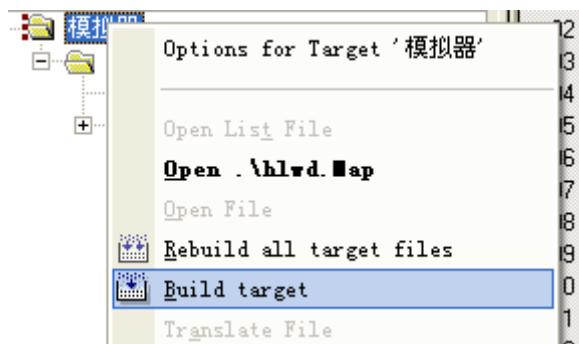


我们还需要设置连接器以定义程序的入口点。通过在“Misc Controls box”中加入 `entry`

`Reset_Handler` 来实现。这个选项定义了程序的入口点为 `Reset_Handler`——它可以在 `Startup.s` 中找到(其实在 RVMDK 3.20 版本中,连接设置有了变化,此时不加也可以——译者注)。



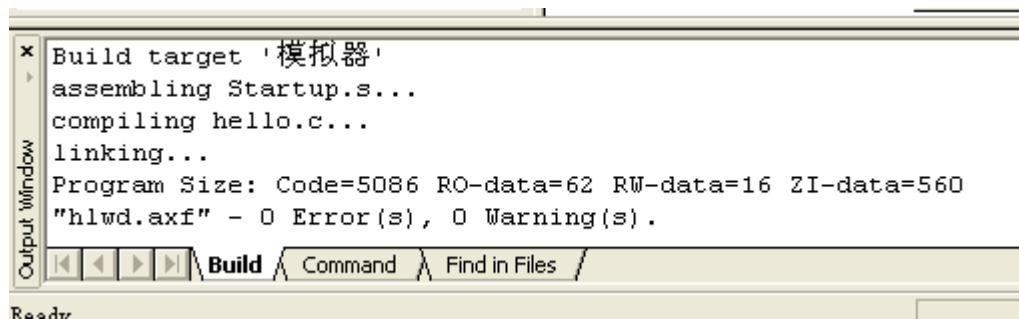
现在我们就可以编译了。如下所示:



也可以使用工具栏来方便地操作:



编译成功后, 输出窗口如下显示:



20.3 使用 UART 输出 “Hello World”

在上一个例子中，我们使用了C标准库中的`printf`函数。但是C标准库并不知道我们使用的硬件是什么，因此如果要“真实”地输出字符串（如通过UART输出），还必须添加一些代码。正如本书曾经提到的，为使输出送到实际的硬件，我们经常需要做所谓的“**retargeting**”工作。与**Retargeting**相关的函数除了可以用于输出文本外，还可以包含处理错误以及终结程序等其它功能。在本例，只介绍文本输出的功能。

在本例中，是打算把“Hello World”消息从LM3S811的UART0送出去，目标系统是Luminous Micro LM3S811评估板。板上晶振为6MHz，但单片机片内配有PLL模块，它把时钟频率上升到50MHz。波特率是115,200，并且使用PC上的超级终端程序来接收从UART发出的数据。

要对`printf`执行**retarget**处理，我们需要实现`fputc`函数。在下面的代码中，我们就创建了这个`fputc`函数，它又呼叫`sendchar`函数，而后者则操作UART输出字符：

```
#include "stdio.h"

#define CR 0x0D //回车符
#define LF 0x0A //换行符
void Uart0Init(void);
void SetClockFreq(void);
int sendchar(int ch);
// 若使用6MHz，则注释掉下一行
#define CLOCK50MHZ
// Register addresses
#define SYSCTRL_RCC ((volatile unsigned long *) (0x400FE060))
#define SYSCTRL_RIS ((volatile unsigned long *) (0x400FE050))
#define SYSCTRL_RCGC1 ((volatile unsigned long *) (0x400FE104))
#define SYSCTRL_RCGC2 ((volatile unsigned long *) (0x400FE108))
#define GPIOA_AFSEL ((volatile unsigned long *) (0x40004420))
#define UART0_DATA ((volatile unsigned long *) (0x4000C000))
#define UART0_FLAG ((volatile unsigned long *) (0x4000C018))
#define UART0_IBRD ((volatile unsigned long *) (0x4000C024))
#define UART0_FBRD ((volatile unsigned long *) (0x4000C028))
#define UART0_LCRH ((volatile unsigned long *) (0x4000C02C))
#define UART0_CTRL ((volatile unsigned long *) (0x4000C030))
#define UART0_RIS ((volatile unsigned long *) (0x4000C03C))
int main (void)
```

```
{  
    SetClockFreq();           // 建立时钟的配置 (50MHz/6MHz)  
    Uart0Init();             // 初始化UART0  
    printf ("Hello world!\n");  
    while (1);  
}  
  
void SetClockFreq(void)  
{  
#ifdef CLOCK50MHZ  
    // 置位BYPASS, 清除 USRSYSDIV 和 SYSDIV  
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF83FFFF) | 0x800 ;  
    // 清零 OSCSRC, PWRDN 和 OEN  
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xFFFFFCFC);  
    //修改 SYSDIV, 设置 USRSYSDIV 和 Crystal 位段的值  
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF87FFC3F) | 0x01C002C0;  
    // 等待PLLLRIS被置位  
    while ((*SYSCTRL_RIS & 0x40)==0); // wait until PLLRIS is set  
    // 清除bypass  
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xFFFFF7FF) ;  
#else  
    // 置位 BYPASS, 清除 USRSYSDIV 和 SYSDIV  
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF83FFFF) | 0x800 ;  
#endif  
    return;  
}  
  
void Uart0Init(void)  
{  
    *SYSCTRL_RCGC1 = *SYSCTRL_RCGC1 | 0x0003; // 使能 UART0 & UART1  
    // clock  
    *SYSCTRL_RCGC2 = *SYSCTRL_RCGC2 | 0x0001; // 使能 PORTA 时钟  
    *UART0_CTRL = 0; // 除能 UART  
#  ifdef CLOCK50MHZ  
    *UART0_IBRD = 27; // 以50MHz频率为基准编程波特率  
    *UART0_FBRD = 9;  
#  else  
    *UART0_IBRD = 3; // 以6MHz频率为基准编程波特率  
    *UART0_FBRD = 17;  
#  endif  
    *UART0_LCRH = 0x60; // 8 bit, 无奇偶  
    *UART0_CTRL = 0x301; // 使能 TX 和 RX, 以及 UART 使能  
    *GPIOA_AFSEL = *GPIOA_AFSEL | 0x3; // 把GPIO管脚交给UART0控制  
    return;  
}  
// 送给UART0一个字符 (printf函数使用它来输出文字)
```

```

int sendchar (int ch)
{
    if (ch == '\n')
    {
        while ((*UART0_FLAG & 0x8)); // 如果UART忙碌中则等待
        *UART0_DATA = CR; // 输入附加的CR以便字符串被正确显示
    }
    while ((*UART0_FLAG & 0x8)); // 如果UART忙碌中则等待
    return (*UART0_DATA = ch); // 输出数据
}

//文本输出的retargeting代码
int fputc(int ch, FILE *f)
{
    return (sendchar(ch));
}

```

代码中的**SetupClockFreq()**用于把系统时钟设置为50MHz。要注意的是这个函数是与具体的器件相关的。另外，还使用了条件编译来允许选择使用6MHz的原始频率。

UART的初始化是由**Uart0Init()**来执行的，它设置了波特率为115200，8个数据位，1个停止位，无奇偶校验，并且启用GPIO的第二功能，从而让GPIO控制器把管脚的控制权交给UART0。在使用UART和GPIO之前，必须先启用这两个模块。代码中的把**SYSCTRL_RCGC1**和**SYSCTRL_RCGC2**分别启用了这两个模块。

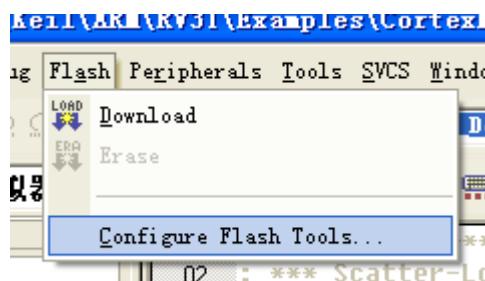
Retargeting的代码是由粗体的**fputc()**执行的。要注意的是，不能使用其它的名字，因为”**fputc**”是编译器预定义的用于字符输出的函数名。**fputc()**实际上只是个封皮，它直接调用**sendchar()**来做真实的工作。**sendchar()**除了输出一般的字符之外，还要在检测到”\n”时输出一个附加的CR，才能在超级中断上正常显示回车换行，否则将回到同一行的起始处，使先前输出的字符被新输出的字符覆盖掉。

在加入**retargeting**代码后，就可以重新编译了。

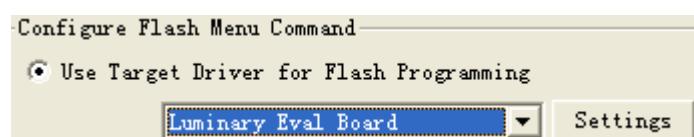
20.4 测试示例程序

如果有硬件设备，则可以把编译好的程序下载到Flash并且运行，步骤如下：

1. 配置flash下载选项：



2. 选择硬件平台



3. 接下来就可以下载程序到flash中了



4. 下载完后，程序将开始运行，在超级终端上应看到如下字样：

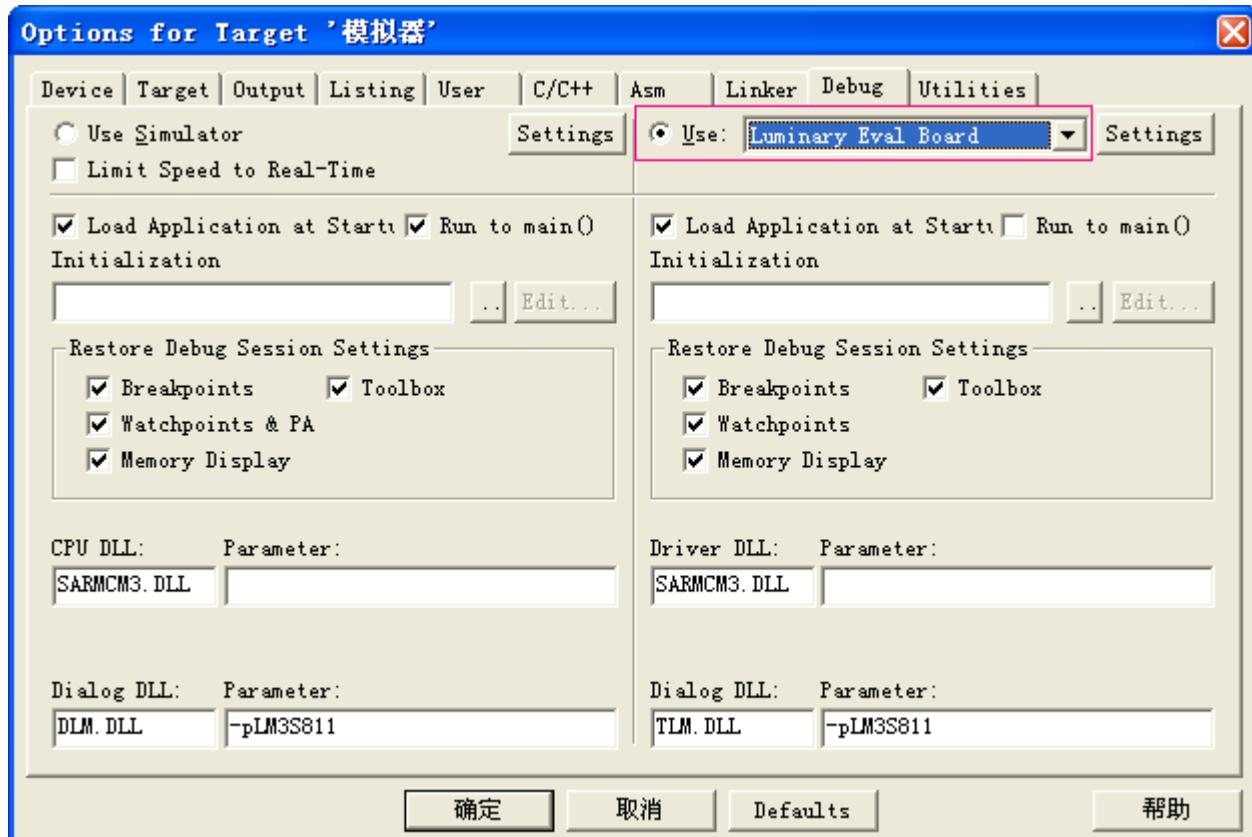


20.5 使用调试器

uVision中附带了调试器，这是一个可视化的源码级调试器。可以把它连接到目标板上（通过JTAG仿真器）。在设置时，单击“魔术棒”按钮，



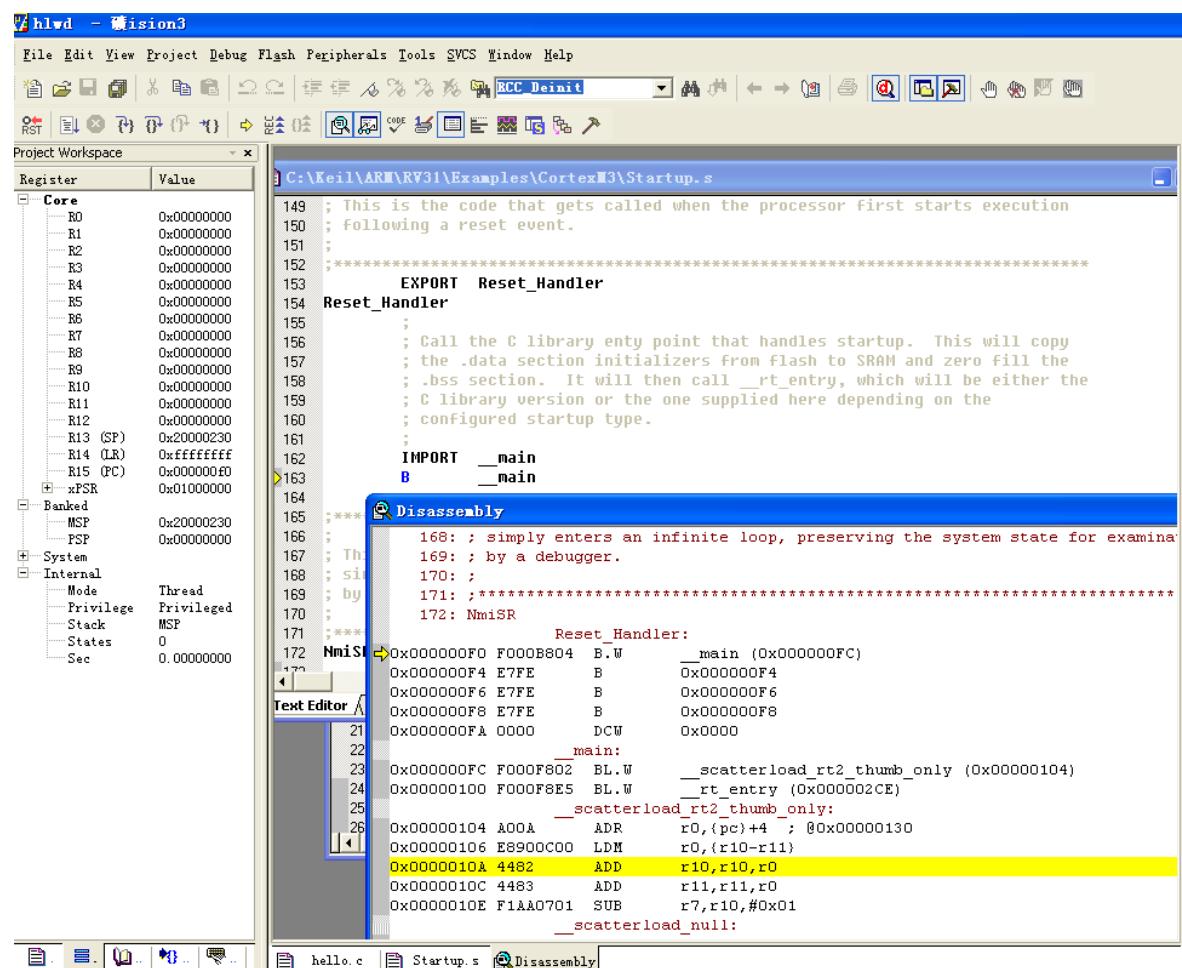
再在弹出的对话框中选择”Debug”选项卡，并且如下设置：



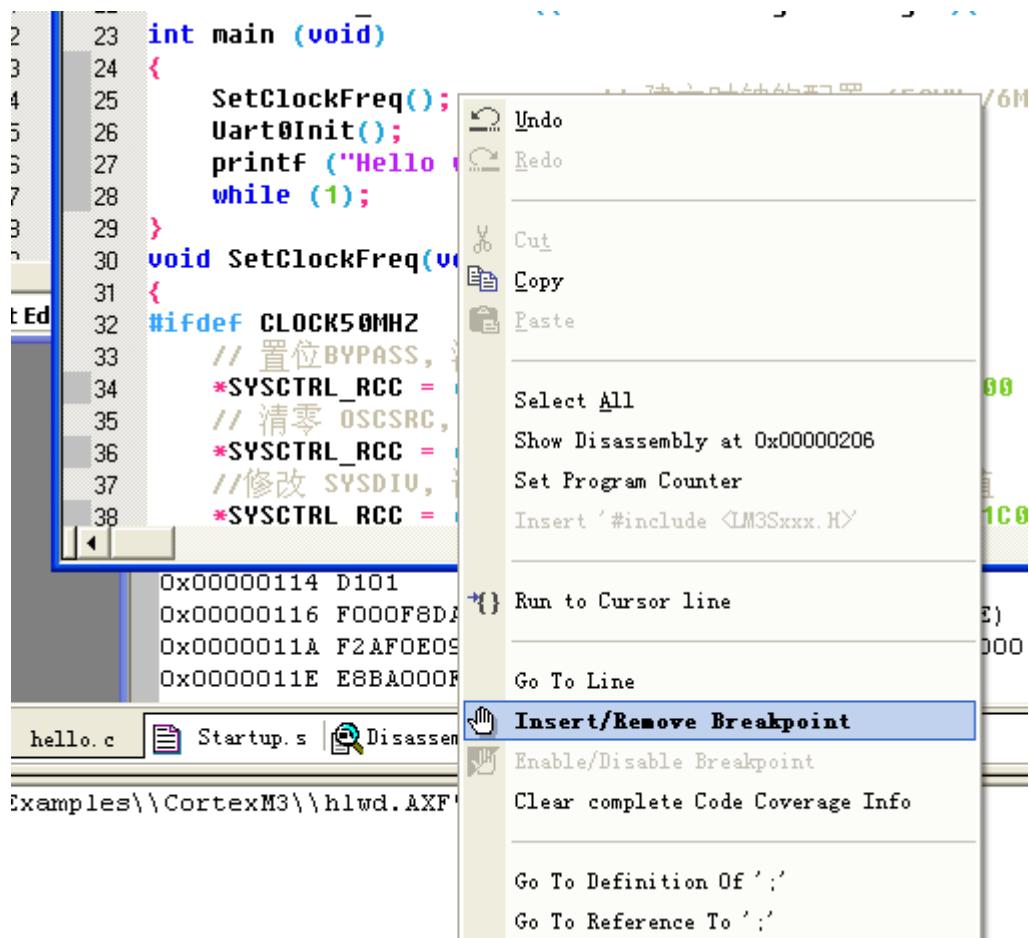
接下来就可以启动调试了。但要注意的是：如果板子已经在运行并且连上了超级终端，则需要关闭超级终端，断开USB电缆，并且在开始调试之前再重新连接。



当调试器开始工作后, IDE会提供一个寄存器视图, 显示当前寄存器的内容(通用寄存器和特殊功能寄存器都有)。还可以从源码级上看到程序当前的执行进度。从下图中我们可以看出刚开始执行时的情况——内核停止在 `Reset_Handler` 上:



为了测试, 我们可以在 `main()` 的入口处设置断点。通过右击代码并且在在菜单中执行“Insert/Remove Breakpoint”即可(也可以在调试选项中如下设置: Run to `main()`)。这样会在运行到 `main()` 时自动停止)。



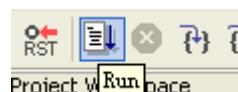
(也可以双击main()入口处“SetClockFreq()”前面的空白，或者是它的行号“25”来快速切换断点)。

```

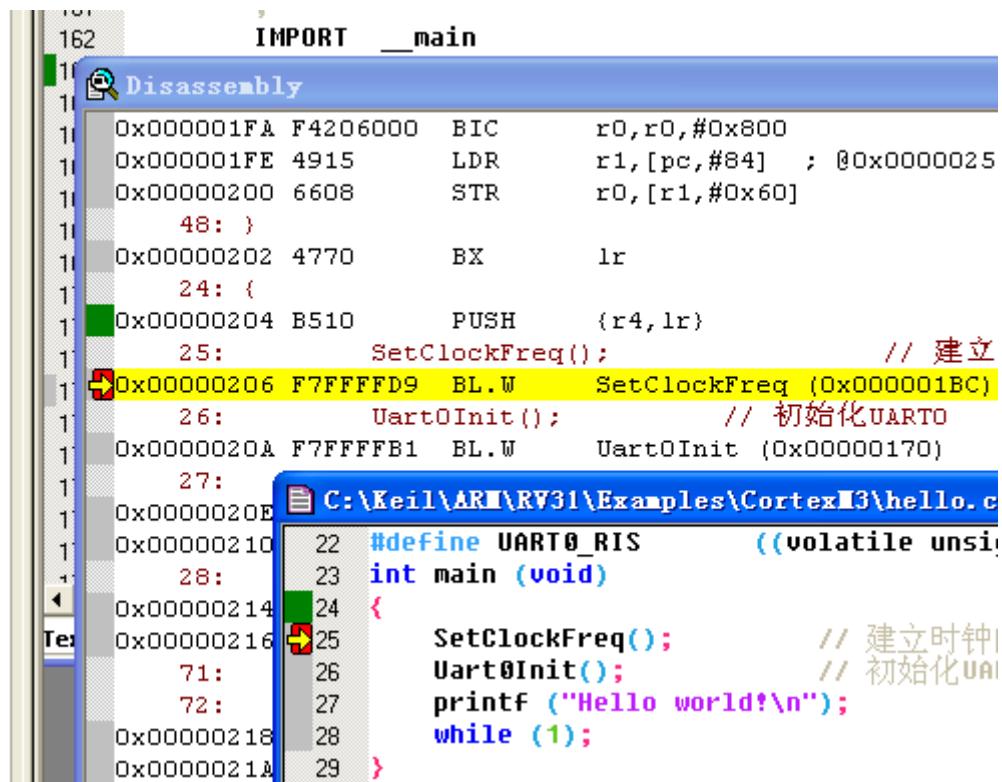
71  22 #define UART0_RIS
72  23 int main (void)
73  24 {
74  25 | SetClockFreq();
75  26 | Uart0Init();
76  27 | printf ("Hello world\r\n");

```

上图的红色实心矩形表示断点已经设好。接下来运行程序：



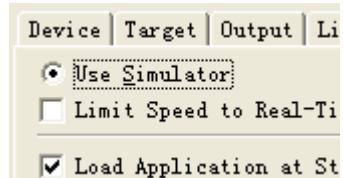
程序将停在断点处：



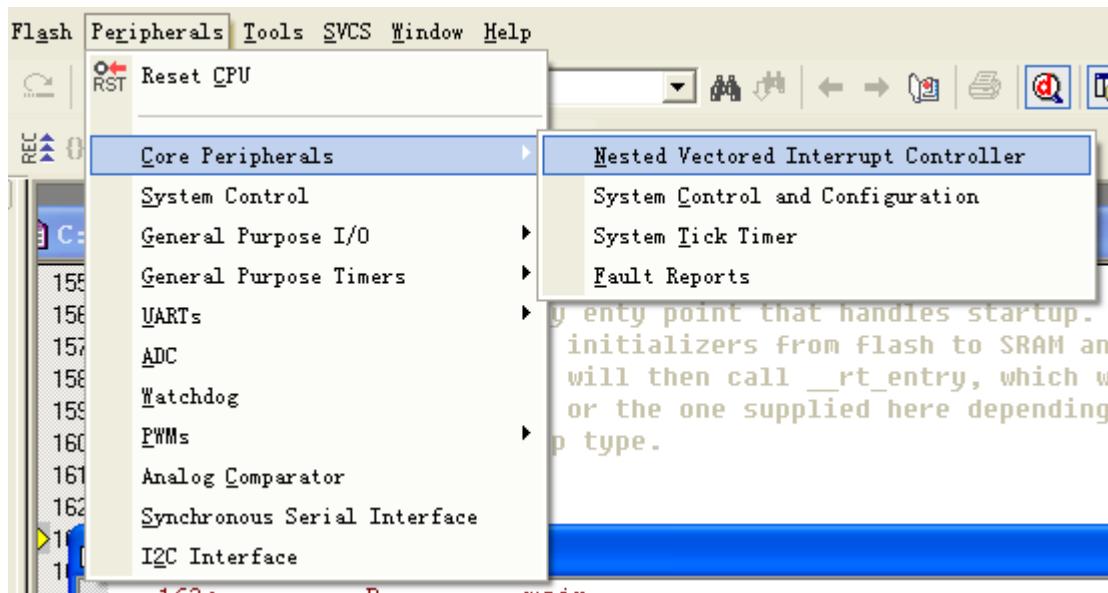
黄色小箭头表示停止后即将执行的语句。从图中还可以看到，反汇编窗口也对应地显示了这个断点所处的指令位置。译者使用的是模拟器，因此还能记录曾经执行过的语句。图中绿色小块就表示已经执行过的语句/指令。细心的读者可能会发现，“{”竟然也被“执行”了（因为是绿色小块）！事实上，“{”被编译后产生了汇编指令。看一下反汇编窗口——原来“{”处的机器码是一条“PUSH {r4,lr}”指令。

20.6 指令模拟器

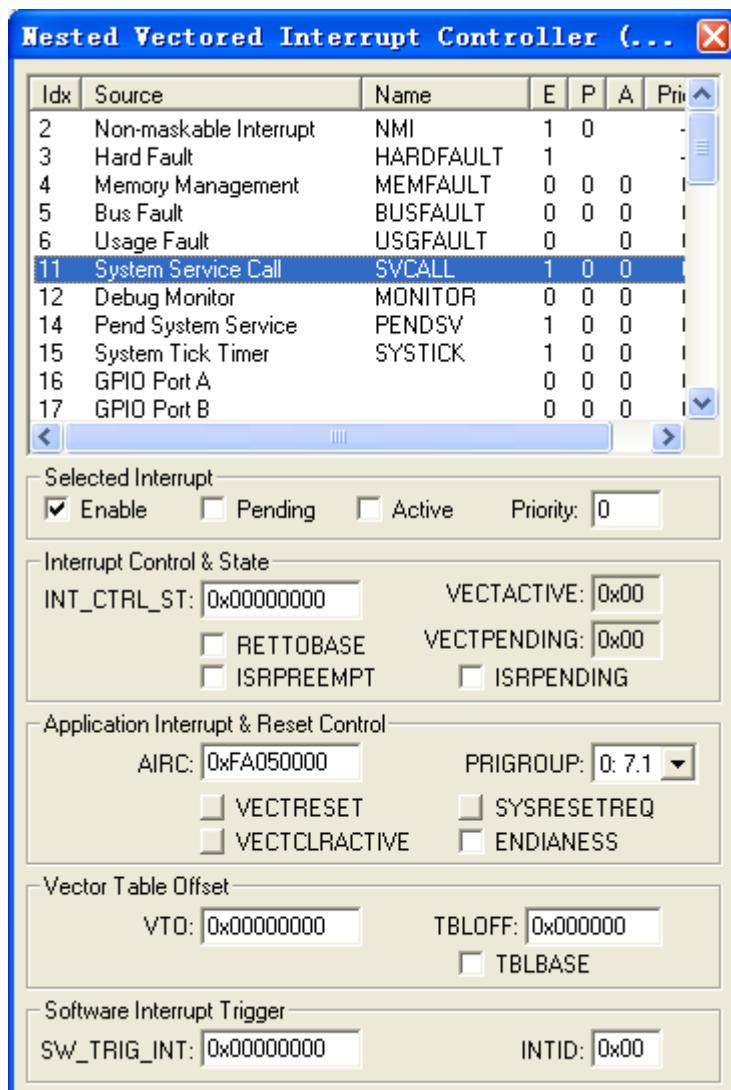
uVision IDE还附带了一个软件模拟器，它可以用来验证算法的各项性能指标，也是学习CM3的好帮手。若欲使用模拟器，只需在刚才的“Debug”选项里点中Simulator：



事实上，调试器还对片上外设包装了非常丰满和直观的可视化操作接口，模拟器也可以模拟它们。比如，欲查看NVIC的内部状态，可以通过如下菜单命令启动：

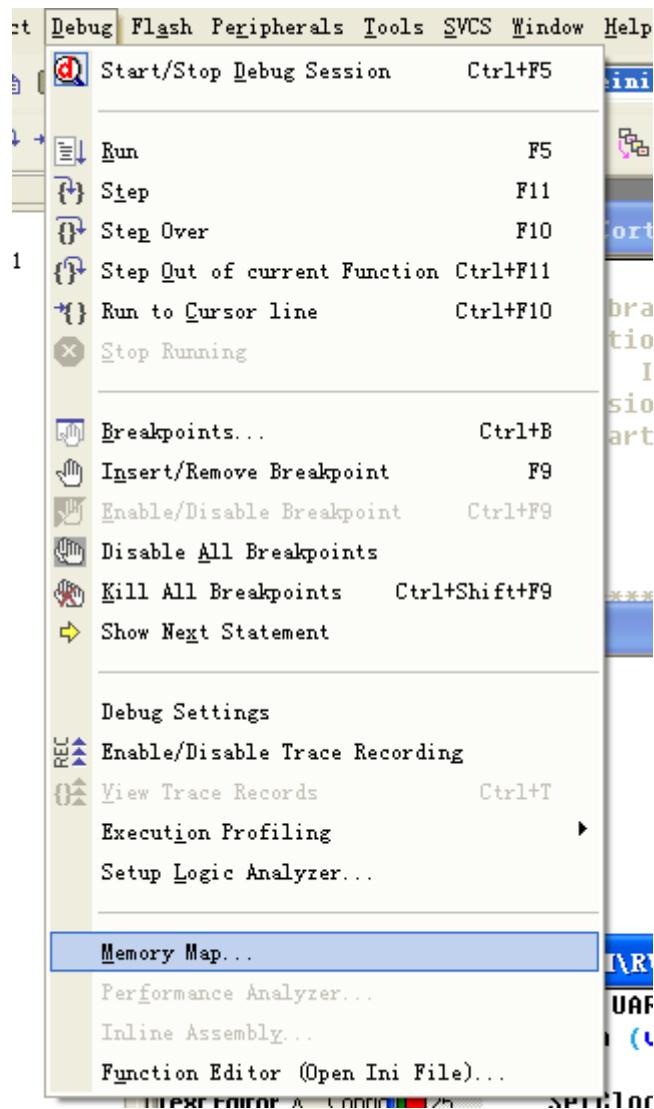


执行上图中的菜单命令后，将弹出如下对话框：

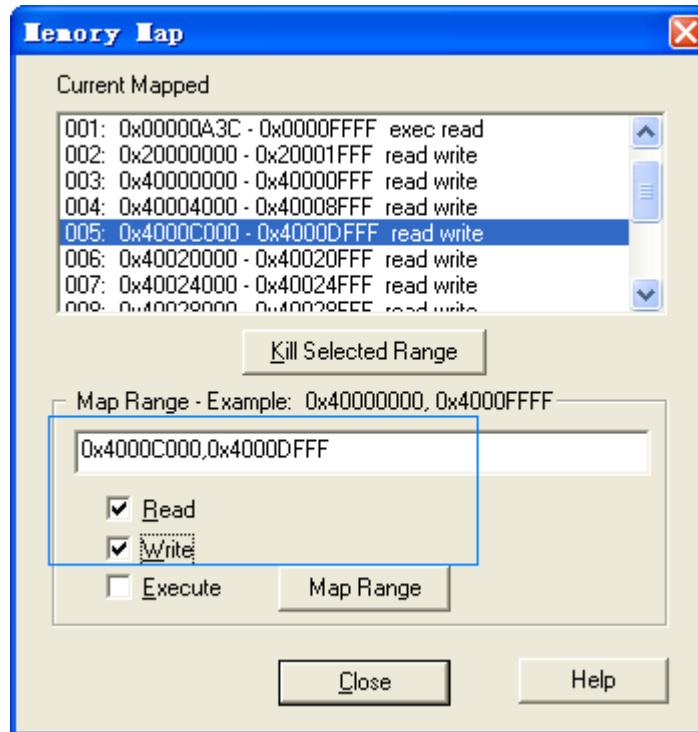


可以在它上面单击不同的异常来查看相关状态，上图中就查看了SVC异常的状态。

在使用模拟器时, 如果被模拟的器件比较新, 则有可能器件中某些地址范围模拟器没有来得及更新。这样在使用模拟器时, 就会被判定成访问落空的地址范围, 因此产生 **fault**。比如, 本例中, 为了保证 **UART** 寄存器的地址范围被支持, 可以使用如下的菜单命令检查:



执行后, 弹出如下对话框:



本例中UART寄存器的地址范围是0x4000_C000-0x4000_DFFF。译者使用的RVMDK版本较新，已经加入了本地址范围。但是原著作者使用的RVMDK版本还没有加入，需要手工添加，语法如上图被框住的部分所示。RVMDK的调试器会自动合并“碎片”，对于译者使用的RVMDK，它会发现手工添加的内容与原有的重合，故而不会有任何影响。

-----译者添加-----

uVision IDE的调试功能非常强大，很多没有想到的功能它都有，简列如下：

- 观察窗口，函数调用栈窗口，监视窗口等
- 计算到目前为止已经经历的周期数
- 计算每条指令被执行的频率（性能分析非常有用）
- 逻辑分析仪
- 源程序中的各种符号窗口（文件名，变量名等）
- 存储器映射窗口

限于篇幅，本书不能详细地讲述RVMDK的方方面面。但好消息是市面上新出了一本专门讲解RVMDK的书，书名为《ARM开发工具RealView MDK使用入门》，作者：李宁。北京航空航天大学出版社出版。

20.7 例程万里行

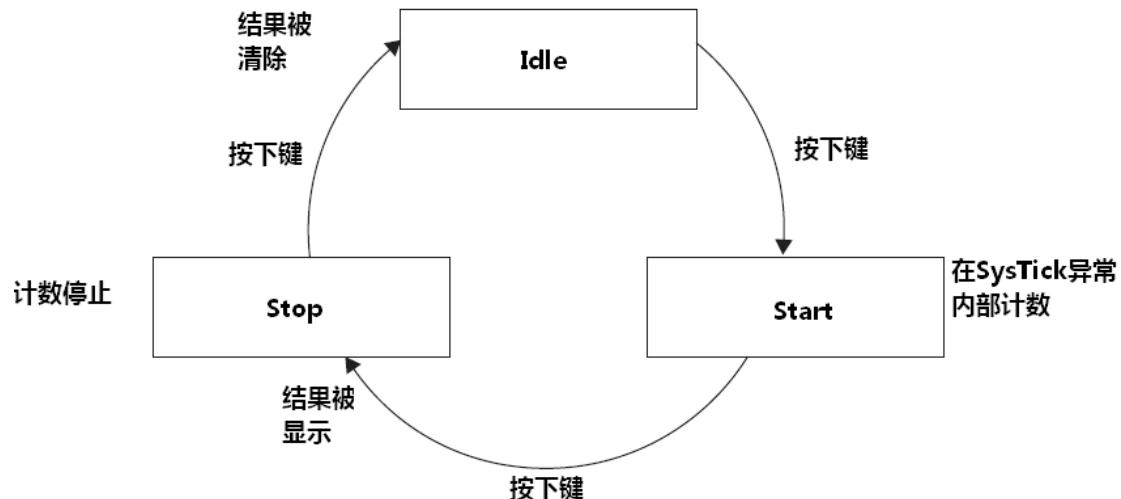
在上例中，向量表在Startup.s中已预先定义，这是由开发工具提供的向量表模板，它只包含了必备的MSP初始值、复位向量、NMI向量以及硬fault向量。但是我们往往还要响应其它中断，因此要添加其它的向量，或者需要把预先提供的向量也改成自己的。在这些场合下，就要更改Startup.s的代码，把向量表中对应的位置写成我们提供的服务例程名。

但问题又来了，ISR不是在Startup.s中实现，而是在其它文件实现的。那么怎么让startup.s汇编出的代码startup.o在连接时知晓ISR的地址呢？这时，就要使用IMPORT指示字。IMPORT后面跟随函数名或变量名，作用相当于C中的extern关键字，指出这些全局符号是在其它源文件中定义的。下一小节作为本书的谢幕，就提供了一个例子演示了IMPORT的使用。

20.8 使用中断实现的秒表示例程序

终于到了最后一节了，来看压轴好戏吧！

在这一小节中，将给出了一个最完整的示例——秒表程序示例。它使用了SysTick异常和UART0中断。秒表程序内部是以状态机的方式实现的，其状态转换图如下：



在上例的基础上，我们使用PC来通过UART以控制秒表程序的执行。为简化示例代码，我们使用固定的50MHz主频。时间测量上，由SysTick提供时基——它以100Hz的频率给出异常请求。本例中，SysTick以内核的50MHz时钟运行(FCLK)，每次响应SysTick中断时，如果秒表在走，则把计数器加1——自增TickCounter变量。

因为使用UART显示文字是个很耗费时间的工作，因此不再使用以前的查询方式，而转用中断来实现（这也是编程基本功），而对于秒表数值的格式化则在mani()中完成（线程模式下）。程序中的主状态机由UART服务例程启动状态转换——每收到一个字符转换一次。

使用上例的创建步骤，我们再创建一个名为 `stopwatch` 的工程。这次添加的代码是 `stopwatch.c`：

```
#include "stdio.h"

#define CR 0x0D // Carriage return
#define LF 0x0A // Linefeed

void Uart0Init(void);
void SysTickInit(void);
void SetClockFreq(void);
void DisplayTime(void);
void PrintValue(int value);
int sendchar(int ch);
int getkey(void);
void Uart0Handler(void);
void SysTickHandler(void);
// 寄存器地址
#define SYSCTRL_RCC      ((volatile unsigned long *) (0x400FE060))
#define SYSCTRL_RIS      ((volatile unsigned long *) (0x400FE050))
#define SYSCTRL_RCGC1    ((volatile unsigned long *) (0x400FE104))
```

```
#define SYSCTRL_RCGC2 ((volatile unsigned long *)(0x400FE108))
#define GPIOPA_AFSEL ((volatile unsigned long *)(0x40004420))
#define UART0_DATA ((volatile unsigned long *)(0x4000C000))
#define UART0_FLAG ((volatile unsigned long *) (0x4000C018))
#define UART0_IBRD ((volatile unsigned long *) (0x4000C024))
#define UART0_FBRD ((volatile unsigned long *) (0x4000C028))
#define UART0_LCRH ((volatile unsigned long *) (0x4000C02C))
#define UART0_CTRL ((volatile unsigned long *) (0x4000C030))
#define UART0_IM ((volatile unsigned long *) (0x4000C038))
#define UART0_RIS ((volatile unsigned long *) (0x4000C03C))
#define UART0_ICR ((volatile unsigned long *) (0x4000C044))
#define NVIC_IRQ_EN0 ((volatile unsigned long *) (0xE000E100))

// 全局变量
volatile int CurrState; // 状态机
volatile unsigned long TickCounter; // 秒表当前值
volatile int KeyReceived; // 指示用户按下了键
volatile int userinput; // 用户按下的键
#define IDLE_STATE 0 // 状态的定义
#define RUN_STATE 1
#define STOP_STATE 2

int main (void)
{
    int CurrStateLocal; // 局部变量
    // 初始化全局变量
    CurrState = 0;
    KeyReceived = 0;
    // 初始化硬件
    SetClockFreq(); // 设置时钟
    Uart0Init();
    SysTickInit();
    printf ("Stop Watch\n");
    while (1)
    {
        CurrStateLocal = CurrState; // 建立一个局部的副本
        // 因为SysTick ISR随时可能修改它的值
        switch (CurrStateLocal) {
        case (IDLE_STATE):
            printf ("\nPress any key to start\n");
            break;
        case (RUN_STATE):
            printf ("\nPress any key to stop\n");
            break;
        case (STOP_STATE):
            printf ("\nPress any key to clear\n");
            break;
        }
    }
}
```

```
        break;
    default:
        CurrState = IDLE_STATE;
        break;
    } // end of switch
    while (KeyReceived == 0)
    {
        if (CurrState==RUN_STATE)
        {
            DisplayTime();
        }
    } // 等待用户输入
    if (CurrStateLocal==STOP_STATE)
    {
        TickCounter=0;
        DisplayTime(); //显示, 以指示结果被清
    }
    else if (CurrStateLocal==RUN_STATE)
    {
        DisplayTime(); // 显示结果
    }
    if (KeyReceived!=0) KeyReceived=0;
}; // end of while loop
} // end of main

void SetClockFreq(void)
{
    // Set BYPASS, clear USRSYSDIV and SYSDIV
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF83FFFF) | 0x800 ;
    // Clr OSCSRC, PWRDN and OEN
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xFFFFFCFC);
    // 修改 SYSDIV, 设置 USRSYSDIV 和 Crystal 的值
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xF87FFC3F) | 0x01C002C0;
    // 等待直到PLLRIS置位
    while ((*SYSCTRL_RIS & 0x40)==0); // 等待直到PLLRIS 置位
    // 清除bypass
    *SYSCTRL_RCC = (*SYSCTRL_RCC & 0xFFFFF7FF) ;
    return;
}
// UART0 初始化
void Uart0Init(void)
{
    *SYSCTRL_RCGC1 = *SYSCTRL_RCGC1 | 0x0003; // 使能 UART0 & UART1 时钟
    *SYSCTRL_RCGC2 = *SYSCTRL_RCGC2 | 0x0001; // 使能 PORTA 时钟
```

```
*UART0_CTRL = 0; // 除能 UART
*UART0_IBRD = 27; // 基于50MHz编程波特率
*UART0_FBRD = 9;
*UART0_LCRH = 0x60; // 8 bit, 无奇偶
*UART0_CTRL = 0x301; // 使能 TX 和 RX, 并使能 UART
*UART0_IM = 0x10; // 使能 UART 接收中断
*GPIOPA_AFSEL = *GPIOPA_AFSEL | 0x3; // 让UART0控制GPIO管脚
*NVIC_IRQ_EN0 = (0x1<<5); // 在NVIC中使能UART中断
return;
}

// SYSTICK 初始化
void SysTickInit(void)
{
#define NVIC_STCSR ((volatile unsigned long *) (0xE000E010))
#define NVIC_RELOAD ((volatile unsigned long *) (0xE000E014))
#define NVIC_CURRVAL ((volatile unsigned long *) (0xE000E018))
#define NVIC_CALVAL ((volatile unsigned long *) (0xE000E01C))

    *NVIC_STCSR = 0; // 除能 SYSTICK
    *NVIC_RELOAD = 499999; // 基于50MHz主频的100Hz装载值
    *NVIC_CURRVAL = 0; // 清除当前值
    *NVIC_STCSR = 0x7; // 使能SYSTICK, 使能中断, 使用内核时钟
return;
}

// SYSTICK 异常服务例程
void SysTickHandler(void)
{
    if (CurrState==RUN_STATE) {
        TickCounter++;
    }
    return;
}

// UART0 RX 中断服务例程
void Uart0Handler(void)
{
    userinput = getkey();
    // 表示收到了按键请求
    KeyReceived++;
    // 释放UART请求
    *UART0_ICR = 0x10;
    // 状态机转换
    switch (CurrState)
    {
        case (IDLE_STATE):
            CurrState = RUN_STATE;
```

```
        break;
    case (RUN_STATE):
        CurrState = STOP_STATE;
        break;
    case (STOP_STATE):
        CurrState = IDLE_STATE;
        break;
    default:
        CurrState = IDLE_STATE;
        break;
    } // end of switch
    return;
}

// 显示时间值
void DisplayTime(void)
{
    unsigned long TickCounterCopy;
    unsigned long TmpValue;
    sendchar(CR);
    TickCounterCopy = TickCounter; // 建立一个局部的副本
    // 因为SysTick ISR随时可能修改它的值
    TmpValue = TickCounterCopy / 6000; // 分钟
    PrintValue(TmpValue);
    TickCounterCopy = TickCounterCopy - (TmpValue * 6000);
    TmpValue = TickCounterCopy / 100; // 秒
    sendchar(':');
    PrintValue(TmpValue);
    TmpValue = TickCounterCopy - (TmpValue * 100);
    sendchar(':');
    PrintValue(TmpValue); // 1/100秒
    sendchar(' ');
    sendchar(' ');
    return;
}

// 显示10进制数值
void PrintValue(int value)
{
    printf ("%d", value);
    return;
}

// 往UART0送出一个字符 (使用printf来输出数据)
int sendchar (int ch)
{
    if (ch == '\n')
```

```

{
    while ((*UART0_FLAG & 0x20)); // 如果TXFIFO满则等待
    *UART0_DATA = CR; // 输出附加的CR, 以在超级终端上得到正确的显示
}
while ((*UART0_FLAG & 0x20)); // 如果TXFIFO满则等待
return (*UART0_DATA = ch); // 输出数据
}

// 获取用户输入
int getkey (void)
{
    // 从串口读取字节
    while (*UART0_FLAG & 0x10); // 如果Rx FIFO空则等待
    return (*UART0_DATA);
}

// retarget输出
int fputc(int ch, FILE *f)
{
    return (sendchar(ch));
}

```

为使用中断, 本例中的UART初始化代码略有改动。在使用中断前, 既要设置UART中断掩蔽寄存器, 又要设置NVIC来打开对应的外中断。对于SysTick, 因为它是NVIC内建的, 每个CM3芯片都一样, 所以初始化代码也是通用的。

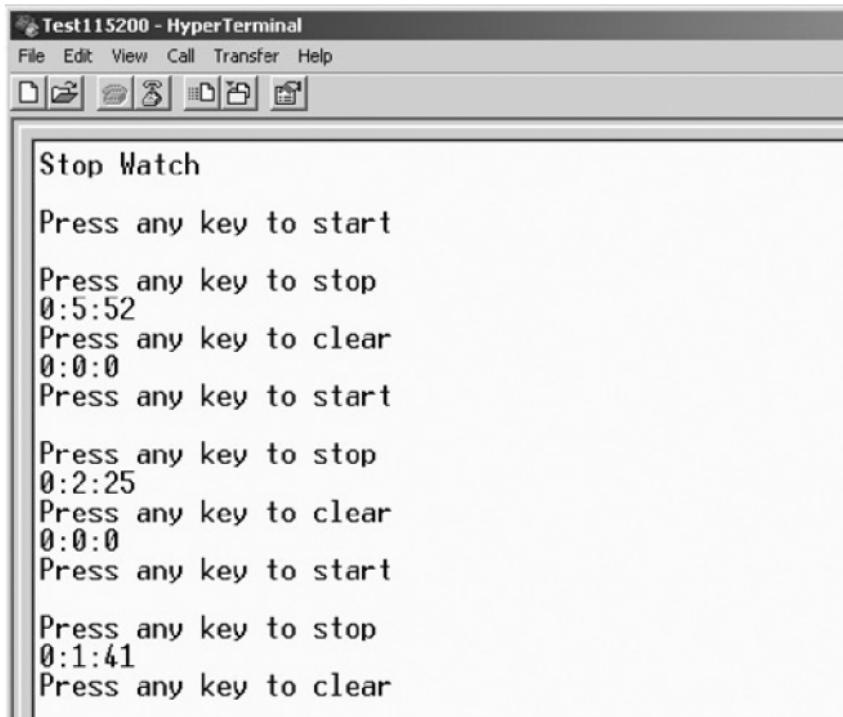
此外, 还添加了若干个函数, 包括UART和SysTick服务例程、显示函数、以及SysTick初始化的函数。根据外设的不同设计, 中断服务例程可能要手工清除中断标志位, 也可能由硬件清零。在本例中, 是通过UART0_ICR来手工清除的。

为了让startup.s能认出我们新添加的两个中断服务例程, 需要如下修改startup.s

099	DCD	0	; Reserved
100	DCD	IntDefaultHandler	; PendSV Handler
101	IMPORT	SysTickHandler	
102	DCD	SysTickHandler	; SysTick Handler
103	DCD	IntDefaultHandler	; GPIO Port A
104	DCD	IntDefaultHandler	; GPIO Port B
105	DCD	IntDefaultHandler	; GPIO Port C
106	DCD	IntDefaultHandler	; GPIO Port D
107	DCD	IntDefaultHandler	; GPIO Port E
108	IMPORT	Uart0Handler	
109	DCD	Uart0Handler	; UART0
110	DCD	IntDefaultHandler	; UART1

注意IMPORT指示字的使用。它们后面跟着的函数名是由其它源文件实现的函数。有了它, 汇编器就知道了这个情况, 从而相应地处理。

本程序的一次运行情况(亦称为一个实例)如下图所示:



The screenshot shows a HyperTerminal window titled "Test115200 - HyperTerminal". The window contains a text-based application for a stopwatch. The application starts with "Stop Watch" and prompts the user to press any key to start. It then displays the current time as 0:5:52, followed by a prompt to press any key to clear. The process repeats, showing times of 0:2:25 and 0:1:41, each followed by a clear prompt. The application ends with a final clear prompt.

```
Stop Watch
Press any key to start
Press any key to stop
0:5:52
Press any key to clear
0:0:0
Press any key to start
Press any key to stop
0:2:25
Press any key to clear
0:0:0
Press any key to start
Press any key to stop
0:1:41
Press any key to clear
```

注意：如果使用了虚拟的COM口，则有可能无法正常使用（因为此时按键无法送至目标板），这是虚拟COM口驱动程序的一个bug导致的。如果遇到这种情况，可能要在另一台没有安装过RVMDK的PC上测试这个示例程序。

以下内容由译者添加，对学习第4章很有用。

也可以使用和添加C源文件相同的方式，来添加汇编源文件。只不过汇编源文件的扩展名是.s。下面给出一个汇编源文件的示例：

```
E:\My Documents\cexprac\Ch3\asmtest.s
001 AREA AsmTest, CODE, READONLY
002 THUMB
003 PRESERVE8
004 export MovMovTTest
005 MovMovTTest
006 push {lr}
007 ;错误的顺序
008 movt r14, #0x5678
009 mov r14, #0x1234
010 ;正确的顺序
011 mov r14, #0x1234
012 movt r14, #0x5678
013 pop {pc}
014
015 end
```

这里练习了mov和movt指令（还刻意演示了push/pop），为在C程序中调用”MovMovTTest”，需要先在某个.h或在使用该函数的.c文件中长明该函数：

```
void MovMovTTest(void);
```

如果写一个接受参数的函数，方法类似，但是要使用ARM的调用标准，如：有下面的汇编函数：

Add3

```
add r0, r0, r1
add r0, r0, r2
bx lr
```

则对应的C声明为：

```
int Add3(int a, int b, int c); //计算a+b+c
```

这种例子虽然看起来很低等，但是只是为了抛砖引玉。读者可以用上面例子所演示的骨架，去练习第4章的各种指令；也可以试着把本书中的汇编子程序包装成可以由C调用的函数。

附录 A

Cortex-M3 指令小结

此附录实际上是从Cortex-M3技术参考手册中译版摘抄并改编的。并且在可能的情况下，使用类C语言的风格来讲解指令的功能。另外要解释的是

U8表示unsigned char，无符号16位整数

U16表示unsigned short，无符号16位整数

S8表示signed char，带符号8位整数

S16表示signed short，带符号16位整数

缺省情况下，如果使用普通的char和short，都是指带符号整数

当借C语言的数组表示法，如Rn[Rm]时，是按整数运算的方式求得Rn+Rm的值，然后把该值当作一

个32位地址，再取出该地址的值。**在计算地址时，并不乘以“数据类型所占用的字节数”，这与C语言的数组/指针运算是概念上的不同，切记切记！**

简单地概括，这里的Rn[Rm]等效于

`*((U32 *) (Rn+Rm))`，其中Rn, Rm均为32位整数类型

还有两条重要的通用规则：

- 凡是在指令中有可选的预移位操作的，预移位后的值是中间结果，不写回被移位的寄存器
- 凡是在{s}的指令中使用了s后缀的，都按照运算结果更新APSР中的标志位。

表 1-1 16位 Cortex-M3 指令汇总

操作	汇编指令
Rd+= Rm+C	ADC <Rd>, <Rm>
Rd= Rn+Imm3	ADD <Rd>, <Rn>, #<immed_3>
Rd+= Imm8	ADD <Rd>, #<immed_8>
Rd=Rn+Rm	ADD <Rd>, <Rn>, <Rm>
Rd+=Rm	ADD <Rd>, <Rm>
Rd=PC+Imm8*4	ADD <Rd>, PC, #<immed_8>*4
Rd=SP+Imm8*4	ADD <Rd>, SP, #<immed_8>*4
Rd=SP+Imm7*4 或 SP+=Imm7*4	ADD <Rd>, SP, #<immed_7>*4 或 ADD SP, SP, #<immed_7>*4
Rd &= Rm	AND <Rd>, <Rm>
Rd = Rm 算术右移 Imm5	ASR <Rd>, <Rm>, #<immed_5>
Rd 算术右移= Rs	ASR <Rd>, <Rs>

操作	汇编指令
按<contd>条件决定是否分支	B<cond> <target address>
无条件分支	B<target address>
Rd &= ~Rs	BIC <Rd>, <Rs>
软件断点	BKPT <immed_8>
带链接分支	BL <Rm>
比较结果不为零时分支	CBNZ <Rn>, <label>
比较结果为零时分支	CBZ <Rn>, <Rm>
将 Rm 取二进制补码后再与 Rn 比较 (注意: 不是取反!!!)	CMN <Rn>, <Rm>
Rn 与 8 位立即数比较, 并根据结果更新标志位的值	CMP <Rn>, #<immed_8>
Rn 与 Rm 比较, 并根据结果更新标志位的值	CMP <Rn>, <Rm>
高寄存器与高或低寄存器比较, 并根据结果更新标志位的值。在实际编程时, 可以无视这两条指令的区别, 当作一条指令来用。	CMP <Rn>, <Rm>
改变处理器状态	CPS <effect>, <iflags>
将高或低寄存器的值复制到另一个高或低寄存器中	CPY <Rd>, <Rm>
Rd^=Rm	EOR <Rd>, <Rm>
以下一条指令为条件; 以下面两条指令为条件; 以下面三条指令为条件; 以下面四条指令为条件	IT<cond> IT<x> <cond> IT<x><y> <cond> IT<x><y><z> <cond>
多个连续的存储器字加载	LDMIA <Rn>!, <register>
将基址寄存器与 5 位立即数偏移的和的地址处的数据加载到寄存器中 Rd= Rn[Imm5*4]	LDR <Rd>, [<Rn>, #<immed_5*4>]
Rd= Rn[Rm]	LDR <Rd>, [<Rn>, <Rm>]
Rd= PC[Imm8*4+4]	LDR <Rd>, [PC, #<immed_8>*4]
Rd= SP[Imm8*4]	LDR <Rd>, [SP, #<immed_8>*4]
Rd= (U8) Rn[Imm5]	LDRB <Rd>, [<Rn>, #<immed_5>]
Rd= (U8) Rn[Rm]	LDRB <Rd>, [<Rn>, <Rm>]
Rd= (U16) Rn[Imm5*2]	LDRH <Rd>, [<Rn>, #<immed_5>*2]
Rd= (U16) Rn[Rm]	LDRH <Rd>, [<Rn>, <Rm>]
加载 Rn+Rm 的地址处的字节, 并带符号扩展到 Rd 中	LDRSB <Rd>, [<Rn>, <Rm>]
加载 Rn+Rm 的地址处的半字, 并带符号扩展到 Rd 中	LDRSH <Rd>, [<Rn>, <Rm>]
Rd= Rm<<Imm5	LSL <Rd>, <Rm>, #<immed_5>
Rd<<= Rs	LSL <Rd>, <Rs>
Rd= Rm>>Imm5	LSR <Rd>, <Rm>, #<immed_5>
Rd>>= Rs	LSR <Rd>, <Rs>
Rd= (U32) Imm8	MOV <Rd>, #<immed_8>

操作	汇编指令
Rd=Rn	MOV <Rd>, <Rn>
Rd=Rm。实际使用时, 可把这两条 MOV 指令当成一条指令来用——译者注	MOV <Rd>, <Rm>
Rd*=Rm	MUL <Rd>, <Rm>
Rd= ~Rm (注意, 是取反, 不是取补码!!!)	MVN <Rd>, <Rm>
Rd= ~Rm + 1	NEG <Rd>, <Rm>
无操作	NOP <C>
Rd =Rm	ORR <Rd>, <Rm>
寄存器出栈	POP <寄存器>
若干寄存器和 PC 出栈	POP <寄存器, PC>
若干寄存器压栈	PUSH <registers>
若干寄存器和 LR 压栈	PUSH <registers, LR>
Rd=Rn 字内的字节反转	REV <Rd>, <Rn>
Rd=Rn 两个半字内的字节反转	REV16 <Rd>, <Rn>
将 Rn 低半字内的字节反转, 再把反转后的值带符号位扩展到 32 位后, 复制到 Rd 中	REVSH <Rd>, <Rn>
Rd 圆圈右移=Rs	ROR <Rd>, <Rs>
Rd= Rm+C	SBC <Rd>, <Rm>
发送事件	SEV <c>
将多个寄存器字保存到连续的存储单元中, 首地址由 Rn 给出。每保存完一个 Rn+4	STMIA <Rn>!, <registers>
Rn[Imm5*4]=Rd	STR <Rd>, [<Rn>, #<immed_5>*4]
Rn[Rm]=Rd	STR <Rd>, [<Rn>, <Rm>]
SP[Imm8*4]=Rd	STR <Rd>, [SP, #<immed_8> * 4]
((U8) (Rn+Imm5)) = (U8) Rd	STRB <Rd>, [<Rn>, #<immed_5>]
((U8) (Rn+Rm)) = (U8) Rd	STRB <Rd>, [<Rn>, <Rm>]
((U16) (Rn+Imm5*2)) = (U16) Rd	STRH <Rd>, [<Rn>, #<immed_5> * 2]
((U16) (Rn+Rm)) = (U16) Rd	STRH <Rd>, [<Rn>, <Rm>]
Rd= Imm8	SUB <Rd>, #<immed_8>
Rd= Rn-Rm	SUB <Rd>, <Rn>, <Rm>
SP= Imm7*4	SUB SP, #<immed_7> * 4
操作系统服务调用, 带 8 位立即数调用代码	SVC <immed_8>
从寄存器中提取字节[7:0], 传送到寄存器中, 并用符号位扩展到 32 位	SXTB <Rd>, <Rm>
从寄存器中提取半字[15:0], 传送到寄存器中, 并用符号位扩展到 32 位	SXTH <Rd>, <Rm>
执行 Rn & Rm, 并根据结果更新标志位	TST <Rn>, <Rm>
从寄存器中提取字节[7:0], 传送到寄存器中, 并用零位扩展到 32 位	UXTB <Rd>, <Rm>
Rd= (U8) Rm	

操作	汇编指令
从寄存器中提取半字[15:0], 传送到寄存器中, 并用零位扩展到32位 Rd= (U16) Rm	UXTH <Rd>, <Rm>
等待事件	WFE <c>
等待中断	WFI <c>

表 1-2列出了 32位 Coortex-M3指令。表 1-2 32位 Coortex-M3指令汇总

操作	汇编指令
Rd=Rn+Imm12+C。有S就按结果更新标志位。S的作用下同。	ADC{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>
Rd= Rn与移位后的Rm及C位相加	ADC{S}.W <Rd>, <Rn>, <Rm>{, <shift>}
Rd= Rn+Imm12	ADD{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>
Rd=Rd与移位后的Rm相加	ADD{S}.W <Rd>, <Rm>{, <shift>}
Rd= Rn+Imm12	ADDW.W <Rd>, <Rn>, #<immed_12>
Rd= Rn & Imm12	AND{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>
Rd=Rn与移位后的Rm按位与	AND{S}.W <Rd>, <Rn>, Rm{, <shift>}
Rd = Rn>>Rm。有S就按结果更新标志位	ASR{S}.W <Rd>, <Rn>, <Rm>
条件分支	B{cond}.W <label>
位区清零	BFC.W <Rd>, #<lsb>, #<width>
将一个寄存器的位区插入另一个寄存器中	BFI.W <Rd>, <Rn>, #<lsb>, #<width>
Rd= Rn & ~Imm12	BIC{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>
Rd&= 移位后的Rn取反	BIC{S}.W <Rd>, <Rn>, {, <shift>}
带链接的分支	BL <label>
带链接的分支 (立即数)	BL<c> <label>
无条件分支	B.W <label>
Rd=Rn中前导零的数目	CLZ.W <Rd>, <Rn>
Rn与12位立即数取补后的值比较	CMN.W <Rn>, #<modify_constant(immed_12)>
Rn与移位后的Rm取补后的值比较	CMN.W <Rn>, <Rm>{, <shift>}
Rn与12位立即数比较	CMP.W <Rn>, #<modify_constant(immed_12)>
Rn与按需移位后的Rm比较 Rm的值不变	CMP.W <Rn>, <Rm>{, <shift>}
数据存储器隔离	DMB <c>
数据同步隔离	DSB <c>
Rd= Rn ^ Imm12	EOR{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>
Rd=Rn与按需移位后的Rm作异或操作 Rm的值不变	EOR{S}.W <Rd>, <Rn>, <Rm>{, <shift>}
指令同步排序 (barrier)	ISB <c>
多存储器寄存器加载, 加载后加4或加载前减4	LDM{IA DB}.W <Rn>{!}, <registers>
Rxf= Rn[ofs12]	LDR.W <Rxf>, [<Rn>, #<offset_12>]
PC= Rn[ofs12]	LDR.W PC, [<Rn>, #<offset_12>]

操作	汇编指令
无此指令	LDR.W PC, #<+/-<offset_8>
Rxf= *Rn; Rn+= ofs8;	LDR.W <Rxf>, [<Rn>], #+/-<offset_8>
Rn+= ofs8; Rxf= *Rn	LDR.W <Rxf>, [<Rn> , #<+/-<offset_8>]!
PC= Rn[ofs8]; Rn+= ofs8	LDR.W PC, [<Rn> , #+/-<offset_8>]!
Rxf=Rn[按需左移后的Rm] 左移只能是0,1,2,3	LDR.W <Rxf>, [<Rn> , <Rm>{, LSL #<shift>}]
PC=Rn[按需左移后的Rm] 左移只能是0,1,2,3	LDR.W PC, [<Rn> , <Rm>{, LSL #<shift>}]
Rxf= PC[ofs12]	LDR.W <Rxf>, [PC, #+/-<offset_12>]
PC= PC[ofs12]	LDR.W PC, [PC, #+/-<offset_12>]
Rxf=(U8) Rn[ofs12]	LDRB.W <Rxf>, [<Rn> , #<offset_12>]
Rxf= (U8) *Rn; Rn+= ofs8	LDRB.W <Rxf>, [<Rn> , #+/-<offset_8>]
Rxf= (U8) Rn[左移后的Rm]; 左移只能是0,1,2,3	LDRB.W <Rxf>, [<Rn> , <Rm>{, LSL #<shift>}]
Rxf= Rn[ofs8]; Rn+= ofs8	LDRB.W <Rxf>, [<Rn> , #<+/-<offset_8>]!
Rxf= PC[ofs12]	LDRB.W <Rxf>, [PC, #+/-<offset_12>]
读取Rn地址加上8位偏移量乘以4的处的双字 到Rxf(低32位), Rxf2(高32位), 前索引。 并且可选在加载后更新Rn	LDRD.W <Rxf>, <Rxf2>, [<Rn> , #+/-<offset_8> * 4]{!}
读取Rn处的双字到Rxf(低32位), Rxf2(高 32位), 并且在加载后Rn+= ofs8*4	LDRD.W <Rxf>, <Rxf2>, [<Rn>], #+/-<offset_8> * 4
Rxf= (U16) Rn[ofs12]	LDRH.W <Rxf>, [<Rn> , #<offset_12>]
Rxf= (U16) Rn[ofs8]; Rn+=ofs8;	LDRH.W <Rxf>, [<Rn> , #<+/-<offset_8>]!
Rxf= (U16) *Rn; Rn+= ofs8;	LDRH.W <Rxf>, [<Rn> , #+/-<offset_8>]
Rxf= (U16) Rn[左移后的Rm]; 左移只能是0,1,2,3	LDRH.W <Rxf>, [<Rn> , <Rm>{, LSL #<shift>}]
Rxf= (U16) PC[ofs12]	LDRH.W <Rxf>, [PC, #+/-<offset_12>]

操作	汇编指令
加载Rn+ofs12地址处的字节，并带符号扩展到Rxf中	LDRSB.W <Rxf>, [<Rn>, #<offset_12>]
加载Rn地址处的字节，并带符号扩展到Rxf中。然后Rn+=ofs8	LDRSB.W <Rxf>. [<Rn>], #+/-<offset_8>
先做Rn+=ofs8，再加载新Rn地址处的字节，并带符号扩展到Rxf中。	LDRSB.W <Rxf>, [<Rn>, #<+/-<offset_8>>]!
先把Rm按要求左移0,1,2,3位，再加载Rn+新Rm地址处的字节，并带符号扩展到Rxf中	LDRSB.W <Rxf>, [<Rn>, <Rm>{, LSL #<shift>}]
加载PC+ofs12地址处的字节，并带符号扩展到Rxf中	LDRSB.W <Rxf>, [PC, #+/-<offset_12>]
加载Rn+ofs12地址处的半字，并带符号扩展到Rxf中	LDRSH.W <Rxf>, [<Rn>, #<offset_12>]
加载Rn地址处的半字，并带符号扩展到Rxf中。然后Rn+=ofs8	LDRSH.W <Rxf>. [<Rn>], #+/-<offset_8>
先做Rn+=ofs8，再加载新Rn地址处的半字，并带符号扩展到Rxf中。	LDRSH.W <Rxf>, [<Rn>, #<+/-<offset_8>>]!
先把Rm按要求左移0,1,2,3位，再加载Rn+新Rm地址处的半字，并带符号扩展到Rxf中	LDRSH.W <Rxf>, [<Rn>, <Rm>{, LSL #<shift>}]
加载PC+ofs12地址处的半字，并带符号扩展到Rxf中	LDRSH.W <Rxf>, [PC, #+/-<offset_12>]
Rd= Rn<<Rm	LSL{S}.W <Rd>, <Rn>, <Rm>
Rd= Rn>>Rm	LSR{S}.W <Rd>, <Rn>, <Rm>
Rd= Racc+Rn*Rm	MLA.W <Rd>, <Rn>, <Rm>, <Racc>
Rd=Racc-Rn*Rm	MLS.W <Rd>, <Rn>, <Rm>, <Racc>
Rd= Imm12	MOV{S}.W <Rd>, #<modify_constant(immed_12)>
先按需移位Rm，然后Rd=新Rm	MOV{S}.W <Rd>, <Rm>{, <shift>}
将16位立即数传送到Rd的高半字中，Rd的低半字不受影响	MOVT.W <Rd>, #<immed_16>
将16位立即数传送到Rd的低半字中，并把高半字清零	MOVW.W <Rd>, #<immed_16>
把特殊功能寄存器的值传送到Rd中	MRS<c> <Rd>, <psr>
把Rn的值传送到特殊功能寄存器中	MSR<c> <psr>_<fields>, <Rn>
Rd= Rn*Rm	MUL.W <Rd>, <Rn>, <Rm>
无操作	NOP.W
Rd= Rn ~Imm12	ORN{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>
先按需要移位Rm，然后 Rd= Rn ~新Rm	ORN{S}.W <Rd>, <Rn>, <Rm>{, <shift>}
Rd= Rn Imm12	ORR{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)

操作	汇编指令
先按需要移位Rm, 然后 Rd= Rn 新Rm	ORR{S}.W <Rd>, <Rn>, <Rm>{, <shift>}
Rd=把Rm的位反转后的值	RBIT.W <Rd>, <Rm>
Rd=Rm字内的字节逆向	REV.W <Rd>, <Rm>
Rd=Rn每个半字内的字节逆向	REV16.W <Rd>, <Rn>
Rd=Rn低半字内的字节逆向后再符号扩展	REVSH.W <Rd>, <Rn>
Rd= Rn圆圈右移Rm	ROR{S}.W <Rd>, <Rn>, <Rm>
Rd= Imm12-Rd	RSB{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>
先按需移位Rm, 然后 Rd= 新Rm-Rn	RSB{S}.W <Rd>, <Rn>, <Rm>{, <shift>}
Rd= Imm12-Rn-C	SBC{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>
先按需移位Rm, 然后 Rd=Rn-新Rm-C	SBC{S}.W <Rd>, <Rn>, <Rm>{, <shift>}
抽取Rn中以1sb号位为最低有效位, 共width宽度的位段, 并带符号扩展到Rd中	SBFX.W <Rd>, <Rn>, #<1sb>, #<width>
带符号除法, Rd= Rn/Rm	SDIV<c> <Rd>, <Rn>, <Rm>
发送事件	SEV<c>
带符号64位乘加, RdHi:RdLo+= Rn*Rm	SMLAL.W <RdLo>, <RdHi>, <Rn>, <Rm>
带符号64位乘法, RdHi:RdLo= Rn*Rm	SMULL.W <RdLo>, <RdHi>, <Rn>, <Rm>
先按需移位Rn, 再把Rn向低Imm位执行带符号饱和操作, 并把结果带符号扩展后写到Rd	SSAT <c> <Rd>, #<imm>, <Rn>{, <shift>}
多个寄存器字连续保存到由Rn给出的首地址中, 并且在Rn上, 每存储一个后自增(IA)/每存储一个前自减(DB)	STM{IA DB}.W <Rn>{!}, <registers>
Rn[ofs12]=Rxf	STR.W <Rxf>, [<Rn>, #<offset_12>]
*Rn=Rxf; Rn+=ofs8	STR.W <Rxf>, [<Rn>], #+/-<offset_8>
先按需左移Rm, 然后 Rn[新Rm]=Rxf, 左移格数只能是0,1,2,3	STR.W <Rxf>, [<Rn>, <Rm>{, LSL #<shift>}]
Rn[ofs8]=Rxf 若有“!”, 则还执行Rn+=ofs8	STR{T}.W <Rxf>, [<Rn>, #+/-<offset_8>]{!}
((U8) (Rn+ofs8)) = (U8) Rxf 若有“!”, 则还执行Rn+=ofs8	STRB{T}.W <Rxf>, [<Rn>, #+/-<offset_8>]{!}
((U8) (Rn+ofs12)) = (U8) Rxf	STRB.W <Rxf>, [<Rn>, #<offset_12>]
((U8) Rn) = (U8) Rxf Rn+=ofs8	STRB.W <Rxf>, [<Rn>], #+/-<offset_8>
先按需左移Rm, 左移格数只能是0,1,2,3,再 *((U8*) (Rn+新Rm)) = (U8) Rxf	STRB.W <Rxf>, [<Rn>, <Rm>{, LSL #<shift>}]
*(Rn+ofs8*4)=Rxf; *(Rn+ofs8*4+4)=Rxf2 若有“!”, 则Rn+=ofs8	STRD.W <Rxf>, <Rxf2>, [<Rn>, #+/-<offset_8>*4]{!}
*Rn=Rxf; *(Rn 4)=Rxf2; Rn+=ofs8*4	STRD.W <Rxf>, <Rxf2>, [<Rn>], #+/-<offset_8> * 4
((U16) (Rn+ofs12)) = (U16) Rxf	STRH.W <Rxf>, [<Rn>, #<offset_12>]
先按需左移Rm, 左移格数只能是0,1,2,3,再 *((U16*) (Rn+新Rm)) = (U16) Rxf	STRH.W <Rxf>, [<Rn>, <Rm>{, LSL #<shift>}]

操作	汇编指令
$*(\text{U16}^*) \text{ (Rn+ofs8)} = (\text{U16}) \text{ Rx}f$ 若有“! ”，则还要执行 $\text{Rn}+=\text{ofs8}$	<code>STRH{T}.W <Rx_f>, [<Rn>, #+/-<offset_8>]{!}</code>
$*(\text{U16}^*) \text{ Rn} = (\text{U16}) \text{ Rx}f$ $\text{Rn}+=\text{ofs8}$	<code>STRH.W <Rx_f>, [<Rn>], #+/-<offset_8></code>
$\text{Rd} = \text{Rn-Imm12}$	<code>SUB{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)></code>
先按需移位 Rm $\text{Rd} = \text{Rn-新Rm}$	<code>SUB{S}.W <Rd>, <Rn>, <Rm>{, <shift>}</code>
$\text{Rd} = \text{Rn-Imm12}$	<code>SUBW.W <Rd>, <Rn>, #<immed_12></code>
先按需圆圈移位 Rm ，然后取出 Rm 的低8位，带符号扩展到32位并存储到 Rd	<code>SXTB.W <Rd>, <Rm>{, <rotation>}</code>
先按需圆圈移位 Rm ，然后取出 Rm 的低16位，带符号扩展到32位并存储到 Rd	<code>SXTH.W <Rd>, <Rm>{, <rotation>}</code>
$\text{PC}+= (\text{U8}^*(\text{Rn+Rm}))^*2$	<code>TBB [<Rn>, <Rm>]</code>
$\text{PC}+= (\text{U16}^*(\text{Rn+Rm}^*2))^*2$	<code>TBH [<Rn>, <Rm>, LSL #1]</code>
Rn 与 Imm12 按位异或，并根据结果更新标志位	<code>TEQ.W <Rn>, #<modify_constant(immed_12)></code>
先按需移位 Rm ，然后 Rn 与 Rm 按位异或，并根据结果更新标志位	<code>TEQ.W <Rn>, <Rm>{, <shift>}</code>
Rn 与 Imm12 按位与，并根据结果更新标志位	<code>TST.W <Rn>, #<modify_constant(immed_12)></code>
先按需移位 Rm ，然后 Rn 与 Rm 按位与，并根据结果更新标志位	<code>TST.W <Rn>, <Rm>{, <shift>}</code>
抽取 Rn 中以 lsb 号位为最低有效位，共 width 宽度的位段，并无符号扩展到 Rd 中	<code>UBFX.W <Rd>, <Rn>, #<lsb>, #<width></code>
无符号除法 $\text{Rd} = \text{Rn/Rm}$	<code>UDIV<c> <Rd>, <Rn>, <Rm></code>
无符号64位乘加， $\text{RdHi:RdLo}+= \text{Rn} * \text{Rm}$	<code>UMLAL.W <RdLo>, <RdHi>, <Rn>, <Rm></code>
无符号64位乘法， $\text{RdHi:RdLo} = \text{Rn} * \text{Rm}$	<code>UMULL.W <RdLo>, <RdHi>, <Rn>, <Rm></code>
先按需移位 Rn ，再把 Rn 向低 Imm 位执行带符号饱和操作，并把结果无符号扩展后写到 Rd 中	<code>USAT <c> <Rd>, #<imm>, <Rn>{, <shift>}</code>
先按需圆圈移位 Rm ，然后取出 Rm 的低8位，无符号扩展到32位并存储到 Rd	<code>UXTB.W <Rd>, <Rm>{, <rotation>}</code>
先按需圆圈移位 Rm ，然后取出 Rm 的低16位，无号扩展到32位并存储到 Rd	<code>UXTH.W <Rd>, <Rm>{, <rotation>}</code>
等待事件	<code>WFE.W</code>
等待中断	<code>WFI.W</code>

附录B

16 位 Thumb 指令及架构版本

表B.1 不同ARM架构版本下对 16 位指令的改动

指令	v4T	v5T	v6	Cortex-M3(v7-M)
BKPT		Y	Y	Y
BLX		Y	Y	只能使用 BLX <reg>
CBZ, CBNZ				Y
CPS			Y	CPSIE <i/f>, CPSID <i/f>
CPY			Y	Y
NOP				Y
IT				Y
REV(多种形式)			Y	Y
SEV				Y
SETEND			Y	
SWI	Y	Y	Y	改为 SVC
SXTB, SXTH			Y	Y
UXTB, UXTH			Y	Y
WFI, WFE				Y

附录C

Cortex-M3 异常快速参考

表C.1 异常一览表

编号	类型	优先级	简介	
0	N/A	N/A	没有异常在运行	
1	复位	-3 (最高)	复位	总是
2	NMI	-2	不可屏蔽中断 (来自外部 NMI 输入脚)	总是
3	硬 fault	-1	所有被除能的 fault, 都将“上访” (escalation) 成硬 fault。只要 FAULTMASK 没有置位, 硬 fault 服务例程就被强制执行。Fault 被除能的原因包括被禁用, 或者被 PRIMASK/BASEPRI 掩蔽	总是
4	Mem Manage fault	可编程	存储器管理 fault, MPU 访问犯规以及访问非法位置均可引发。企图在“非法执行区”取指也会引发此 fault	NVIC SHCSR.16 E000_ED24
5	总线 fault	可编程	从总线系统收到了错误响应, 原因可能是预取流产 (Abort) 或数据流产, 或者企图访问协处理器	NVIC SHCSR.17 E000_ED24
6	用法 Fault	可编程	由于程序错误导致的异常。通常是使用了一条无效指令, 或者是非法的状态转换, 例如尝试切换到 ARM 状态	NVIC SHCSR.18 E000_ED24
7-10	保留	N/A	N/A	
11	SVCall	可编程	执行系统服务调用指令 (SVC) 引发的异常	总是 E000_ED1F
12	调试监视器	可编程	调试监视器 (断点, 数据观察点, 或者是外部调试请求)	NVIC DEMCR.16 E000_EDFC
13	保留	N/A	N/A	
14	PendSV	可编程	为系统设备而设的“可悬挂请求” (pendable request)	总是 E000_ED22
15	SysTick	可编程	系统滴答定时器 (也就是周期性溢出的时基定时器——译注)	SysTick CTRLSTAT.1 E000_E010
16-255	IRQ	E000_E400	多达 240 条外部中断	NVIC SETENA 寄存器阵列

表C.2 自动入栈后堆栈中的内容以及SP的调整

地址	寄存器	被保存的顺序
旧SP (N-0)	原先已压入的内容	-
(N-4)	xPSR	2
(N-8)	PC	1
(N-12)	LR	8
(N-16)	R12	7
(N-20)	R3	6
(N-24)	R2	5
(N-28)	R1	4
新SP (N-32)	R0	3

注意：如果启用了堆栈的双字对齐特性，但是SP却没能对齐到双字，则堆栈帧的顶部有可能从 $((OLD_SP-4) \text{ AND } 0xFFFF_FFF8)$ 处开始，且其余的内容被向下错位一个字

附录D

NVIC 寄存器小结

表D.1 中断控制器类型寄存器ICTR 0xE000_E004

位段	名称	类型	复位值	描述
4:0	INTLINESUM	R	-	中断输入的数量, 以 32 为粒度, 如 0=1 至 32 1=33 至 64 2=65 至 96 ...

表D.2 SysTick控制及状态寄存器 (地址: 0xE000_E010)

位段	名称	类型	复位值	描述
16	COUNTFLAG	R	0	如果在上次读取本寄存器后, SysTick 已经数到了 0, 则该位为 1。如果读取该位, 该位将自动清零
2	CLKSOURCE	R/W	0	0=外部时钟源(STCLK) 1=内核时钟(FCLK)
1	TICKINT	R/W	0	1=SysTick 倒数到 0 时产生 SysTick 异常请求 0=数到 0 时无动作
0	ENABLE	R/W	0	SysTick 定时器的使能位

表D.3 SysTick重装载数值寄存器 (地址: 0xE000_E014)

位段	名称	类型	复位值	描述
23:0	RELOAD	R/W	0	当倒数至零时, 将被重装载的值

表D.4 SysTick当前数值寄存器 (地址: 0xE000_E018)

位段	名称	类型	复位值	描述
23:0	CURRENT	R/Wc	0	读取时返回当前倒计数的值, 写它则使之清零, 同时还会清除在 SysTick 控制及状态寄存器中的 COUNTFLAG 标志

表D.5 SysTick校准数值寄存器 (地址: 0xE000_E01C)

位段	名称	类型	复位值	描述
31	NOREF	R	-	1=没有外部参考时钟 (STCLK 不可用) 0=外部参考时钟可用
30	SKEW	R	-	1=校准值不是准确的 10ms 0=校准值是准确的 10ms

23:0	TENMS	R/W	0	10ms 的时间内倒计数的格数。芯片设计者应该通过 Cortex-M3 的输入信号提供该数值。若该值读回零，则表示无法使用校准功能
-------------	--------------	-----	----------	---

表D.6 SETENA/CLRENA寄存器族

SETENAs: 0xE000_E100 – 0xE000_E11C ; CLRENAAs: 0xE000_E180 – 0xE000_E19C

名称	类型	地址	复位值	描述
SETENA0	R/W	0xE000_E100	0	中断 0-31 的使能寄存器，共 32 个使能位 位[n]，中断#n 使能（异常号 16+n）
SETENA1	R/W	0xE000_E104	0	中断 32-63 的使能寄存器，共 32 个使能位
...
SETENA7	R/W	0xE000_E11C	0	中断 224-239 的使能寄存器，共 16 个使能位
CLRENA0	R/W	0xE000_E180	0	中断 0-31 的除能寄存器，共 32 个除能位 位[n]，中断#n 除能（异常号 16+n）
CLRENA1	R/W	0xE000_E184	0	中断 32-63 的除能寄存器，共 32 个除能位
...
CLRENA7	R/W	0xE000_E19C	0	中断 224-239 的除能寄存器，共 16 个除能位

表D.7 SETPEND/CLRPEND寄存器族

SETPENDs: 0xE000_E200 – 0xE000_E21C ; CLRPENDs: 0xE000_E280 – 0xE000_E29C

名称	类型	地址	复位值	描述
SETPEND0	R/W	0xE000_E200	0	中断 0-31 的悬起寄存器，共 32 个悬起位 位[n]，中断#n 悬起（异常号 16+n）
SETPEND1	R/W	0xE000_E204	0	中断 32-63 的悬起寄存器，共 32 个悬起位
...
SETPEND7	R/W	0xE000_E21C	0	中断 224-239 的悬起寄存器，共 16 个悬起位

CLRPEND0	R/W	0xE000_E280	0	中断 0-31 的解悬寄存器, 共 32 个解悬位 位[n], 中断#n 解悬 (异常号 16+n)
CLRPEND1	R/W	0xE000_E284	0	中断 32-63 的解悬寄存器, 共 32 个解悬位
...
CLRPEND7	R/W	0xE000_E29C	0	中断 224-239 的解悬寄存器, 共 16 个解悬位

表D.8 ACTIVE寄存器族 0xE000_E300_0xE000_E31C

名称	类型	地址	复位值	描述
ACTIVE0	RO	0xE000_E300	0	中断 0-31 的活动状态寄存器, 共 32 个状态位 位[n], 中断#n 活动状态 (异常号 16+n)
ACTIVE1	RO	0xE000_E304	0	中断 32-63 的活动状态寄存器, 共 32 个状态位
...
ACTIVE7	RO	0xE000_E31C	0	中断 224-239 的活动状态寄存器, 共 16 个状态位

表D.9 中断优先级寄存器阵列 0xE000_E400 - 0xE000_E4EF

名称	类型	地址	复位值	描述
PRI_0	R/W	0xE000_E400	0 (8 位)	外中断#0 的优先级
PRI_1	R/W	0xE000_E401	0 (8 位)	外中断#1 的优先级
...
PRI_239	R/W	0xE000_E4EF	0 (8 位)	外中断#239 的优先级

表D.10 CPUID寄存器 0xE000_ED00

位段	名称	复位值	描述
31:24	R	0x41	实现者代码, ARM=0x41

23:20	R	0x0/0x1/0x02	实现定义的变种号
19:16	R	0xF	常量
15:4	R	0xC23	Part 编号
3:0	R	0x0/0x1	修订号

表D.11 中断控制及状态寄存器ICSR 0xE000_ED04

位段	名称	类型	复位值	描述
31	NMIPENDSET	R/W	0	写 1 以悬起 NMI。因为 NMI 的优先级最高且从不掩蔽，在置位此位后将立即进入 NMI 服务例程。
28	PENDSVSET	R/W	0	写 1 以悬起 PendSV。读取它则返回 PendSV 的状态
27	PENDSVCLR	W	0	写 1 以清除 PendSV 悬起状态
26	PENDSTSET	R/W	0	写 1 以悬起 SysTick。读取它则返回 PendSV 的状态
25	PENDSTCLR	W	0	写 1 以清除 SysTick 悬起状态
23	ISRPREEMPT	R	0	为 1 时，则表示一个悬起的中断将在下一步时进入活动状态（用于单步执行时的调试目的）
22	ISRPENDING	R	0	1=当前正有外部中断被悬起（不包括 NMI）
21:12	VECTPENDING	R	0	悬起的 ISR 的编号。如果不止一个中断悬起，则它的值是这引动中断中，优先级最高的那一个。
11	RETTTOBASE	R	0	当从异常返回后将回到基级(base level)，且没有其它异常悬起时，此位为 1。若是在线程模式下，在某个服务例程中，有不止一级的异常处于活动状态，或者在异常没有活动时执行了异常服务例程（此时执行返回指令将产生 fault。此乃高危行为，大虾专用），则此位为 0
9:0	VECTACTIVE	R	0	当前活动的 ISR 编号，该位段指出当前运行中的 ISR 是哪个中断的（提供异常序号），包括 NMI 和硬 fault。如果多个异常共享一个服务例程，该例程可根据本位段的值来判定是哪一个异常的响应导致它的执行。把本位段的值减去 16，就得到了外中断的编号，并可以用此编号来操作外中断相关的使能/除能等寄存器。

表D.12 向量表偏移量寄存器(VTOR) 0xE000_ED08

位段	名称	类型	复位值	描述
7-28	TBLOFF	RW	0	向量表起始地址
29	TBLBASE	R	-	向量表是在 Code 区 (0)，还是在 RAM 区 (1)

表D.13 应用程序中断及复位控制寄存器(AIRCR) 0xE000_ED0C

位段	名称	类型	复位值	描述
31:16	VECTKEY	RW	-	访问钥匙: 任何对该寄存器的写操作, 都必须同时把 0x05FA 写入此段, 否则写操作被忽略。若读取此半字, 则 0xFA05
15	ENDIANESS	R	-	指示端设置。1=大端(BE8), 0=小端。此值是在复位时确定的, 不能更改。
10:8	PRIGROUP	R/W	0	优先级分组
2	SYSRESETREQ	W	-	请求芯片控制逻辑产生一次复位
1	VECTCLRACTIVE	W	-	清零所有异常的活动状态信息。通常只在调试时用, 或者在 OS 从错误中恢复时用。
0	VECTRESET	W	-	复位 CM3 处理器内核(调试逻辑除外), 但是此复位不影响芯片上在内核以外的电路

表D.14 系统控制寄存器 0xE000_ED10

位段	名称	类型	复位值	描述
4	SEVONPEND	RW	-	发生异常悬起时请发送事件, 用于在一个新的中断悬起时从 WFE 指令处唤醒。不管这个中断的优先级是否比当前的高, 都唤醒。如果没有 WFE 导致睡眠, 则下次使用 WFE 时将立即唤醒
3	保留	-	-	-
2	SLEEPDEEP	R/W	0	当进入睡眠模式时, 使能外部的 SLEEPDEEP 信号, 以允许停止系统时钟
1	SLEEPONEXIT	R/W	-	激活“SleepOnExit”功能
0	保留	-	-	-

表D.15 配置与控制寄存器 0xE000_ED14

位段	名称	类型	复位值	描述
9	STKALIGN	RW	0(r1 1(r2	开 在响应异常的自动入栈操作时, 强制 SP 对齐到双字地址上。修订版 0 无此功能 开 始)
8	BFHFNIGN	RW	0	在硬 fault 与 NMI 服务例程中忽略数据总线 fault
7:5	保留	-	-	-
4	DIV_0_TRP	RW	0	除数为零时陷入用法 fault
3	UNALIGN_TRP	RW	0	访问未对齐时陷入用法 fault
2	保留	-	-	-
1	USERSETMPEND	RW	0	如果为 1, 则允许用户代码设置 STIR
0	NONBASETHRDEN	RW	0	非基于线程模式使能位。如果为 1, 则允许

异常服务例程通过修改 `EXC_RETURN`, 使其
在线程模式下执行

表D.16 系统异常优先级寄存器 0xE000_ED18 - 0xE000_ED23

地址	名称	类型	复位值	描述
0xE000_ED18	PRI_4			存储器管理 <code>fault</code> 的优先级
0xE000_ED19	PRI_5			总线 <code>fault</code> 的优先级
0xE000_ED1A	PRI_6			用法 <code>fault</code> 的优先级
0xE000_ED1B	-	-	-	-
0xE000_ED1C	-	-	-	-
0xE000_ED1D	-	-	-	-
0xE000_ED1E	-	-	-	-
0xE000_ED1F	PRI_11			SVC 优先级
0xE000_ED20	PRI_12			调试监视器的优先级
0xE000_ED21	-	-	-	-
0xE000_ED22	PRI_14			PendSV 的优先级
0xE000_ED23	PRI_15			SysTick 的优先级

表D.17 系统Handler控制及状态寄存器SHCSR 0xE000_ED24

位段	名称	类型	复位值	描述
18	USGFAULTENA	R/W	0	用法 <code>fault</code> 服务例程使能位
17	BUSFAULTENA	R/W	0	总线 <code>fault</code> 服务例程使能位
16	MEMFAULTENA	R/W	0	存储器管理 <code>fault</code> 服务例程使能位
15	SVCALLPENDED	R/W	0	SVC 悬起中。本来已经要 SVC 服务例程, 但是却被更高优先级异常取代
14	BUSFAULTPENDED	R/W	0	总线 <code>fault</code> 悬起中, 细节同上。
13	MEMFAULTPENDED	R/W	0	存储器管理 <code>fault</code> 悬起中, 细节同上
12	USGFAULTPENDED	R/W	0	用法 <code>fault</code> 悬起中, 细节同上
11	SYSTICKACT	R/W	0	SysTick 异常活动中
10	PENDSVACT	R/W	0	PendSV 异常活动中
9	-	-	-	-
8	MONITORACT	R/W	0	Monitor 异常活动中
7	SVCALLACT	R/W	0	SVC 异常活动中
6:4	-	-	-	-
3	USGFAULTACT	R/W	0	用法 <code>fault</code> 异常活动中
2	-	-	-	-
1	BUSFAULTACT	R/W	0	总线 <code>fault</code> 异常活动中
0	MEMFAULTACT	R/W	0	存储器管理 <code>fault</code> 异常活动中

表 D.18 存储器管理 fault 状态寄存器(MFSR) 0xE000_ED28

位段	名称	类型	复位值	描述
7	MMARVALID	-	0	=1 时表示 MMAR 有效
6:5	-	-	-	-
4	MSTKERR	R/Wc	0	入栈时发生错误
3	MUNSTKERR	R/Wc	0	出栈时发生错误
2	-	-	-	-
1	DACCVIOL	R/Wc	0	数据访问违例
0	IACCVIOL	R/Wc	0	取指访问违例

表 D.19 总线 fault 状态寄存器(BFSR) 0xE000_ED29

位段	名称	类型	复位值	描述
7	BFARVALID	-	0	=1 时表示 BFAR 有效
6:5	-	-	-	-
4	STKERR	R/Wc	0	入栈时发生错误
3	UNSTKERR	R/Wc	0	出栈时发生错误
2	IMPRECISERR	R/Wc	0	不精确的数据访问违例 (violation)
1	PRECISERR	R/Wc	0	精确的数据访问违例
0	IBUSERR	R/Wc	0	取指时的访问违例

表 D.20 用法 fault 状态寄存器(UFSR), 地址: 0xE000_ED2A

位段	名称	类型	复位值	描述
9	DIVBYZERO	R/Wc	0	表示除法运算时除数为零(只有在 DIV_0_TRP 置位时才会发生)
8	UNALIGNED	R/Wc	0	未对齐访问导致的 fault
7:4	-	-	-	-
3	NOCP	R/Wc	0	试图执行协处理器相关指令
2	INVPC	R/Wc	0	在异常返回时试图非法地加载 EXC_RETURN 到 PC。包括非法的指令, 非法的上下文以及非法的值。The return PC 指向的指令试图设置 PC 的值 (要理解此位的含义, 还需学习后面的讨论中断级异常的章节)
1	INVOKESTATE	R/Wc	0	试图切入 ARM 状态
0	UNDEFINSTR	R/Wc	0	执行的指令其编码是未定义的——解码不能

表 D.21 硬 fault 状态寄存器 0xE000_ED2C

位段	名称	类型	复位值	描述
31	DEBUGEVT	R/Wc	0	硬 fault 因调试事件而产生

30	FORCED	R/Wc	0	硬 fault 是总线 fault , 存储器管理 fault 或是用法 fault 上访的结果
29:2	-	-	-	-
1	VECTBL	R/Wc	0	硬 fault 是在取向量时发生的
0	-	-	-	-

表 D.22 调试 fault 状态寄存器(DFSR) 0xE000_ED30

位段	名称	类型	复位值	描述
4	EXTERNAL	R/Wc	0	EDBGREQ 信号有效
3	VCATCH	R/Wc	0	发生向量加载
2	DWTTRAP	R/Wc	0	发生 DWT 匹配
1	BKPT	R/Wc	0	执行到 BKPT 指令
0	HALTED	R/Wc	0	在 NVIC 中请求 HALT

表 D.23 存储管理地址寄存器(MMAR) 0xE000_ED34

位段	名称	类型	复位值	描述
31:0	MMAR	R	-	触发存储管理 fault 的地址

表 D.24 总线 fault 地址寄存器(BFAR) 0xE000_ED38

位段	名称	类型	复位值	描述
31:0	BFAR	R	-	触发总线 fault 的地址

表 D.25 辅助 fault 地址寄存器(AFAR) 0xE000_ED3C

位段	名称	类型	复位值	描述
31:0	AFAR	R	-	由芯片制造商决定 (可选)

表 D.26 MPU 类型寄存器 MPUTR 0xE000_ED90

位段	名称	类型	复位值	描述
23:16	IREGION	R	0	MPU 支持的指令 region 数量。因为 ARMv7-M 只使用单个统一的 MPU, 此位段永远为零
15:8	DREGION	R	0	MPU 支持的数量。若系统中配了 MPU 则为 8, 否则为零
0	SEPARATE	R	0	固定为零

表 D.27 MPU 控制寄存器 MPUCR (地址 : 0xE000_ED94)

位段	名称	类型	复位值	描述
----	----	----	-----	----

2	PRIVDEFENA	RW	0	是否为特权级打开缺省存储器映射（即背景 region）。 1=特权级下打开背景 region 0=不打开背景 region。任何访问违例以及对 region 外地址区的访问都将引起 fault
1	HFNMIENA	RW	0	1=在 NMI 和硬 fault 服务例程中不强制除能 MPU 0=在 NMI 和硬 fault 服务例程中强制除能 MPU
0	ENABLE	RW	0	使能 MPU

表 D.28 MPU region 号寄存器 MPURNR (地址 : 0xE000_ED98)

位段	名称	类型	复位值	描述
7:0	REGION	RW	-	选择下一个要配置的 region。因为只支持 8 个 region, 所以事实上只有[2:0]有意义

表 D.29 MPU region 号寄存器 MPURNR (地址 : 0xE000_ED9C)

位段	名称	类型	复位值	描述
31:N	ADDR	RW	-	Region 基址字段。N 取决于 region 容量, 以使基址在数值上能被容量整除。在 MPU region 属性及容量寄存器中有个 SZENABLE 位段, 它决定 ADDR 中有多少个位被采用。
4	VALID	RW	-	决定是否理会写入REGION字段的值 1=MPU region 号寄存器被REGION覆盖 0=MPU region 号寄存器的值保持不变
3:0	REGION	RW	-	MPU region 覆写位段

表D.30 MPU region属性及容量寄存器MPURASR (地址 : 0xE000_EDA0)

位段	长度	名称	功能																																				
31:29	3	-	保留																																				
28	1	XN	1=此区禁止取指 2=此区允许取指																																				
27	1	-	保留																																				
26:24	3	AP	访问许可, 如下表所示																																				
			<table border="1"> <thead> <tr> <th>值</th><th>特权级下的许可</th><th>用户级下的许可</th><th>典型用法</th></tr> </thead> <tbody> <tr> <td>0b000</td><td>禁地</td><td>禁地</td><td>该区没有存储器, 是空地址</td></tr> <tr> <td>0b001</td><td>RW</td><td>禁地</td><td>OS和系统软件使用的数据区</td></tr> <tr> <td>0b010</td><td>RW</td><td>RO</td><td>禁止在用户级下更改的高危地带</td></tr> <tr> <td>0b011</td><td>RW</td><td>RW</td><td>共享内存, 或彻底开放的设备</td></tr> <tr> <td>0b100</td><td>n/a</td><td>n/a</td><td>n/a</td></tr> <tr> <td>0b101</td><td>RO</td><td>禁地</td><td>OS使用的常量数据</td></tr> <tr> <td>0b110</td><td>RO</td><td>RO</td><td>常量数据或只读存储器的地址区</td></tr> <tr> <td>0b111</td><td>RO</td><td>RO</td><td>常量数据或只读存储器的地址区</td></tr> </tbody> </table>	值	特权级下的许可	用户级下的许可	典型用法	0b000	禁地	禁地	该区没有存储器, 是空地址	0b001	RW	禁地	OS和系统软件使用的数据区	0b010	RW	RO	禁止在用户级下更改的高危地带	0b011	RW	RW	共享内存, 或彻底开放的设备	0b100	n/a	n/a	n/a	0b101	RO	禁地	OS使用的常量数据	0b110	RO	RO	常量数据或只读存储器的地址区	0b111	RO	RO	常量数据或只读存储器的地址区
值	特权级下的许可	用户级下的许可	典型用法																																				
0b000	禁地	禁地	该区没有存储器, 是空地址																																				
0b001	RW	禁地	OS和系统软件使用的数据区																																				
0b010	RW	RO	禁止在用户级下更改的高危地带																																				
0b011	RW	RW	共享内存, 或彻底开放的设备																																				
0b100	n/a	n/a	n/a																																				
0b101	RO	禁地	OS使用的常量数据																																				
0b110	RO	RO	常量数据或只读存储器的地址区																																				
0b111	RO	RO	常量数据或只读存储器的地址区																																				
23:22	2	—	保留																																				
21:19	3	TEX	类型扩展																																				
18	1	S	Sharable (可否共享) 1=共享可 0=共享不可																																				
17	1	C	Cachable (可否缓存) 1=缓存可 0=缓存不可																																				
16	1	B	Buffable (可否缓冲) 1=缓冲可 0=缓冲不可																																				
15:8	8	SRD	子region除能位段。每设置SRD的一个位, 就会除能与之对应的一个子region。容量大于128字节的region都被划分成8个容量相同的子region。容量小于等于128字节的region不能再分。更多信息, 请参见对子Region的论述。																																				
7:6	2	-	保留																																				
5:1	5	REGIONSIZE	Region容量, 单位是字节。容量为 $1 \ll (\text{REGIONSIZE} + 1)$, 但是最小容量为32字节																																				
0	1	SZENABLE	1=使能此region 0=除能此region																																				

表 D.31 调试停机控制及状态寄存器 DHCSR (地址 : 0xE000_EDF0)

位段	名称	类型	复位值	描述
31:15	KEY	W	-	调试钥匙。必须在任何写操作中把该位段写入A05F, 否则忽略写操作

25	S_RESET_ST	R	-	内核已经或即将复位, 读后清零
24	S_RETIRE_ST	R	-	在上次读取以后指令已执行完成, 读后清零
19	S_LOCKUP	R	-	1=内核进入锁定状态
18	S_SLEEP	R	-	1=内核睡眠中
17	S_HALT	R	-	1=内核已停机
16	S_REGRDY	R	-	1=寄存器的访问已经完成
15:6	保留	-	-	
5	C_SNAPSTALL	RW	0*	打断一个 <code>stalled</code> 存储器访问
4	保留	-	-	
3	C_MASKINTS	RW	0*	调试期间关中断, 只有在停机后方可设置
2	C_STEP	RW	0*	让处理器单步执行, 在 C_DEBUGEN=1 时有效
1	C_HALT	RW	0*	喊停处理器, 在 C_DEBUGEN=1 时有效
0	C_DEBUGEN	RW	0*	使能停机模式的调试

表 D.32 调试内核寄存器选择者寄存器 DCRSR (地址 : 0xE000_EDF4)

位段	名称	类型	复位值	描述
16	REGWnR	W	-	1=写寄存器 0=读寄存器
15:5	保留	-	-	-
4:0	REGSEL	W	-	00000=R0 00001=R1 ... 01111=R15 10000=xPSR 10001=MSP 10010=PSP 10100=特殊功能寄存器组 [31:24]: CONTROL [23:16]: FAULTMASK [15:8]: BASEPRI [7:0]: PRIMASK

表 D.33 调试内核寄存器数据寄存器 DCRDR (地址 : 0xE000_EDF8)

位段	名称	类型	复位值	描述
31:0	DATA	R/W	-	读回来的寄存器的值, 或欲写入寄存器的值, 寄存器由 DCRSR 选择

表 D.34 调试乃及监视器控制寄存器 DEMCR (地址 : 0xE000_EDFC)

位段	名称	类型	复位值	描述
24	TRCENA	RW	0*	跟踪系统的使能位。在使用 DWT, ETM, ITM 和

				TPIU 前, 必须先设置此位
23:20	保留			
19	MON_REQ	RW	0	1=调试监视器异常不是由硬件调试事件触发, 而是由软件手工悬起的
18	MON_STEP	RW	0	让处理器单步执行, 在 MON_EN=1 时有效
17	MON_PEND	RW	0	悬起监视器异常请求, 内核将在优先级允许时响应
16	MON_EN	RW	0	使能调试监视器异常
15:11	保留			
10	VC_HARDERR	RW	0*	发生硬 fault 时停机调试
9	VC_INTERR	RW	0*	指令/异常服务错误时停机调试
8	VC_BUSERR	RW	0*	发生总线 fault 时停机调试
7	VC_STATERR	RW	0*	发生用法 fault 时停机调试
6	VC_CHKERR	RW	0*	发生用法 fault 使能的检查错误时停机调试 (如未对齐, 除数为零)
5	VC_NOCPERR	RW	0*	发生用法 fault 之无处理器错误时停机调试
4	VC_MMERR	RW	0*	发生存储器管理 fault 时停机调试
3:1	保留			
0	VC_CORERESET	RW	0*	发生内核复位时停机调试

*: DEMCR 中的控制位是在上电复位时得到复位的。系统复位(例如, 往 NVIC 应用程序中断及复位寄存器中写命令)不会影响到它们

表D.35 软件触发中断寄存器STIR 0xE000_EF00

位段	名称	类型	复位值	描述
8:0	INTID	W	-	影响编号为 INTID 的外部中断, 其悬起位被置位。 例如, 写入 8, 则悬起 IRQ #8

表 D36 中断优先级寄存器阵列 0xE000_E400 – 0xE000_E4EF

名称	类型	地址	复位值	描述
PERIPHID4	R	0xE000_EFD0	0x04	外设 ID 寄存器 4
PERIPHID5	R	0xE000_EFD4	0	外设 ID 寄存器 5
PERIPHID6	R	0xE000_EFD8	0	外设 ID 寄存器 6
PERIPHID7	R	0xE000_EFDC	0	外设 ID 寄存器 7
PERIPHID0	R	0xE000_EFE0	0	外设 ID 寄存器 0
PERIPHID1	R	0xE000_EFE4	0	外设 ID 寄存器 1
PERIPHID2	R	0xE000_EFE8	0x0B/0x1B	外设 ID 寄存器 2
PERIPHID3	R	0xE000_EFEC	0	外设 ID 寄存器 3
PCELLID0	R	0xE000_EFF0	0x0D	组件 ID 寄存器 0
PCELLID1	R	0xE000_EFF4	0xE0	组件 ID 寄存器 1
PCELLID2	R	0xE000_EFF8	0x05	组件 ID 寄存器 2
PCELLID3	R	0xE000_EFFC	0xB1	组件 ID 寄存器 3

附录E

Cortex-M3 疑难解答

E.1 简介

使用 CM3，一大令人闹心之处就是会有那么多的 **faults**，不知什么时候就会来一个。来的是哪个，为什么来，如何避免？这连珠炮一般的问题简直成了学习的拦路虎了。

其实，只要我们掌握和理解得深入，不粗心，规范设计和编程，这些 **faults** 就成了纸老虎。如果善加利用，甚至还能变害为利——在系统出现故障时由它们为我们提供有用的诊断信息——这才是 CM3 设计这些 **faults** 的初衷。

当 **fault** 发生时，首先要弄清楚的就是 **fault** 源，下表列出了相关的寄存器

表 E.1 CM3 中的 fault 状态寄存器组

地址	寄存器	全名	尺寸
0xE000_ED28	MMSR	MemManage fault 状态寄存器	字节
0xE000_ED29	BFSR	总线 fault 状态寄存器	字节
0xE000_ED2A	UFSR	用法 fault 状态寄存器	半字
0xE000_ED2C	HFSR	硬 fault 状态寄存器	字
0xE000_ED30	DFSR	调试 fault 状态寄存器	字
0xE000_ED3C	AFSR	辅助 fault 状态寄存器	字

上表的另一可视化视图如下图所示：

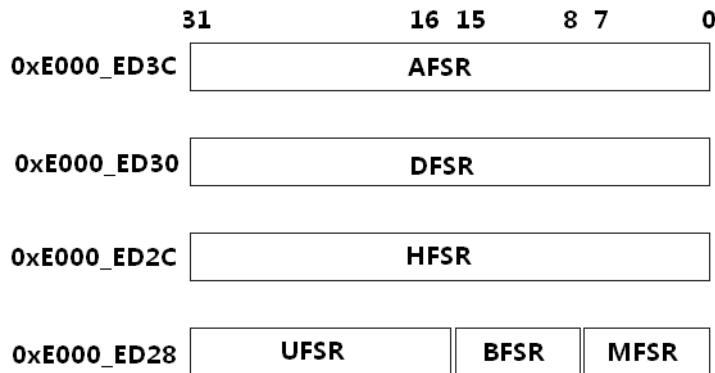


图 E.1 各 fault 状态寄存器的地址组织

因为 MMSR, BFSR 和 UFSR 的地址是相连的，所以可以使用按字加载指令一次性地全部读进来。在这种情况下，这个三合一的 **fault** 状态亦有一个名字：可配置 **fault** 状态寄存器(CFSR)。

另一个提供重要信息的寄存器是什么？它远在天边，近在眼前——人尽皆知的程序计数器 PC！进入 **fault** 服务例程后，当时的 PC 值在(SP-0x24)处。因为 CM3 中有两个堆栈指针，**fault** 服务例程还要判定发生 **fault** 时使用的是哪一个堆栈——MSP 还是 PSP。

进一步地，对于总线 **fault** 和存储器管理 **fault**，有时还能精确定位肇事指令的地址——当 MMAVALID/BFARVALID 位被置位时，即是精确 **fault**。此时，存储器管理 **fault** 的地址存储在 MMAR 中，总线 **fault** 的地址则存储在 BFAR 中。在物理实现上，MMAR 与 BFAR 其实是同一个寄存器，因

此同一时刻只能用一个——这是因为同一时刻只能出现一个 **fault** (但是访问地址还是不同的), 如表 E.2 所示。

表 E.2 CM3 中的 **fault** 地址寄存器

地址	寄存器	全名	尺寸
0xE000_ED34	MMAR	MemManage fault 地址寄存器	字
0xE000_ED38	BFAR	总线 fault 地址寄存器	字

最后, 在执行 **fault** 服务例程时, **LR** 寄存器的值也常常是一个线索, 间接反映了发生 **fault** 时的情景。如果 **fault** 是由无效的 **EXC_RETURN** 值导致的, 则进入 **fault** 时, **LR** 的值则是上次异常返回时使用的 **EXC_RETURN** 值。**Fault** 服务例程可以据此上报有问题的 **LR** 值, 从而使开发人员可以检查为何会使用非法的 **EXC_RETURN** (常常是粗心造成的)。

E.2 设计 **Fault** 服务例程

用于开发阶段的 **fault** 服务例程, 与用于实际系统中的服务例程, 在绝大多数场合下是截然不同的。对于软件开发, **fault** 服务例程应关注于准确及时地上报发生 **fault** 时上下文; 而实际系统中的 **fault** 服务例程则要把这当作是危急关头来处理, 它要尽可能地想办法来恢复系统, 实在不可救要时可能只有重启。这里我们主要讨论前者, 因为后者是比较有技术含量的, 而且不同的应用需要不同的策略。

对于比较复杂的软件, 通常不直接在 **fault** 服务例程中报告与 **fault** 相关的状态, 而是把它倾倒 (专业术语: **dump**) 到一块专用的内存中 (主要包括 **fault** 状态寄存器, 通用寄存器, 自动入栈的内容等), 接着悬起 **PendSV**。等到 **PendSV** 服务例程执行后, 再上报问题。这是因为上报过程执行的工作可能比较多, 夜长梦多——有可能在上报过程中又不小心触发了其它的 **fault**, 使得处理器被锁死。因此先暂记下来, 稍后转交 **PendSV** 处理。如果软件很简单, 则可以斟情简化 **fault** 的处理过程, 甚至直接在服务例程中上报。

E.2.1 上报 **fault** 状态寄存器

Fault 服务例程最基本的工作就是上报 **fault** 状态寄存器的值, 即上图所列出的那些。

E.2.2 上报入栈的 **PC**

定位入栈 **PC** 的流程如下图所示

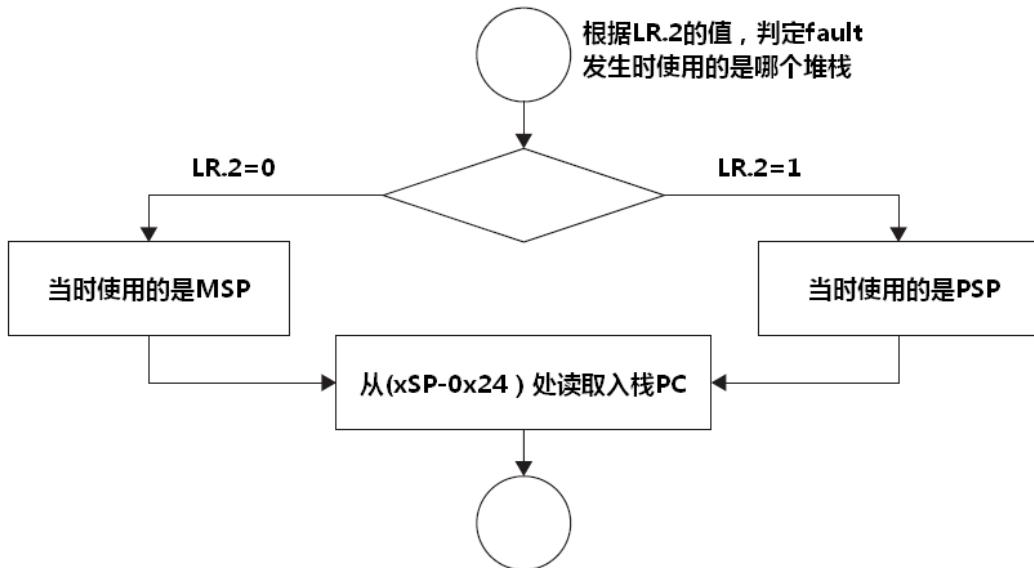


图 E.2 定位入栈 PC 的流程图

上图所示的工作流程可以由如下代码来演示：

```

TST LR, #0x4           ; EXC_RETURN.2=0?
ITTEE EQ                ; 如是为零，则
MRSEQ R0, MSP           ; 把MSP加载入R0中
LDREQ R0,[R0,#24]       ; 从MSP中获取入栈的PC
MRSNE R0, PSP           ; 否则，就把PSP加载入R0中
LDRNE R0,[R0,#24]       ; 从PSP中获取入栈的PC

```

为了辅助调试，应该同时创建一个反汇编的指令列表（如果使用 RVMDK，则自动创建），从而可以容易地定位问题所在。

E.2.3 上报 fault 地址寄存器

如果 MMARVALID 或 BFARVALID 为 1，则可以提供出事时的地址。但要注意的是，当 MMARVALID/BFARVALID 被清除后，fault 地址寄存器中的值可能被擦除。因此，必须先读 BFAR/MMAR，再读 BFARVALID/MMARVALID。如果后者为零，则丢弃读出的地址值。最后一步再清除 BFARVALID/MMARVALID。

如果先读取了状态位，则有可能在下一步操作前被其它 fault 异常抢占，这也是一种紊乱危象，有可能导致下述的错误处理序列：

1. 读取 BFARVALID/MMARVALID
2. 发现 VALID 位有效，于是准备读 BFAR/MMAR
3. 高优先级异常抢占了 fault 服务例程，而且它又触发了另一个 fault，导致另一个 fault 服务例程被执行
4. 高优先级 fault 服务例程清除了 BFARVALID/MMARVALID，导致 BFAR/MMAR 被擦除。
5. 回到先前的 fault 服务例程，此时再读取 BFAR/MMAR 时，内容已经丢失了。

可见，后读取 VALID 位，可以减少这种紊乱危象出现的概率（但如果还没来得及读取 BFAR/MMAR 就出现了被抢占的情况，则只能看人品了，所以最好第一件事就是读取 BFAR/MMAR——译者注）。

清除 fault 状态位

在 fault 上报完毕后，一定不要忘记清除 FSR 中的 fault 状态位。否则下次再发生 fault 时，就分不清 FSR 中的状态位是反映新来的 fault，还是反映以前的 fault 了。而且，如果 fault 地

址有效位没有清除，下次发生 `fault` 时，`BFAR/MMAR` 的值就无法更新。

E.2.4 其它注意事项

我们经常需要在 `fault` 服务例程的开始处保存 `LR` 的值。然而，如果 `fault` 是由于堆栈操作错误导致的，此时再把 `LR` 压栈就更添乱了。但我们已经知道，`R3-R0` 以及 `R12` 的值已经被保存，因此我们可以在呼叫其它函数之前先把 `LR` 的值拷贝到它们中去（事实上在出现堆栈错误时，是无法保证寄存器已经正常入栈了的。此时的问题比较棘手。可能行得通的作法是，在 `SRAM` 中专门开出一个块服务于 `fault` 服务例程，并把通用寄存器的值保存到那里，但这种办法无法用于嵌套的情况——这已经很钻牛角尖了）。

E.3 在 C 中上报入栈的寄存器和各 fault 状态寄存器

大多数的 CM3 项目还是以 C 语言为主的。然而，在 C 中不方便定位和直接访问堆栈帧（入栈的寄存器）。因为在标准 C 语言中是不能获取 `SP` 指针的。因此，如果使用 C 来写 `fault` 服务例程，最好配合一小段汇编码来获取 `SP` 的值，再把该值以一个参数传送给 `fault` 上报函数。

译注：在使用 MDK 自带的 ARM 编译器时，可以使用 `__builtin_frame_address()` 函数来获取堆栈帧的地址。在 GNU 工具中也可以这样做。此法方便，并且可取代上文的通用作法，但降低了编译器间的可移植性。

这个机制与第 12 章讲的 `SVC` 范例相同（“在 C 中使用 `SVC`”）。下例就以嵌入式汇编的方式来演示。这个例子可以在 RealView MDK 中编译。

示例程序的第一部分是个汇编封皮。在使用前，要在向量表中的硬 `fault` 入口地址项中，填写好该封皮的入口地址。这个封皮代码把正确的堆栈指针值拷贝到 `R0` 中，以作为参数来传送给 C 函数。

```
// 使用汇编写就的硬 fault 服务例程
// 该例程提取堆栈帧的位置并且把它传递
// 给 C 程序
__asm void hard_fault_handler_asm(void)
{
    IMPORT hard_fault_handler_c
    TST    LR, #4
    ITE    EQ
    MRSEQ  R0, MSP
    MRSNE  R0, PSP
    B      hard_fault_handler_c
}
```

示例程序的第二部分，也是主体部分，使用 C 语言来写。在这里，我们主要是演示如何访问入栈的寄存器和 `fault` 状态寄存器。

```
// 使用 C 写就的硬 fault 服务例程
// 第一个参数即堆栈帧的位置
void hard_fault_handler_c(unsigned int * hardfault_args)
{
    unsigned int stacked_r0;
    unsigned int stacked_r1;
    unsigned int stacked_r2;
```

```
unsigned int stacked_r3;
unsigned int stacked_r12;
unsigned int stacked_lr;
unsigned int stacked_pc;
unsigned int stacked_psr;

stacked_r0 = ((unsigned long) hardfault_args[0]);
stacked_r1 = ((unsigned long) hardfault_args[1]);
stacked_r2 = ((unsigned long) hardfault_args[2]);
stacked_r3 = ((unsigned long) hardfault_args[3]);

stacked_r12 = ((unsigned long) hardfault_args[4]);
stacked_lr = ((unsigned long) hardfault_args[5]);
stacked_pc = ((unsigned long) hardfault_args[6]);
stacked_psr = ((unsigned long) hardfault_args[7]);

printf ("[Hard fault handler]\n");
printf ("R0 = %x\n", stacked_r0);
printf ("R1 = %x\n", stacked_r1);
printf ("R2 = %x\n", stacked_r2);
printf ("R3 = %x\n", stacked_r3);
printf ("R12 = %x\n", stacked_r12);
printf ("LR = %x\n", stacked_lr);
printf ("PC = %x\n", stacked_pc);
printf ("PSR = %x\n", stacked_psr);

printf ("BFAR = %x\n", (*((volatile unsigned long *)(0xE000ED38))));;
printf ("CFSR = %x\n", (*((volatile unsigned long *)(0xE000ED28))));;
printf ("HFSR = %x\n", (*((volatile unsigned long *)(0xE000ED2C))));;
printf ("DFSR = %x\n", (*((volatile unsigned long *)(0xE000ED30))));;
printf ("AFSR = %x\n", (*((volatile unsigned long *)(0xE000ED3C))));;

exit(0); // terminate
return;
}
```

请注意：如果发生了堆栈溢出或其它错误，使 **SP** 指向了无效的存储空对空区域，则上段代码会失能。在大多数情况下，这种错误都会影响 **C** 代码，因为所有 **C** 代码都需要堆栈。

E.4 理解发生 fault 的原因

在收集到所需的信息后，就需要分析问题了。表 E.3-表 E.7 列出了导致 **faults** 的典型原因。

表 E.3 MemManage fault 状态寄存器提供的讯息

位	可能的原因
MSTKERR	入栈时发生错误 (异常响应序列开始时) 1) 堆栈指针的值被破坏 2) 堆栈容易过大, 已经超出 MPU 允许的 region 范围
MUNSTKERR	出栈时发生错误 (异常响应序列终止时)。入栈时没有发生错误, 出栈时却出错, 总令人有些匪夷所思, 可能的原因是 1. 异常服务例程破坏了堆栈指针 2. 异常服务例程更改了 MPU 配置
DACCVIOL	内存访问保护违例。这是 MPU 发挥作用的体现。常常是用户应用程序企图访问特权级 region 所致
IACCVIOL	1. 内存访问保护违例。常常是用户应用程序企图访问特权级 region。在这种情况下, 入栈的 PC 给出的地址, 就是产生问题的代码之所在 2. 跳转到不可执行指令的 regions 3. 异常返回时, 使用了无效的 EXC_RETURN 值 4. 向量表中有无效的向量。例如, 异常在向量建立之前就发生了, 或者加载的是用于传统 ARM 内核的可执行映像 5. 在异常处理期间, 入栈的 PC 值被破坏了

表 E.4 总线 fault 状态寄存器提供的讯息

位	可能的原因
STKERR	(自动) 入栈期间出错 1. 堆栈指针的值被破坏 2. 堆栈用量太大, 到达了未定义存储器的区域 3. PSP 未经初始化就使用
UNSTKERR	(自动) 出栈期间出错。如果没有发生过 STKERR, 则最可能的就是在异常处理期间把 SP 的值破坏了
IMPRECISERR	与设备之间传送数据的过程中发生总线错误。可能是因为设备未经初始化而引起; 或者在用户级访问了特权级的设备, 或者传送的数据单位尺寸不能为设备所接受。此时, 有可能是 LDM/STM 指令造成了非精确总线 fault。
PRECISERR	在数据访问期间的总线错误。通过 BFAR 可以获取具体的地址。发生 fault 的原因同上。
IBUSERR	同 MemManage fault 中的 IACCVIOL

表 E.5 用法 fault 状态寄存器提供的讯息

位	可能的原因
DIVBYZERO	当 DIV_0_TRP 置位时则发生了除数为零的情况。引发此 fault 的指令可以从入栈的 PC 读取
UNALIGNED	当 UNALIGN_TRP 置位时发生未对齐访问。引发此 fault 的指令可以从入栈的 PC

	读取
NOCP	企图执行一个协处理器指令。引发此 fault 的指令可以从入栈的 PC 读取
INVPC	<ol style="list-style-type: none"> 异常返回时使用了无效的 EXC_RETURN, 例如 <ol style="list-style-type: none"> 当 EXC_RETURN=0xFFFF_FFF1 时却要返回线程模式 当 EXC_RETURN=0xFFFF_FFF9 时却要返回 handler 模式 无效的异常活动状态, 例如 <ol style="list-style-type: none"> 当前异常的活动状态已经清除了, 却在此时执行异常返回。往往是因为滥用 VECTCLRACTIVE 或清除了 SHCSR 中活动状态所致 在尚有异常的活动位置位时, 却要返回线程模式 由于堆栈指针错误导致了 IPSR 的值不正确。对于 INVPC fault, 入栈的 PC 指出了该 fault 服务例程在何处抢占了其它代码。这个问题往往是由比较隐晦的程序错误造成的, 欲详细调查该问题的原因, 最好使用 ITM 的跟踪功能。 ICI/IT 位对当前指令无效。当 LDM/STM 指令被异常打断后, 在异常服务例程中又更改了入栈的 PC。结果在中断返回时, 非零的 ICI 位段作用到了不使用 ICI 位段的指令上。如果是其它原因破坏了 PSR 的值, 也可能导致此 fault。
INVSTATE	<ol style="list-style-type: none"> 加载到 PC 中的跳转地址值是偶数 (LSB=0)。通过检查入栈 PC 的值, 一下子就可以查出该问题。 向量地址的 LSB=0, 诊断方法同上。 入栈的 PSR 在异常处理过程中被破坏, 使得在返回时内核尝试进入 ARM 状态。
UNDEFINSTR	<ol style="list-style-type: none"> 使用了 CM3 不支持的指令 代码段中的数据被破坏 连接时加载了 ARM 目标码。请检查编译阶段的设置 指令对齐的问题。例如, 在使用 GNU 工具链时, 忘记了在 .ascii 后使用 .align, 就有可能导致下一条指令没有对齐

表 E.6 硬 **fault** 状态寄存器提供的讯息

位	可能的原因
DEBUGEVF	因调试事件导致的 fault <ol style="list-style-type: none"> 断点/观察点事件 在硬 fault 服务例程的执行过程中, 没有使能监视器异常 (MON_EN=0) 也没有使能停机调试 (C_DEBUGEN=0), 却执行了 BKPT 指令。缺省时, 有些 C 编译器可能会在半主机代码中使用 BKPT 指令。
FORCED	这是 fault “上访”的情况 <ol style="list-style-type: none"> 试图在 SVC/监视器服务例程中执行 SVC/BKPT, 或者在其它拥有相同或更高优先级的服务例程中执行 SVC/BKPT。 发生了 fault, 但是它的服务例程被除能 发生了 fault, 但是当前处理器在响应同级或更高优先级的异常 发生了 fault, 但是它被掩蔽了
VECTBL	取向量失败, <ol style="list-style-type: none"> 在取向量过程中发生总线 fault 向量表偏移量设置有误

表 E.7 调试 **fault** 状态寄存器提供的讯息

位	可能的原因
EXTERNAL	EDBGREQ 信号置为有效
VCATCH	发生了向量抓捕事件
DWTTRAP	发生了 DWT 观察点事件
BKPT	1. 执行了 BKPT 指令 2. FPB 单元产生了断点事件