第1章

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

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”，而且还又要马儿跑，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。相信用不了多久，CM3就一定能在32位嵌入式处理器市场中脱颖而出，像当年8051推动整个业界那样，再次放飞设计师的梦想，实现多年的夙愿！

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。

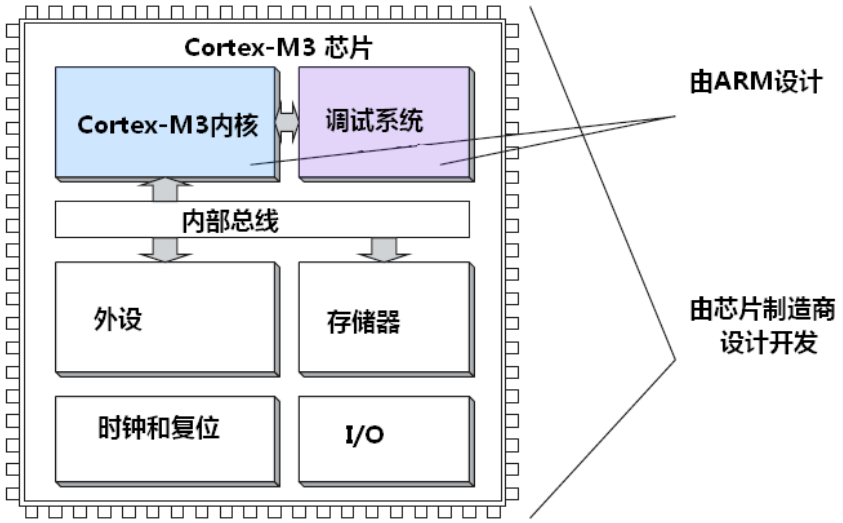
降低成本还有一招，就是使基础代码在所有系统中都可以重用，至少要方便移植。CM3的内核架构非常精工细作，使它与C语言成为了一个梦幻绝配。优质的C程序代码三下五除二就可以移植并重用，使升级和移植一下子从拦路虎变成了纸老虎。

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！

Cortex-M3 处理器内核 vs. 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。



ARM及ARM架构的背景

一路走来

让我们回顾一下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。

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）。

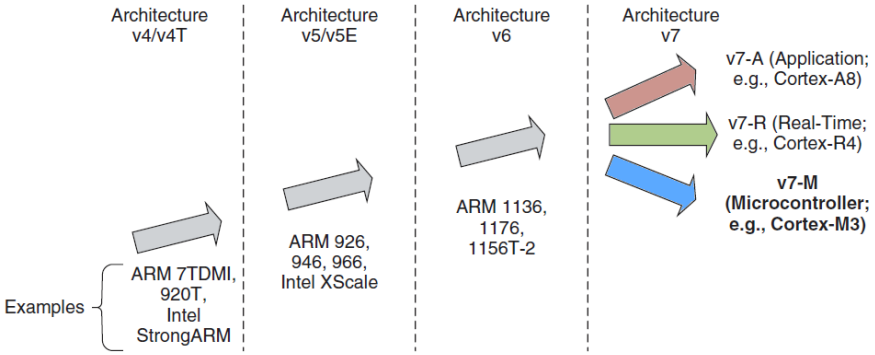


图1.2 ARM处理器架构进化史

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。

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。

指令系统的开发

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。

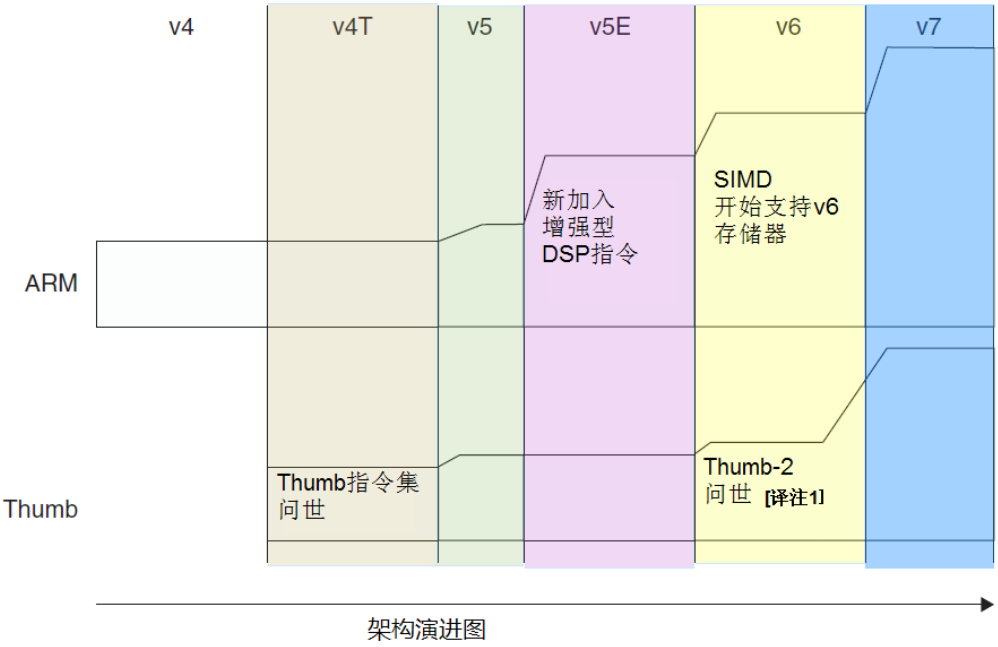


图1.3 指令集演进图

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。

Thumb-2指令集体系体系结构（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。

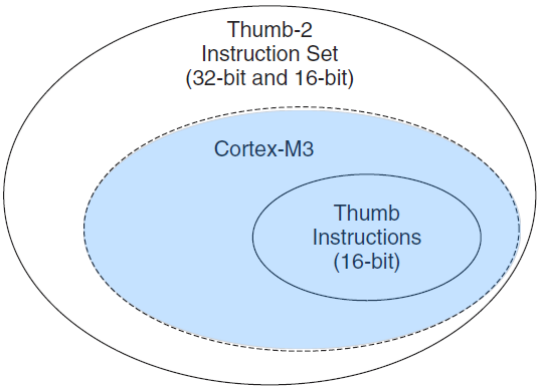


图1.4 Thumb-2指令集与Thumb指令集的关系

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，能精妙地游走于不同状态之间，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”。

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也是汽车电子的好伙。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-handing，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

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。

第2章

Cortex-M3概览

内容提要:

- 简介
- 寄存器组
- 操作模式和特权极别
- 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

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。

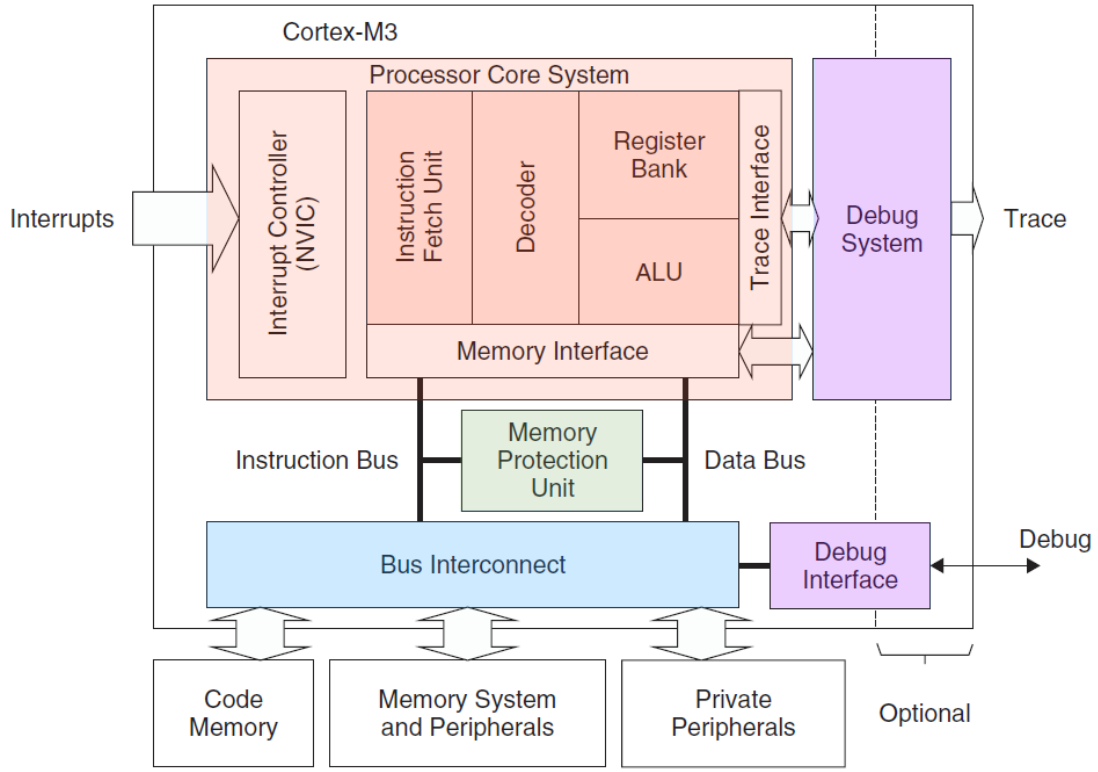
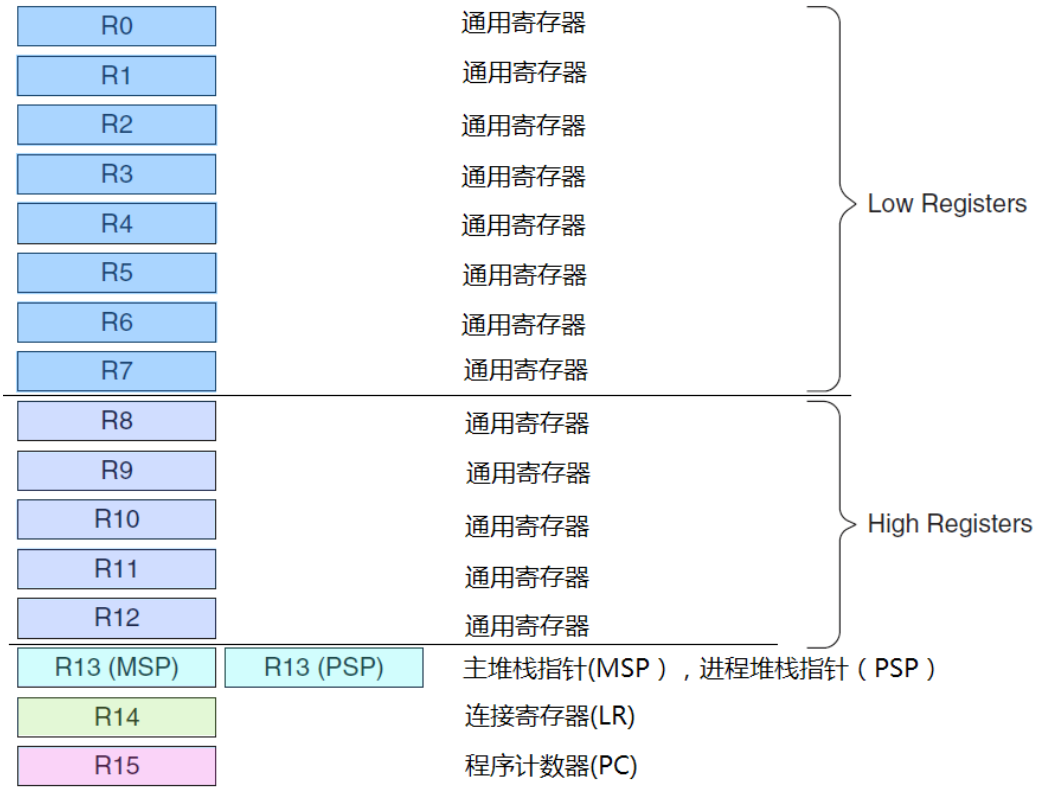


图 2.1 Cortex-M3 的一个简化视图

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。



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。

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）。

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。

R15：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）。

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）

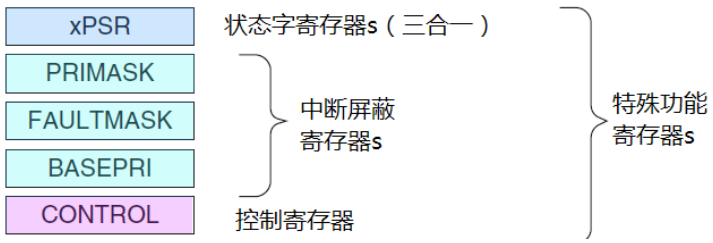


图 2.3： Cortex-M3 中的特殊功能寄存器集合

表 2.1 寄存器及其功能

寄存器	功能
xPSR	记录 ALU 标志（0 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。

操作模式和特权极别

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：处理者模式(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。

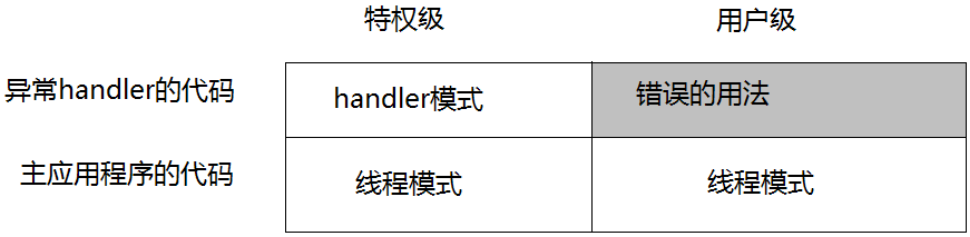


图 2.4 Cortex-M3 下的操作模式和特权级别

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，特权极访问。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）。

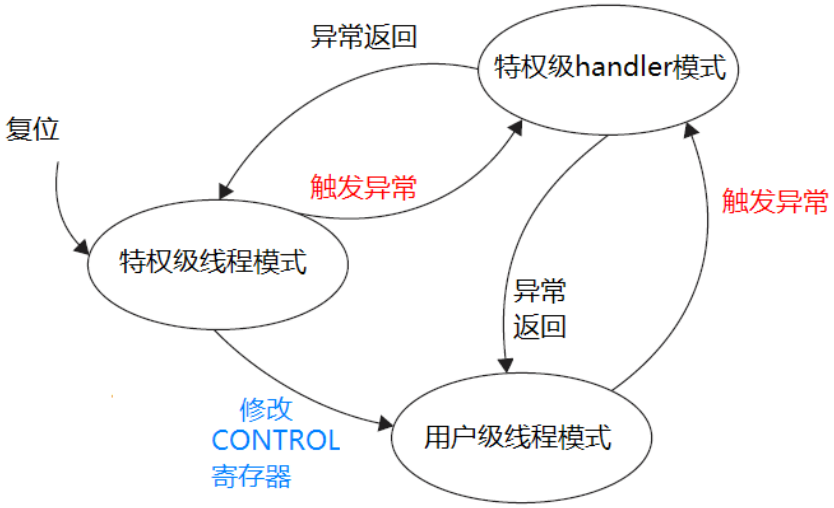


图 2.5 合法的操作模式转换图

通过引入特权级和用户级，就能够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。

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“私交”——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

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。

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。

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。

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。

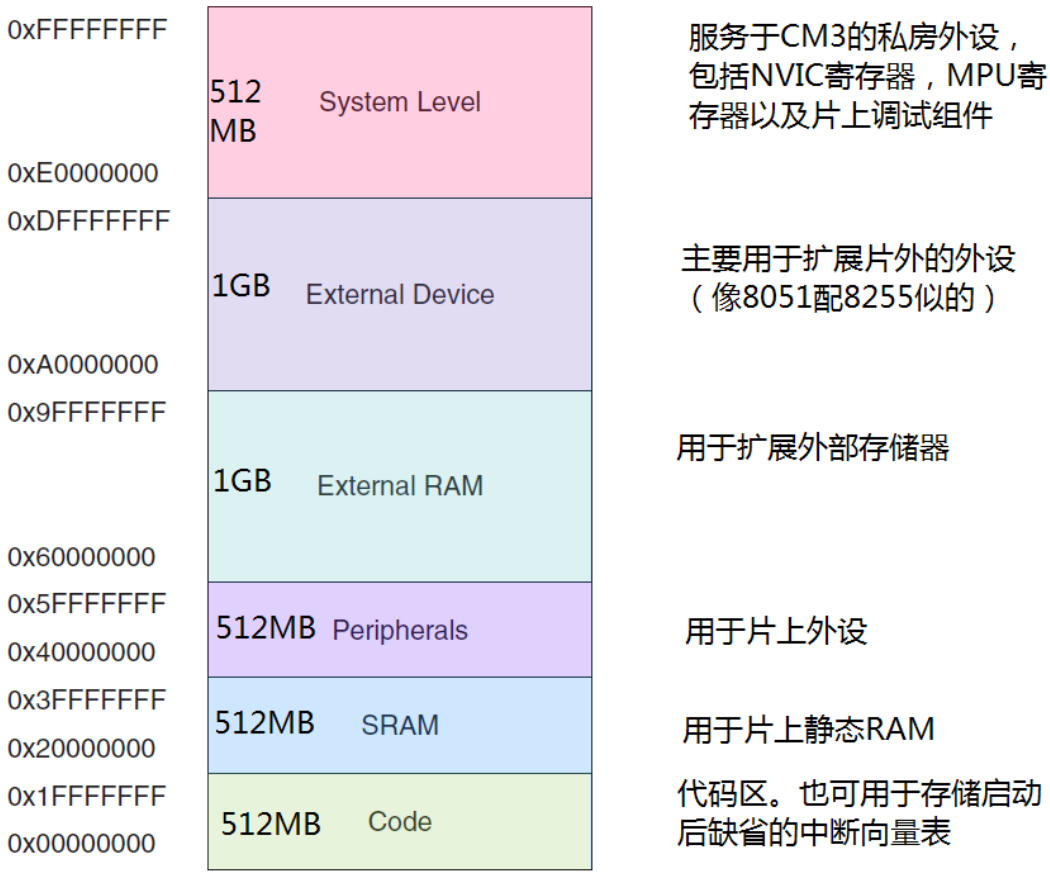
中断可屏蔽

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。

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。

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。

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）。

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。

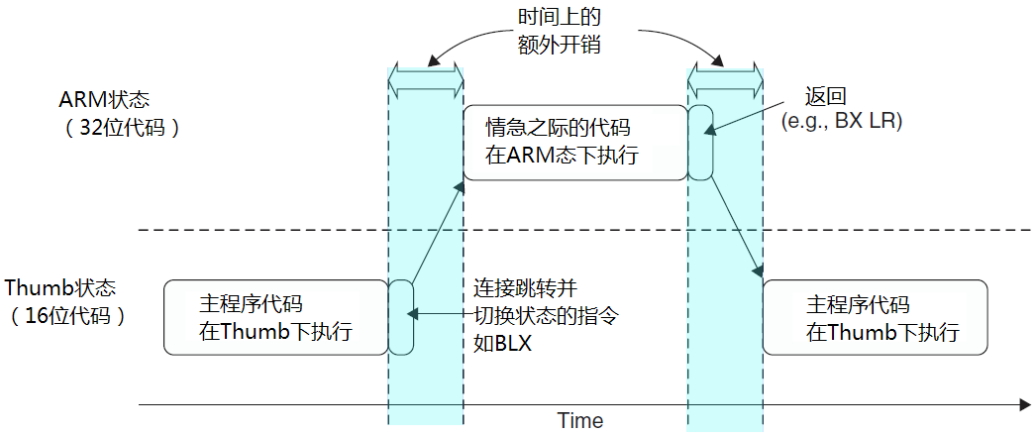


图 2.7 在诸如 ARM7 处理器上的状态切换模式图

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。

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。

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 DPs 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 CoreSignt 产品家族的 JTAG-DP 模块。这下就有 3 个 DPs 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“指令追踪宏单元（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。

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。

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。

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）。

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。

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。

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。

第3章

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

寄存器组

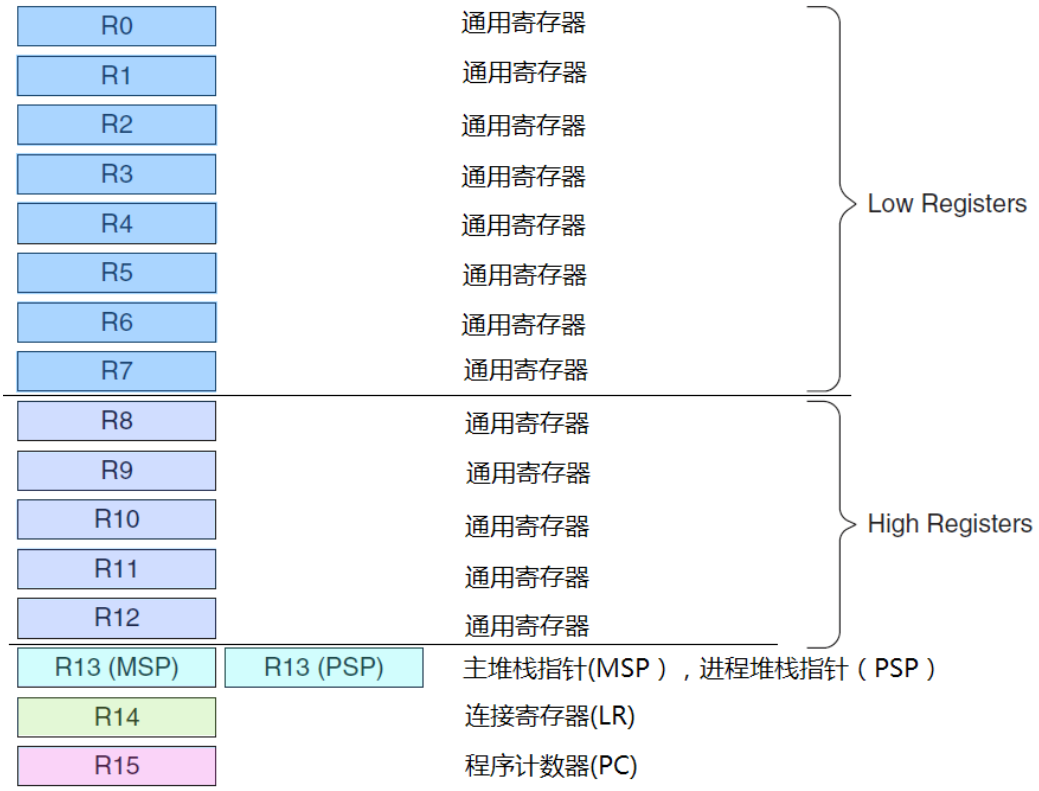
如我们所见，CM3 拥有通用寄存器 R0-R15 以及一些特殊功能寄存器。R0-R12 是最“通用目的”的，但是绝大多数的 16 位指令只能使用 R0-R7（低组寄存器），而 32 位的 Thumb-2 指令则可以访问所有通用寄存器。特殊功能寄存器有预定义的功能，而且必须通过专用的指令来访问。

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。

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。



特殊功能寄存器：

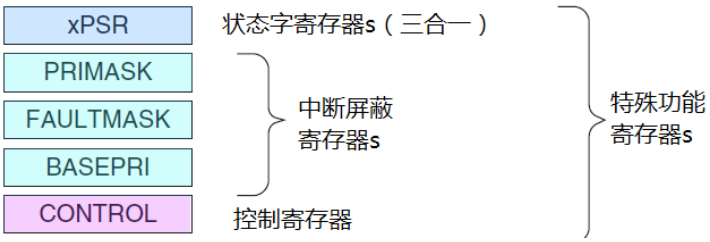


图 3.1 Cortex-M3 的寄存器组

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，你引用到的是当前正在使用的那一个，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）。

要注意的是，并不是每个应用都必须用齐两个堆栈指针。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，以及一个栈顶指针组成，用于实现“先进后出”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。

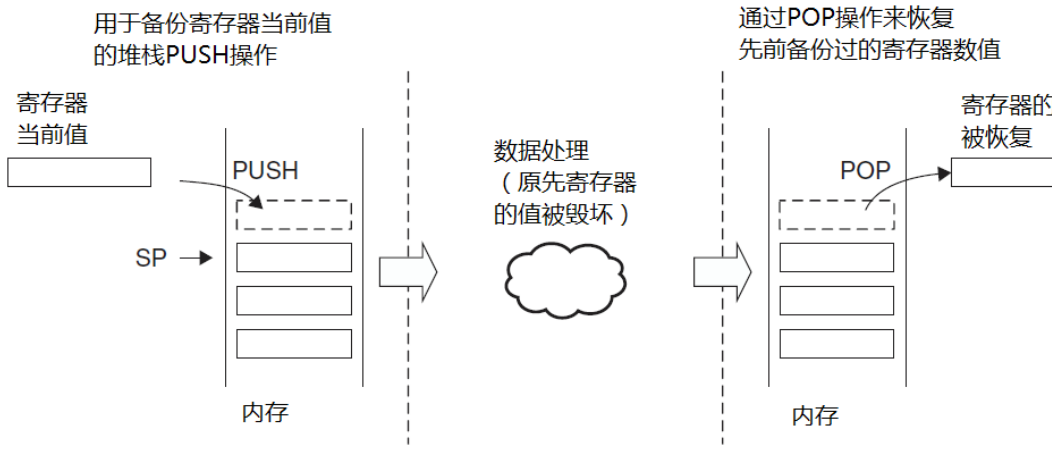


图 1.2 堆栈内存的基本概念

在执行 PUSH 和 POP 操作时，那个通常被称为 SP 的地址寄存器，会自动被调整，以避免后续的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。它俩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，ortex-M3 中的堆栈以这种方式来使用的，这就是所谓的“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）。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
    …                             ; 执行处理
    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，你可以使用 SP 表示 R13。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,……。这样一来，R13 的最低两位被硬线连接到 0,并且总是读出 0（Read As Zero）。

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。

程序计数器 R15

R15 是程序计数器，在汇编代码中你也可以使用名字“PC”来访问它。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。

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 和 MRS 指令访问，而且它们也没有存储器地址。

```
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
```

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”。

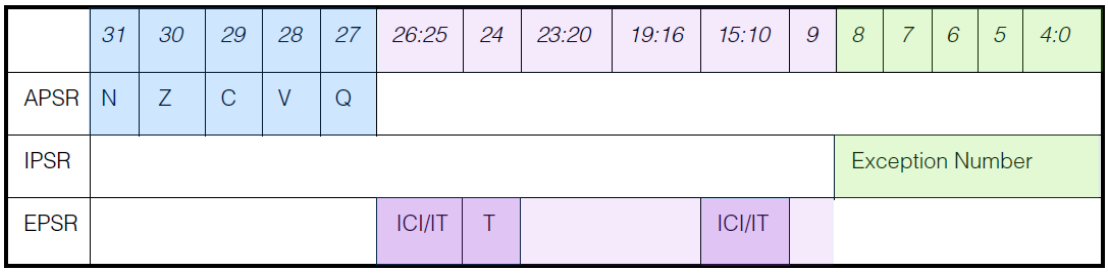


图 3.3 Cortex-M3 中的程序状态寄存器（xPSR）

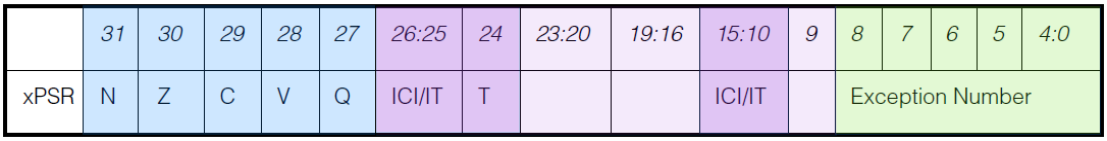


图 3.4 合体后的程序状态寄存器(xPSR)

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 s

名字	功能描述
PRIMASK	这是个只有 1 个位的寄存器。当它置 1 时，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，包括中断和 fault，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，PRIMASK 和 BASEPRI 对于暂时关闭中断是非常重要的。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。

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
```

控制寄存器（CONTROL）

控制寄存器用于定义特权级别，还用于选择当前使用哪个堆栈指针。

表 3.3 Cortex-M3 的 CONTROL 寄存器

位	功能
CONTROL[1]	堆栈指针选择 0=选择主堆栈指针 MSP（复位后缺省值） 1=选择进程堆栈指针 PSP 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。
CONTROL[0]	0=特权级的线程模式 1=用户级的线程模式 Handler 模式永远都是特权级的。

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。

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
```

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 s 以及调试组件的寄存器 s。除此之外，还禁止使用 MSR 访问刚才讲到的特殊功能寄存器——除了 APSR 有例外。谁若是以身试法，则将 fault 伺候。

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，异常返回后将回到产生异常之前的特权级。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。

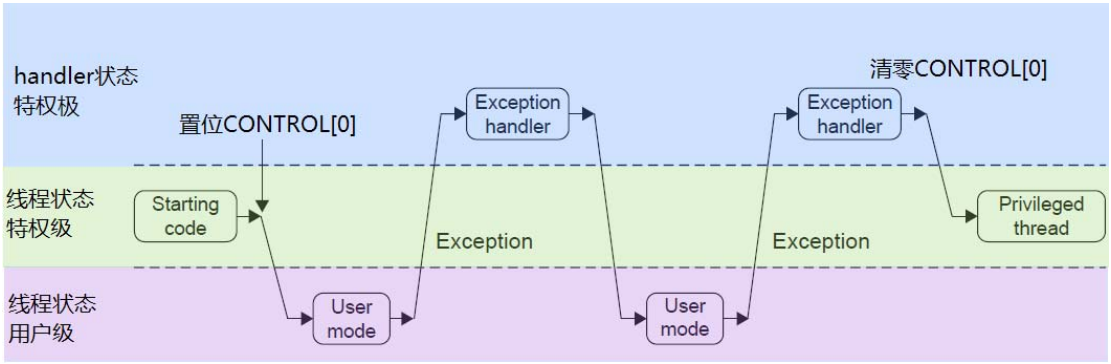


图 3.7 特权级和处理器模式的改变图

把代码按特权级和用户极分开对待，有利于使架构更加安全和健壮。例如，当某个用户代码出问题时，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，你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全自动的，就在出入异常服务例程时由硬件处理。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。

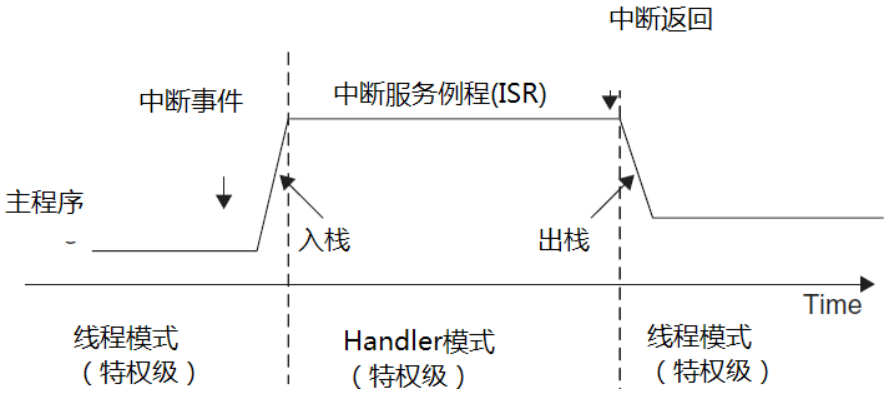


图 3.8 中断前后的状态转换

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极都要发生变化，如下图所示。

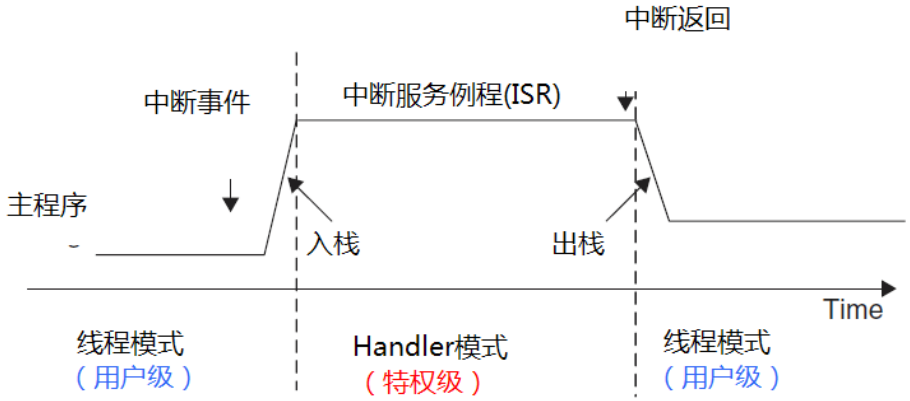


图 3.9 中断前后的状态转换＋特权等级切换

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选择修改 CONTROL[0]。

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 FAULTMASK 被置位。
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。

向量表 s

当一个发生的异常被 CM3 内核接受，对应的异常 handler 就会执行。为了决定 handler 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异常 handler 的入口地址。向量表的存储位置是可以设置的，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	表项地址偏移量	异常向量
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 会计算出偏移移量是 11x4=0x2C，然后从那里取出服务例程的入口地址并跳入。0 号异常的功能则是个另类，它并不是什么入口地址，而是给出了复位后 MSP 的初值。

栈内存操作

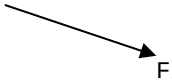
在 Cortex-M3 中，除了可以使用 PUSH 和 POP 指令来处理堆栈外，内核还会在异常处理的始末自动地执行 PUSH 与 POP 操作。本节让我们来检视一下具体的动作，第 9 章则讨论异常处理时的自动栈操作。

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，但是其地址由 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放入内存，以便将来能恢复之——当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 指针的值也根着自减/自增。

```
…（主程序）
; R0=X, R1=Y, R2=Z
BL      Fx1
```



```
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 的值没有被改变）
```

图 3.10 基本的堆栈操作：每次处理单个寄存器

译者添加：

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     {R0-R2}          ;弹出 R0-R2
POP     {R3-R5,R8, R12}  ;弹出 R3-R5，R8，以及 R12
```

注意：不管在寄存器列表中，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 PUSH 指令按照从大到小的顺序依次入栈，POP 则按从小到大的顺序依次出栈。如果不按升序写寄存器，有些汇编器可能会给出一个语法错误。

PUSH/POP 对子还有这样一种特殊形式，形如

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

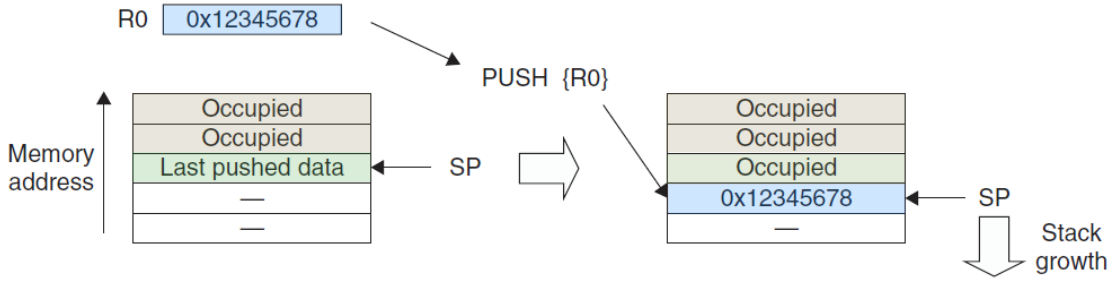
请注意：POP 的最后一个寄存器是 PC，并不是先前 PUSH 的 LR。这其实是一个返回的小技巧。因为总要把先前 LR 的值弹出来，再使用此值返回，干脆绕过 LR，直接传给 PC！那不怕 LR 的值没有被恢复吗？不怕，因为 LR 在子程序返回时的唯一用处就是提供返回地址，在返回后，先前保存的返回地址就没有利用价值了，所以只要 PC 得到了正确的值，不恢复也没关系。

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，后续章节讲到指令系统时再介绍。

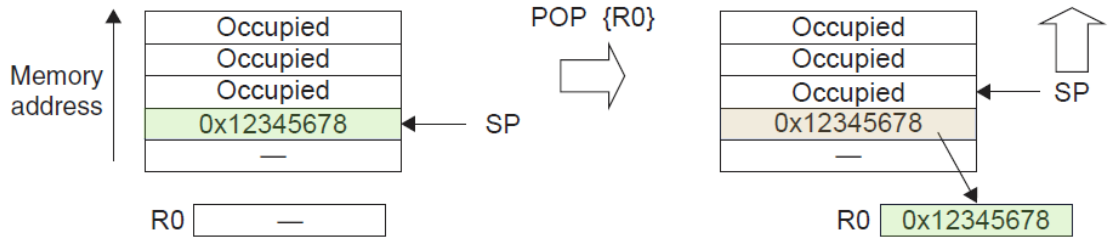
图 3.10 中的子程序返回后，R0-R2 的值仍然是执行前的——仿佛什么事都没有发生一样。

Cortex-M3 堆栈的实现

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。



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。



译注[9]：虽然 POP 后被压入的数值还保存在栈中，但它已经无效了，因为为下次的 PUSH 将覆盖它的值！

在进入 ISR 时，CM3 会自动把一些寄存器压栈，这里使用的是进入 ISR 之前使用的 SP 指针（MSP 或者是 PSP）。离开 ISR 后，只要 ISR 没有更改过 CONTROL[1]，就依然使用先前的 SP 指针来执行出栈操作。

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。

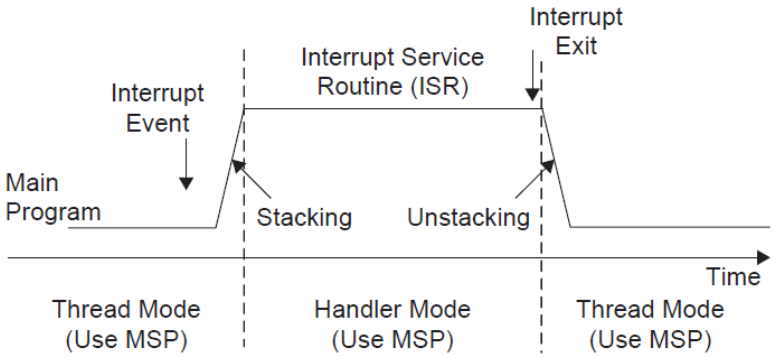


图 3.15 CONTROL[1]=0 时的堆栈使用情况

当 CONTROL[1]=1 时，线程模式将不再使用 PSP，而改用 MSP（handler 模式永远使用 MSP）。

［译注 10］：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。

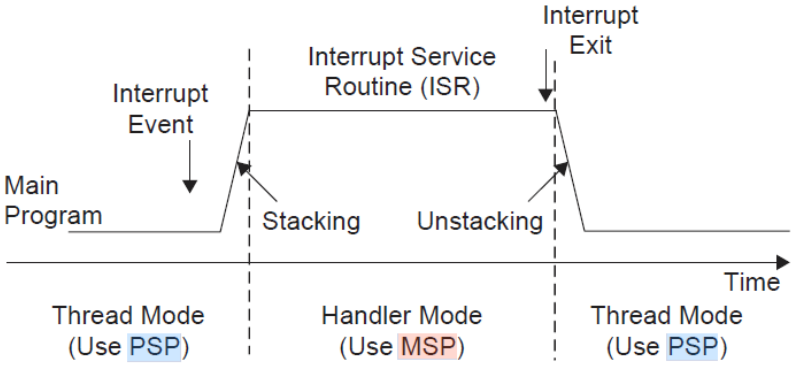


图 3.16 CONTROL[1]=0 时的堆栈切换情况

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      ; 写入 R0 的值到主堆栈中
MRS     R0,     PSP     ; 读取进程堆栈指针到 R0
MSR     PSP,    R0      ; 写入 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。

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。
-

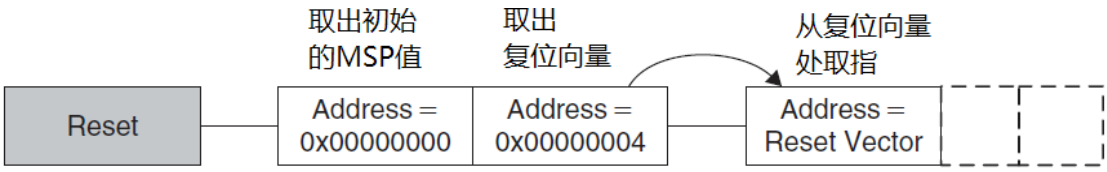


图 3.17 复位序列

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，0 地址处提供 MSP 的初始值，然后就是向量表（向量表在以后还可以被移至其它位置）。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。

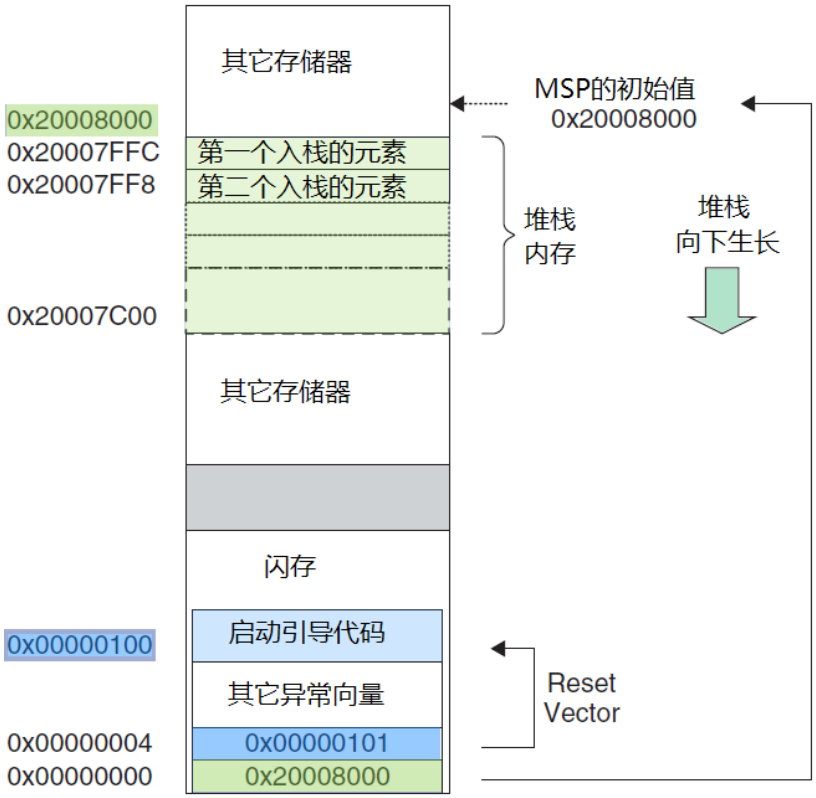


图 3.18　初始 MSP 及 PC 初始化的一个范例

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 条指令还没执行就会被 NMI 或是其它 fault 打断。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。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。

第4章

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 中还有一个快速的查阅参考。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。

汇编语言基础

为了给以后的学习扫清障碍，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。

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，通常都给出本指令执行结果的存储地。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操作，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_SETEN0     EQU     0xE000E100
NVIC_IRQ0_ENABLE    EQU     0x1
    …
    LDR     R0, =NVIC_IRQ_SETEN0    ;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       ; 把立即数传送到指令中
    STR     R1, [R0]                ; *R0=R1，执行完此指令后 IRQ #0 被使能。
```

注意：常数定义必须顶格写

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
```

(DCI 也必须空格写——译注)

类似地，你还可以使用 DCB 来定义一串字节常数——允许以字符串的形式表达，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。

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 中的标志 s，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=Euqal, NE= Not Equal, LT= Less Than, GT= Greater Than。还有若干个其它的条件。例如： BEQ <Label> ；仅当 EQ 满足时转移

在 Cortex-M3 中，对条件后缀的使用有限制，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。

汇编语言：统一的汇编语言

为了最有力地支持 Thumb-2，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以相同的语法格式书写，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   ; UAL 语法允许的等值写法（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，你可以让汇编器决定用哪个，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 位指令以缩小代码体积，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，但如果没有给出后缀，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，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，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 指令则无任何限制。不过，把 R15(PC)作为目的寄存器却很容易走火入魔——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。对 PC 的使用额外的戒律。如果感兴趣，可以参考《ARMv7-M 架构应用级参考手册》。

指令集

Cortex-M3 支持的指令在表 4.2-表 4.9 列出。其中

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

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多格——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	加载多个字，并且在加载后自增基址寄存器
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。如果调试被使能，则进入调试状态（停机）。或者如果调试监视器异常被使能，则调用一个调试异常，否则调用一个 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——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底 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 位整数先用 2 进制表达，再旋转 180 度——译注）
REV	对一个 32 位整数做按字节反转
REVH/ REV16	对一个 32 位整数的高低半字都执行字节反转
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	带进位的逻辑右移一格（最高位用 C 填充，且不影响 C 的值——译注）
SFBX	从一个 32 位整数中提取任意的位段，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多个字到若干寄存器
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
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	在本地的处理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）

未支持的指令

有若干条 Thumb 指令没有被 Cortex-M3 支持，下表列出了没有被支持的指令，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 态下运行，故以此指令为代表的，凡是试图切入 ARM 态的操作，都将引发一个用法 fault。
~~SETEND~~	由 v6 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

有少量在 ARMv7-M 中列出的指令不被 CM3 支持。其中 v7M 允许 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
~~MCRR~~	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
~~MRRC~~	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至协处理器中
~~STC~~	从协处理器中传送一串数值到地址连续的一段地址空间中

改变处理器状态指令（CPS）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 v6 中定义的某些位在 CM3 中不存在。

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
~~PLI~~	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
~~YIELD~~	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。

近距离地检视指令

从现在起，我们将介绍一些在 ARM 汇编代码中很通用的语法。有些指令可以带有多种参数，比如预移位操作，但本章并不会讲得面面俱到。

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

- 两个寄存器间传送数据
- 寄存器与存储器间传送数据
- 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 处读取一个字节到 Rd
LDRH Rd, [Rn, #offset]	从地址 Rn+offset 处读取一个半字到 Rd
LDR Rd, [Rn, #offset]	从地址 Rn+offset 处读取一个字到 Rd
LDRD Rd1, Rd2, [Rn, #offset]	从地址 Rn+offset 处读取一个双字(64 位整数)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
LDRD 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。每读一个字后 Rd 自增一次，16 位宽度
STMIA Rd!, {寄存器列表}	存储多个字到 Rd 处。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。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。每读一个字前 Rd 自减一次，32 位宽度
STMIA.W Rd!, {寄存器列表}	存储多个字到 Rd 处。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

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_{efore})或访问后(A_{fter})。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，每存一次曾一次，先存储后自增
STMDB.W R8,     {R0-R3} ;   R8 值的“一个内部复本”先自减后存储，但是 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 ← 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。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]!**	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）

LDRD.W Rd1, Rd2, [Rn, #offset]!	
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]!	

CM3 除了支持“预索引”，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，后索引是忠实使用基址寄存器 Rd 的值作为数据传送的地址的。待到数据传送后，再执行 Rd ← 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 ← 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——相当于有一个“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	

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，则汇编器会给你报错，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
```

看起来令人扫兴，不是吗？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]
```

如果 r3 给出了某 32 位整数数组的下标，则这条指令即可取出该下标处的数组元素。还有一个注意事项：左移的位数只能是 1、2 或者 3。（最常用的就是 2，对应 long 类型）。

PUSH/POP 作为堆栈专用操作，也属于数据传送指令类（具体关系参见译注 13——译者注）。

通常 PUSH/POP 对子的寄存器列表是一致的，但是 PC 与 LR 的使用方式有所通融，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，这些寄存器都是关键部位。因此除了 APSR 在某些场合下可以“露点”之外，其它的都不能“走光”——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，甚至当机。如果以身试法，则 fault 伺候（MemManage fault，如果被除能则“上访”成硬 fault）。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。

程序写多了你就会感觉到，程序中会经常使用立即数。最典型的就是：当你要访问某个地址时，你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，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？当前是要用两条指令来完成了。

如果某指令需要使用 32 位立即数，也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，马上要讲到的一条伪指令能让汇编器来

自动产生这种数组，并且负责计算 offset。这里提到的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，程序很大时可能也需要手工布设(LTORG 指示字)。

不过，为了书写的方便，汇编器通常都支持”LDR Rd, = imm32”伪指令。例如：

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

酷吧！它的名字也是 LDR，但它是伪指令，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，汇编器都在遇到 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。通过组合使用 MOVW 和 MOVT 也能产生 32 位立即数，不过有点麻烦。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。

LDR 伪指令 vs. ADR 伪指令

Both LDR 和 ADR 都有能力产生一个地址，但是语法和行为不同。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       ; R0= 0x4000
    …
address1
0x4000:         DCD     0x0         ;0x4000 处记录的是一个数据
```

ADR 指令则是“厚道人”，它决不会修改 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，故而它的代码效率常常比 LDR 的要高。

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，这里只列出最常用的使用方式。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
```

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`
`ADD Rd, Rm ; Rd += Rm`
`ADD Rd, #imm ; Rd += imm`
	常规加法
	imm 的范围是 im8（16 位指令）或 im12（32 位指令）

`ADC Rd, Rn, Rm ; Rd = Rn+Rm+C`
`ADC Rd, Rm ; Rd += Rm+C`
`ADC Rd, #imm ; Rd += imm+C`
	带进位的加法
	imm 的范围是 im8（16 位指令）或 im12（32 位指令）

`ADDW Rd, #imm12 ; Rd += imm12`	带 12 位立即数的常规加法

`SUB Rd, Rn ; Rd -= Rn`
`SUB Rd, Rn, #imm3 ; Rd = Rn-imm3`
`SUB Rd, #imm8 ; Rd -= imm8`
`SUB Rd, Rn, Rm ; Rd = Rm-Rm`
	常规减法

`SBC Rd, Rm ; Rd -= Rm+C`
`SBC.W Rd, Rn, #imm12 ; Rd = Rn-imm12-C`
`SBC.W Rd, Rn, Rm ; Rd = Rn-Rm-C`
	带借位的减法

`RSB.W Rd, Rn, #imm12 ; Rd = imm12-Rn` `RSB.W Rd, Rn, Rm ; Rd = Rm-Rn`	反向减法
`MUL Rd, Rm ; Rd *= Rm` `MUL.W Rd, Rn, Rm ; Rd = Rn*Rm`	常规乘法
`MLA Rd, Rm, Rn, Ra ; Rd = Ra+Rm*Rn` `MLS Rd, Rm, Rn, Ra ; Rd = Ra-Rm*Rn`	乘加与乘减 （译者添加）
`UDIV Rd, Rn, Rm ; Rd = Rn/Rm （无符号除法）` `SDIV Rd, Rn, Rm ; Rd = Rn/Rm （带符号除法）`	硬件支持的除法

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` `SMLAL RL, RH, Rm, Rn ;[RH:RL]+= Rm*Rn`	带符号的 64 位乘法
`UMULL RL, RH, Rm, Rn ;[RH:RL]= Rm*Rn` `SMLAL RL, RH, Rm, Rn ;[RH:RL]+= Rm*Rn`	无符号的 64 位乘法

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 正巧是 Rd,则按 DataOp 的计算方式来更新。

其中，DataOp 可以是所有“传统”的 32 位数据操作指令，包括：

ADD/ADC/ SUB/SBC/RSB/ AND/ORR/EOR/ BIC/ORN

CM3 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`
`LSL Rd, Rn ; Rd <<= Rn`
`LSL.W Rd, Rm, Rn ; Rd = Rm<<Rn`
	逻辑左移

`LSR Rd, Rn, #imm5 ; Rd = Rn>>imm5`
`LSR Rd, Rn ; Rd >>= Rn`
`LSR.W Rd, Rm, Rn ; Rd = Rm>>Rn`
	逻辑右移

`ASR Rd, Rn, #imm5 ; Rd = Rn »» imm5`
`ASR Rd, Rn ; Rd »» = Rn`
`ASR.W Rd, Rm, Rn ; Rd = Rm»»Rn`
	算术右移

`ROR Rd, Rn ; Rd ⟲ = Rn` `ROR.W Rd, Rm, Rn ; Rd = Rm ⟲ Rn`	圆圈右移

`RRX.W Rd, Rn ; Rd = (Rn>>1)+(C<<31)`
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`
	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

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

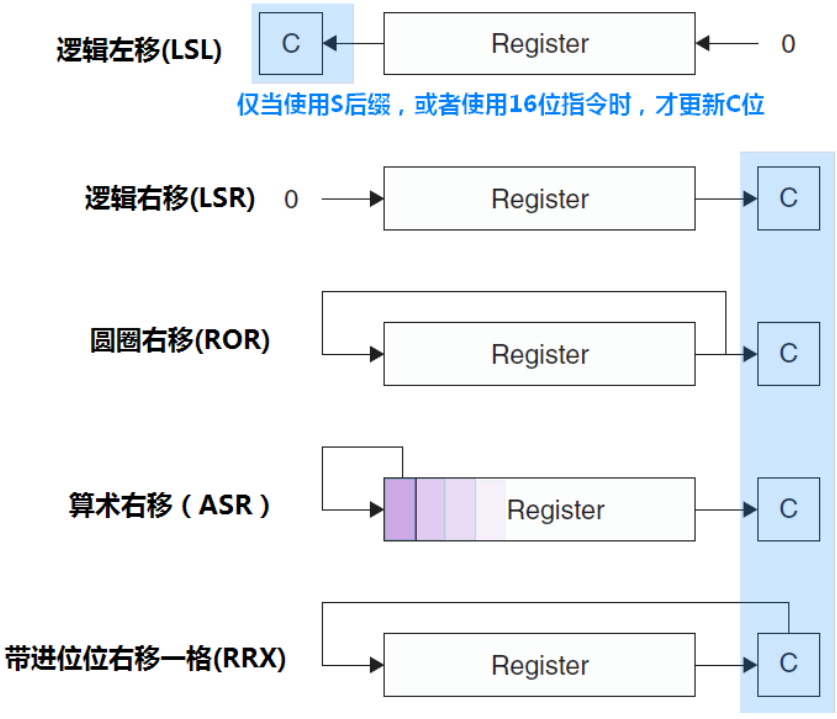


图 4.1 移位与循环指令

> **为啥没有圆圈左移？**
>
> 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。但是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。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，就必须把所有高位全填 1。至于正数或无符号数，则只需简单地把高位清 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 2 个半字拼接成的。有时，需要把这些子元素倒腾倒腾，颠来倒去，如表 4.23 所示

表 4.23 数据序转指令

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，相信看过图 4.2 后就会豁然开朗了：

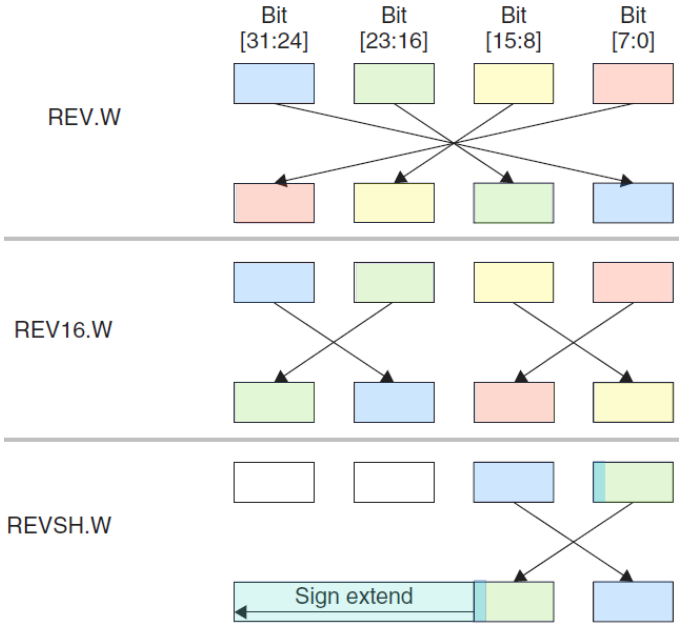


图 4.2 反序操作

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用得比较多，而且在这里面有大量的使用技巧。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	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

汇编语言：子程呼叫与无条件转移指令

最基本的无条件转移指令有两条：

```
B       Label       ;转移到 Label 处对应的地址
BX      reg         ;转移到由寄存器 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)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 fault 伺候。

呼叫子程序时，需要保存返回地址，正点的指令是：

```
BL      Label       ;转移到 Label 处对应的地址，并且把转移前的下条指令地址保存到 LR
BLX     reg         ;转移到由寄存器 reg 给出的地址，根据 REG 的 LSB 切换处理器状态，
                    ;并且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常人想不到的绝活，常见形式有：

```
MOV     PC,     R0          ;转移地址由 R0 给出
LDR     PC,     [R0]        ;转移地址存储在 R0 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，
                            ;从而实现转移（这也是 OS 惯用的一项必杀技——译注）
LDMIA   SP!,    {…, PC}     ;POP 的另一种等效写法
```

同理，使用这些密技，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"。

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）。

担任条件转移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 种转移判据，如下表 4.26 所示

表 4.26 转移及条件执行判据

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 || 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（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
ITTET   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——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，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
```

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，因为有了 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。

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，它们的目的就是更新标志位，因此是会影响标志位的，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，但是它无条件更新标志位。它的用法和 CMP 的相同：

```
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，但是它无条件更新标志位。它的用法和 CMP 的相同：

```
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
```

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，如果存储器的映射关系，或者内存保护区的设置可以在运行时更改，（通过写 MMU/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 MMU/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。

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都执行完毕后，才提交(commit)在它后面的存储器访问动作。
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都执行完毕后，才执行它在后面的指令（亦即任何指令都要等待——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——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，通通清洗缓冲区。大虾们可以在有绝对信心时使用 DMB，新手还是保险点好。

同 DMB/DSB 相比，ISB 指令看起来似乎最小白。不过它还有其它的用场——对于高级底层技巧：“自我更新”(self-mofifying)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。

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“饱和削顶失真”，理解饱和运算就更加容易。

CM3 中的饱和运算指令分为两种：一种是“没有直流分量”的饱和——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。

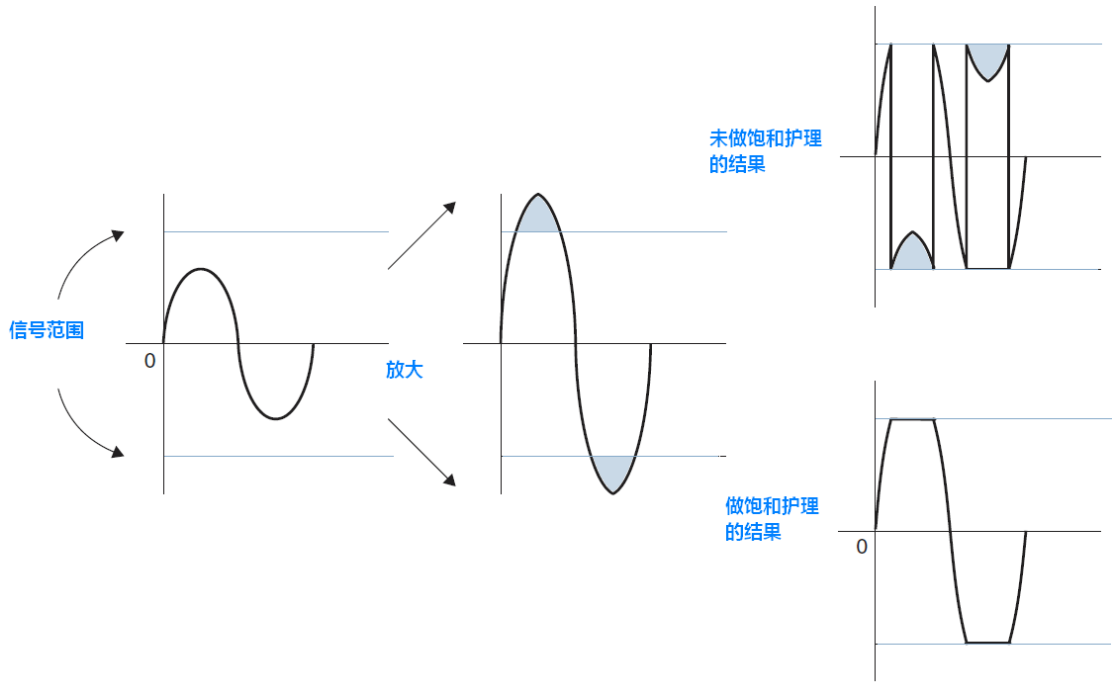


图 4.3　带符号饱和运算

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，这也是 APSR“露点”的部位。

Rn 存储“放大后的信号”，（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)	0xFFFFF800(-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

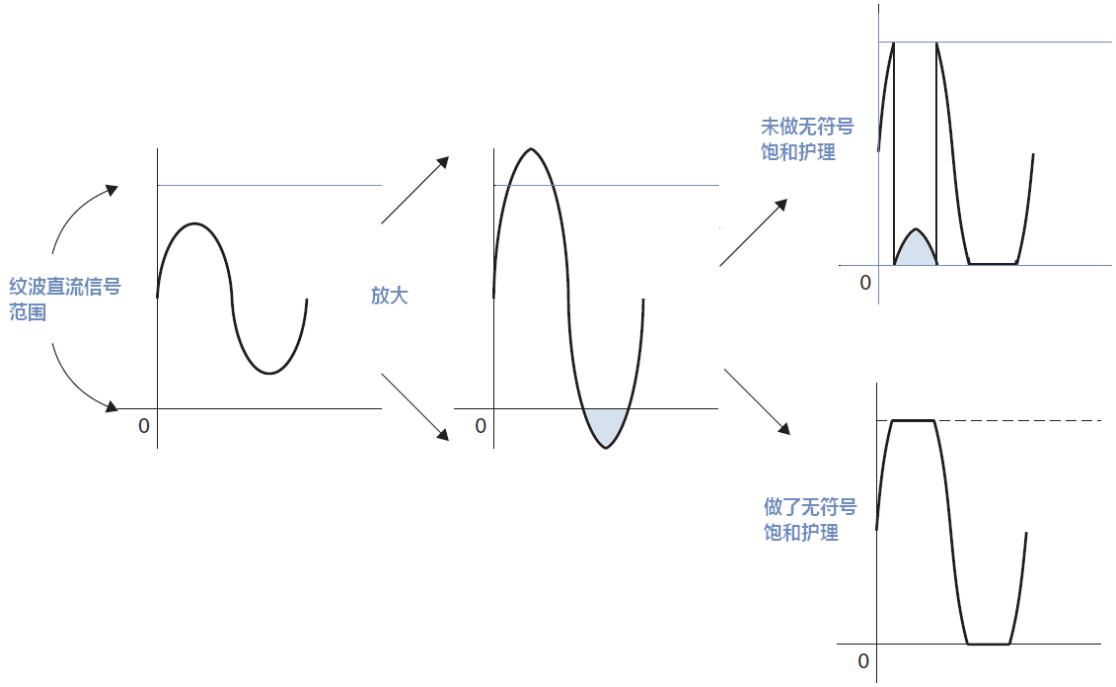


图 4.4　无符号饱和运算

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)	0xFFF(4095)	1
0xFFF(4095)	0xFFF(4095)	无变化
0x1000(4096)	0xFFF(4095)	1
0x800(2048)	0x800(2048)	无变化
0	0	无变化
0x80000000(-2G)	0	1
0xFFFFFB32(-1230)	0	1

CM3 中的一些有用的新指令

这里列出几条从 v6 和 v7 开始才支持的最新指令。

MRS 和 MSR

这两条指令是访问特殊功能寄存器的“绿色通道”——当然必须在特权级下，除 APSR 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（同时还包含了 PRIMASK 的功能，因为 faults 的优先级更高）
CONTROL	控制寄存器（堆栈选择，特权等级）

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
```

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 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 除外）。

［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 结构中的小型 if 块

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
```

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。

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。

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=0x**FFFF**9988

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（不信你去写段程序完成它的功能，看看要执行多久）。

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。

SXTB, SXTH, UXTB, UXTH

这 4 个指令是为优化 C 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
```

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，这对于系统程序和单片机程序往往非常有用。

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。

位段不支持首尾拼接。如 BFC R0, #27, #9 将产生不可预料的结果——译者注

例如，

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

执行完后，R0= 0x1234C00F

BFI（位段插入指令），则把某个寄存器按 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=0xFFFFFFFB

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

LDRD/STRD

CM3 在一定程度上支持对 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。

TBB,TBH

高级语言都提供了“分类讨论”式控制结构，如 C 的 switch，Basic 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 的语法格式为：

```
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

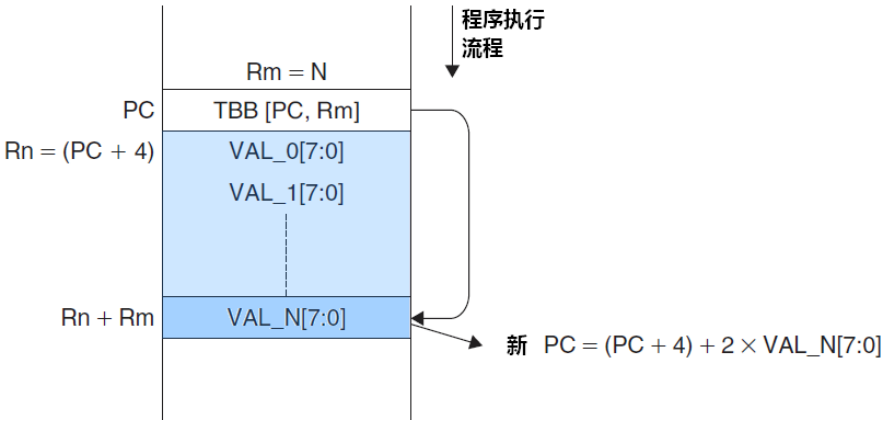


图 4.5 TBB 功能演示

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，甚至在特殊情况下，还要严重违反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格式。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：

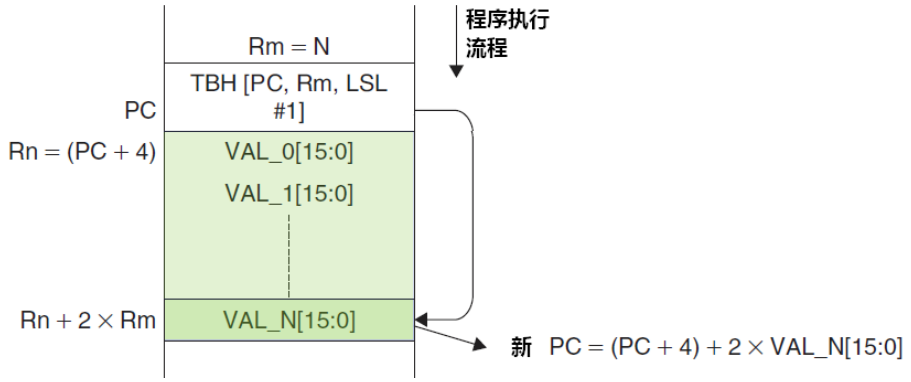


图 4.6 TBH 功能演示

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
```