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

## 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。

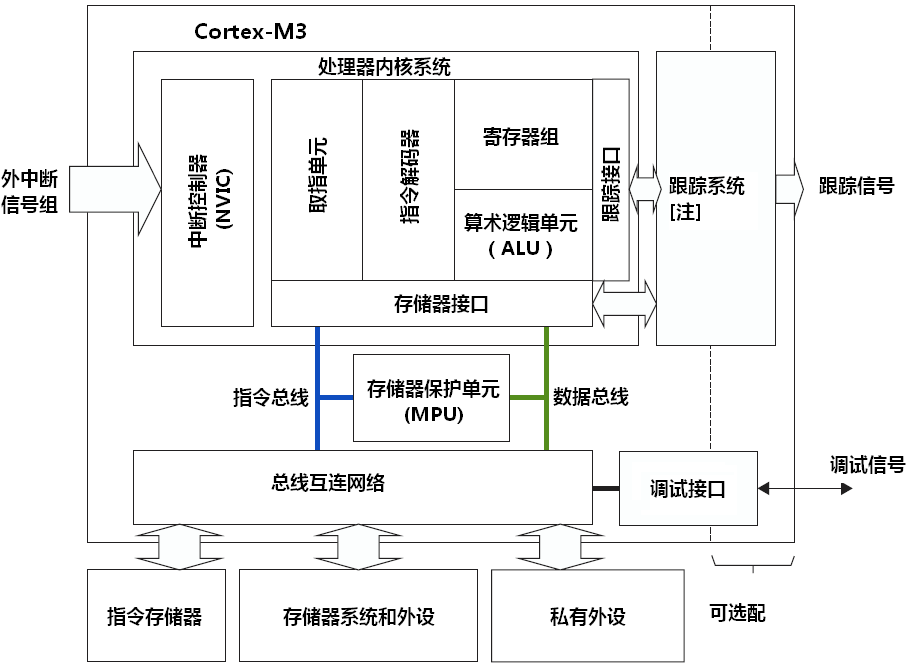


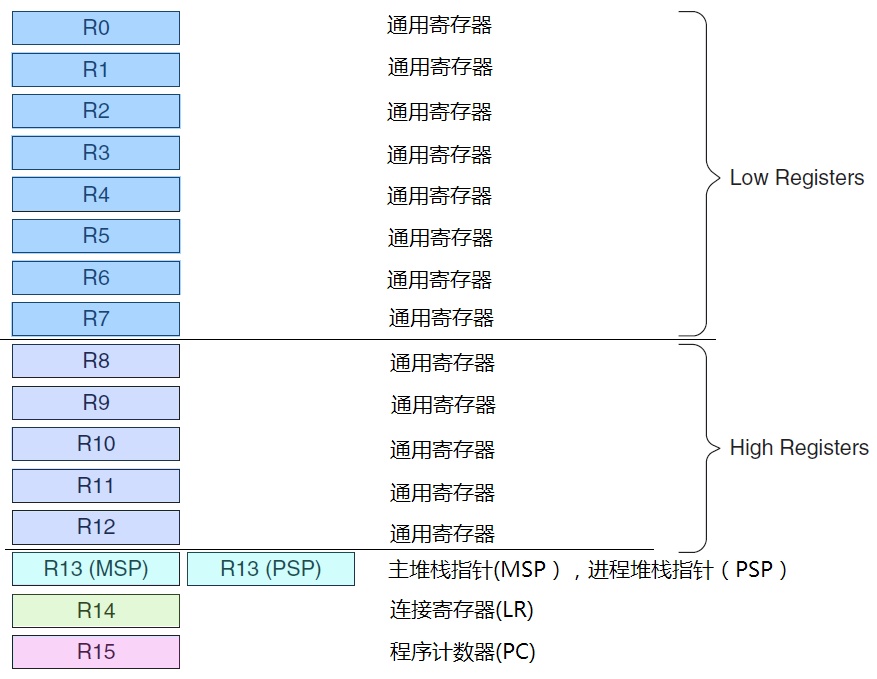
图2.1 Cortex-M3的一个简化视图

［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。

### 2.2.4 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）。

### 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）

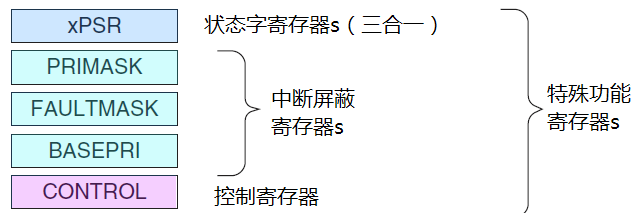


图2.3： Cortex-M3中的特殊功能寄存器集合

表2.1 寄存器及其功能

|  |  |
| --- | --- |
| 寄存器 | 功能 |
| xPSR | 记录ALU标志（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。

## 2.3 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。

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”。

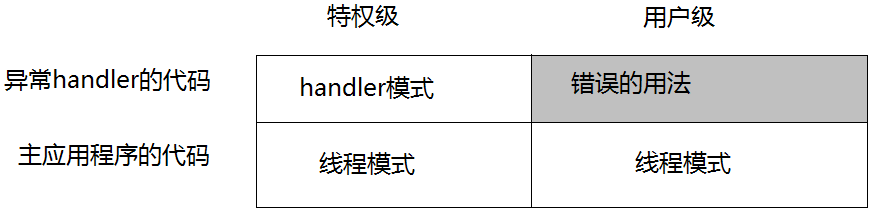


图2.4 Cortex-M3下的操作模式和特权级别

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）。

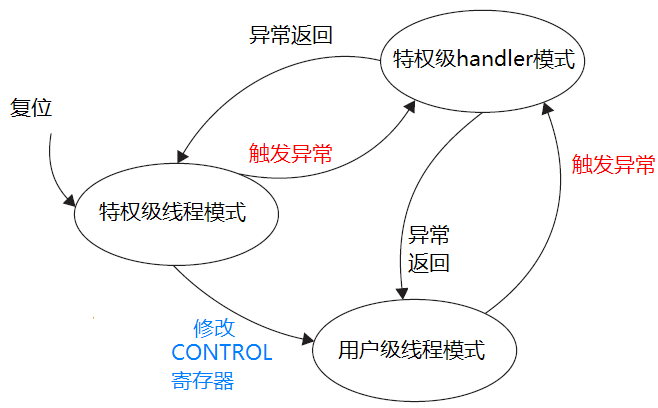


图2.5 合法的操作模式转换图

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。

## 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。

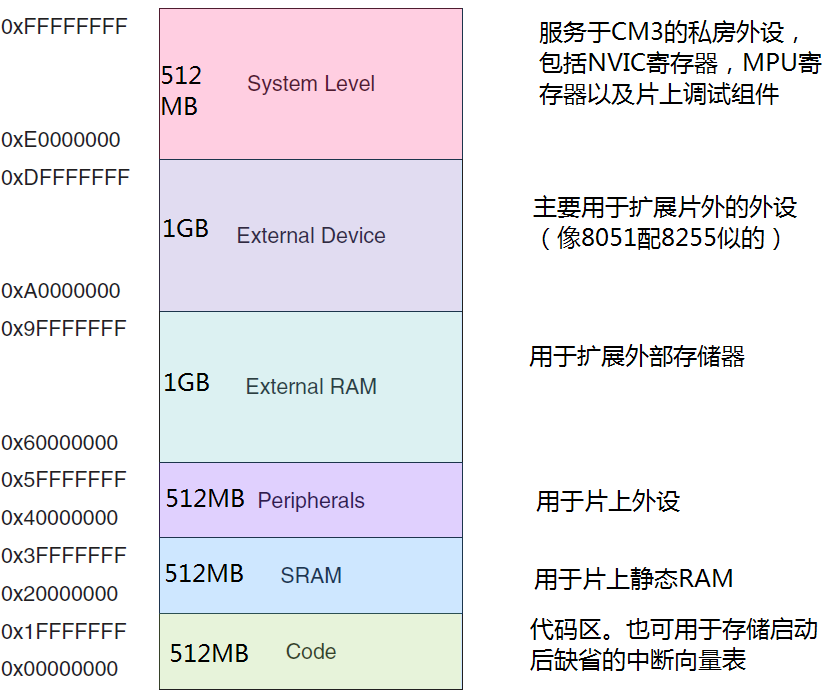
### 2.4.5 中断可屏蔽

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。

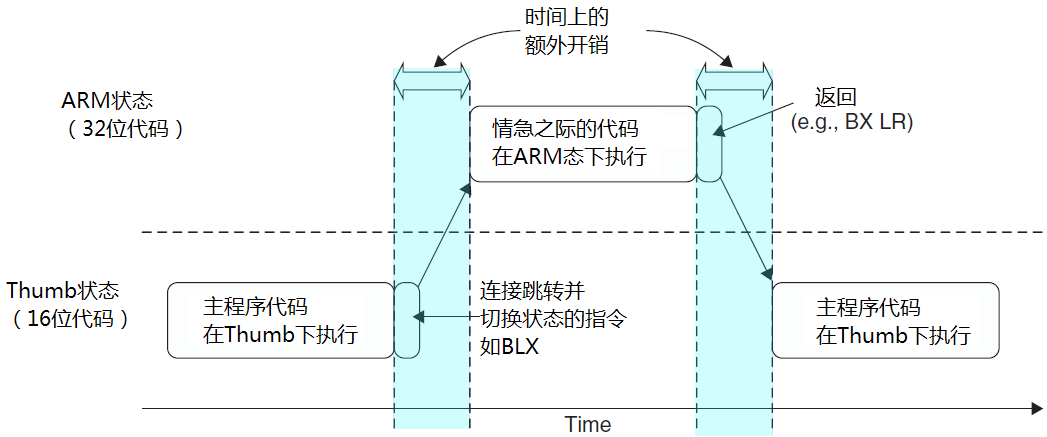


图2.7 在诸如ARM7处理器上的状态切换模式图

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“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－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。