第7章

# 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)  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） |

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级抢占（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：



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：

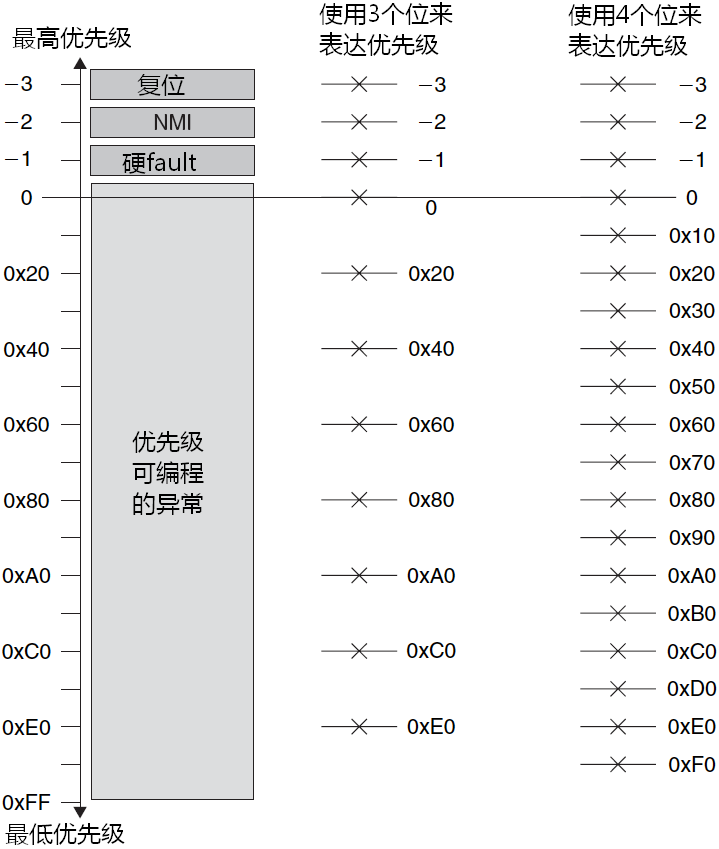


图7.3 3位表达的优先级 vs. 4位表达的优先级

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。

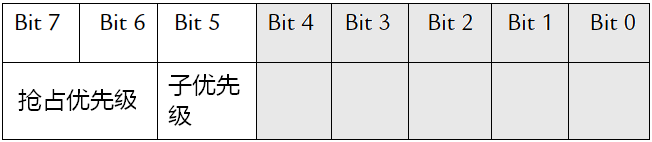


图7.4 3位优先级，从比特5处分组时，优先级位段的划分

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。

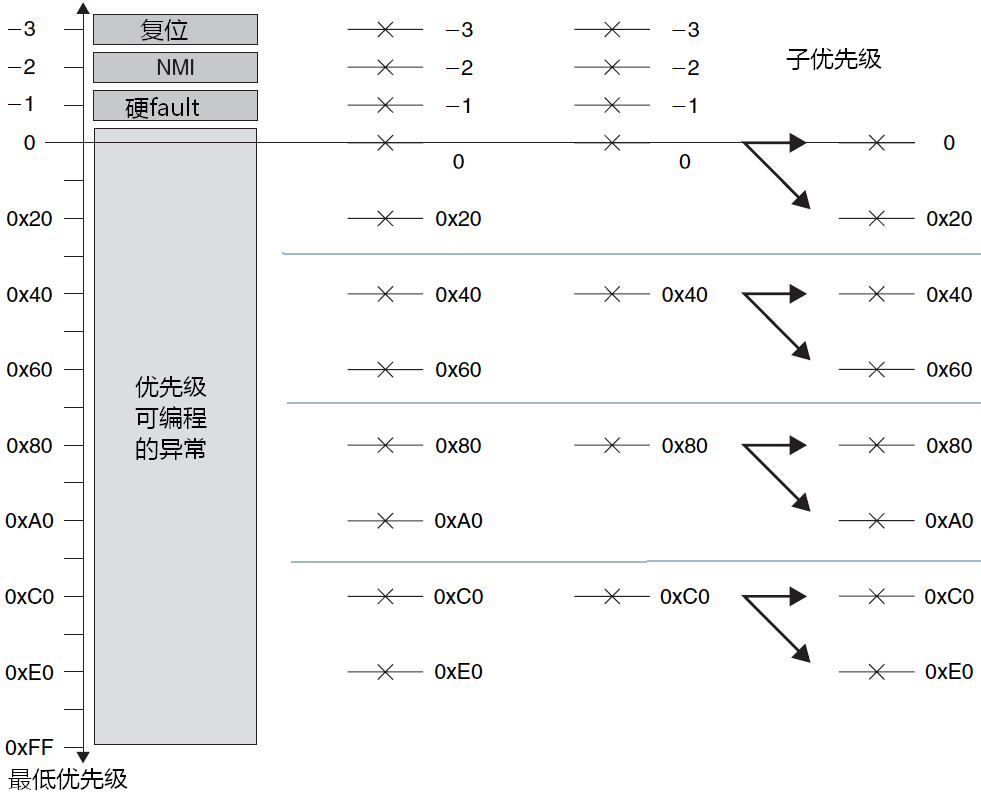


图7.5 三位优先级，从比特5处分组

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。

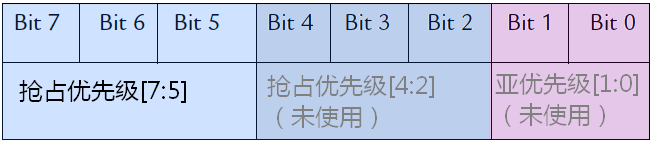


图7.6 3位优先级，从比特1处分组

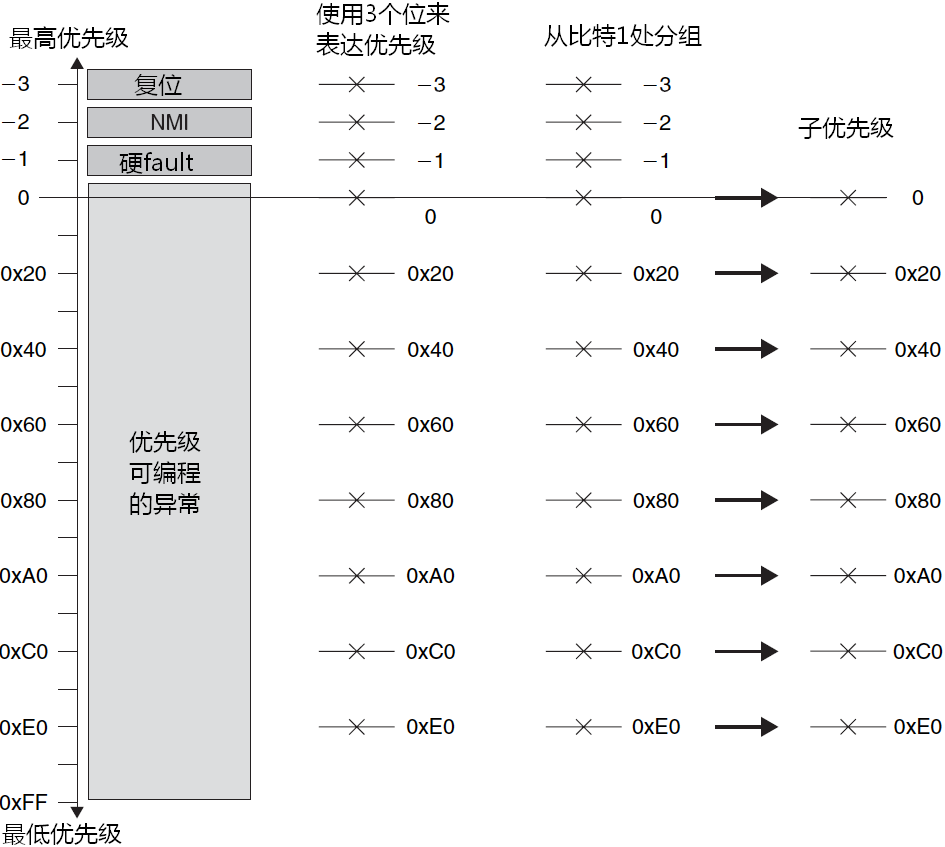


图7.7 3位优先级，从比特1处分组，详细情况

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一个。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。

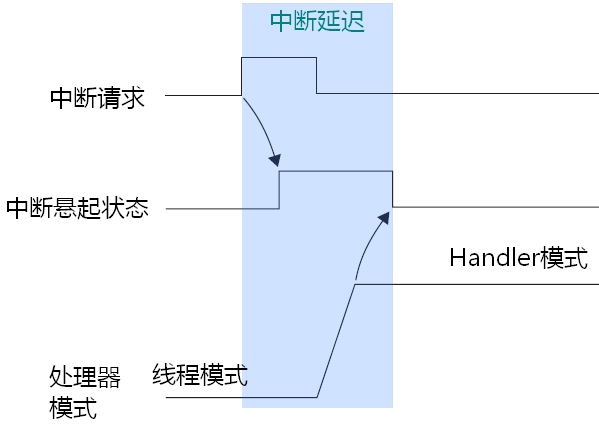


图7.8 中断悬起示意图

当中断输入脚被置为有效（asser）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。

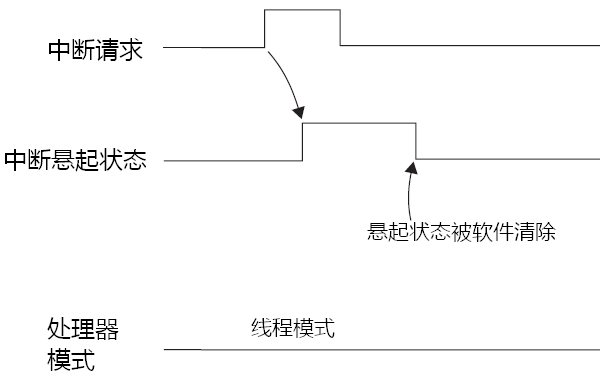


图7.9 中断在得到处理器响应之前被清除悬起状态

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）。

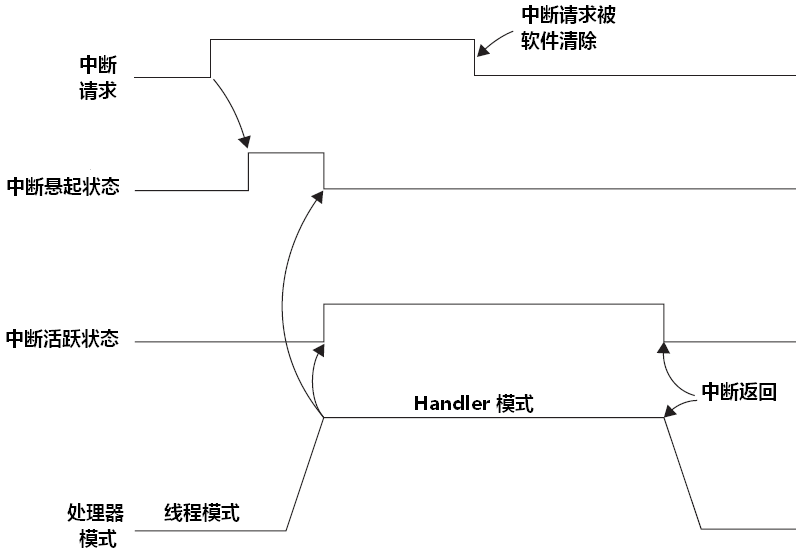


图7.10 在处理器进入服务例程后对中断活跃状态的设置

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。

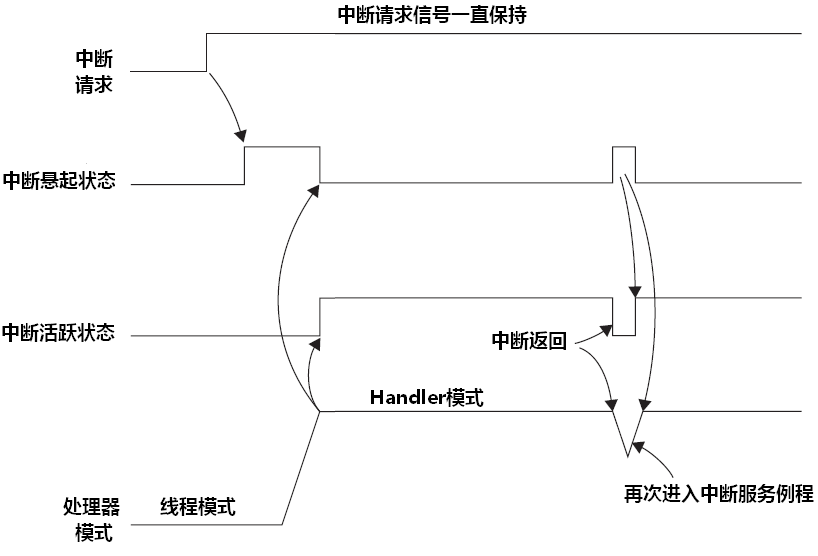


图7.11 一直维持的中断请求导致服务例程返回后再次悬起该中断

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。

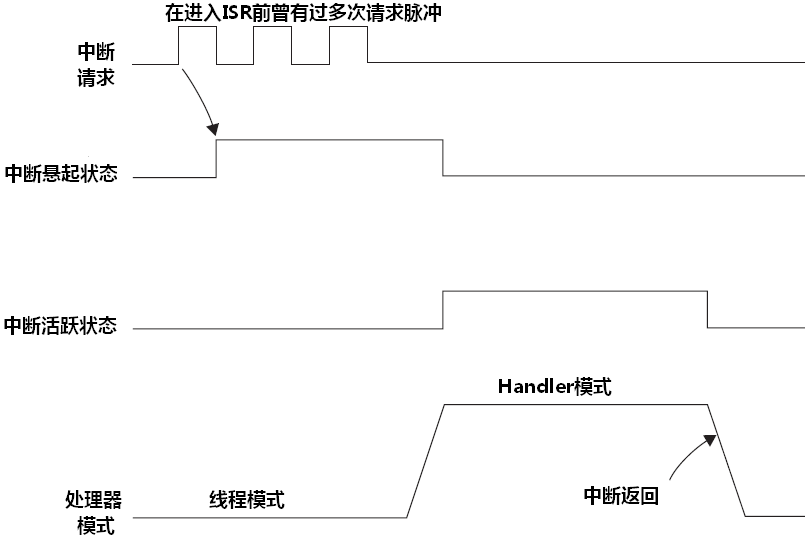


图7.12 中断请求过快导致一部分请求错失的情况

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。

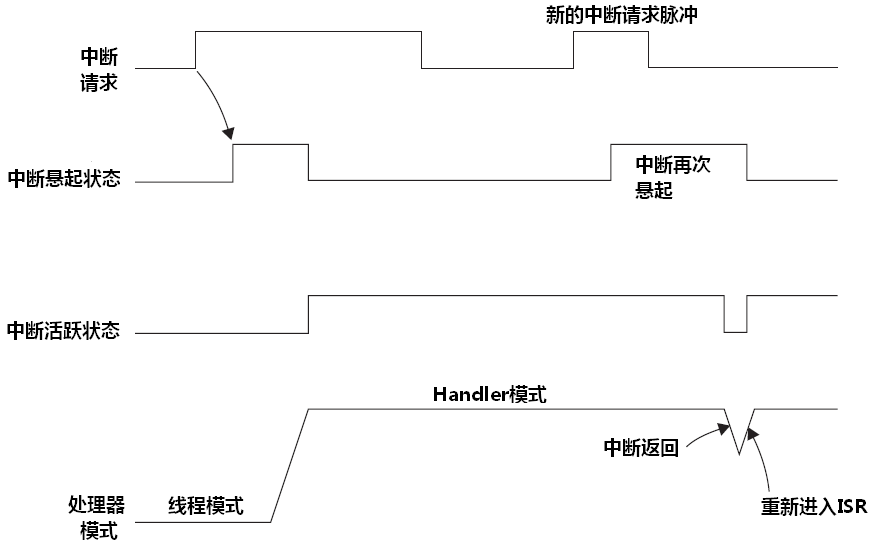


图7.13 在执行ISR时中断悬起再次发生

## 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（取向量的除外），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。

由取指和堆栈操作产生的fault总是精确的。

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MemMange 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MemMange 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

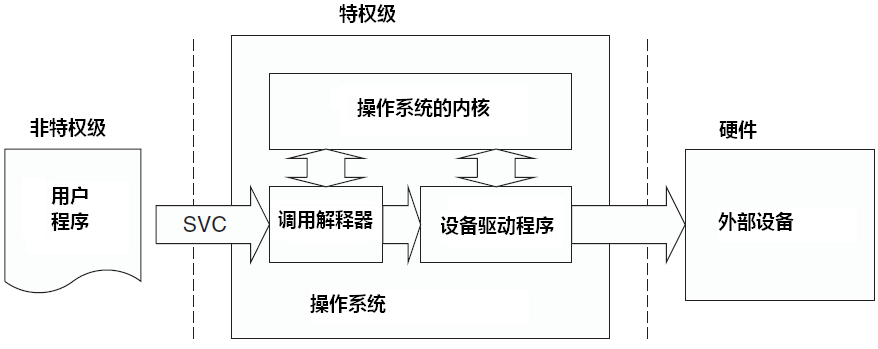


图7.14 SVC作为操作系统函数门户示意图

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。

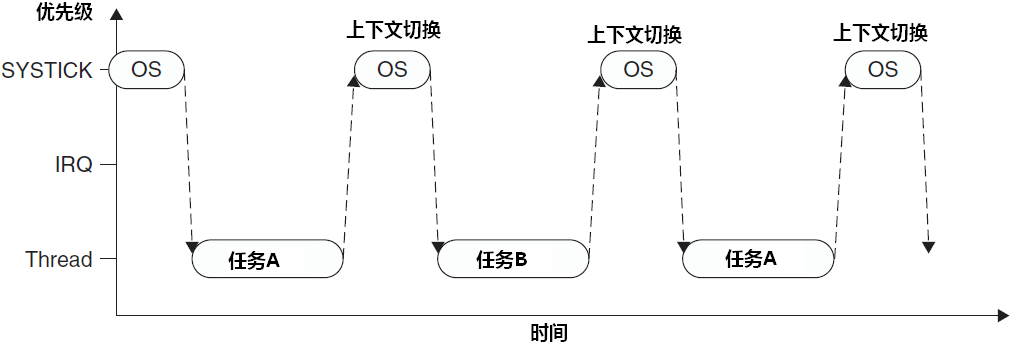


图7.15 两个任务间通过SysTick进行轮转调度的简单模式

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，将触犯用法fault异常。

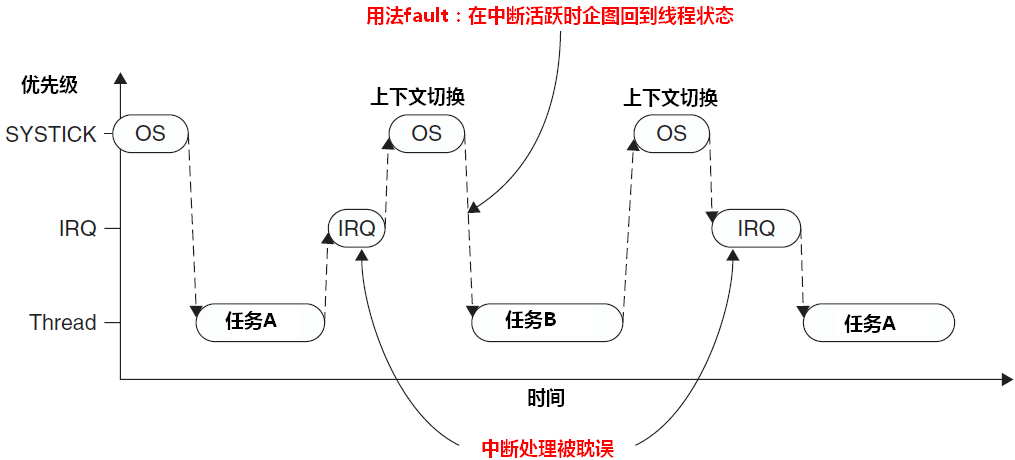


图7.16 发生IRQ时上下文切换的问题

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

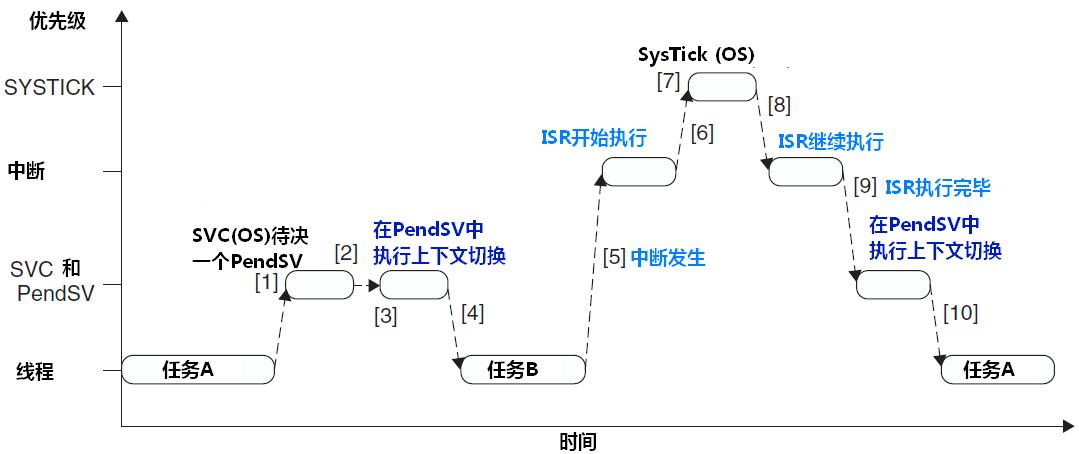


图7.17 使用PendSV控制上下文切换

个中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。