

***Package na***

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

<https://cloud.tencent.com/developer/article/1764431>

# C语言10大基础算法，学C语言必会源码（珍藏版）

# RTT控制台的rt\_kprintf函数的实现和控制台的兼容

### kservice.c

|  |
| --- |
| RT\_WEAK int rt\_kprintf(const char \*fmt, ...)  {  va\_list args;  rt\_size\_t length;  static char rt\_log\_buf[RT\_CONSOLEBUF\_SIZE];  va\_start(args, fmt);  /\* the return value of vsnprintf is the number of bytes that would be  \* written to buffer had if the size of the buffer been sufficiently  \* large excluding the terminating null byte. If the output string  \* would be larger than the rt\_log\_buf, we have to adjust the output  \* length. \*/  length = rt\_vsnprintf(rt\_log\_buf, sizeof(rt\_log\_buf) - 1, fmt, args);  if (length > RT\_CONSOLEBUF\_SIZE - 1)  length = RT\_CONSOLEBUF\_SIZE - 1;  #ifdef RT\_USING\_DEVICE  if (\_console\_device == RT\_NULL)  {  rt\_hw\_console\_output(rt\_log\_buf);  }  else  {  rt\_device\_write(\_console\_device, 0, rt\_log\_buf, length);  }  #else  rt\_hw\_console\_output(rt\_log\_buf);  #endif /\* RT\_USING\_DEVICE \*/  va\_end(args);  return length;  } |

### device.c

|  |
| --- |
| rt\_size\_t rt\_device\_write(rt\_device\_t dev,  rt\_off\_t pos,  const void \*buffer,  rt\_size\_t size)  {  /\* parameter check \*/  RT\_ASSERT(dev != RT\_NULL);  RT\_ASSERT(rt\_object\_get\_type(&dev->parent) == RT\_Object\_Class\_Device);  if (dev->ref\_count == 0)  {  rt\_set\_errno(-RT\_ERROR);  return 0;  }  /\* call device\_write interface \*/  if (device\_write != RT\_NULL)  {  return device\_write(dev, pos, buffer, size);  }  /\* set error code \*/  rt\_set\_errno(-RT\_ENOSYS);  return 0;  } |

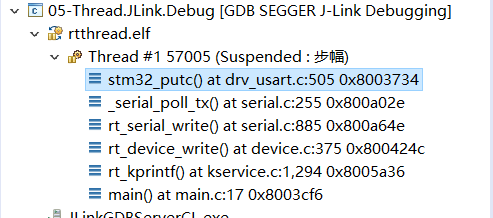
### serial.c

|  |
| --- |
| static rt\_size\_t rt\_serial\_write(struct rt\_device \*dev,  rt\_off\_t pos,  const void \*buffer,  rt\_size\_t size)  {  struct rt\_serial\_device \*serial;  RT\_ASSERT(dev != RT\_NULL);  if (size == 0) return 0;  serial = (struct rt\_serial\_device \*)dev;  if (dev->open\_flag & RT\_DEVICE\_FLAG\_INT\_TX)  {  return \_serial\_int\_tx(serial, (const rt\_uint8\_t \*)buffer, size);  }  #ifdef RT\_SERIAL\_USING\_DMA  else if (dev->open\_flag & RT\_DEVICE\_FLAG\_DMA\_TX)  {  return \_serial\_dma\_tx(serial, (const rt\_uint8\_t \*)buffer, size);  }  #endif /\* RT\_SERIAL\_USING\_DMA \*/  else  {  return \_serial\_poll\_tx(serial, (const rt\_uint8\_t \*)buffer, size);  }  } |

|  |
| --- |
| rt\_inline int \_serial\_poll\_tx(struct rt\_serial\_device \*serial, const rt\_uint8\_t \*data, int length)  {  int size;  RT\_ASSERT(serial != RT\_NULL);  size = length;  while (length)  {  /\*  \* to be polite with serial console add a line feed  \* to the carriage return character 为了礼貌地使用串行控制台，请在回车符中添加换行符  \*/  if (\*data == '\n' && (serial->parent.open\_flag & RT\_DEVICE\_FLAG\_STREAM))  {  serial->ops->putc(serial, '\r');  }  serial->ops->putc(serial, \*data);  ++ data;  -- length;  }  return size - length;  } |

### drv\_uart.c

|  |
| --- |
| static int stm32\_putc(struct rt\_serial\_device \*serial, char c)  {  struct stm32\_uart \*uart;  RT\_ASSERT(serial != RT\_NULL);  uart = rt\_container\_of(serial, struct stm32\_uart, serial);  UART\_INSTANCE\_CLEAR\_FUNCTION(&(uart->handle), UART\_FLAG\_TC);  #if defined(SOC\_SERIES\_STM32L4) || defined(SOC\_SERIES\_STM32F7) || defined(SOC\_SERIES\_STM32F0) \  || defined(SOC\_SERIES\_STM32L0) || defined(SOC\_SERIES\_STM32G0) || defined(SOC\_SERIES\_STM32H7) \  || defined(SOC\_SERIES\_STM32G4)  uart->handle.Instance->TDR = c;  #else  uart->handle.Instance->DR = c;  #endif  while (\_\_HAL\_UART\_GET\_FLAG(&(uart->handle), UART\_FLAG\_TC) == RESET);  return 1;  } |



|  |
| --- |
|  |

# stm32 NVIC 中断 实时系统

### 知识点总结，cm3为例

1、中断也是异常。

2、代码的临界段也称为临界区，指处理时不可分割的代码区域，一旦这部分代码开始执行，则不允许任何中断打断。而临界段代码执行完毕后，要立即打开中断。

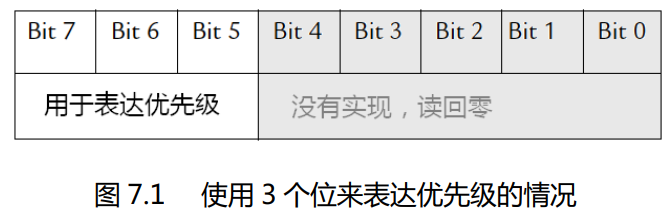
3、如果一个发生的异常不能被即刻响应，就称它被“悬起” (pending)，不过，少数 fault异常是不允许被悬起的

4、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，直到该异常能够执行为止，这与传统的 ARM 是完全不同的。

5、优先级的数值越小，则优先级越高

6、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（但不能编程为负数）**

7、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



8、

* **这种优先级分组规定：亚优先级至少是 1 个位。因此抢占优先级最多是 7 个位，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亚优先级，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。**
* 当然还有凌架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”。

9、

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |

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。如 IRQ #3会比 IRQ #5 先得到响应。

10、

11、***NMI***是Non-Maskable Interrupt(不可屏蔽中断)缩写, 在STM32F4中, NMI被连接到时钟安全系统CCS(Clock security system)异常上。

12、中断的输入和悬起行为。

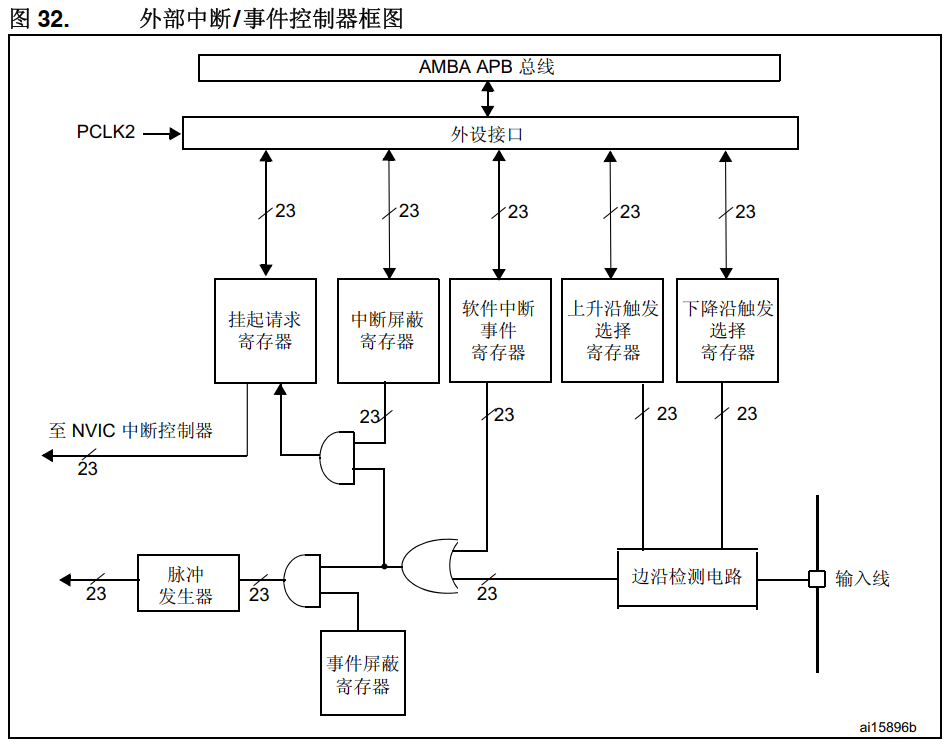
1、这也适用于 NMI，只是 NMI 将会立即无条件执行，除了特殊情况：

2、若当前已经在执行 NMI 服务例程，或者 CPU 被调试器喊停(halted)，或者被一些***严重的系统错误锁定（Lock up），则新的 NMI 请求也将悬起***。

13、

当中断输入脚被 assert 后，该中断就被悬起。即使后来中断源取消了中断请求，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。（中断悬起会被记录下来，直到响应中断）

但是，如果在某个中断得到响应之前，其悬起状态被清除了则中断被取消。（例如，在 ***PRIMASK 或FAULTMASK 置位的时候软件清除了悬起状态标志***），如下的框图所示，***中断屏蔽寄存器最终控制了挂起请求寄存器***。

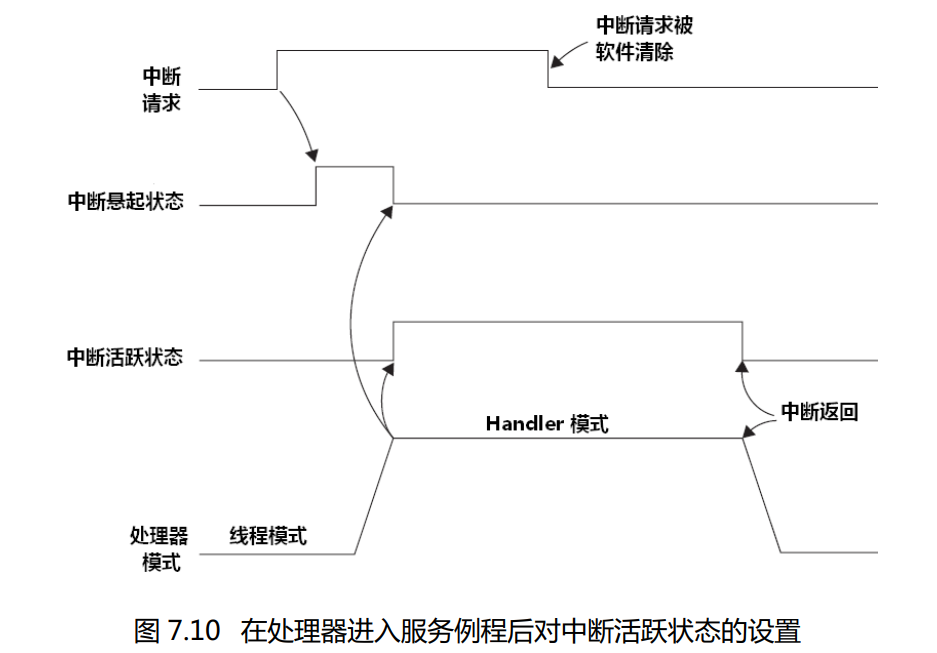


14、

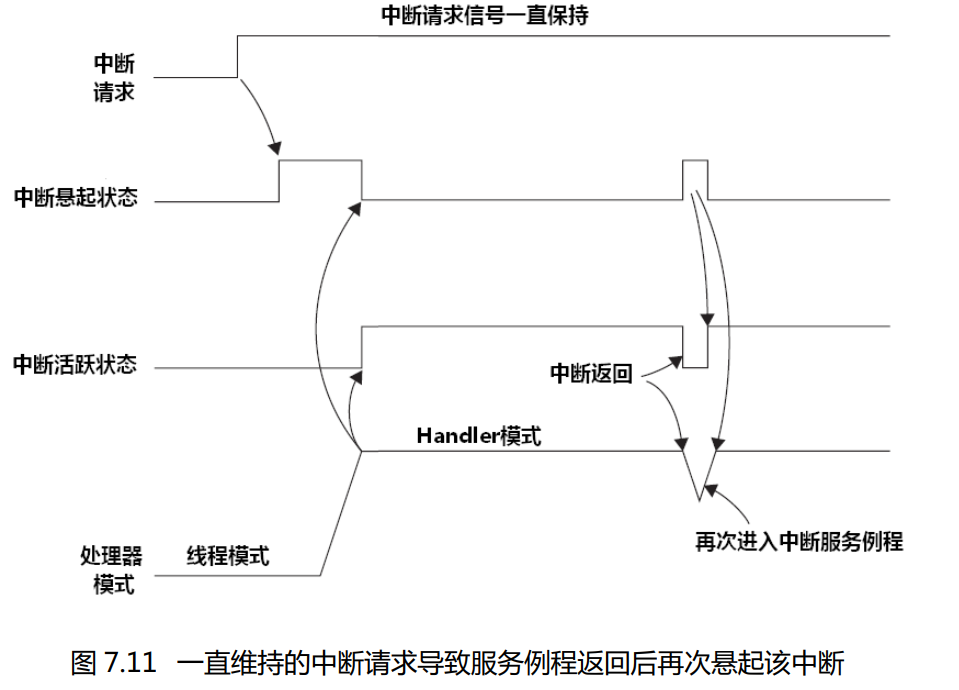
***当某中断的服务例程开始执行时，就称此中断进入了“活跃”状态，并且其悬起位会被硬件自动清除***，如图 7.10 所示。在一个中断活跃后，直到其服务例程执行完毕，并且返回（亦称为中断退出，第九章详细讨论）了，才能对该中断的新请求予以响应（即单实例）。

当然，***新请求的响应亦是由硬件自动清零悬起标志位***。

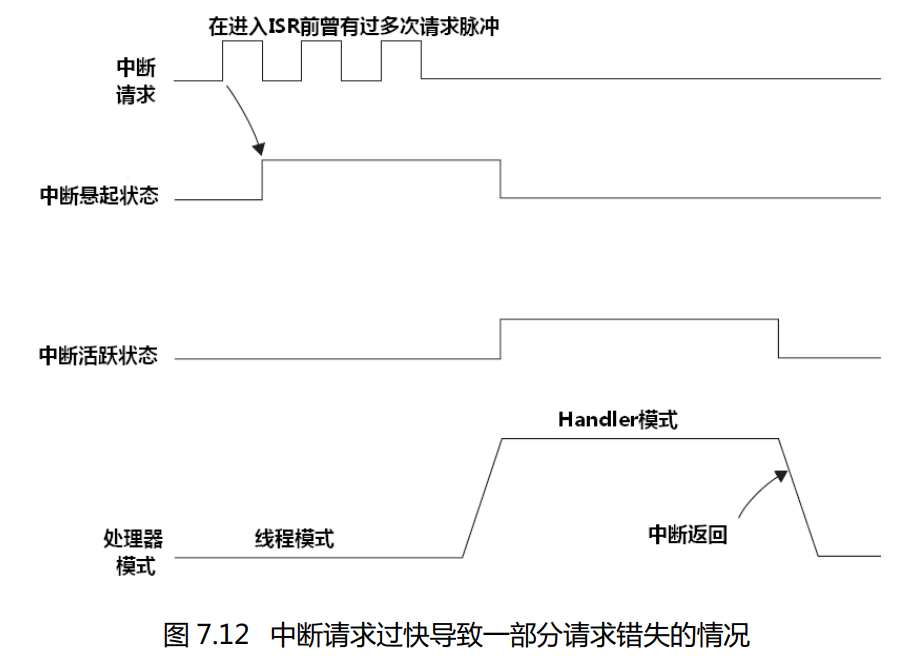
中断服务例程也可以在执行过程中把自己对应的中断重新悬起（使用时要注意避免进入“死循环”——译注）。



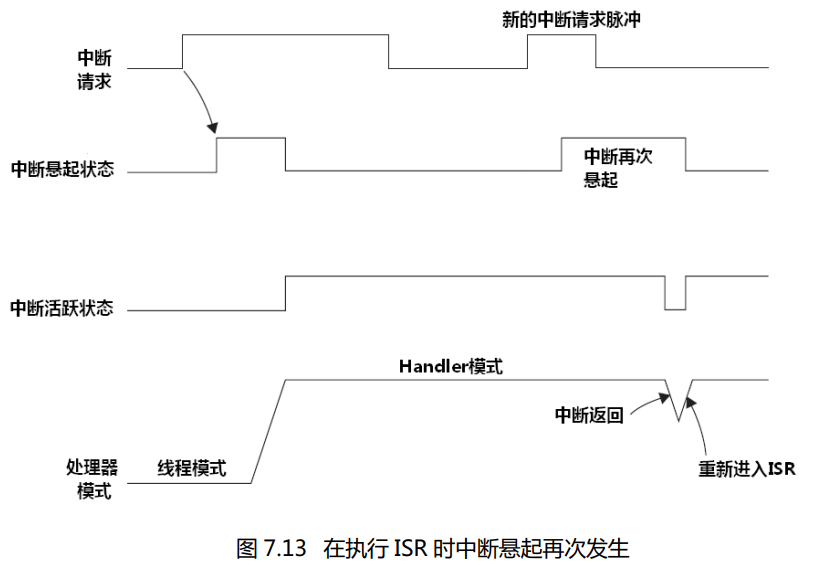
***如果中断源咬住请求信号不放，该中断就会在其上次服务例程返回后再次被置为悬起状态***，如图 7.11 所示。这一点上 CM3 和传统的 ARM7TDMI 是相同的。



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



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



15、

16、

17、

18、

19、

### 异常（中断）

z 异常类型

z 优先级的定义

z 向量表

z 中断输入以及悬起行为

z Fault 异常

z SVC 和 PendSV

#### 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***。

1、其中，编号为 ***1－15*** 的对应***系统异常***，

2、***大于等于 16*** 的则全是***外部中断***。

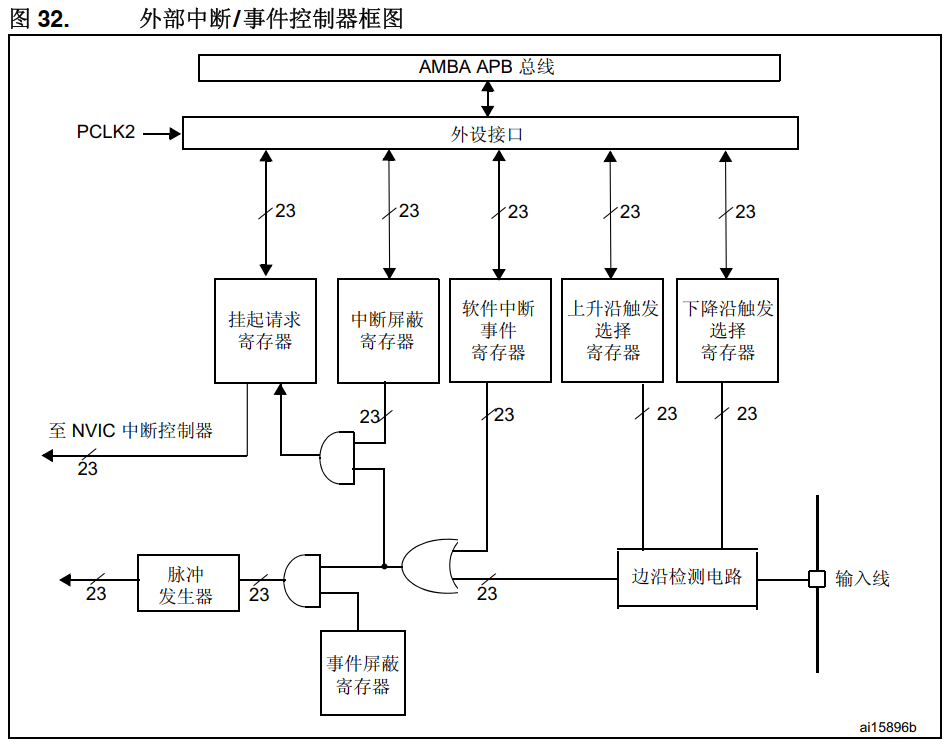
除了个别异常的优先级被定死外， 其它异常的优先级都是可编程的（所有能打断正常执行流的事件都称为异常——译者注）。

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。

#### 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 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） |

#### 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异常的编号。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，它们没有必然联系。常见的情况是，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，直到该异常能够执行为止，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。现在NVIC 的悬起状态寄存器的出现解决了这个问题，即使后来设备已经释放了请求信号，曾经的中断请求也不会错失。

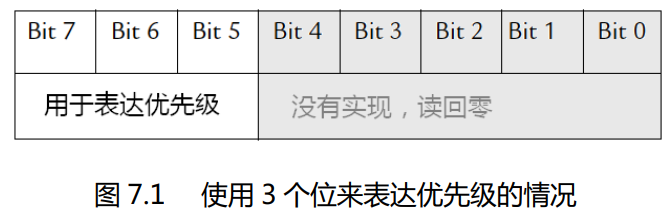
#### 优先级的定义

在 CM3 中，优先级对于异常来说很关键的，它会影响一个异常是否能被响应，以及何时可以响应。**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（但不能编程为负数）**。

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，以达到减少优先级数的目的（可见，不管使用多少位，优先级号是以MSB 对齐的——译注）。

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：

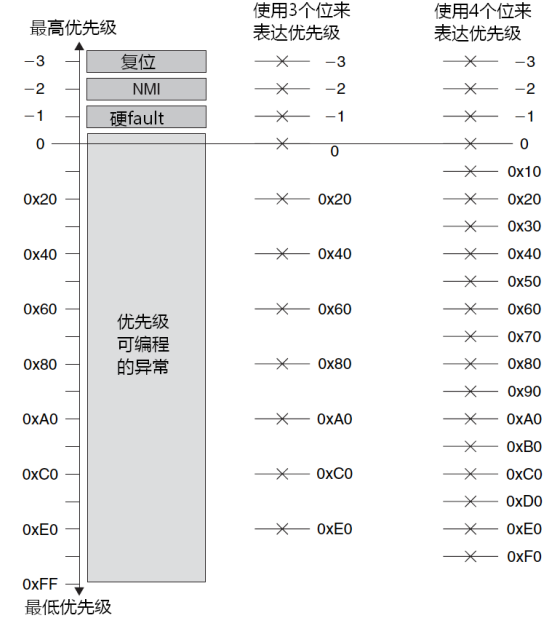


在图中， [4:0]没有被实现，所以读它们总是返回零，写它们则忽略写入的值。因此，对在图中， [4:0]没有被实现，所以读它们总是返回零，写它们则忽略写入的值。因此，对于 3 个位的情况，我们能够使用的 8 个优先级为： 0x00（最高）， 0x20， 0x40， 0x60， 0x80，

0xA0， 0xC0 以及 0xE0。

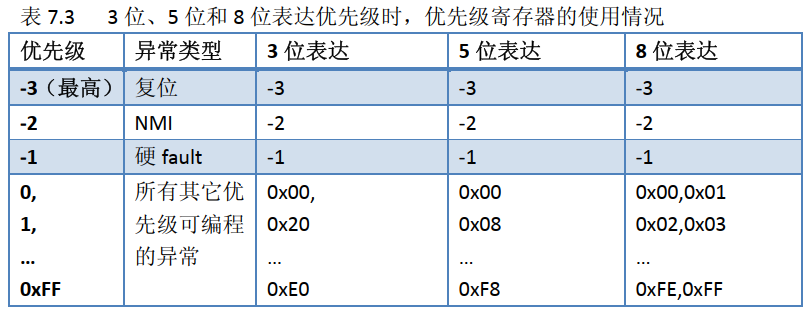
如果使用更多的位来表达优先级，则能够使用的值也更多，同时需要的门也更多——带来更多的成本和功耗。 CM3 允许的最少使用位数为 3 个位，亦即至少要支持 8 级优先级

下图给出 3 个优先级位和 4 个优先级位的对比：



***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，甚至会反转小于等于 7 的高优先级。如， 8 号优先级因为损失了 MSB，现在反而变成 0 号了！

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：



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，分别是抢占优先级和亚优先级，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***亚优先级***，如表 7－4 所示

表 7.4 抢占优先级和亚优先级的表达，位数与分组位置的关系

|  |  |  |
| --- | --- | --- |
| 分组位置 | 表达抢占优先级的位段 | 表达亚优先级的位段 |
| **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 |

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。亚优先级则处理“内务”：当抢占优先级相同的异常有不止一个悬起时，就优先响应亚优先级最高的异常。

**这种优先级分组规定：亚优先级至少是 1 个位。因此抢占优先级最多是 7 个位，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亚优先级，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。**

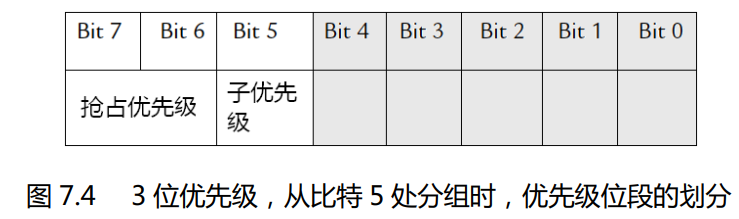
当然还有凌架于法律之上的三位老大：复位， NMI 和硬 fault。它们无论何时出现，都立即无条件抢占所有优先级可编程的“平民异常”。

在计算抢占优先级和亚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。**

##### 举例

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），则你得到 4 级抢占优先级，且在每个抢占优先级的内部有 2 个亚优先级，如图 7.4所示。



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

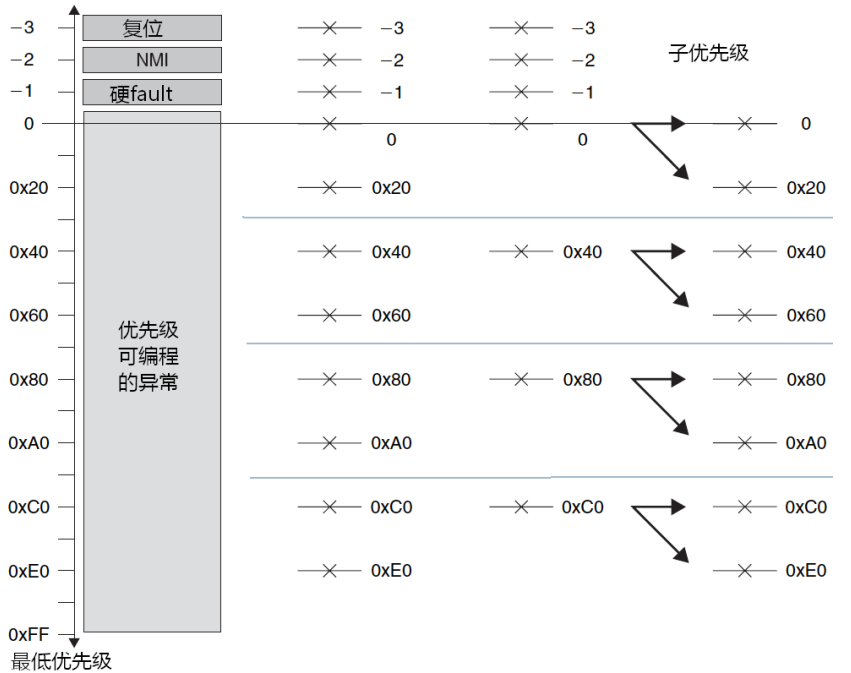
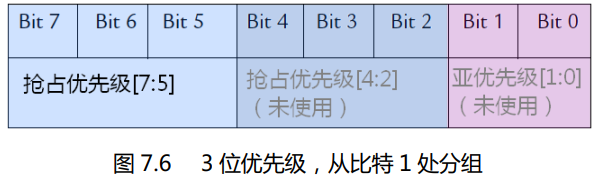
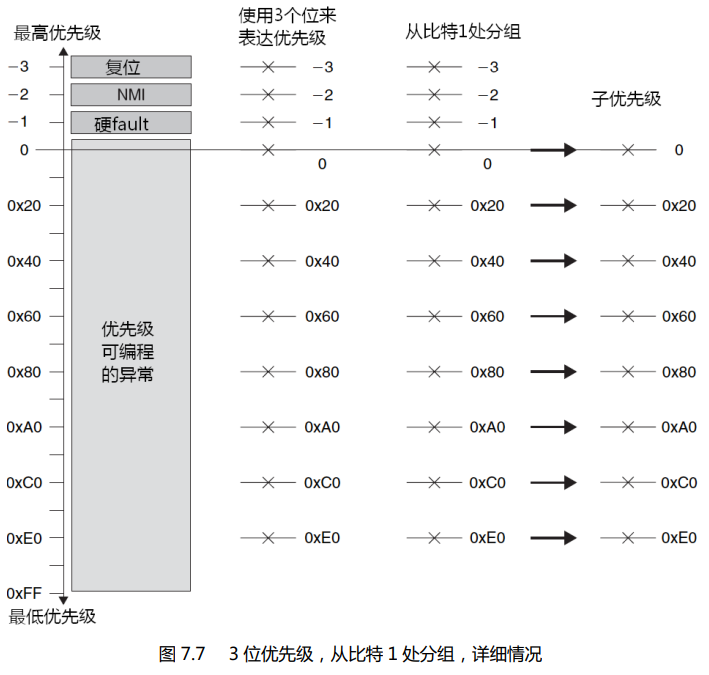


图 7.5 三位优先级，从比特 5 处分组

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

2^3=8



***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。如 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。

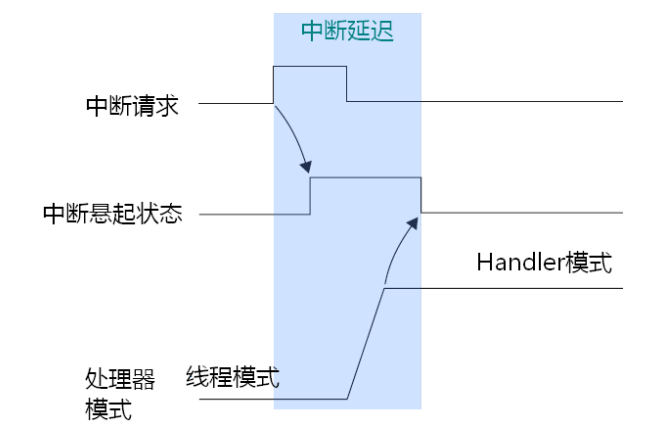
#### 中断输入及悬起行为

NMI是Non-Maskable Interrupt(不可屏蔽中断)缩写, 在STM32F4中, NMI被连接到时钟安全系统CCS(Clock security system)异常上。

本节开始讨论中断的输入和悬起行为。

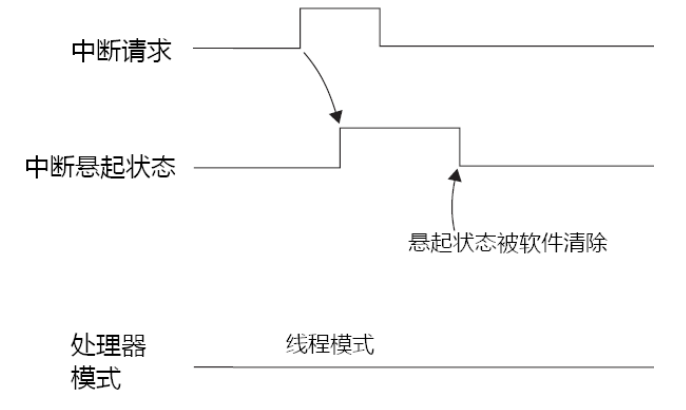
1、这也适用于 NMI，只是 NMI 将会立即无条件执行，除了特殊情况：

2、若当前已经在执行 NMI 服务例程，或者 CPU 被调试器喊停(halted)，或者被一些***严重的系统错误锁定（Lock up），则新的 NMI 请求也将悬起***。



当中断输入脚被 assert 后，该中断就被悬起。即使后来中断源取消了中断请求，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。

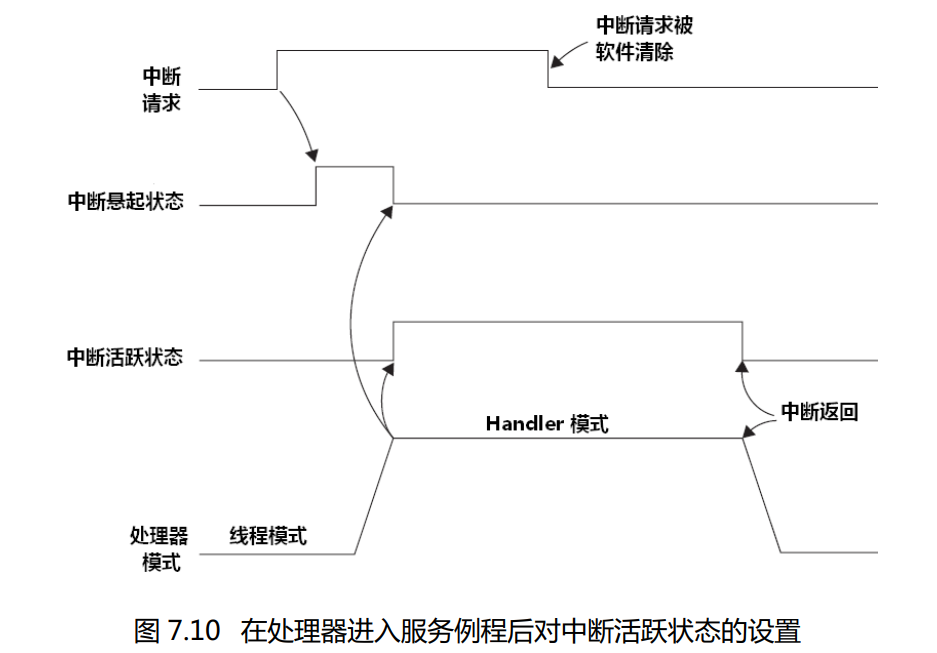


中断在得到处理器响应之前被清除悬起状态

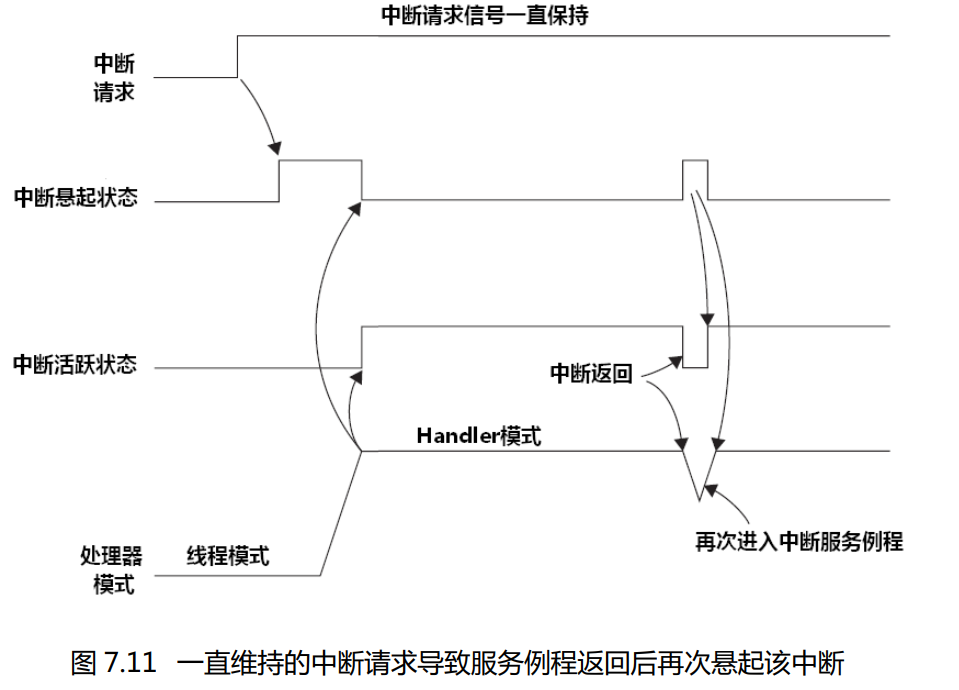
当某中断的服务例程开始执行时，就称此中断进入了“活跃”状态，并且其悬起位会被硬件自动清除，如图 7.10 所示。在一个中断活跃后，直到其服务例程执行完毕，并且返回（亦称为中断退出，第九章详细讨论）了，才能对该中断的新请求予以响应（即单实例）。

当然，新请求的响应亦是由硬件自动清零悬起标志位。

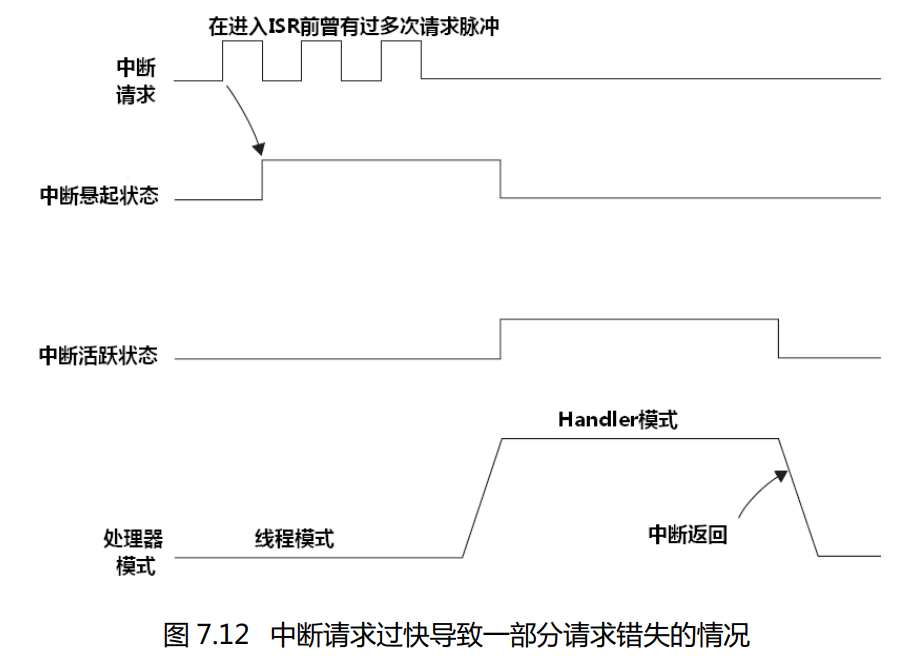
中断服务例程也可以在执行过程中把自己对应的中断重新悬起（使用时要注意避免进入“死循环”——译注）。



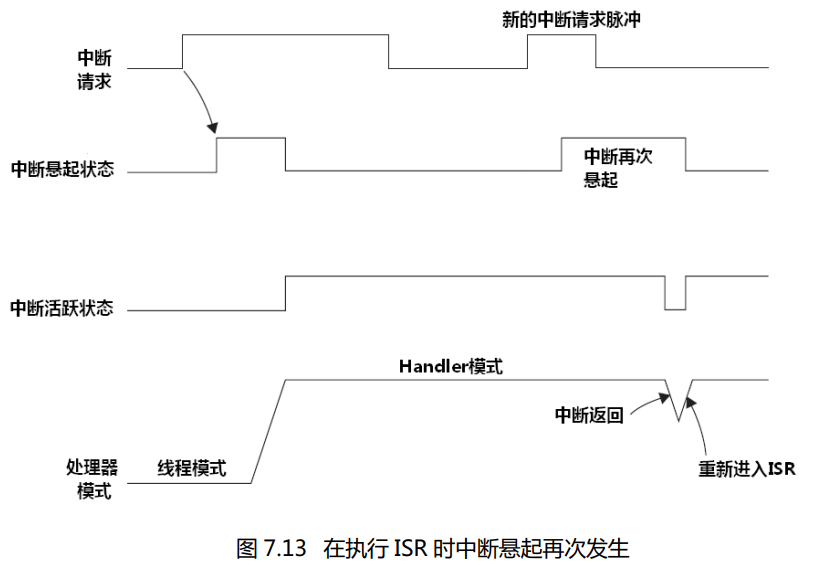
如果中断源咬住请求信号不放，该中断就会在其上次服务例程返回后再次被置为悬起状态，如图 7.11 所示。这一点上 CM3 和传统的 ARM7TDMI 是相同的。



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

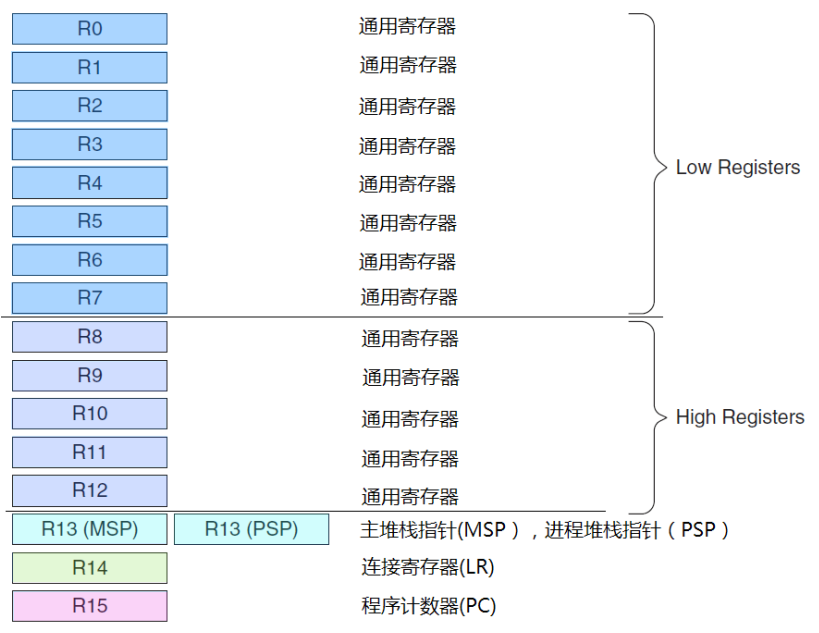


如果在服务例程执行时，中断请求释放了，但是在服务例程返回前又重新被置为有效，则 CM3 会记住此动作，重新悬起该中断。如图 7.13 所示



### 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。

z 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）

z 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）。

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）。

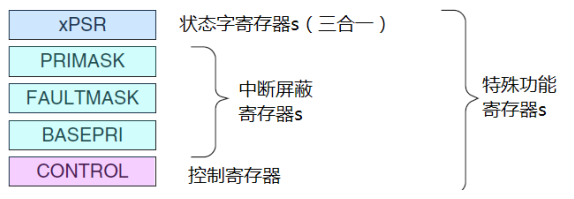
#### 特殊功能寄存器

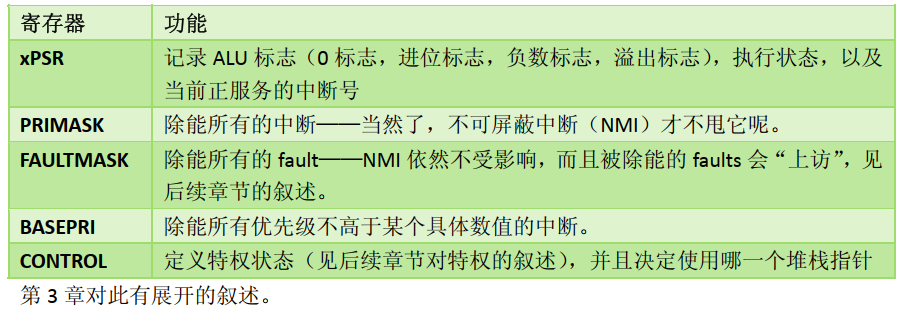
Cortex‐M3 还在内核水平上搭载了若干特殊功能寄存器，包括

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）





#### primask暂时屏蔽中断寄存器

在许多应用中,需要暂时屏蔽所有的中断一执行一些对时序要求严格的任务,这个时候就 可以使用 PRIMASK 寄存器,PRIMASK 用于禁止除 复位、NMI(不可屏蔽中断) 和 HardFalut(硬故障寄存器) 外的所有异常和中断.

#### faultmask寄存器

FAULTMASK比PRIMASK更狠，它可以连HardFault都屏蔽掉，使用方法和PRIMASK类似，FAULTMASK会在退出时自动清零。

#### basepri优先级屏蔽中断寄存器

BASEPRI它屏蔽操作很细腻,它能屏蔽优先级低于BASEPRI的中断,优先级阈值存在BASEPRI寄存器中,如果向BASEPRI写入0会停止屏蔽中断功能。

注意！freeos开关中断就是控制basepri寄存器，优先级低于阈值关闭，高于阈值不关闭。

### 临界段

#### 什么是临界段

代码的***临界段***也称为***临界区***，***指处理时不可分割的代码区域***，一旦这部分代码开始执行，则***不允许任何中断打断***。

为确保临界段代码的执行不被中断，在进入临界段之前须关中断，而***临界段代码执行完毕后，要立即打开中断***。

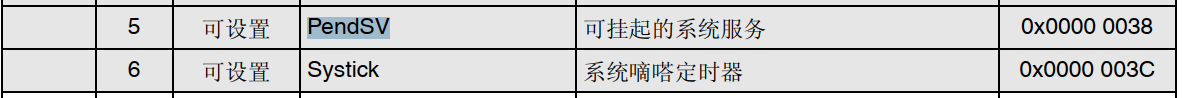
#### 临界段的作用

其实***在RTOS中，使用最多的临界段是OS本身的调用***，

但是***我们用户也是需要对临界资源进行保护的***（临界资源是一次仅允许一个线程使用的共享资源），特别是一些全局变量，当线程正在使用的时候不希望有人来打断我的操作，就行很多时候我们写代码时，需要集中精力，不希望别人打断我们的思路一样。这样子使得系统的运行更加稳定健壮。

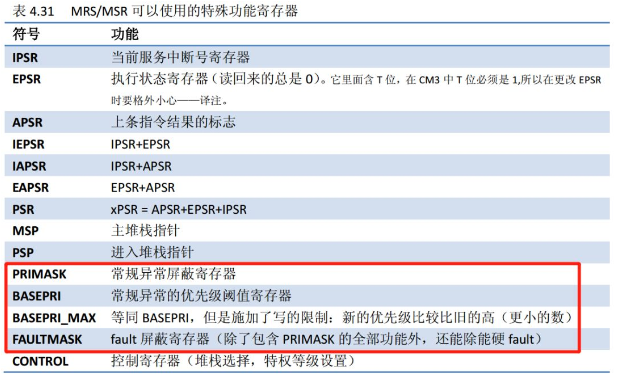
#### 什么时候会打断代码的执行？

顾名思义，代码正在正常运行的时候，基本不会被打断，能被打断的都是系统发生了异常（***中断也是异常***），在OS中，除了外部中断能将正在运行的代码打断，还有线程的调度——PendSV，系统产生 PendSV中断，在 PendSV Handler 里面实现线程的切换。我们要将这项东西屏蔽掉，保证当前只有一个线程在使用临界资源。



### 如何关闭中断？

其实，在我们常用的MCU中，一般为Cortex-M内核的，M内核是有一些指令能快速关闭中断，一起来看看Cortex-M权威指南吧（以Cortex-M3为例）。



### rtt中的rt\_hw\_interrupt\_disable（）和rt\_hw\_interrupt\_enable（level）

|  |
| --- |
| void rt\_interrupt\_enter(void)  {  rt\_base\_t level;  level = rt\_hw\_interrupt\_disable();  rt\_interrupt\_nest ++;  RT\_OBJECT\_HOOK\_CALL(rt\_interrupt\_enter\_hook,());  rt\_hw\_interrupt\_enable(level);  RT\_DEBUG\_LOG(RT\_DEBUG\_IRQ, ("irq has come..., irq current nest:%d\n",  rt\_interrupt\_nest));  }  RTM\_EXPORT(rt\_interrupt\_enter);  rt\_hw\_interrupt\_disable:  MRS r0, PRIMASK  CPSID I  BX LR  /\*  \* void rt\_hw\_interrupt\_enable(rt\_base\_t level);  \*/  .global rt\_hw\_interrupt\_enable  .type rt\_hw\_interrupt\_enable, %function  rt\_hw\_interrupt\_enable:  MSR PRIMASK, r0  BX LR |

#### 一、PRIMASK

PRIMSK=1 ;关闭中断

PRIMSK=0;打开中断

#### rt\_hw\_interrupt\_enable(rt\_base\_t level);有参数，无返回值

该功能将设置指定的中断状态，该***状态应由rt\_hw\_intterrupt\_disable函数返回值（入参level）决定***。

如果入参level的值是0，该函数将打开系统中断状态。

rt\_hw\_interrupt\_enable的主要功能是：***当入参为0时，打开中断；而当入参为1时，无作用***。

#### rt\_base\_t rt\_hw\_interrupt\_disable(void);有返回值，无参数

此功能***将返回当前系统中断状态并禁用系统中断***。即：先返回“当前系统中断状态” 然后“禁用系统中断”

rt\_hw\_interrupt\_disable的主要功能是***：关闭中断并且赋值给level***。

系统默认中断打开状态（PRIMASK = 0），因此level1 = rt\_hw\_interrupt\_disable()，先返回系统当前中断状态则返回level1 = 0；

***而当再次调用level2 = rt\_hw\_interrupt\_disable()时，由于中断已经被禁用，因此返回的是level2 = 1***；

#### 代码示例一：双重临界段嵌套

|  |
| --- |
| 1 代码示例一：临界段嵌套  2 rt\_base\_t rt\_hw\_interrupt\_disable();//有返回值，但无参数！！！！  3  4 rt\_hw\_interrupt\_disable PROC  5 EXPORT rt\_hw\_interrupt\_disable  6 MRS r0, PRIMASK  7 CPSID I  8 BX LR  9 ENDP  10  11  12 void rt\_hw\_interrupt\_enable(rt\_base\_t level);//无返回值，但有参数！！！！  13  14 rt\_hw\_interrupt\_enable PROC  15 EXPORT rt\_hw\_interrupt\_enable  16 MSR PRIMASK, r0  17 BX LR  18 ENDP  1 PRIMASK = 0; /\* PRIMASK 初始值为 0,表示没有关中断 \*/ (1)  2 rt\_base\_t level1; (2)  3 rt\_base\_t level2;  4  5 /\* 临界段代码 \*/  6 {  7 /\* 临界段 1 开始 \*/  8 level1 = rt\_hw\_interrupt\_disable(); /\* 关中断,level1=0,PRIMASK=1 \*/ (3)  9 {  10 /\* 临界段 2 \*/  11 level2 = rt\_hw\_interrupt\_disable(); /\* 关中断,level2=1,PRIMASK=1 \*/ (4)  12 {  13  14 }  15 rt\_hw\_interrupt\_enable(level2); /\* 开中断,level2=1,PRIMASK=1 \*/ (5)  16 }  17 /\* 临界段 1 结束 \*/  18 rt\_hw\_interrupt\_enable(level1); /\* 开中断,level1=0,PRIMASK=0 \*/ (6)  19 } |

#### 代码示例二：单一临界段

|  |
| --- |
| 1 ;/\*  2 ; \* rt\_base\_t rt\_hw\_interrupt\_disable();  3 ; \*/  4 rt\_hw\_interrupt\_disable PROC  5 EXPORT rt\_hw\_interrupt\_disable  6 MRS r0, PRIMASK  7 CPSID I  8 BX LR  9 ENDP  10  11 ;/\*  12 ; \* void rt\_hw\_interrupt\_enable(rt\_base\_t level);  13 ; \*/  14 rt\_hw\_interrupt\_enable PROC  15 EXPORT rt\_hw\_interrupt\_enable  16 MSR PRIMASK, r0  17 BX LR  18 ENDP  1 PRIMASK = 0; /\* PRIMASK 初始值为 0,表示没有关中断 \*/  2 rt\_base\_t level1;  3  4 /\* 临界段代码 \*/  5 {  6 /\* 临界段开始 \*/  7 level1 = rt\_hw\_interrupt\_disable();/\* 关中断,level1=0,PRIMASK=1 \*/  8 {  9  10 }  11 /\* 临界段结束 \*/  12 rt\_hw\_interrupt\_enable(level1); /\* 开中断,level1=0,PRIMASK=0 \*/( 注意点) |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |