第9章

# 中断的具体行为

* 中断/异常的响应序列
* 异常返回
* 嵌套的中断
* 咬尾中断
* 晚到（的高优先级）中断
* 异常返回值
* 中断延迟
* 异常响应期间的faults

**译注：在本章中，如无特殊说明，不分辨“中断”与“异常”这两个术语，可以互换使用。**

## 9.1 中断／异常的响应序列

当CM3开始响应一个中断时，会在它小小的体内奔涌起三股暗流：

* + 入栈： 把8个寄存器的值压入栈
  + 取向量：从向量表中找出对应的服务程序入口地址
  + 选择堆栈指针MSP/PSP，更新堆栈指针SP，更新连接寄存器LR，更新程序计数器PC

### 9.1.1 入栈

响应异常的第一个行动，就是自动保存现场的必要部分：依次把xPSR, PC, LR, R12以及R3-R0由硬件自动压入适当的堆栈中：如果当响应异常时，当前的代码正在使用PSP，则压入PSP，也就是使用进程堆栈；否则就压入MSP，使用主堆栈。一旦进入了服务例程，就将一直使用主堆栈。

假设入栈开始时，SP的值为N，则在入栈后，堆栈内部的变化如表9.1表示。又因为AHB接口上的流水线操作本性，地址和数据都在经过一个流水线周期之后才进入。另外，在自动入栈的过程中，把寄存器写入堆栈内存的时间顺序，并不是与写入的空间顺序相对应的。但是机器会保证：正确的寄存器将被保存到正确的位置，如图9.1和表9.1的第3列所示。

表9.1 入栈顺序以及入栈后堆栈中的内容

|  |  |  |
| --- | --- | --- |
| 地址 | 寄存器 | 被保存的顺序 |
| 旧SP (N-0) | 原先已压入的内容 | - |
| (N-4) | xPSR | 2 |
| (N-8) | PC | 1 |
| (N-12) | LR | 8 |
| (N-16) | R12 | 7 |
| (N-20) | R3 | 6 |
| (N-24) | R2 | 5 |
| (N-28) | R1 | 4 |
| 新SP (N-32) | R0 | 3 |

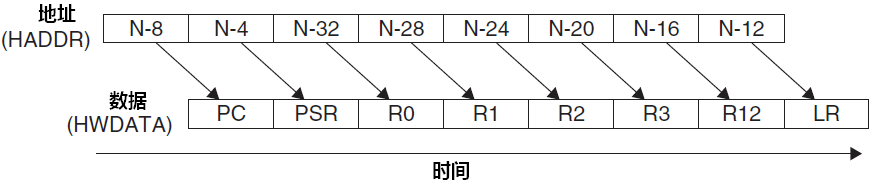


图9.1 内部入栈序列

Cortex-M3 r2p0修订版的区别

在r2p0中，自动打开了“双字对齐的堆栈工作模式”，简称双字对齐模式。在双字对齐的模式下，SP的值必须能被8整除。如果没有打开双字对齐，则与图9.1和表9.1所示的相同。但如果打开了双字对齐，可是SP却不能被8整除，则没有对齐的一个字被空出来，所有入栈寄存器的地址，依序减4。如：PC不再是N-8，而是N-12，其它寄存器亦如此。

上文所提到的这个自动压入的8字数据块，通常被称作“异常堆栈帧（exception stack frame）”。在CM3修订版2之前，缺省配置下的堆栈帧可以始于任何字对齐的地址。到了修订版2，则改为缺省配置下堆栈帧要双字对齐。之所以这样做，是为了满足AAPCS所规定的过程调用标准。这个功能其实在CM3的修订版1中就有了，只是缺省时没有打开。如欲在修订版1中开启此功能，需要手动在NVIC配置控制寄存器中置位STKALIGN位。当需要除能此特性时，也只需清除此位。关于该寄存器的更多细节，请参阅第12章（双字堆栈对齐节）。

CM3在看不见的内部“搅浑”了入栈的顺序，这是有深层次的原因的。先把PC与xPSR的值保存，就可以更早地启动服务例程指令的预取——因为这需要修改PC；同时，也做到了在早期就可以更新xPSR中IPSR位段的值。

细心的读者一定在猜测：为啥袒护R0-R3以及R12呢，R4-R11就是下等公民？原来，在ARM上，有一套的C函数调用标准约定（《C/C++ Procedure Call Standard for the ARM Architecture》，AAPCS, Ref5）。个中原因就在它上面：它使得中断服务例程能用C语言编写，编译器优先使用入栈了的寄存器来保存中间结果（当然，如果程序过大也可能要用到R4-R11，此时编译器负责生成代码来push它们。但是，ISR应该短小精悍，不要让系统如此操心——译者注）。

如果读者再仔细看，会发现R0-R3, R12是最后被压进去的。这里也有一番良苦用心：为的是可以更容易地使用SP基址来索引寻址，（这也方便了LDM等多重加载指令。因为LDM必须加载地址连续的一串数据，而现在R0-R3, R12的存储地址连续了——译者注）。这种顺序也舒展了参数的传递过程：使之可以方便地通过读取入栈了的R0-R3取出（主要为系统软件所利用，多见于SVC与PendSV中的参数传递）。

### 9.1.2 取向量

当数据总线（系统总线）正在为入栈操作而忙得风风火火时，指令总线（I-Code总线）可不是凉快地坐那儿看热闹——它正在为响应中断紧张有序地执行另一项重要的任务：从向量表中找出正确的异常向量，然后在服务程序的入口处预取指。由此可以看到各自都有专用总线的好处：入栈与取指这两个工作能同时进行。

### 9.1.3 更新寄存器

在入栈和取向量操作完成之后，执行服务例程之前，还要更新一系列的寄存器：

* + SP：在入栈后会把堆栈指针（PSP或MSP）更新到新的位置。在执行服务例程时，将由MSP负责对堆栈的访问。
  + PSR：更新IPSR位段（地处PSR的最低部分）的值为新响应的异常编号。
  + PC：在取向量完成后，PC将指向服务例程的入口地址，
  + LR：在出入ISR的时候，LR的值将得到重新的诠释，这种特殊的值称为“EXC\_RETURN”，在异常进入时由系统计算并赋给LR，并在异常返回时使用它。EXC\_RETURN的二进制值除了最低4位外全为1，而其最低4位则有另外的含义（后面讲到，见表9.3和表9.4）。

以上是在响应异常时通用寄存器的变化。另一方面，在NVIC中，也会更新若干个相关有寄存器。例如，新响应异常的悬起位将被清除，同时其活动位将被置位。

## 9.2 异常返回

当异常服务例程执行完毕后，需要很正式地做一个“异常返回”动作序列，从而恢复先前的系统状态，才能使被中断的程序得以继续执行。从形式上看，有3种途径可以触发异常返回序列，如表9.2所示。而不管使用哪一种，都需要用到先前储到LR的EXC\_RETURN。

表9.2 触发中断返回的指令

|  |  |
| --- | --- |
| 返回指令 | 工作原理 |
| BX <reg> | 当LR存储了EXC\_RETURN时，使用BX LR即可返回 |
| POP {PC}和  POP {…,PC} | 在服务例程中，LR的值常常会被压入栈。此时即可使用POP指令把LR存储的EXC\_RETURN往PC里弹，从而启动处理器的中断返回序列 |
| LDR与LDM | 把PC作为目的寄存器，亦可启动中断返回序列 |

有些处理器使用特殊的返回指令来标示中断返回，例如8051就使用reti。但是在CM3中，是通过把EXC\_RETURN往PC里写来识别返回动作的。因此，可以使用上述的常规返回指令，从而为使用C语言编写服务例程扫清了最后的障碍（无需特殊的编译器命令，如\_\_interrupt）。

在启动了中断返回序列后，下述的处理就将进行：

1. 出栈：先前压入栈中的寄存器在这里恢复。内部的出栈顺序与入栈时的相对应，堆栈指针的值也改回先前的值。

2. 更新NVIC寄存器：伴随着异常的返回，它的活动位也被硬件清除。对于外部中断，倘若中断输入再次被置为有效，悬起位也将再次置位，新一次的中断响应序列也可随之再次开始。

## 9.3 嵌套的中断

在CM3内核以及NVIC的深处，就已经内建了对中断嵌套的全力支持，根本无需使用汇编去写封皮代码(wrapper code)。事实上，我们要做的就只是为每个中断适当地建立优先级，不用再操心别的。表现在：

第一、 NVIC和CM3处理器会根据优先级的设置来控制抢占与嵌套行为。因此，在某个异常正在响应时，所有优先级不高于它的异常都不能抢占之，而且它自己也不能抢占自己。

第二、 有了自动入栈和出栈，就不用担心在中断发生嵌套时，会使寄存器的数据损毁，从而可以放心地执行服务例程。

然而，有一件事情却必须更加一丝不苟地处理了，否则有功能紊乱甚至死机的危险。这就是计算主堆栈容量的最小安全值。我们已经知道，所有服务例程都只使用主堆栈。所以当中断嵌套加深时，对主堆栈的压力会增大：每嵌套一级，就至少再需要8个字，即32字节的堆栈空间——而且这还没算上ISR对堆栈的额外需求，并且何时嵌套多少级也是不可预料的。如果主堆栈的容量本来就已经所剩无几了，中断嵌套又突然加深，则主堆栈有被用穿的凶险。这就好像已经表现出了高血压危象的时候，情绪又一激动，就容易中风一般。中风是一大杀手，而堆栈溢出同样是很致命的，它会使入栈数据与主堆栈前面的数据区发生混迭，使这些数据被破坏；若在服务例程返回前混迭区的数据又被更改了，则堆栈内容被破坏。这么一来在执行中断返回后，系统极可能功能紊乱，甚至当场被一击秒杀——程序跑飞/死机！

另一个要注意的，是相同的异常是不允许重入的。因为每个异常都有自己的优先级，并且在异常处理期间，同级或低优先级的异常是要阻塞的。因此对于同一个异常，只有在上次实例的服务例程执行完毕后，方可继续响应新的请求。由此可知，在SVC服务例程中，就不得再使用SVC指令，否则将fault伺候。

## 9.4 咬尾中断

CM3为缩短中断延迟做了很多努力，第一个要提的，就是新增的“咬尾中断”（Tail-Chaining）机制。

当处理器在响应某异常时，如果又发生其它异常，但它们优先级不够高，则被阻塞——这个我们已经知道。那么在当前的异常执行返回后，系统处理悬起的异常时，倘若还是先POP，然后又把POP出来的内容PUSH回去，这不成了砸锅炼铁再铸锅，白白浪费CPU时间吗，可知还有多少紧急的事件悬而未决呀！正因此，CM3不会傻乎乎地POP这些寄存器，而是继续使用上一个异常已经PUSH好的成果，消灭了这种铺张浪费。这么一来，看上去好像后一个异常把前一个的尾巴咬掉了，前前后后只执行了一次入栈／出栈操作。于是，这两个异常之间的“时间沟”变窄了很多，如图9.2所示。

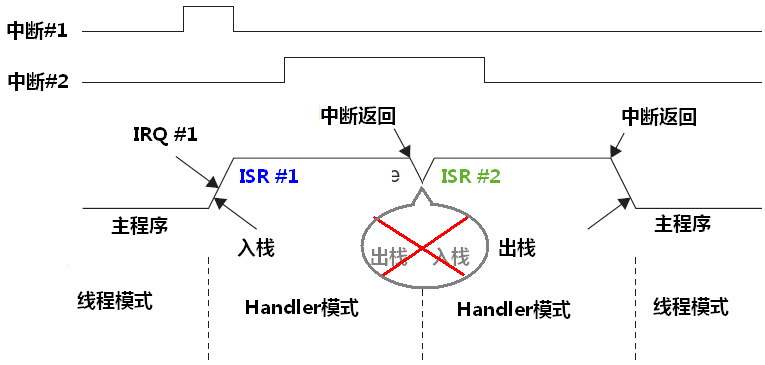


图9.2 异常咬尾示意图

为进一步帮助读者理解，译者从另外文献上截取并改编下图：

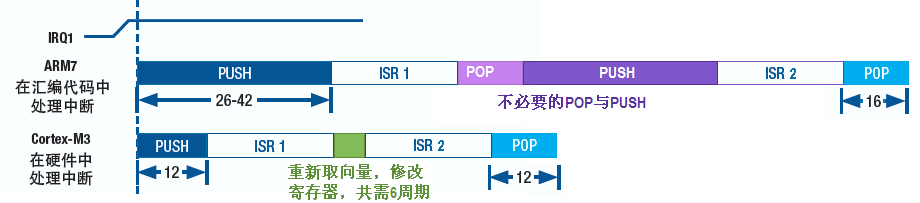


图9.2B 异常咬尾与常规处理的比较（以ARM7TDMI为例）

## 9.5 晚到（的高优先级）异常

CM3的中断处理还有另一个机制，它强调了优先级的作用，这就是“晚到的异常处理”。当CM3对某异常的响应序列还处在早期：入栈的阶段，尚未执行其服务例程时，如果此时收到了高优先级异常的请求，则本次入栈就成了为高优先级中断所做的了——入栈后，将执行高优先级异常的服务例程。可见，它虽然来晚了，却还是因优先级高而受到偏袒，低优先级的异常为它“火中取栗”。

比如，若在响应某低优先级异常#1的早期，检测到了高优先级异常#2，则只要#2没有太晚，就能以“晚到中断”的方式处理——在入栈完毕后执行ISR #2，如图9.3所示。如果异常#2来得太晚，以至于已经执行了ISR #1的指令，则按普通的抢占处理，这会需要更多的处理器时间和额外32字节的堆栈空间。

在ISR #2执行完毕后，则以刚刚讲过的“咬尾中断”方式，来启动ISR #1的执行。

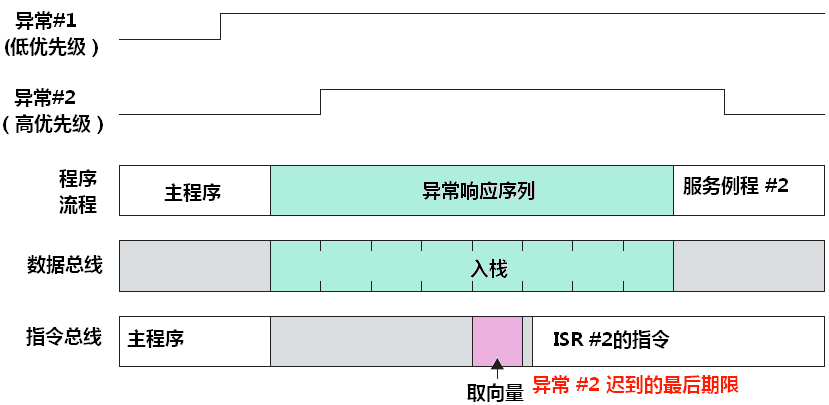


图9.3 晚到异常的处理模式图

## 9.6 异常返回值

前面已经讲到，在进入异常服务程序后，将自动更新LR的值为特殊的EXC\_RETURN。这是一个高28位全为1的值，只有[3:0]的值有特殊含义，如表9.3所示。当异常服务例程把这个值送往PC时，就会启动处理器的中断返回序列。因为LR的值是由CM3自动设置的，所以只要没有特殊需求，就不要改动它。

表9.3 EXC\_RETURN位段详解

|  |  |
| --- | --- |
| 位段 | 含义 |
| [31:4] | EXC\_RETURN的标识：必须全为1 |
| 3 | 0=返回后进入Handler模式  1=返回后进入线程模式 |
| 2 | 0=从主堆栈中做出栈操作，返回后使用MSP，  1=从进程堆栈中做出栈操作，返回后使用PSP |
| 1 | 保留，必须为0 |
| 0 | 0=返回ARM状态。  1=返回Thumb状态。**在CM3中必须为1** |

总结一下表9.3，可以得出，合法的EXC\_RETURN值共3个，如表9.4所示

表9.4 合法的EXC\_RETURN值及其功能

|  |  |
| --- | --- |
| EXC\_RETURN数值 | 功能 |
| 0xFFFF\_FFF1 | 返回handler模式 |
| 0xFFFF\_FFF9 | 返回线程模式，并使用主堆栈(SP=MSP) |
| 0xFFFF\_FFFD | 返回线程模式，并使用线程堆栈(SP=PSP) |

如果主程序在线程模式下运行，并且在使用MSP时被中断，则在服务例程中LR=0xFFFF\_FFF9（主程序被打断前的LR已被自动入栈）。

如果主程序在线程模式下运行，并且在使用PSP时被中断，则在服务例程中LR=0xFFFF\_FFFD（主程序被打断前的LR已被自动入栈）。

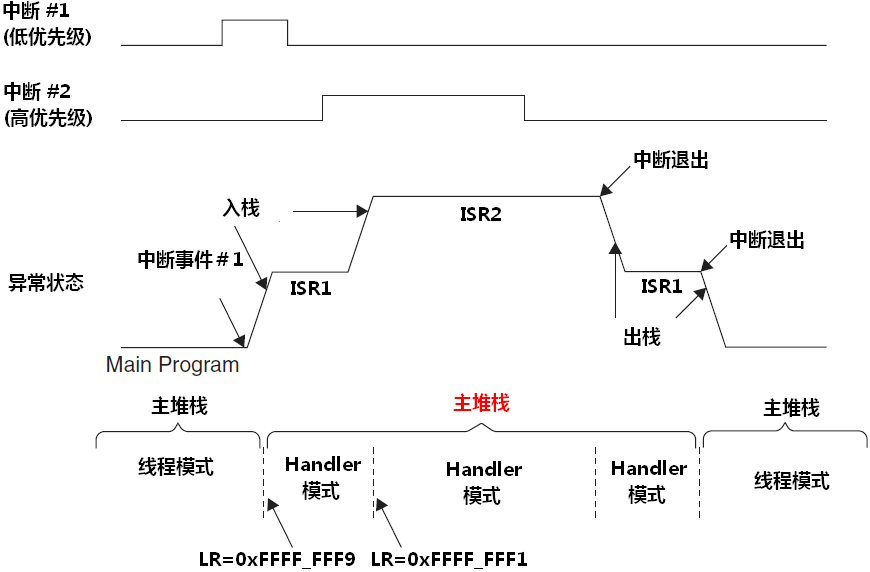


图9.4 LR的值在异常期间被设置为EXC\_RETURN（线程模式使用主堆栈）

如果主程序在Handler模式下运行，则在服务例程中LR=0xFFFF\_FFF1（主程序被打断前的LR已被自动入栈）。这时的所谓“主程序”，其实更可能是被抢占的服务例程。事实上，在嵌套时，更深层ISR所看到的LR总是0xFFFF\_FFF1，如图9.5所示。

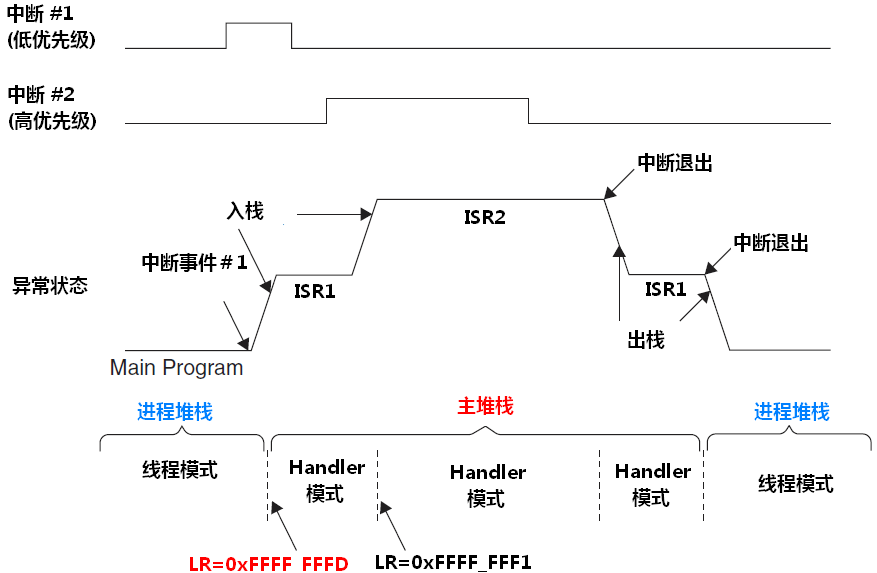


图9.5 LR的值在异常期间被设置为EXC\_RETURN（线程模式使用进程堆栈）

由EXC\_RETURN的格式可见，我们不能把0xFFFF\_FFF0-0xFFFF\_FFFF中的地址作为任何返回地址。其实也并不用担心会弄错，因为CM3已经把这个范围标记成“取指不可区”了。

## 9.7 中断延迟

在设计实时系统时，必须对中断延迟进行严肃和仔细地估算。在这里，中断延迟的定义是：从检测到某中断请求，到执行了其服务例程的第一条指令时，已经流逝了的时间。在CM3中，若存储器系统够快，且总线系统允许入栈与取指同时进行，同时该中断可以立即响应，则中断延迟是雷打不动的12周期（满足硬实时所要求的确定性）。在与时间赛跑的这12个周期里，处理器内部一直开足马力，进行了入栈、取向量、更新寄存器以及服务例程取指的一系列操作。但若存储器太慢以至于引入等待周期，或者还有其它因素，则会引入额外的延时——反正如果有拖后腿的，那绝不可能是CM3内核。

当处理咬尾中断时，省去了堆栈操作，因此切入新异常服务例程的耗时可以短至6周期。

有些指令需要较多的周期才能完成。它们是除法指令，双字传送指令LDRD/STRD以及多重数据传送指令(LDM/STM)。

对于前两者，CM3将为了保证中断及时响应而取消它们的执行，待返回后重新开始——这牺牲了一点性能，以及某些子程序的一点个人利益，但换来了对意外事件的更快救援。

对于LDM/STM，则有另外的处理方式。因为它们不照前两者那么浑然一体——它们其实是一串LDR/STR的速度优化版。于是，为了加速中断的响应，CM3支持LDM/STM指令的中止和继续，就好像它们只是普通的一串LDR/STR一样。为了实现“指令撕裂与粘合”的目的，需要记录中断时数据传送的进程。为此，CM3在xPSR中开出若干个“ICI位”，记录下一个即将传送的寄存器是哪一个（LDM/STM在汇编时，都把寄存器号升序排序）。在服务例程返回后，xPSR被弹出，CM3再从ICI bits中获取当时LDM/STM执行的进度，从而可以继续传送。

这个办法听起来是个好主意，只是在个别情况下还有一点限制：IF-THEN(IT)指令的执行也需要在xPSR中使用几个位，可它需要的位刚好与ICI位重合（类似C中的union）——both ICI bits和IT条件都记录在EPSR中。所以，如果在IF-THEN中使用了LDM/STM，则不再记录LDM/STM的执行进度。但尽管如此，及时响应中断依然是首要任务。此时只好把LDM/STM取消，待中断返回后继续执行

译注：仔细的读者可能会注意到，xPSR中有很多位空着没用，从而可能想不通，为啥要让“有人可怜没人爱，有人却忙不过来”。这可能是因为在其它款式中，这些位被用掉了，或者还有其它什么难言之隐。

另外，如果在总线接口上还有未完成的(outstanding)数据传送，例如有一个带缓冲的写操作未完成，处理器也只能等待此传送完成。这是迫不得以的——只有这样，才能保证在发生了总线fault时，其服务例程能够安全地抢占其它程序。

当多个中断同时请求时，也会发生中断延迟，这表现在只有优先级最高的得到立即响应，所有其它的中断将被延迟。另外，在中断嵌套时，每个中断都会阻塞同级和低优先级的中断。最后，如果中断被掩蔽（也就是俗称的关中，在多任务系统下满地都有），则在掩蔽期间也会附加中断延迟。

## 9.8 异常响应期间的faults

Faults是运行时发生各种故障的表现，在中断响应期间的故障也不例外。中断响应的每一步骤都可以触发faults。

### 9.8.1 入栈期间

如果在入栈期间引起了总线fault，则本次入栈操作将被强行中止，并且把总线异常悬起或者在允许时立即响应。若除能了总线fault，则此次故障将成为“硬伤”——上访至硬fault。在总线fault被使能的情况下，如果它的优先级比正在响应的异常高，则抢占之，否则将悬起直到引起fault的异常执行完毕。这种情况被称为“入栈错误”(stacking error)，由总线fault状态寄存器(BFSR，地址：0xE000\_ED29)的STKERR位指示（位偏移：4）。

如果入栈操作引起MPU访问违例，则产生存储管理fault，并且必须能立即执行MemFault服务例程，否则将无条件上访成硬fault。在发生入栈时访问违例时，存储管理fault寄存器(MFSR，地址:0xE000\_ED28)中的MSTKERR位（位偏移：4）被置位，用于指示该fault。

入栈是自动完成的，因此不可能产生用法fault——译者。

### 9.8.2 出栈期间

如果在中断返回时的出栈期间引起了总线fault，则本次出栈操作将被强行中止，并且把总线异常悬起或立即响应。若除能了总线fault，则此次故障将成为“硬伤”——上访至硬fault。其它情况下，只要总线fault的优先级比当前的高（也包括比当前最深嵌套的优先级高），则可以立即响应。这种情况称为“出栈错误”（unstacking error），由BFSR.3指示（UNSTKERR位）。

类似地，如果是因MPU访问违例造成的MemManage fault，由MFSR.3（MUNSTKERR）指示。且MemManage fault的服务例程必须能立即执行，否则无条件硬fault。

### 9.8.3 取向量期间

在取向量期间发生总线fault，这是非常罕见的一种情况，这也是最严重的，因此直接上硬fault（MPU的限制则管不着取向量操作——译者注）。这种情况，由硬fault状态寄存器（HFSR，地址：0xE000\_ED2C）中的VECTTBL位（位偏移：1）来指示。

### 9.8.4 无效返回时

如果LR中的EXC\_RETURN不是合法的值（合法值见表9.4，包括企图返回ARM状态），则引起用法fault。如果用法fault被除能，也上访成硬fault。此时，用法Fault状态寄存器(UFSR,地址：0xE000\_ED2A)中的INVPC位（位偏移：2），或者是INVSTATE位（位偏移：1）置位。