第12章

# 编程进阶与系统行为

* 在系统中使用双堆栈
* 双字的堆栈对齐方式
* 非基级的线程模式
* 性能评估
* 当处理器被锁定时

阅读理解本章，需要有操作系统的基本概念：定义、作用及地位

## 12.1 在系统中使用双堆栈

CM3的出现，让单片机业界也能出双枪李向阳。v7-M架构的一个重要能力，就是提供了这个双堆栈的设计，允许把用户应用程序的堆栈与特权级/操作系统内核(kernel)的堆栈分开。如果再辅以MPU，还能进一步地阻止用户程序访问内核的堆栈，同时也消除了内核数据被破坏的可能。

要在CM3中创建可靠扛打的系统，必须两手抓，两手都要硬。典型情况下，一个真正健壮的CM3软件系统是要使用实时操作系统内核的，通常会符合如下的要求：

* + 服务例程使用MSP（在“非基级线程模式”中会讲到例外情况）
  + 尽管异常服务例程使用MSP，但是它们在形式上返回后，内容上却可以依然继续——而且此时还能使用PSP，从而实现“可抢占的系统调用”，大幅提高实时性能
  + 通过SysTick，实时内核的代码每隔固定时间都被调用一次，运行在特权级水平上，负责任务的调度、任务时间管理以及其它系统例行维护
  + 用户应用程序以线程的形式运行，使用PSP，并且在用户级下运行
  + 内核在执行关键部位的代码时，使用MSP，并且在辅以MPU时，MSP对应的堆栈只允许特权级访问

如图12.1所示，假设系统内存是一块SRAM，则我们可以通过MPU，把它分为两个regions，其中一个用于用户级，另一个用于特权级。另外别忘了CM3的堆栈是“向下生长的满栈”，因此需要把这两个SP初始为指向这两个regions的顶端。

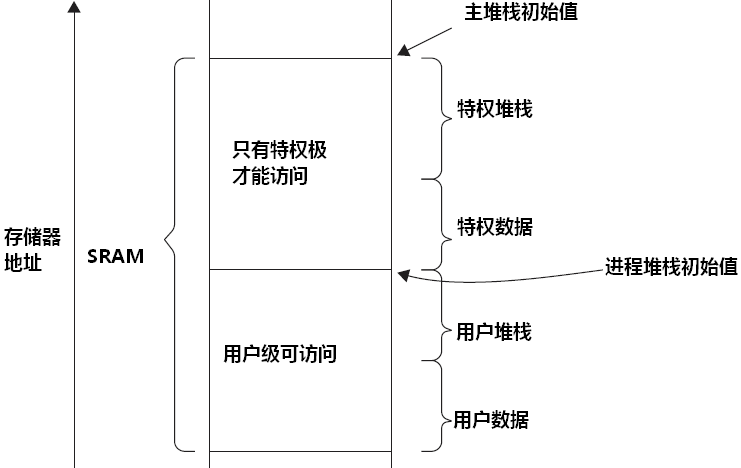


图12.1 用户级/特权级下的内存配置示范

上电后，通过从向量表中取出0号向量，仅初始化了MSP。因此，需要额外的工作来建立完整的双堆栈系统。对于使用汇编写成的代码，只需寥寥几句：

; 这段代码在用户可访问内存中，但从特权级开始执行

BL MpuSetup ; 建立MPU regions，并使能存储器保护

LDR R0, =PSP\_TOP ; 读取进程堆栈的栈顶

MSR PSP, R0 ; 并用它来初始化进程堆栈

BL SystickSetup ; 配置SysTick，并建立SysTick异常向量，供OS日后使用

MOV R0, #0x3 ; 设置CONTROL寄存器，让用户程序使用PSP

MSR CONTROL,R0 ; 并且切入用户级

B UserAppStart ; 到了这里已经进入了用户级，开始跳入用户程序入口

这个函数最好用汇编写。如果非要用C，则会破坏C函数的堆栈帧：因为C函数常常把多出来的局部变量放到堆栈中，所以在切换堆栈指针时，函数的局部变量可能丢失。在Cortex-M3 TRM（Ref1）中，已经做出明确建议：使用形如SVC的ISR来调用内核，然后通过修改EXC\_RETURN的值来切换堆栈指针。

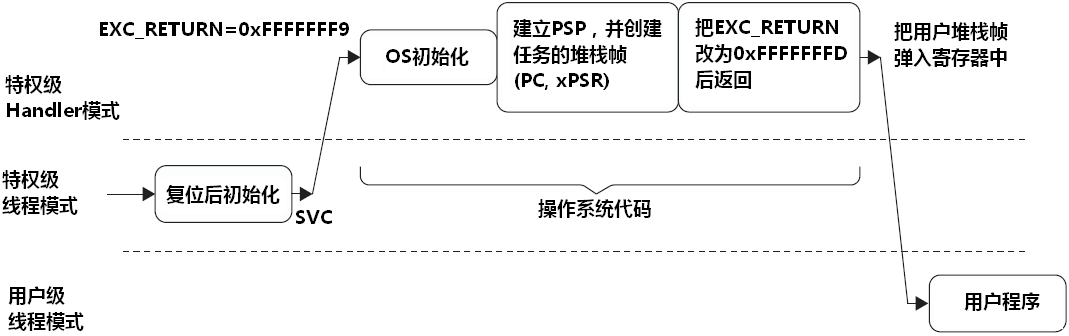


图12.2 简单OS中的堆栈初始化

在操作系统中，对于EXC\_RETURN的修改，只是再寻常不过基本需求。在开始调度用户程序后，一定还伴随着SysTick异常，它周期性把执行权转入操作系统，从而使例行的系统管理以及必要轮转调度得以维持——差不多就是系统的心跳吧，如图12.3所示：

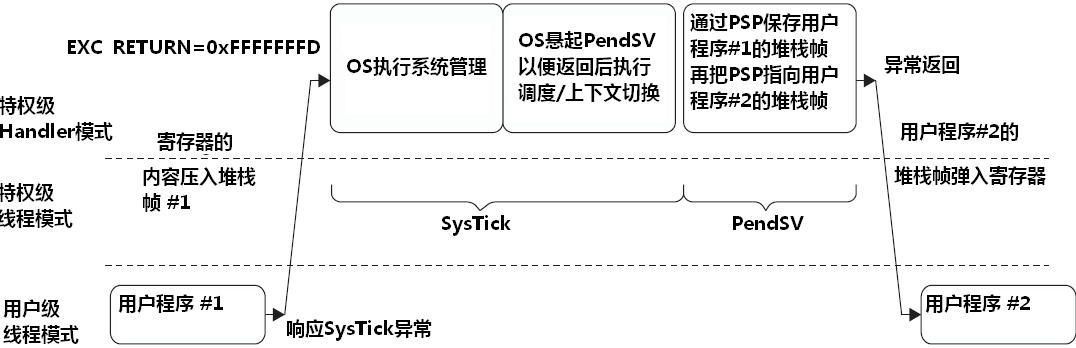


图12.3 SysTick异常推动简单轮转调度模式图

在这里，使用PendSV（一个优先级最低的异常）来执行上下文切换，从而消灭了在中断服务例程中出现上下文切换的可能，读者应该对此还记忆犹新吧。

然而，也有不少的程序不需要上操作系统。即便如此，使用两个栈也依然对于提升程序的可靠性大有用场。其中一个可行的方案是，以MSP启动CM3时，把MSP初始化成某进程的堆栈（process stack）。这样，就可以使初始化代码使用进程堆栈中运行（虽然还使用MSP）。在正式执行应用程序前，先执行如下的初始化代码：

; 从特权级启动，MSP指向一个用户程序的堆栈

MpuSetup(); // 建立MPU regions并使能存储器保护

SystickSetup(); // 建立SysTick异常向量，由其服务例程作为时基的管理

SwitchStackPointer(); // 呼叫一个汇编程序来切换到PSP

/\*; ------在 SwitchStackPointer 中-----

PUSH {R0, R1, LR}

MRS R0, MSP ; 读取MSP到R0，稍后使用

LDR R1, =MSP\_TOP

MSR MSP, R1 ; 让MSP指向新的MSP\_TOP

MSR PSP, R0 ; 把当前的MSP存储到PSP中

MOV R0, #0x3

MSR CONTROL,R0 ; 切换到用户级，并使用PSP指向的堆栈作为当前堆栈

POP {R0, R1, PC}

; ------ 回到C程序中 -----\*/

; 现在已经进入了用户级，使用PSP，并且没有弄丢局部变量

UserApplicationStart(); // 在用户级下开始执应用程序

## 12.2 双字的堆栈对齐方式

在符合AAPCS的应用程序中，对于响应异常时的堆栈操作，是有必要对齐到原始（primitive）的数据尺寸的（1,2,4或8字节）。这是CM3的一个可配置选项。欲使能此特性，需要把NVIC配置控制寄存器的STKALIGN置位（在附录D的表D.17给出定义），如下面汇编代码所演示：

LDR R0, =0xE000ED14 ; R0=NVIC CCR的基址

LDR R1, [R0]

ORR.W R1, R1, #0x200 ; 设置STKALIGN位

STR R1, [R0] ; 更新NVIC CCR

如果使用C语言，则代码如下：

#define NVIC\_CCR ((volatile unsigned long \*)(0xE000ED14))

\*NVIC\_CCR= \*NVIC\_CCR | 0x200; /\* 设置STKALIGN位\*/

如果在入栈时STKALIGN位为1，则xPSR的位9功能启用，指示在入栈时SP的值是否为了对齐而作出了调整。在出栈时，会检查入栈的xPSR.9，再根据它的值把SP的值调整回去。

注意：切勿在异常服务例程中改动STKALIGN位的值，否则会使出栈时数据发生错位，彻底破坏各个寄存器的值，这常常是致命错误（跑飞，死机等）。

要注意的是，这个特性是在CM3修订版1开始后才引入的，早期基于版本0的产品则无此功能。当需要符合AAPCS时，需要启用此特性。此外，当程序的一部分是使用C开发，且程序中包含了对双字数据的处理时，也推荐启用此功能。

在最新的修订版2中，该特性不需手动使能，而是在缺省时已使能。在使用C开发时，如果程序包含了需要双字尺寸的数据类型（double, long long / INT64）时，推荐使能此特性。

## 12.3 非基级的线程模式

在CM3中，原则上异常服务程序要在handler模式下执行，但是也允许在服务例程中切换到线程模式。通过设置NVIC配置与控制寄存器的“非基级线程模式允许”位（NONBASETHRDENA，位偏移：0），可以在服务例程中把处理器切换入线程模式。为什么要这么做？如果中断服务例程是用户程序的一部分，可能需要让它在线程模式下执行，以限制它访问特权级下的资源，此时可以让此功能派上用场（对于让CM3在线程模式下赋予用户级访问权限的配置，不在本节中完成，而是在上电初始化时就一次性地做好了——译者注）。

#### 小心地使用此功能

如果使用此功能，则需要手工调整堆栈指针，还要重建堆栈中的数据。这种乾坤大挪移可是高度危险的作业，一不小心就很容易把整个系统弄垮。所以必须格外严肃地对待。另外，在使用时，系统设计者还必须保证服务例程能正确地返回。因为在线程模式下是不允许作中断返回的，所以必须用一点手腕才行。如果放任不管，则中断无法退出，这会永远阻塞其它同级和更低优先级中断。通常，由系统软件负责完成这种工作。

在启用本功能时，必须伴随着一个“服务例程重定向”动作：中断向量指向一个运行在特权级的服务例程，但它却是应该只访问用户级内存的，因此必须先在头部切入用户级，调用真正干活的服务例程，再在最后回到特权级。演示代码如下所示：

redirect\_handler

PUSH {LR}

SVC #0 ; 呼叫系统服务，用于把特权级别改为用户级

BL User\_IRQ\_Handler

SVC #1 ; 执行完中断处理后，回到特权级

POP {PC} ; 启动本次中断的返回序列

上例中，字体不同的中间两行是在线程模式+PSP下执行的。在这段代码中，通过首尾的两个系统调用来完成乾坤大挪移：

使用SVC #0，它先使能非基级线程模式，再拷贝主堆栈中被压入的8个寄存器到进程堆栈并更新PSP的值，最后修改EXC\_RETURN，以使返回后进入“线程模式.用户级+PSP堆栈”

使用SVC #1来使一切归位，它除能非基级模式，恢复PSP先前的位置，并且修改EXC\_RETURN以返回到特权级，继续使用主堆栈。

在最后执行到返回指令后，则终结了本次异常处理序列。虽然redirect\_handler的内部有这么多的暗箱操作，但是在表面上看还是很傻很天真的，也就5行“安分守己”的指令而已。

听起来很神吧，那就让我们把这个内幕曝光。这可是一道大荤菜，可以尝尝系统程序大肉的味道。这个菜是很“油”的，最好边吃边看图12.4来帮助消化。使用上一章讲到的SVC服务例程框架，在这里搭成了真正能干活的系统服务：

svc\_handler

; 小测试：请读者为本段代码加注释

TST LR, #0x4 ; 测试EXC\_RETURN.2

ITE EQ ; 如果为零则

MRSEQ R0, MSP ; 先前使用的是主堆栈，把MSP的值加载到R0

MRSNE R0, PSP ; 否则，先前使用的是进程堆栈，把PSP的值加载到R0

LDR R1, [R0, #24] ; 读取入栈的返回值

LDRB R0, [R1, #-2] ; 提出8位立即数调用代号

CBZ r0, svc\_service\_0

CMP r0, #1

BEQ svc\_service\_1

B.W Unknown\_SVC\_Request

svc\_service\_0 ; 0号服务：切换到“线程模式+PSP”

MRS R0, PSP ; 读取PSP

SUB R0, R0, #0x20 ; 开出32字节的空间存储8个寄存器

MSR PSP, R0 ; 更新PSP的值

MOV R1, #8\*4 ; R1作为拷贝堆栈帧（8个寄存器）的循环变量

svc\_service\_0\_copy\_loop

SUBS R1, R1, #1\*4

LDR R2, [SP, R1]

STR R2, [R0, R1]

CMP R1, #0

BNE svc\_service\_0\_copy\_loop

STRB R1, [R0, #0x1C] ; 在进程堆栈中清零IPSR

LDR R0, =0xE000ED14 ; 加载NVIC中CCR（配置与控制寄存器）的地址

LDR r1, [r0]

ORR r1, #1

STR r1, [r0] ; 使能非基级线程模式（这里的地址不在位带操作区）

ORR LR, #0xC ; 修改EXC\_RETURN，以使得返回后进入 线程模式+PSP

BX LR ; 启动异常返回序列，执行动作

svc\_service\_1 ; 1号服务：从线程模式+PSP返回到handler模式

MRS R0, PSP ; 读取PSP到R0，以便于后续的一系列归位处理

LDR R1, [R0, #0x18] ; 读取压入PSP中的返回地址（即svc #1后面的

; POP {PC}）

STR R1, [SP, #0x18] ; 因为将要返回到handler模式，所以把它转移到MSP

ADD R0, R0, #0x20 ; 把PSP的值归位——刚响应外部中断时的值

MSR PSP, R0 ; 用归位后的值更新PSP

LDR R0, =0xE000ED14 ; NVIC中配置与控制寄存器（CCR）的地址

LDR r1, [r0] ; 再次读取NVIC中的CCR

BIC r1, #1

STR r1, [r0] ; 清除NONBASETHRDENA位，

BIC LR, #0xC ; 修改EXC\_RETURN以返回handler模式，MSP亦归位

BX LR

使用SVC是必须的，因为只有通过异常返回才能改动IPSR的值。软件触发中断也能用，但那种偏方是旁门邪道，因为它是不精确的，而且可能被阻塞（回顾前一章），带来了隐患——使得堆栈拷贝与切换操作不被立即执行。

图12.4给出了上述代码的工作序列图，如果吃不消的话就赶快看吧：

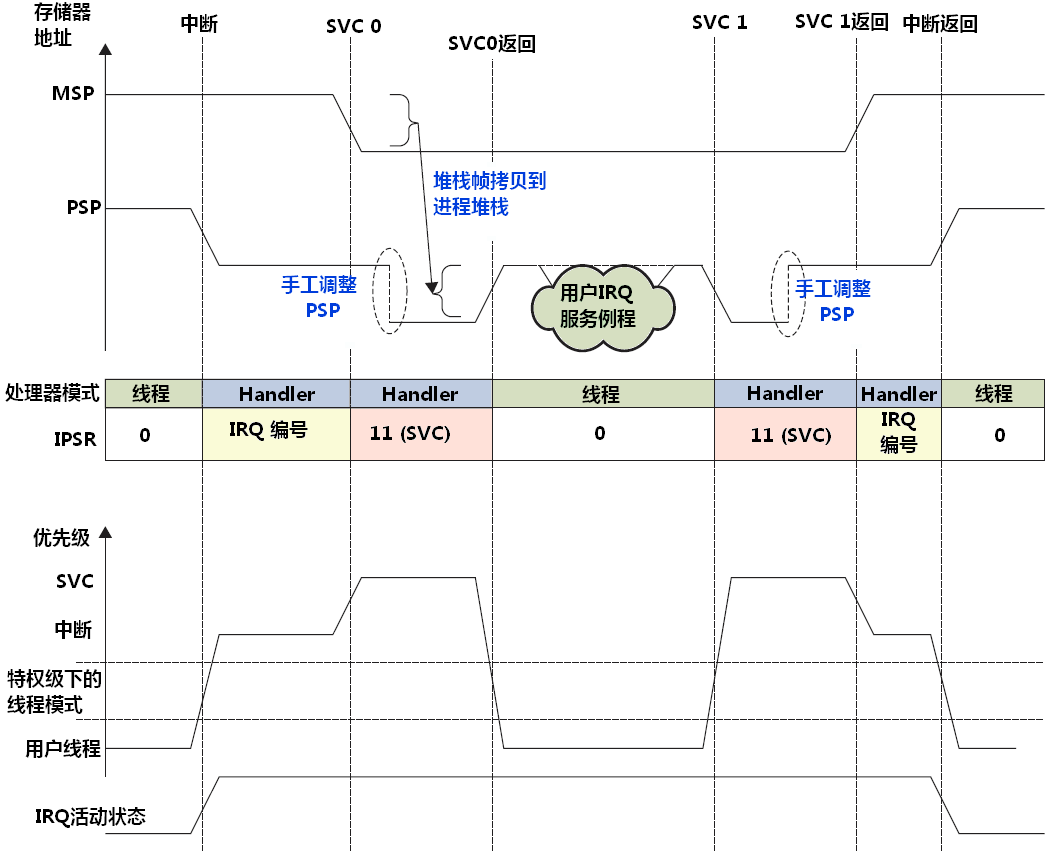


图12.4 非基级线程模式操作模式图

手工调整PSP也是必须的。如果没有第一次调整，则在借SVC0返回的形式进入用户IRQ服务例程后，会使PSP回到进入中断前的状态。然后在执行“svc #1”时，将重新把寄存器压入栈——但此时的寄存器已经是被用户IRQ服务例程用过的了！结果，虽然PSP的值与两次调整后的还相同，但是PSP中寄存器内容已经被破坏了！

对MSP的调整也是很有魄力的，它突破了嵌套的异常在返回时，一定要从MSP出栈的教条。这段代码中对MSP和PSP的把玩，是不是很精湛？在真实的操作系统中，还有更刺激的动作。

## 12.4 性能评估

为了让CM3能尽情地释放她的青春能量，还需要我们清扫路上其它石子。

第一，要消灭存储器等待周期。在MCU/SoC的设计期间，就应该优化存储器系统，最起码的 要求，也要允许取指和数据访问能并行不悖，这才对得起“哈佛结构”的称号。此外，应尽可地使用32位的存储器。对于软件开发人员，还应该划清代码与数据的界线，使得程序代码从代码区执行（使用I-Code总线），而绝大部分数据都从数据区访问（使用System总线，而不要使用D-Code总线），哪怕是多浪费点内存。只有这样，才能使取指与访问数据同时进行。

第二，如果没有必要，中断向量表也放到代码区中。只有这样，才能使取向量(I-Code总线)与入栈(System总线)同时进行。如果向量表在RAM中，就会出现取向量与入栈抢总线的情况，必然导致额外的中断延迟被引入（当然在极个别情况下，如果把SRAM放到Code区，则使用D-Code总线入栈。但如果就为了放向量表而专配一个SRAM，代价未免也太大了）。

第三、限制使用非对齐访问。前面讲到，CM3总线内部其实只接受对齐访问，而由总线接口来堵窟窿：把一个非对齐的访问拆成若干个对齐的访问，来实现这种透明性。可见，一次非对齐访问可能要数次对齐访问才能完成（最坏情况下3次）。而且节省内存的正道，在于优良的程序结构和算法设计，从来不在这种见缝插针地乱挤上。除非是客观上被定死的（常见于某些早期网络协议的报文头部），否则应在心里暗下决心：决不染指非对齐访问，在设计数据结构及定义变量时，都高度自觉。在ARM汇编器中，提供了ALIGN指示字（GNU AS中也有类似的汇率器指示字），可以保证产生所需的对齐方式。

虽然我们会在绝大多数场合下使用C来开发，但是在为某个关键的功能启动“汇编级待遇”时，不要忘了使用下述的技巧，它们经常能产生意想不到的特效：

1. 使用带偏移量寻址的LDR/STR指令，进一步地，还可以对偏移量作侈位预处理（LSL用得最多）。使用这种强大的寻址方式，常常能省去分立的地址增减/乘除计算操作。重温一下上一章中使能中断和使能异常子程的不同，相信会有切身的体会
2. 把上下文相关的变量放到一起——也就是说使它的地址是连续的。这样就可以创造使用LDM/STM指令的机会。只要遇到连续地址的数据传送，就使用LDM/STM。一条传送14个字的LDM指令，可远比14个LDR要快多了，而且代码也巨幅精简
3. 当遇到很小的“if then”块时，如果使用条件跳转指令，则会使流水线被清洗，花不少时间。这时，应使用IF-THEN指令（ITxxx）。IT指令在张开双臂时，最多能保护4个孩子。
4. 如果旧时需要两条Thumb指令才能完成的操作，现在可以由一条Thumb-2指令完成，则应使用Thumb-2指令。
5. 为使自己成长为大虾，要学会使用CM3的新好指令。尤其是在ARMv6后才新出来的，都是无数前人经验的结晶，常常能有戏剧般地优化（回顾RBIT与CLZ的梦幻组合）

## 12.5 当处理器被锁定(Lockup)时

这确实是很扎手的问题：本来就已经因为出错而进入fault服务例程了，结果fault服务例程也触犯了fault条件，升级为硬fault的。可如果硬fault服务例程也脑子进水了怎么办？一错再错，最终使CM3在万般无奈下进入锁定状态。万万要避免它，因为一旦锁定就不可救要了——几乎只能复位，这在使命-关键（mission-critical）系统中是决不允许的（像那种大型交换机、体外循环机等设备）。

### 12.5.1 锁定情形下的众生相

在锁定下，寄存器和存储器都被“冻结”，PC的值被强制为0xFFFF\_FFFx，并且原地打转地定死在那里一直取指。与此同时，CM3的另一条名叫“LOCKUP”的输出信号线将被置为有效，芯片厂商可以检测此信号，并且在系统复位发生器上触发一个复位。

具体地说，下列场合会导致锁定：

* + 在硬fault服务例程中产生faults（双重fault）
  + 在NMI服务例程中产生faults
  + 在复位序列（初始的MSP与PC读取）中产生总线fault

在双重fault下，NMI还能响应（再次证明了它的第一优先地位）。然而在NMI服务例程退出后，又回到锁定状态。此时，当前优先级为-1，因此可以响应NMI——NMI的优先级是-2，比当前的高。

在产品中出现锁定就等同于是大限已到。但是在调试阶段也许还能让系统起死回生：如果连接了调试器，则可以喊停（halt）处理器，然后手工修改PC的值。然而这也往往是无力的：因为上下文没有了——所有的寄存器，以及中断系统，都已经物是人非，需要重新初始化，才能返回到正常的操作中。

那为什么不直接复位，好让它早点在下个轮回中转世投胎呢？原来，系统的生命是开发者赋予的，因此就要对它的生死负责。哪怕死了，也要明白死因才行。如果当场就复位了，则所有寄存器的值都归位了，不再有机会去查明当时的情况。

如果不是使命-关键系统，则可以使用一个看门狗，它可以使系统从锁定状态中复位。

还要注意的是，如果在响应NMI或硬fault的入栈/出栈阶段触发了总线fault，则不会导致锁定，只是会悬起总线fault，如图12.5所示。

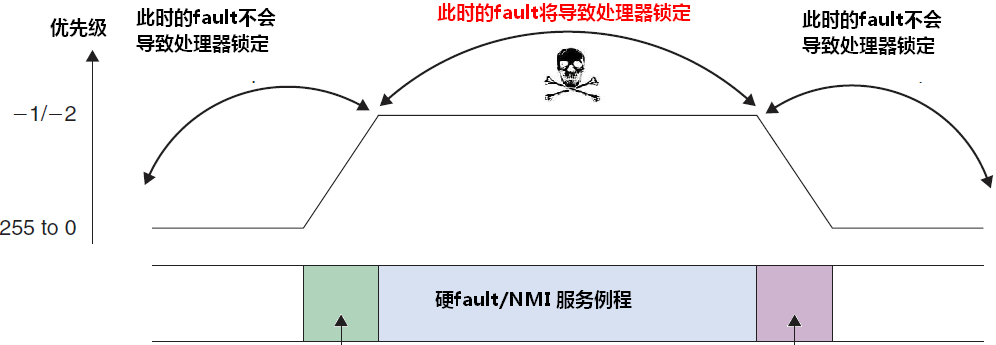


图12.5 只有在硬fault/NMI 服务例程中的fault才锁定系统

### 12.5.2 避免被锁定

既然被锁定就等同于死机，我们唯一能做的也只能是避免锁定状态。因为锁定只出现于NMI和硬fault的服务例程中，所以当我们在设计它们时，一定要分外地小心，就好像亲手给自己的爱人做大手术那样地一丝不苟。比如，我们应该尽量避免不必要的堆栈访问，这是有原因的。对于NMI来说，因为在进入NMI时常常是在危急关头，如：掉电，短路等硬件故障。此时，有可能存储系统已经失能了。而对于硬fault来说，有可能就是因为SP指针指飞了（干扰、堆栈溢出等），以致前面的堆栈操作触发了本次硬fault，再操作堆栈还不当场被秒杀？ 如下面代码所警示：

hard\_fault\_handler

**~~PUSH {R4-R7,LR}~~** ; 除非确保堆栈是安全可用的（谁能确保？），否则不要这样做

...

值此危难关头，必须沉着冷静。在我们设计硬fault，总线fault以及存储管理fault的服务例程时，值得先花点工夫去查一查SP的值，看它是否在可接受的范围，然后再做后续工作。对于NMI服务例程来说，它做的通常是应急工作，设计系统时就应该让这种应急工作极简单（比如，只改变一个I/O脚的电平，最多也就是修改若干寄存器的值，就可以开启相关的应急硬件——译者注），因此常常可以只使用R0-R3以及R12就完全够用，无需堆栈操作。

简化硬fault和NMI的服务例程确实是个好主意：它们只做必需的，然后悬起PendSV，让诸如错误报告等其它工作在PendSV中处理，当然，软件中断兴许也能凑和着用。

除此之外，我们还必须杜绝在硬NMI/fault例程中使用SVC指令，这也是斩立决的——因为SVC的优先级总是没有NMI和硬fault的高，而且它又不允许悬起（悬起时触发fault）。这看起来很容易做到，那是饱汉不知道饿汉饥——当程序变得复杂，并且如果NMI/硬fault服务例程中调用了其它目标文件中的函数，就不能保证这些函数中没有使用过SVC。因此，在开发软件时，必须仔细地计划如何实现SVC。或者获取所调用函数的说明文档，确保不会出事。