第11章

# 玩转异常系统

* 使用中断
* 异常/中断服务例程
* 软件中断
* 异常服务例程的示范
* 使用SVC
* SVC示范：用于输出数据的函数
* 在C中使用SVC

NMI，Faults，SVC，PendSV，IRQ #0, IRQ #1，……

自动栈操作、向量式、抢占、咬尾、晚到……

CM3把“中断/异常”这个概念捧到了登峰造极的境界，为实时系统的开发垫上了那么一个宽大的肩。如果在CM3上开发却不能善用这炙手可热的能力，那说不定都会有一种暴殄天物的感觉！

## 11.1 使用中断

任何一个有点型的嵌入式系统，就没有不使用中断机制的。在CM3中，NVIC为我们搞定了使用中断时的很多例行任务，如优先级检查、入栈/出栈、取向量等。不过在NVIC能行使职能之前，还需要我们做好如下的初始化工作：

* + 建立堆栈
  + 建立向量表
  + 分配各中断的优先级
  + 使能中断

### 11.1.1 建立堆栈

当开发的程序比较简单时，可以从头到尾都只使用MSP。这时，只需要保证开出一个容量够大的堆栈，再把MSP初始化到其顶即可——这也是单片机开发最常见的做法。

堆栈用穿是非常致命的错误，必须非常严肃地计算安全容量。在计算时，除了要计入最深函数调用时对堆栈的需求，还需要判定最多可能有多少级中断嵌套。一个笨方法（但是很保险）是假设每个中断都可以嵌套。对于每一级嵌套的中断，至少需要8个字（32字节），而且如果ISR过于复杂，还可能有更多的堆栈需求。

因为CM3中的堆栈是以“向下生长的满栈”来操作SP的。在简单的场合中，经常可以把SP初始化为SRAM的末尾，这么一来就使所有的空闲内存都能为堆栈所用——反正不用白不用，用了也白用，如图11.1所示。

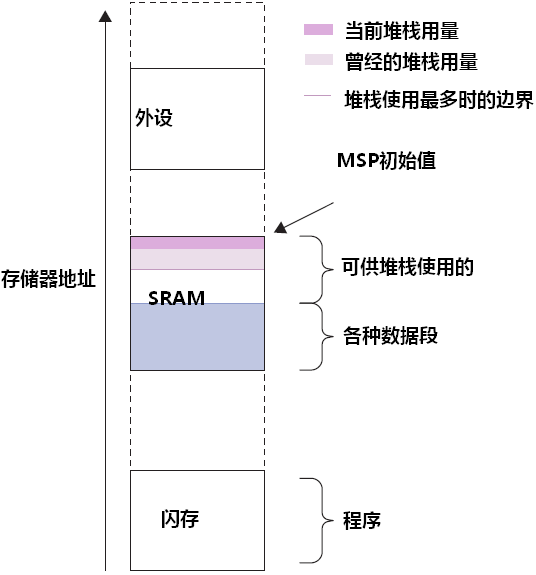


图10.1 简单程序中典型的存储器分配

从图中我们可以看出，这种分配方式能给堆栈区留下最大的容量——所有剩余内存，而有省事又省心——省去了令人头痛的堆栈需求计算了。

然而，对于比较大型的或者是有高性能指标的嵌入式系统，往往需要两个堆栈配合使用。这时，就只好勇敢地面对。必须保证各堆栈都有足够的容量，尤其是主堆栈，最容易栽在它上面。要注意的是，进程堆栈除了要满足本进程的最大需求量，还需要额外留出8个字，用于容纳第一级中断时被保护的寄存器。

（译者添加）事实上，准确计算主堆栈需求往往是不可能的任务，也容易过于保守而浪费宝贵的血液资源。在调试阶段时，最好先选用内存更大点的器件，然后开出足够大的内存给主堆栈。然后在调试程序时，允许随时把主堆栈曾经的最大用量输出（通过调试串口或仿真器等），这样时间长了就能估算对主堆栈的需求，正如图10.1中边界的作用。

### 11.1.2 建立向量表

如果在程序执行的从头到尾，都只给每个中断提供固定的中断服务例程（这也是目前单片机开发的绝大多数情况），则可以把向量表放到ROM中。在这种情况下不需要运行时重建向量表。然而，如果想让自己的设备能随机应变地对付各种复杂情况，就常常需要动态地改变中断服务例程，更新向量表就是必需的了。此时，向量表必须被转移到可读写存储器中（如内存）。

在把向量表重定位之前，往往要把现有的向量表往新的位置复制一份。需要拷贝的向量主要是系统异常的服务例程，如各种fault的、NMI的以及SVC的等等。如果没有建立好这些向量就启用了新的向量表，就可能会在响应异常时把不可预料的地址取出，程序极有可能跑飞。

当我们把所有必要的向量都填好后，就可以启用了新的向量表了。然后继续往里面加入新的中断向量，例如：

; 该子程序根据异常类型建立相应的异常向量

; 对于IRQ，异常号=中断号+16

; 入口条件：R0=异常类型编号

; 入口条件：R1=向量地址

PUSH {R2, LR}

LDR R2, =0xE000ED08 ; 向量表偏移量寄存器的地址

LDR R2, [R2] ; 获取向量表的首地址

STR R1, [R2, R0, LSL #2] ; 在VectTblOffset+ExcpType\*4处写入向量

; ExcpType\*4

POP {R2, PC} ; Return

### 11.1.3 建立中断优先级

在复位后，对于所有优先级可编程的异常，其优先级都被初始化为0。而对于NMI和硬fault，由于它们要在危难之际挺身而出，所以把它们的优先级定死为-2和-1（高于任何其它异常）。在编程优先级寄存器时，我们可以利用它们能按字节访问的好处，以简化程序代码，如：

; 把IRQ #4的优先级设为0xC0

LDR R0, =0xE000E400 ; 加载外部空优先级寄存器阵列的起始地址

LDR R1, =0xC0 ; 优先级

STRB R1, [R0, #4] ; 为IRQ #4设置优先级（按字节写）

在CM3中，允许使用3个位到8个位来表达优先级。为了确定具体的位数，可以先往一个优先级寄存器中写0xFF，再读回来，读出多少个1，就表示使用多少个位来表达优先级，如下所示（下段代码演示了RBIT配CLZ的绝技）：

; 检测系统使用多少个位来表达优先级

LDR R0, =0xE000E400 ; 加载IRQ #0的优先级配置寄存器

LDR R1, =0xFF

STRB R1, [R0] ; 按字节写，写入0xFF

LDRB R1, [R0] ; 读回（如果是3位，则应读取回0xE0）

**RBIT R2, R1 ; 反转，使之以LSB对齐**

**CLZ R1, R2 ; 计算前导零个数（例如，如果是3个1则返回5）**

MOV R2, #8

SUB R2, R2, R1 ; 得到表达优先级的位数

MOV R1, #0x0

STRB R1, [R0] ; 存储结果

如果程序可能要跨器件移植（常见于比较底层的基础设施函数），那么最好只使用最高3个有效位，对应的优先级为：0x00, 0x20, 0x40, 0x60, 0x80, 0xA0, 0xC0以及0xE0。所有的CM3芯片都一定支持3个位表达的优先级。

还要提醒的是，不要忘记为系统异常（包括faults）建立优先级。如果程序中有非常紧急的外部中断，它们甚至需要比系统异常还紧急，可是却因故不能连接到NMI上，就要把系统异常的优先级调低，才能保证紧急的中断能够抢占系统异常，从而不被延误。

### 11.1.4 使能中断

在向量表与优先级都建立好后，就到了最后一步：开中断的时候了。

然而，在打开中断之前，可能还有两个步骤不能省略：

1. 如果把向量表重定位到了RAM中，且这块RAM所在的存储器区域是写缓冲的，向量更新就可能被延迟。为了以防万一，必须在建立完所有向量后追加一条“数据同步隔离(DSB)”指令（见第4章），以等待缓冲写入后再继续，从而确保所有数据都已落实。
2. 开中断前可能已经有中断悬起，或者请求信号有效了，这往往是不可预料的。比如，在上电期间，信号线上有发生过毛刺，就可能会被意外地判定成一次中断请求脉冲。另外，在某些外设，如UART，在串口连接瞬间的一些噪音也可以被误判为接收到的数据，从而使中断被悬起。

在NVIC中进行中断的使能与除能时，都是使用各自的寄存器阵列(SETENA/CLRENA)来完成的：通过往适当的位写1来发出命令，而写0则不会有任何效果。这就让每个中断都可以自顾地使能和除能，而不必担心会破坏其它中断的设置。这改变了以前必须“读-改-写”的三步曲，从而在根本上消灭了在此地产生紊乱危象的可能；否则，必须使用互斥访问等机制来完成修改。下例就演示了通过置位SETENA中的位来使能中断；通过**置位**CLRENA中的位来除能中断：

1. 使能中断

; 根据IRQ号来使能中断的子程序

EnableIRQ

; 入口条件：R0=中断号

PUSH {R0-R2, LR}

AND.W R1, R0, #0x1F ; 为该IRQ产生移位量

MOV R2, #1

LSL R2, R2, R1 ; 位旗标 = (0x1 << (N & 0x1F))

AND.W R1, R0, #0xE0 ; 若IRQ编号>31则为它生成下标偏移量

LSR R1, R1, #3 ; 地址偏移量= (N/32)\*4（每个IRQ一个位）

LDR R0, =**0xE000E100** ; 加载SETENA寄存器阵列的首地址

STR R2, [R0, R1] ; 写入该中断的位旗标，从而使能该中断

POP {R0-R2, PC} ; 子程返回

1. 除能中断

几乎是照抄上一个例程，就得到了对应的除能中断的子程序：

DisableIRQ

; 入口条件：R0=中断号

PUSH {R0-R2, LR}

AND.W R1, R0, #0x1F ; 为该IRQ产生移位量

MOV R2, #1

LSL R2, R2, R1 ; 位旗标 = (0x1 << (N & 0x1F))

AND.W R1, R0, #0xE0 ; 若IRQ编号>31则为它生成下标偏移量

LSR R1, R1, #3 ; 地址偏移量= (N/32)\*4（每个IRQ一个位）

LDR R0, =**0xE000E180**  ; 加载CLRENA寄存器阵列的首地址

STR R2, [R0, R1] ; 写入该中断的位旗标，从而除能该中断

POP {R0-R2, PC} ; 子程返回

访问NVIC寄存器的小贴士

在NVIC中，绝大多数寄存器都可以按字/半字/字节的形式访问。对于不同的场合，应灵活使用适当的形式，以简化程序的开发。比如，对优先级寄存器的按字节访问，就消除了按字/半字访问时，需要“读-改-写”的序列（为的是不影响其它中断的优先级）。

## 11.2 异常/中断服务例程

在CM3中，中断服务例程可以纯用C来写。与ARM7的情况相比，后者则往往需要首尾都加以汇编封皮，用以保证所有寄存器都保护了。另外，在中断嵌套时，处理器需要切换到另外的模式，以防止信息丢失。这些拖跨系统实时性和带来入门难度的繁文缛节在CM3中都被消灭了，使得编程时舒心很多。

如果用汇编来写ISR，其骨架看上去差不多如下所示：

irq1\_handler

; 处理中断请求

...

; 消除在设备中的IRQ请求信号

...

; 中断返回

BX LR

如果ISR逻辑比较复杂，则常常需要更多的寄存器，这时就要启用R4-R11了。但是它们不是CM3自动入栈的，所以使用前必须手工PUSH。下一个例子演示一个保险的笨方法：保护了所有的寄存器。其实如果内存够用，使用笨方法作为起点也不失为一个不错的主意，等到日后优化程序时再去掉没有使用的寄存器。

irq1\_handler

PUSH {R4-R11, LR} ; 保存所有可能用到的，又没有被自动入栈的寄存器

; 处理中断请求

...

; 消除在设备中的IRQ请求信号

...

; 中断返回

POP {R4-R11, PC}

因为POP也是启动中断返回的一条途径，所以我们把寄存器出栈与中断返回合并在一条POP中，使程序更精练。

有些外设的中断请求信号需要ISR手工清除，如：外设的中断请求是持续的电平信号——显然，对于稍纵即逝的脉冲型的请求，是无需手工清除的。若电平型中断请求没有清除，则中断返回后将再次触发已经服务过的中断。以前在ARM7中，外设必须使用这种“电平保持”的方式[译注]，直到中断被响应，因为那个时候的中断控制器没有保存悬起状态。在CM3中就解决了这个问题：只要检测到过曾经出现的中断请求，NVIC就会记住它，因此硬件只需给一个脉冲，无需再一直保持请求电平，持续的电平反而成为一种讨厌的事了。而且当其服务例程得到执行时，NVIC自动把悬起状态清除。对于这种情况，就不必在ISR中软件清除请求信号了。

译注：很多厂家都在设计ARM7芯片时添加了自己的中断控制器，这些中断控制器也常常能记住请求脉冲。

## 11.3 软件触发中断

触发中断有多种方法：

* + 外部中断输入
  + 设置NVIC的悬起寄存器中设置相关的位（第8章）
  + 使用NVIC的软件触发中断寄存器（STIR）（第8章）

系统中总是会有一些中断没有用到，此时就可以当作软件中断来使用。软件中断的功用与SVC类似，两者都能用于让任务进入特权级下，以获取系统服务。不过，若要使用软件中断，必须在初始化时把NVIC配置与控制寄存器的USERSETMPEND位置位，否则是不允许用户级下访问STIR的（附录D的表D.17有该寄存器的详细说明）。

但是软件中断没有SVC专业：比如，它们是不精确的，也就是说，抢占行为不一定会立即发生，即使当时它没有被掩蔽，也没有被其它ISR阻塞，也不能保证马上响应。这也是写缓冲造成的，会影响到与操作NVIC STIR相临的后一条指令：如果它需要根据中断服务的结果来决定如何工作（如条件跳转），则该指令可能会误动作——这也可以算是紊乱危象的一种表现形式。为解决这个问题，必须使用一条DSB指令，如下例所示：

MOV R0, #SOFTWARE\_INTERRUPT\_NUMBER

LDR R1,=0xE000EF00 ; 加载NVIC软件触发中断寄存器的地址

STR R0, [R1] ; 触发软件中断

**DSB**  ; 执行数据同步隔离指令

...

那是否这样就万事大吉了呢？不幸的是，还不能高兴得太早，因为还有另一个隐患：如果欲触发的软件中断被除能了，或者执行软件中断的程序自己也是个异常服务程序，软件中断就有可能无法响应。因此，必须在使用前检查这个中断已经在响应中了。为达到此目的，可以让软件中断服务程序在入口处设置一个标志。

最后要注意的是，虽然是出于好心置位USERSETMPEND，但容易烧香引出鬼来：因为用户程序可能会以软件的方式触发任何一个中断，制造出各种“假象”。如果系统中包含了不受信任的用户程序，就必须全体接种疫苗——每个异常服务例程都必须检查该异常是否允许。其实，通向天堂是有路的——干嘛不用更专业的SVC来实现系统服务呢？

## 11.4 异常服务例程的范例

回忆第7章，我们曾提到，不管应用程序多简单，都必须在向量表中包含下列三项：复位向量、NMI向量以及硬fault向量，这是因为后两者无需使能就可以发生。在程序运行后，有时还会把向量表重定位的SRAM中。下面就演示一种重定位的情况：把向量表转移到SRAM的起始处，并且在它的后面定义数据区——存储各种全局和静态变量。程序有点长，但很多部分以前都见过了，不要怕！

STACK\_TOP EQU 0x20002000 ; MSP初始值

NVIC\_SETEN EQU 0xE000E100 ; SETENA寄存器阵列的起始地址

NVIC\_VECTTBL EQU 0xE000ED08 ; 向量表偏移寄存器的地址

NVIC\_AIRCR EQU 0xE000ED0C ; 应用程序中断及复位控制寄存器的地址

NVIC\_IRQPRI EQU 0xE000E400 ; 中断优先级寄存器阵列的起始地址

AREA | Header Code|, CODE

DCD STACK\_TOP ; MSP初始值

DCD Start ; 复位向量

DCD Nmi\_Handler ; NMI服务例程

DCD Hf\_Handler ; 硬fault服务例程

ENTRY

Start ; 主程序开始

; 初始化各寄存器

MOV r0, #0

MOV r1, #0

...

; 把各个向量拷贝到新向量表中

LDR r0, =0

LDR r1, =VectorTableBase

LDMIA r0!, {r2-r5} ; 拷贝4个字（MSP, Reset, NMI, 硬fault）

STMIA r1!, {r2-r5}

DSB ; 数据同步隔离

; 执行向量表重定位：

LDR r0, =NVIC\_VECTTBL

LDR r1, =VectorTableBase

STR r1, [r0]

...

; 设置优先级组寄存器，划分抢占优先级与亚优先级

LDR r0, =NVIC\_AIRCR

LDR r1, =0x05FA0500 ; 从位5处划分（共2个位表达抢占优先级）

STR R1, [r0]

; 建立IRQ0的向量

MOV r0, #0 ; IRQ#0

LDR r1, =Irq0\_Handler

BL SetupIrqHandler

; 建立IRQ #0的优先级

LDR r0, =NVIC\_IRQPRI

LDR r1, =0xC0 ; IRQ#0的优先级

STRB r1, [r0,#0] ; 写入优先级寄存器中，用了按字节传送

DSB ; 数据同步隔离，保证开中断前一切都已各就各位

MOV r0, #0 ; 选择IRQ #0

BL EnableIRQ

...

;------------------------

; 各函数

SetupIrqHandler

; 入口条件：R0 = IRQ编号

; 入口条件：R1 = IRQ服务例程的入口地址

PUSH {R0, R2, LR}

LDR R2, =NVIC\_VECTTBL ; 获取向量表的地址

LDR R2, [R2]

ADD R0, #16 ; 异常号 = IRQ编号 + 16

LSL R0, R0, #2 ; 乘以4 (每个向量4字节)

ADD R2, R0 ; 找出向量地址

STR R1, [R2] ; 写入服务例程

POP {R0, R2, PC} ; 返回

EnableIRQ

; 入口条件：R0=中断号

PUSH {R0-R2, LR}

AND.W R1, R0, #0x1F ; 为该IRQ产生移位量

MOV R2, #1

LSL R2, R2, R1 ; 位旗标 = (0x1 << (N & 0x1F))

AND.W R1, R0, #0xE0 ; 若IRQ编号>31则为它生成下标偏移量

LSR R1, R1, #3 ; 地址偏移量= (N/32)\*4（每个IRQ一个位）

LDR R0, =NVIC\_SETEN ; 加载SETENA寄存器阵列的首地址

STR R2, [R0, R1] ; 写入该中断的位旗标，从而使能该中断

POP {R0-R2, PC} ; 子程返回

;------------------------

; 异常服务程序

Hf\_Handler

... ; 在此添加硬fault的处理代码

BX LR

Nmi\_Handler

... ; 在此添加NMI的响应代码

BX LR

Irq0\_Handler

... ; 在此添加IRQ #0的响应代码

BX LR ; Return

;------------------------

AREA | Header Data|, DATA

ALIGN 4

; 重定位的向量表

**VectorTableBase SPACE 256** ; 保留256字节作向量表

VectorTableEnd ; (256 / 4 = 最多支持64个异常)

MyData1 DCD 0 ; 定义变量

MyData2 DCD 0

END ; 文件结尾

这个例子是长了点，让我们再从后往前看。在程序的尾部，定义了数据存储区。通过SPACE汇编指示字，我们为向量表开出了256字节的内存空间，从而可以容纳64个异常向量。如果把256改成别的数，就能改变向量表的长度。在向量表的后面，还定义了两个变量。第一个变量MyData1紧挨着向量表，所以它的地址是0x2000\_0100，第二个MyData2是为0x2000\_0104。（不过，通常情况下，强烈反对使用这种以计算的方式来求得变量地址。因为很容易出错，而且只要以后再新插入新的变量定义，则所有插入位置后面的变量地址也都要重新计算，因为它们被“拱”到后面去了——译者注）。

再看程序的起头，在那里我们一上来就定义了若干个地址常数（NVIC寄存器的地址），由整个程序使用。通过使用一个有意义的名字取代直接抄地址，程序就更容易理解，也减少了出错。

在初始的向量表中，包含了复位向量、NMI向量，以及硬fault向量，它们是三要素。后面的代码还给出了服务例程的骨架。在开发应用程序时，必须根据程序的指标来实现这三要素的服务例程，不可省略。

这里的服务例程都是使用BX LR返回的，但是真到了写程序时，往往利用POP ｛…,PC｝的形式来使程序更精练（当然也可以使用LDMIA指令）。

进入主程序后，先初始化寄存器，然后，就通过LDM/STM，把向量一次多个地拷贝到新的向量表中。如果后来又添加了新的向量，则可以在LDM/STM中增加数量，或者再多用一对LDM/STM，这些都是很简单的事。

在准备好了向量表后，就可以编程NVIC，启用新的向量表了。但是在启用前，为了保证在向量拷贝都完成后才做下一步，我们还用了DSB指令来隔离。

接下来继续做与中断设置相关的工作，第一个就是建立优先级组。

这些初始化都是一劳永逸的。本例中，使用了两个子程序来完成中断的建立，从而使程序结构更清晰。其中SetupIrqHandler负责在向量表建立中断服务例程的入口地址，而EnableIRQ则用于在NVIC中使能一个中断。在为一个中断建立好优先级后，就可以使能它。如果还需要除能中断，则可以照葫芦画瓢地就能当场造出一个DisableIRQ来，只是SETENA改成了CLRENA。

## 11.5 使用SVC

SVC是用于呼叫OS所提供API的正道。用户程序只需知道传递给OS的参数，而不必知道各API函数的地址。

SVC指令带一个8位的立即数，可以视为是它的参数，被封装在指令本身中，如：

SVC 3 ;呼叫3号系统服务

则3被封装在这个SVC指令中。因此在SVC服务例程中，需要读取本次触发SVC异常的SVC指令，并提取出8位立即数所在的位段，来判断系统调用号，工作流程如11.2所示：

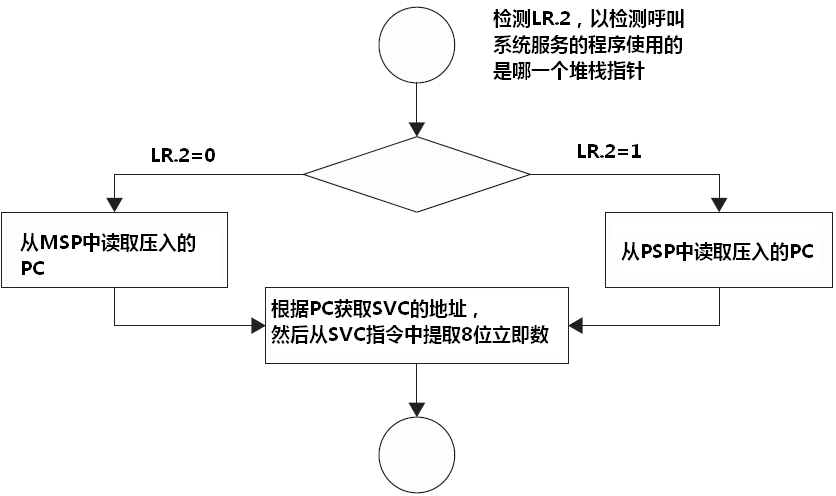


图11.2 提出SVC中立即数的一种途径

实现上图功能的代码如下所示：

svc\_handler

TST LR, #0x4 ; 测试EXC\_RETURN的比特2

ITE EQ ; 如果为0,

MRSEQ R0, MSP ; 则使用的是主堆栈，故把MSP的值取出

MRSNE R0, PSP ; 否则, 使用的是进程堆栈，故把MSP的值取出

LDR R1, [R0,#24] ; 从栈中读取PC的值

LDRB R0, [R1,#-2] ; 从SVC指令中读取立即数放到R0

; 准备调用系统服务函数。这需要适当调整入栈的PC的值以及LR(EXC\_RETURN)，来进入OS内部

BX LR ; 借异常返回的形式，进入OS内部，最终调用系统服务函数

一旦获取了调用号，就可以用它来调用系统服务函数了。有理由相信，OS应该使用TBB/TBH查表跳转指令来加速定位正确的服务函数。然而，如果你是设计OS的人，必须检查这个参数的合法性，以免因数字超出跳转表的范围而跳飞。

因为不能在SVC服务例程中嵌套使用SVC，所以如果有需要，就要直接调用SVC函数，例如，使用BL指令。

## 11.6 SVC示范：用于输出函数

在前面的例子中，我们写了若干个函数用于输出。但是有的时候，可能有一些障碍，使得我们不能用BL指令。例如，需要调用的函数是在另外的目标文件中，这就会导致有的时候我们无法定位子程序的入口地址；另外，如果跳转的目的地太远，也有诸多不便；或者，当使用OS时，这些输出函数已经被OS包装成系统调用了。在这些场合下，我们就需要使用SVC来作为传送门，如下面示例代码所示：

LDR R0, =HELLO\_TXT

SVC 0 ; 请求显示字符串的系统服务。服务代号：0

MOV R0, #’A’

SVC 1 ; 请求显示单一字符的系统服务。服务代号：1

LDR R0, =0xC123456

SVC 2 ; 请求显示16进制数的系统服务。服务代号：2

MOV R0, #1234

SVC 3 ; Display decimal value in R0

在使用SVC之前，我们需要先建立SVC服务例程向量，作法与建立IRQ的一样，只是需要把异常号改为11。这一次，通过巧妙地使用Thumb-2指令，我们还可以进一步优化代码：

SetupExcpHandler

; 入口条件：R0 = 异常号

; 入口条件：R1 = 异常服务例程

PUSH {R0, R2, LR}

LDR R2, =NVIC\_VECTTBL

LDR R2, [R2] ; 读取向量表的地址

STR.W R1, [R2, R0, LSL #2] ; 表中[R2+R0<<2]的位置就是为该向量的

POP {R0, R2, PC} ; 快速返回

对于SVC服务例程，可以使用前面所述的方式提取服务代号。如果那些请求系统服务的程序还传递了其它参数（通过R0-R3），则需找出正确的堆栈，再从堆栈中，读取进入SVC时自动压入的R0-R3值。

一个具体而微的SVC服务例程如下所示：

svc\_handler

;开始读取参数

TST LR, #0x4 ; 测试EXC\_RETURN的比特2

ITE EQ ; 如果为0,

MRSEQ R0, MSP ; 则使用的是主堆栈，故把MSP的值取出

MRSNE R0, PSP ; 否则, 使用的是进程堆栈，故把MSP的值取出

LDR R0, [R1,#0] ; 从堆栈中读取R0的值

LDR R1, [R1,#24] ; 从堆栈中读取当时的PC

LDRB R1, [R1,#-2] ; 提取SVC指令中的8位立即数

; 现在：R0存储了参数，R1存储了服务代号

PUSH {LR} ; 保护LR的值，因为后面将使用的BL指令

CBNZ R1, svc\_handler\_1

BL Puts ; 调用Puts

B svc\_handler\_end

svc\_handler\_1

CMP R1, #1

BNE svc\_handler\_2

BL Putc ; 调用Putc

B svc\_handler\_end

svc\_handler\_2

CMP R1, #2

BNE svc\_handler\_3

BL PutHex ; 调用PutHex

B svc\_handler\_end

svc\_handler\_3

CMP R1, #3

BNE svc\_handler\_4

BL PutDec ; 调用PutDec

B svc\_handler\_end

svc\_handler\_4

B error ; 未能识别的服务代号

...

svc\_handler\_end

POP {PC} ; Return

译者添加：事实上，根据具体的系统，不必总是教条主义，以化简参数提取的工作。比如，如果系统调用的参数不超过3个，就可以把系统调用号存储到R0中，把参数放到R1-R3中，而省去提取服务号的操作。

上例中，需要把svc\_handler的代码与那些输出函数的放在一起，以确保它们能在跳转的范围内。

细心的读者可能会问：为什么不直接从R0-R3中读取参数，却绕个大圈子从堆栈中读取取呢，它们不是一样的么？原来，这与晚到中断机制有关。仔细地想一想，如果在入栈期间，不巧来了另外的高优先级异常，则会使后者的服务例程先执行。待返回后，再以咬尾中断的方式执行SVC服务例程。我们知道，咬尾处理时，取消了前一个服务例程返回时的自动出栈动作。从而，在发生了晚到+咬尾的情况后，再执行SVC服务例程时，R0-R3已经被高优先级的服务例程用过了，它们的值十有八九被改过的。因此，必须从堆栈中读取。为帮助理解，在这里举一个具体的例子：

1. 用户程序把参数放到R0中，并执行SVC指令，请求系统服务
2. CM3为SVC开启了异常的响应序列，开始自动入栈，即把xPSR, PC, LR, R12, R3-R0压入堆栈
3. 入栈期间，来了一个高优先级的中断
4. 入栈完毕后，按晚到中断处理，先执行高优先级中断的服务例程。返回后，再以咬尾中断处理，此时，没有自动出栈的动作。
5. SVC服务例程以咬尾的方式开始执行。可见，此时的R0已经被高优先级服务例程用过了，不再保证是用户程序放入的参数。然而，先前入栈的R0-R3却依然保持不变（除非高优先级服务例程暗中使坏，篡改了堆栈的内容）

编程技巧：善用LDR/STR中的多种寻址方式

对比SetupIrqhandler和SetupExcpHandler的代码，我们可以看到，在SetupIrqHander中，目标地址是用3条计算出来的，然后才使用存储指令。

而SetupExcpHandler就聪明多了，它通过对偏移寄存器做移位预处理，把计算地址巧妙地合并在存储指令的内部，使得本来3条指令做的事1条指令就搞定了。

这个小小的例子还给了我们另外的启示：学习时要求甚解，熟能生巧。CM3中有很多新指令，它们单独使用或者组合使用，能让温柔小女生的力气大增，蜕变成爱情女神。比如，CLZ与RBIT的组合使用，就快速地求得了芯片中表达优先级的位数。此外，它们还对“优先级位图调度算法”有决定性的化简意义（both时间上的和空间上的），有兴趣的读者可以拿它们去化简uC/OS-II中的调度函数，看看能不能去掉那个256字节的查找表。

## 11.7 在C中使用SVC

如前所述，因为晚到中断的关系，SVC中不能再使用寄存器来传递参数，而是必须使用堆栈。因此，需要使用一段汇编代码来给SVC函数传参数。如果SVC服务例程的主部由C来写，则必须在前面伴随一个汇编写的封皮，用于把堆栈中的参数提取到寄存器中。下面给出一段代码来演示这个工作。这些代码是要使用ARM娘家的编译(armcc)和汇编(armasm)工具来处理的，RVDS和Keil RVMDK都使用这个工具链。

// 汇编封皮，用于提出堆栈帧的起始位置，并放到R0中，然后跳转至实际的SVC服务例程中

\_\_asm void svc\_handler\_wrapper(void)

{

IMPORT svc\_handler

TST LR, #4

ITE EQ

MRSEQ R0, MSP

MRSNE R0, PSP

B svc\_handler

}

// 不必写下BX LR来返回，而是由svc\_handler来做决定

接下来的SVC服务例程的主体就可以由C来写了，它使用R0作为输入参数（这也是堆栈帧的起始位置），用于进一步提取服务代号，并且传递参数（通过堆栈中的R0-R3）。

恭喜呀！终于看到第一段C代码了（而且还是一段很另类的C程序哦）！

// 使用C写成的SVC服务例程，接受一个指针参数（pwdSF）：堆栈栈的起始地址。

// pwdSF[0] = R0 , pwdSF[1] = R1

// pwdSF[2] = R2 , pwdSF[3] = R3

// pwdSF[4] = R12, pwdSF[5] = LR

// pwdSF[6] = 返回地址（入栈的PC）

// pwdSF[7] = xPSR

**unsigned long** svc\_handler(unsigned int\* pwdSF)

{

unsigned int svc\_number;

unsigned int svc\_r0;

unsigned int svc\_r1;

unsigned int svc\_r2;

unsigned int svc\_r3;

int retVal; //用于存储返回值

svc\_number = ((char \*) pwdSF[6])[-2]; // 没想到吧，C的数组能用得这么绝！

svc\_r0 = ((unsigned long) pwdSF[0]);

svc\_r1 = ((unsigned long) pwdSF[1]);

svc\_r2 = ((unsigned long) pwdSF[2]);

svc\_r3 = ((unsigned long) pwdSF[3]);

printf (“SVC number = %xn”, svc\_number);

printf (“SVC parameter 0 = %x\n”, svc\_r0);

printf (“SVC parameter 1 = %x\n”, svc\_r1);

printf (“SVC parameter 2 = %x\n”, svc\_r2);

printf (“SVC parameter 3 = %x\n”, svc\_r3);

//做一些工作，并且把返回值存储到retVal中

**pwdSF[0]=retVal;**

**return 0;**

}

注意，这个函数返回的其实不是0！进一步地，灰色的文字只是用于哄编译器开心的——让它认为这个函数是个有返回值的函数，而且确实返回一个数值了，于是不再吵闹着说有错或警告什么的。那返回的是啥？当然是retVal啦！有点迷糊么？那还不快往下看！

原来，SVC服务例程不能像普通的C函数那样——通过把原型声明为”unsigned int func()”,再在末尾来一句”return xx;”来返回。因为这种常规的作法在所有的ARM中其实是把返回值放到R0里。但是别忘了，这个函数可是异常服务例程，它的返回可是享受“异常返回”的待遇的——伴随着一个硬件控制的自动出栈行为，这会从堆栈中重建R0的值，从而覆盖“return”指定的值。因此，它必须把返回值写到堆栈中R0的位置，才能借自动出栈之机返回自己的值（pwdSF[0]=retVal）。

这下可真相大白了！虽然内部暗流汹涌，但是从应用程序的表面上看还是风平浪静——对于系统服务函数来说，这种独特的返回方式与普通的return xx效果是相同的，依然可以用普通的形式接收返回值。怎么样，这招够狠吧！其实，在写系统软件时，这根本算不上耍狠，只不过是寻常的基本功罢了，要不然怎么说C是“低级高级语言”呢。而病毒/木马所采用的“堆栈/缓冲区溢出攻击”，那才算真正的狠招呢，但是它们原理是一脉相承的。可见，对底层理解得深刻，能让我们写出更好，更强大的程序来。

在RVDS和Keil RVMDK中，为了方便我们放参数，提供了“\_\_svc”编译器指示字。举例来说，如果需要在3号服务请求中传递4个参数，则可以类似下例写：

*unsigned long* **\_\_svc(0x03)** CallSvc3(unsigned long svc\_r0, unsigned long

svc\_r1, unsigned long svc\_r2, unsigned long svc\_r3);

当C程序调用这种函数时，则编译器会自动生成SVC指令，如下所示：

int Func(void)

{

unsigned long p0, p1, p2, p3; //传递给SVC服务例程的4个函数

unsigned long svcRet; //系统服务的返回值

. . .

svcRet=CallSvc3(p0, p1, p2, p3); // 呼叫3号系统服务，并且传递4个参数，依次为：p1,p2,p3,p4，再接收返回值到svcRet中（别忘了，这个返回值的来历不寻常）

. . .

return;

}

如欲获知\_\_svc的官方说明，可以查阅《RVCT 3.0 Compiler and Library Guide(Ref6)》。

如果使用的是GNU的工具链，里面没有\_\_svc关键字。但是GCC支持内联汇编，可以实现此功能。例如，如果需要呼叫3号系统服务，同时传递一个参数，还接收一个返回值（两者都通过R0），则可以使用如下的内联汇编来呼叫SVC：

int MyDataIn ＝ 0x123;

\_\_asm \_\_volatile (”mov R0, %0\n”

”svc 3 \n” : ”” : ”r” (MyDataIn) );

上段内联汇编码中，两个“：”后面分别对应输入数据——由r(MyDataIn)指定，以及输出数据——即上段代码中是””，语法模式如下所示：

\_\_asm ( assembler\_code : output\_list : input\_list )

在第19章中，给出了使用GNU工具链的更多汇编例子。如欲获取有关内联汇编的详细信息，还请参阅GNU工具链的说明文档。