第3章

# Cortex-M3基础

* 寄存器组
* 特殊功能寄存器组
* 操作模式
* 异常和中断
* 向量表
* 存储器保护单元
* 堆栈区的操作
* 复位序列

## 3.1 寄存器组

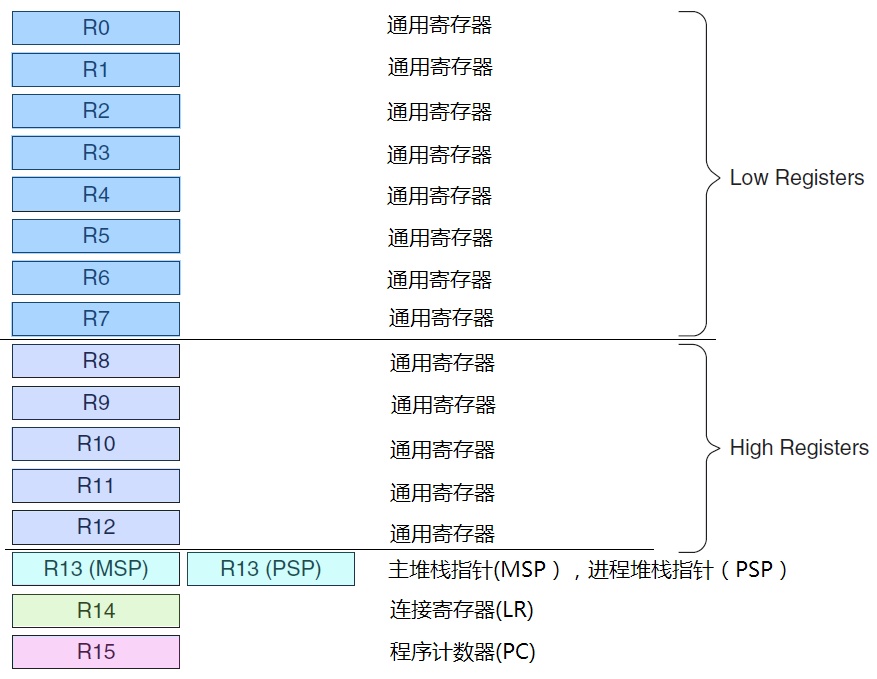
如我们所见，CM3拥有通用寄存器R0-R15以及一些特殊功能寄存器。R0-R12是最“通用目的”的，但是绝大多数的16位指令只能使用R0-R7（低组寄存器），而32位的Thumb-2指令则可以访问所有通用寄存器。特殊功能寄存器有预定义的功能，而且必须通过专用的指令来访问。

### 3.1.1 通用目的寄存器R0-R7

R0-R7也被称为低组寄存器。所有指令都能访问它们。它们的字长全是32位，复位后的初始值是不可预料的。

### 3.1.2 通用目的寄存器R8-R12

R8-R12也被称为高组寄存器。这是因为只有很少的16位Thumb指令能访问它们，32位的thumb-2指令则不受限制。它们也是32位字长，且复位后的初始值是不可预料的。



### 3.1.3 特殊功能寄存器：

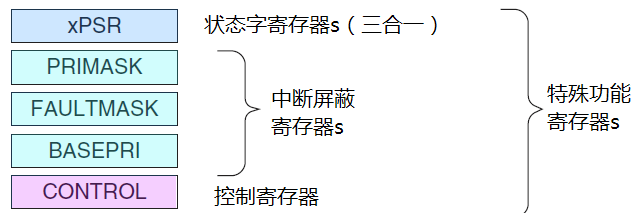


图3.1 Cortex-M3的寄存器组

### 3.1.4 堆栈指针R13

R13是堆栈指针。在CM3处理器内核中共有两个堆栈指针，于是也就支持两个堆栈。当引用R13（或写作SP）时，引用到的是当前正在使用的那一个，另一个必须用特殊的指令来访问（MRS,MSR指令）。这两个堆栈指针分别是：

* 主堆栈指针（MSP），或写作SP\_main。这是缺省的堆栈指针，它由OS内核、异常服务例程以及所有需要特权访问的应用程序代码来使用。
* 进程堆栈指针（PSP），或写作SP\_process。用于常规的应用程序代码（不处于异常服用例程中时）。

译注：在本章中，如无特殊说明，“异常”与“中断”都是指当发生“事件”时，处理器改变正常执行流，去响应该事件的情况。只不过异常之于处理器是同步的，中断之于处理器是异步的。因此常混合使用二术语，ISR和ESR也混合使用，请读者不必工于辨析这两个术语的不同，在这里这不是重点。

要注意的是，并不是每个程序都要用齐两个堆栈指针才算圆满。简单的应用程序只使用MSP就够了。堆栈指针用于访问堆栈，并且PUSH指令和POP指令默认使用SP。

堆栈的PUSH与POP

堆栈是一种存储器的使用模型。它由一块连续的内存和一个栈顶指针组成，用于实现“后进先出”的缓冲区。其最典型的应用，就是在数据处理前先保存寄存器的值，再在处理任务完成后从中恢复先前保护的这些值。

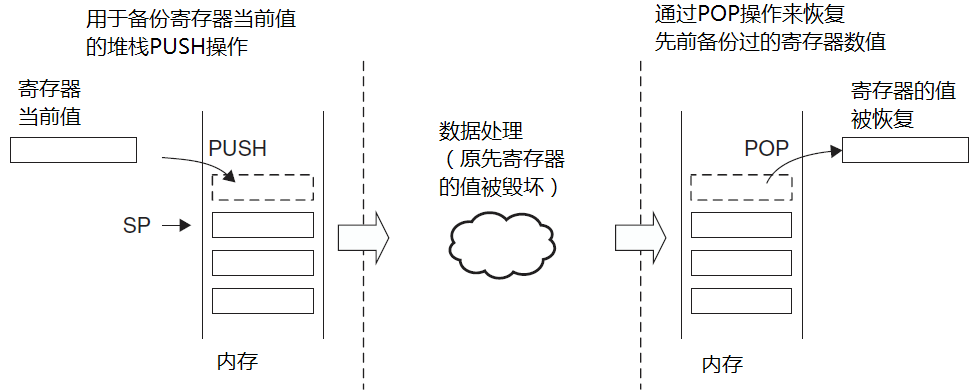


图1.2 堆栈内存的基本概念

在执行PUSH和POP操作时，那个通常被称为SP的地址寄存器，会由硬件自动调整它的值，以避免后续操作破坏先前的数据。本书的后续章节还要围绕着堆栈展开更详细的论述。

在Cortex-M3中，有专门的指令负责堆栈操作——PUSH和POP。它俩的汇编语言语法如下例所演示

PUSH {R0} ; \*(--R13)=R0。R13是long\*的指针

POP {R0} ; R0= \*R13++

请注意后面C程序风格的注释，它诠释了所谓的“向下生长的满栈”（本章后面在讲到堆栈内存操作时还要展开论述），Cortex-M3就是以这种方式使用堆栈的。因此，在PUSH新数据时，堆栈指针先减一个单元。通常在进入一个子程序后，第一件事就是把寄存器的值先PUSH入堆栈中，在子程序退出前再POP曾经PUSH的那些寄存器。另外，PUSH和POP还能一次操作多个寄存器，如下所示：

subroutine\_1

PUSH {R0-R7, R12, R14} ; 保存寄存器列表

… ; 执行处理

POP {R0-R7, R12, R14} ; 恢复寄存器列表

BX R14 ; 返回到主调函数

在程序中为了突出重点，可以一直把R13写作SP。在程序代码中，both MSP和PSP都被称为R13/SP。不过，我们可以通过MRS/MSR指令来指名道姓地访问具体的堆栈指针。

MSP，亦写作SP\_main，这是复位后缺省使用堆栈指针，服务于操作系统内核和异常服务例程；而PSP，亦写作SP\_process，典型地用于普通的用户线程中。

寄存器的PUSH和POP操作永远都是4字节对齐的——也就是说他们的地址必须是0x4,0x8,0xc,……。事实上，R13的最低两位被硬线连接到0,并且总是读出0（Read As Zero）。

### 3.1.5 连接寄存器R14

R14是连接寄存器（LR）。在一个汇编程序中，你可以把它写作both LR和R14。LR用于在调用子程序时存储返回地址。例如，当你在使用BL(分支并连接，Branch and Link)指令时，就自动填充LR的值。

main ;主程序

…

BL function1 ; 使用“分支并连接”指令呼叫function1

; PC= function1，并且LR=main的下一条指令地址

…

Function1

… ; function1的代码

BX LR ; 函数返回（如果function1要使用LR，必须在使用前PUSH，

; 否则返回时程序就可能跑飞了——译注）

尽管PC的LSB总是0（因为代码至少是半字对齐的），LR的LSB却是可读可写的。这是历史遗留的产物。在以前，由位0来指示ARM/Thumb状态。因为其它有些ARM处理器支持ARM和Thumb状态并存，为了方便汇编程序移植，CM3需要允许LSB可读可写。

### 3.1.6 程序计数器R15

R15是程序计数器，在汇编代码中一般我们都都叫它的外号“PC”。因为CM3内部使用了指令流水线，读PC时返回的值是当前指令的地址+4。比如说：

0x1000: MOV R0, PC ; R0 = 0x1004

如果向PC中写数据，就会引起一次程序的分支（但是不更新LR寄存器**）**。CM3中的指令至少是半字对齐的，所以PC的LSB总是读回0。然而，在分支时，无论是直接写PC的值还是使用分支指令，都必须保证加载到PC的数值是奇数（即LSB=1），用以表明这是在Thumb状态下执行。倘若写了0，则视为企图转入ARM模式，CM3将产生一个fault异常。

## 3.2 特殊功能寄存器组

Cortex-M3中的特殊功能寄存器包括：

* 程序状态寄存器组（PSRs或曰xPSR）
* 中断屏蔽寄存器组（PRIMASK, FAULTMASK,以及BASEPRI）
* 控制寄存器（CONTROL）

它们只能被专用的MSR/MRS指令访问，而且它们也没有与之相关联的访问地址。

MRS <gp\_reg>, <special\_reg> ;读特殊功能寄存器的值到通用寄存器

MSR <special\_reg>, <gp\_reg> ;写通用寄存器的值到特殊功能寄存器

### 3.2.1 程序状态寄存器（PSRs或曰PSR）

程序状态寄存器在其内部又被分为三个子状态寄存器：

* 应用程序PSR（APSR）
* 中断号PSR（IPSR）
* 执行PSR（EPSR）

通过MRS/MSR指令，这3个PSRs即可以单独访问，也可以组合访问（2个组合，3个组合都可以）。当使用三合一的方式访问时，应使用名字“xPSR”或者“PSR”。

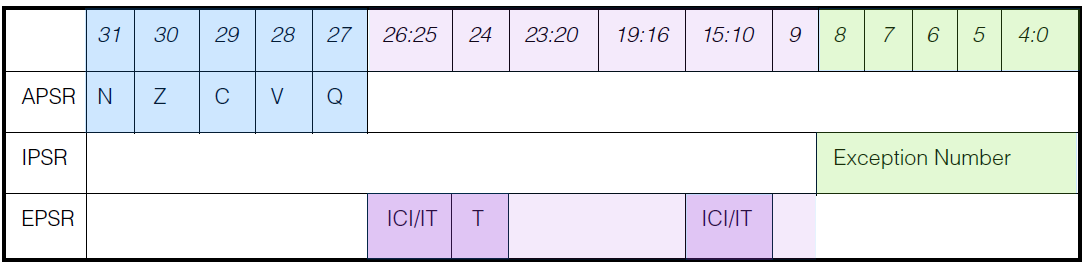


图3.3 Cortex-M3中的程序状态寄存器（xPSR）

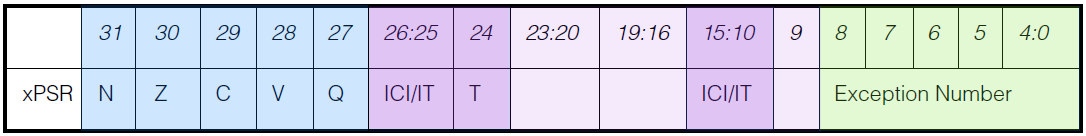


图3.4 合体后的程序状态寄存器(xPSR)

### 3.2.2 PRIMASK, FAULTMASK和BASEPRI

这三个寄存器用于控制异常的使能和除能。

表3.2 Cortex-M3的屏蔽寄存器组

|  |  |
| --- | --- |
| 名字 | 功能描述 |
| PRIMASK | 这是个只有单一比特的寄存器。在它被置1后，就关掉所有可屏蔽的异常，只剩下NMI和硬fault可以响应。它的缺省值是0，表示没有关中断。 |
| FAULTMASK | 这是个只有1个位的寄存器。当它置1时，只有NMI才能响应，所有其它的异常，甚至是硬fault，也通通闭嘴。它的缺省值也是0，表示没有关异常。 |
| BASEPRI | 这个寄存器最多有9位（由表达优先级的位数决定）。它定义了被屏蔽优先级的阈值。当它被设成某个值后，所有优先级号大于等于此值的中断都被关（优先级号越大，优先级越低）。但若被设成0，则不关闭任何中断，0也是缺省值。 |

对于时间-关键任务而言，恰如其分地使用PRIMASK和BASEPRI来暂时关闭一些中断是非常重要的。而FAULTMASK则可以被OS用于暂时关闭fault处理机能，这种处理在某个任务崩溃时可能需要。因为在任务崩溃时，常常伴随着一大堆faults。在系统料理“后事”时，通常不再需要响应这些fault——人死帐清。总之FAULTMASK就是专门留给OS用的。

要访问PRIMASK, FAULTMASK以及BASEPRI，同样要使用MRS/MSR指令,如：

MRS R0, BASEPRI ;读取BASEPRI到R0中

MRS R0, FAULTMASK ;似上

MRS R0, PRIMASK ;似上

MSR BASEPRI, R0 ;写入R0到BASEPRI中

MSR FAULTMASK, R0 ;似上

MSR PRIMASK, R0 ;似上

只有在特权级下，才允许访问这3个寄存器。

译者添加：

其实，为了快速地开关中断，CM3还专门设置了一条CPS指令，有4种用法

CPSID I ;PRIMASK=1， ;关中断

CPSIE I ;PRIMASK=0， ;开中断

CPSID F ;FAULTMASK=1, ;关异常

CPSIE F ;FAULTMASK=0 ;开异常

### 3.2.3 控制寄存器（CONTROL）

控制寄存器有两个用途，其一用于定义特权级别，其二用于选择当前使用哪个堆栈指针。由两个比特来行使这两个职能。

表3.3 Cortex-M3的CONTROL寄存器

|  |  |
| --- | --- |
| 位 | 功能 |
| CONTROL[1] | 堆栈指针选择  0=选择主堆栈指针MSP（复位后的缺省值）  1=选择进程堆栈指针PSP  在线程或基础级（没有在响应异常——译注），可以使用PSP。在handler模式下，只允许使用MSP，所以此时不得往该位写1。 |
| CONTROL[0] | 0=特权级的线程模式  1=用户级的线程模式  **Handler模式永远都是特权级的**。 |

#### CONTROL[1]

在Cortex-M3的handler模式中，CONTROL[1]总是0。在线程模式中则可以为0或1。

因此，仅当处于特权级的线程模式下，此位才可写，其它场合下禁止写此位。改变处理器的模式也有其它的方式：在异常返回时，通过修改LR的位2，也能实现模式切换。这是LR在异常返回时的特殊用法，颠覆了对LR的传统使用方式，将在第5章中展开论述。

#### CONTROL[0]

仅当在特权级下操作时才允许写该位。一旦进入了用户级，唯一返回特权级的途径，就是触发一个（软）中断，再由服务例程改写该位。

CONTROL寄存器也是通过MRS和MSR指令来操作的：

MRS R0, CONTROL

MSR CONTROL, R0

## 3.3 操作模式

Cortex-M3支持2个模式和两个特权等级。

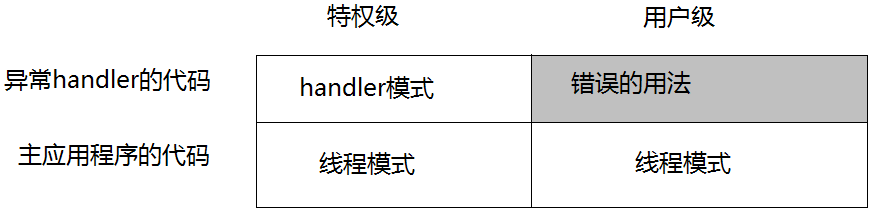


图3.6 操作模式和特权等级

当处理器处在线程状态下时，既可以使用特权级，也可以使用用户级；另一方面，handler模式总是特权级的。在复位后，处理器进入线程模式＋特权级。

在线程模式＋用户级下，对系统控制空间（SCS）的访问将被阻止——该空间包含了配置寄存器组以及调试组件的寄存器组。除此之外，还禁止使用MRS/MSR访问刚才讲到的，除了APSR之外的特殊功能寄存器。如果以身试法，则对于访问特殊功能寄存器的，访问操作被忽略；而对于访问SCS空间的，将fault伺候。

译注：原文的意思是越权访问一律产生fault( If a program running at the user access level tries to access SCS or special registers, a fault exception will occur)。但译者使用Keil MDK开发环境的模拟器和STM32单片机作实验时却发现，对特殊功能寄存器越权访问时，仅忽略访问操作，并不产生fault。另外，译者发现，当使用模拟器时，即使访问了SCS中的地址（译者使用的地址是0xE000E100），模拟器竟然也允许读写！后来译者又使用STM32单片机来实验，STM32单片机则的确产生了总线fault并上访成了硬fault。因此，如果使用指令模拟器，则要小心。附：译者使用的MDK版本号是3.20

在特权级下的代码可以通过置位CONTROL[0]来进入用户级。而不管是任何原因产生了任何异常，处理器都将以特权级来运行其服务例程，异常返回后，系统将回到产生异常时所处的级别。用户级下的代码不能再试图修改CONTROL[0]来回到特权级。它必须通过一个异常handler，由那个异常handler来修改CONTROL[0]，才能在返回到线程模式后拿到特权级。

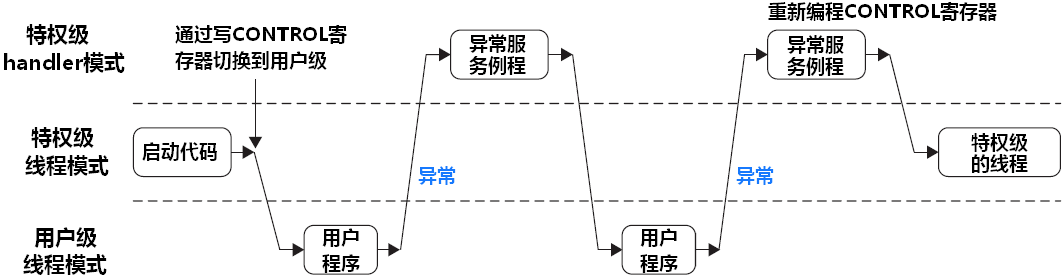


图3.7 特权级和处理器模式转换图

把代码按特权级和用户极分开对待，有利于使CM3的架构更加安全和健壮。例如，当某个用户程序代码出问题时，不会让它成为害群之马，因为用户级的代码是禁止写特殊功能寄存器和NVIC中寄存器的。另外，如果还配有MPU，保护力度就更大，甚至可以阻止用户代码访问不属于它的内存区域。

为了避免系统堆栈因应用程序的错误使用而毁坏，我们可以给应用程序专门配一个堆栈，不让它共享操作系统内核的堆栈。在这个管理制度下，运行在线程模式的用户代码使用PSP，而异常服务例程则使用MSP。这两个堆栈指针的切换是智能全自动的，就在异常服务的始末由CM3硬件处理。第8章将详细讨论此主题。

如前所述，特权等级和堆栈指针的选择均由CONTROL负责。当CONTROL[0]=0时，在异常处理的始末，只发生了处理器模式的转换，如下图所示。

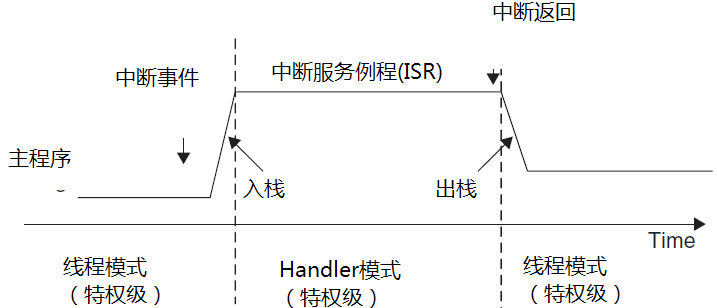


图3.8 中断前后的状态转换

但若CONTROL[0]=1（线程模式+用户级），则在中断响应的始末，both 处理器模式和特权等极都要发生变化，如下图所示。

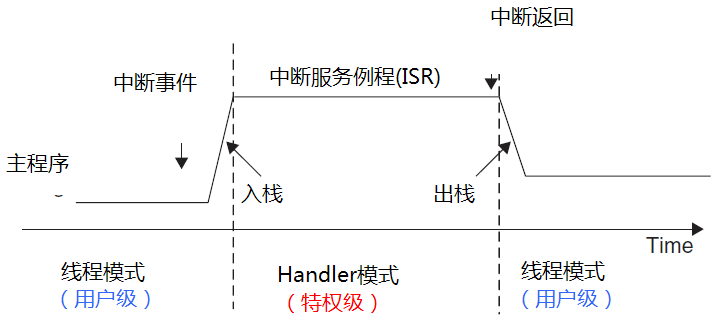


图3.9 中断前后的状态转换＋特权等级切换

CONTROL［0］只有在特权级下才能访问。用户级的程序如想进入特权级，通常都是使用一条“系统服务呼叫指令（SVC）”来触发“SVC异常”，该异常的服务例程可以视具体情况而修改CONTROL[0]。

## 3.4 异常与中断

Cortex-M3支持大量异常，包括16-4-1=11个系统异常，和最多240个外部中断——简称IRQ。具体使用了这240个中断源中的多少个，则由芯片制造商决定。由外设产生的中断信号，除了SysTick的之外，全都连接到NVIC的中断输入信号线。典型情况下，处理器一般支持16到32个中断，当然也有在此之外的。

作为中断功能的强化，NVIC还有一条NMI输入信号线。NMI究竟被拿去做什么，还要视处理器的设计而定。在多数情况下，NMI会被连接到一个看门狗定时器，有时也会是电压监视功能块，以便在电压掉至危险级别后警告处理器。NMI可以在任何时间被激活，甚至是在处理器刚刚复位之后。

表3.4列出了Cortex-M3可以支持的所有异常。有一定数量的系统异常是用于fault处理的，它们可以由多种错误条件引发。NVIC还提供了一些fault状态寄存器，以便于fault服务例程找出导致异常的具体原因。

表3.4 Cortex-M3中的异常类型

|  |  |  |  |
| --- | --- | --- | --- |
| 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”成硬fault。除能的原因包括当前被禁用，或者被PRIMASK或BASPRI掩蔽。 |
| 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） |
| 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 |

第7-9章给出了异常操作的详细信息。

## 3.5 向量表

当CM3内核响应了一个发生的异常后，对应的异常服务例程(ESR)就会执行。为了决定ESR的入口地址，CM3使用了“向量表查表机制”。这里使用一张向量表。向量表其实是一个WORD（32位整数）数组，每个下标对应一种异常，该下标元素的值则是该ESR的入口地址。向量表在地址空间中的位置是可以设置的，通过NVIC中的一个重定位寄存器来指出向量表的地址。在复位后，该寄存器的值为0。因此，在地址0处必须包含一张向量表，用于初始时的异常分配。

表3.5 向量表结构

|  |  |  |
| --- | --- | --- |
| 异常类型 | 表项地址偏移量 | 异常向量 |
| 0 | 0x00 | MSP的初始值 |
| 1 | 0x04 | 复位 |
| 2 | 0x08 | NMI |
| 3 | 0x0C | 硬fault |
| 4 | 0x10 | MemManage fault |
| 5 | 0x14 | 总线fault |
| 6 | 0x18 | 用法fault |
| 7-10 | 0x1c-0x28 | 保留 |
| 11 | 0x2c | SVC |
| 12 | 0x30 | 调试监视器 |
| 13 | 0x34 | 保留 |
| 14 | 0x38 | PendSV |
| 15 | 0x3c | SysTick |
| 16 | 0x40 | IRQ #0 |
| 17 | 0x44 | IRQ #1 |
| 18-255 | 0x48-0x3FF | IRQ #2 - #239 |

举个例子，如果发生了异常11（SVC），则NVIC会计算出偏移移量是11x4=0x2C，然后从那里取出服务例程的入口地址并跳入。要注意的是这里有个另类：0号类型并不是什么入口地址，而是给出了复位后MSP的初值。

## 3.6 栈内存操作

在Cortex-M3中，除了可以使用PUSH和POP指令来处理堆栈外，内核还会在异常处理的始末自动地执行PUSH与POP操作。本节让我们来检视一下具体的动作，第9章则讨论异常处理时的自动栈操作。

### 3.6.1 堆栈的基本操作

笼统地讲，堆栈操作就是对内存的读写操作，但是访问地址由SP给出。寄存器的数据通过PUSH操作存入堆栈，以后用POP操作从堆栈中取回。在PUSH与POP的操作中，SP的值会按堆栈的使用法则自动调整，以保证后续的PUSH不会破坏先前 PUSH进去的内容。

堆栈的功能就是把寄存器的数据临时备份在内存中，以便将来能恢复之——在一个任务或一段子程序执行完毕后恢复。正常情况下，PUSH与POP必须成对使用，而且参与的寄存器，不论是身份还是先后顺序都必须完全一致。当PUSH/POP指令执行时，SP指针的值也根着自减/自增。

…（主程序）

; R0=X, R1=Y, R2=Z

BL Fx1

Fx1

PUSH {R0 } ;把R0存入栈 & 调整SP

PUSH {R1} ;把R1存入栈 & 调整SP

PUSH {R2} ;把R2存入栈 & 调整SP

… ;执行Fx1的功能，中途可以改变R0-R2的值

POP {R2} ;恢复R2早先的值 & 再次调整SP

POP {R1} ;恢复R1早先的值 & 再次调整SP

POP {R0} ;恢复R0早先的值 & 再次调整SP

BX LR ;返回

;回到主程序

;R0=X, R1=Y, R2=Z （调用Fx1的前后R0-R2的值完好无损，从寄存器上下文来看，就好像什么都没发生过一样）

图3.10 基本的堆栈操作：每次处理单个寄存器

堆栈操作的进一步探讨

如果参与的寄存器比较多，这种PUSH和POP岂不是又臭又长？放心，PUSH/POP指令足够体贴，支持一次操作多个寄存器。像这样：

PUSH {R0-R2} ;压入R0-R2

PUSH {R3-R5,R8, R12} ;压入R3-R5,R8，以及R12

在POP时，可以如下操作：

POP {R3-R5,R8, R12} ;弹出R3-R5，R8，以及R12

POP {R0-R2} ;弹出R0-R2

注意：在寄存器列表中，不管寄存器的序号是以什么顺序给出的，汇编器都将把它们升序排序。然后先push序号大的寄存器，所以也就先pop序号小的寄存器。（这是译者在实验中发现的）。如果不按升序写寄存器，也许有些汇编器会给出语法错误。

PUSH/POP对子还有这样一种特殊形式，形如

PUSH {R0-R3, LR}

POP {R0-R3, **PC**}

请注意：POP的最后一个寄存器是PC，并不是先前PUSH的LR。这其实是一个返回的小技巧。与其按部就班地把先前LR的值弹回LR，再复制给PC来返回；不如干脆绕过LR，直接传给PC！那不怕LR的值没有被恢复吗？不怕，因为LR在子程序调用中的唯一用处，就是在返回时提供返回地址。因此，在返回后，先前保存的返回地址就没有利用价值了，所以只要PC得到了正确的值，不恢复也没关系。

PUSH指令等效于与使用R13作为地址指针的STMDB指令，而POP指令则等效于使用R13作为地址指针的LDMIA指令——STMDB/LDMIA还可以使用其它寄存器作为地址指针。至于这两个指令的细节，第4章讲到指令系统时再介绍。

## 3.7 Cortex-M3的堆栈实现

Cortex-M3使用的是“向下生长的满栈”模型。堆栈指针SP指向最后一个被压入堆栈的32位数值。在下一次压栈时，SP先自减4，再存入新的数值。

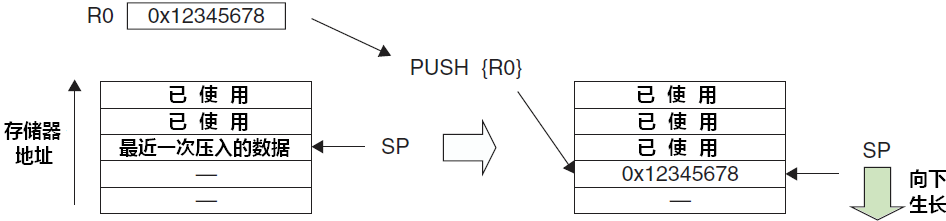


图3.13 Cortex-M3堆栈的PUSH实现方式

POP操作刚好相反：先从SP指针处读出上一次被压入的值，再把SP指针自增4。

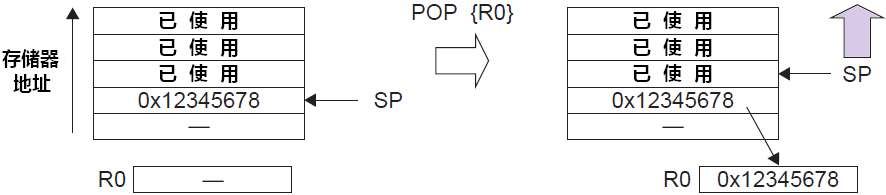


图3.14 Cortex-M3堆栈的POP实现方式

译注：虽然POP后被压入的数值还保存在栈中，但它已经无效了，因为为下次的PUSH将覆盖它的值！

在进入ESR时，CM3会自动把一些寄存器压栈，这里使用的是发生本异常的瞬间正在使用的SP指针（MSP或者是PSP）。离开ESR后，只要ESR没有更改过CONTROL[1]，就依然使用发生本次异常的瞬间正在使用的SP指针来执行出栈操作。

### 3.7.1 再论Cortex-M3的双堆栈机制

我们已经知道了CM3的堆栈是分为两个：主堆栈和进程堆栈，CONTROL[1]决定如何选择。

当CONTROL[1]=0时，只使用MSP，此时用户程序和异常handler共享同一个堆栈。这也是复位后的缺省使用方式。

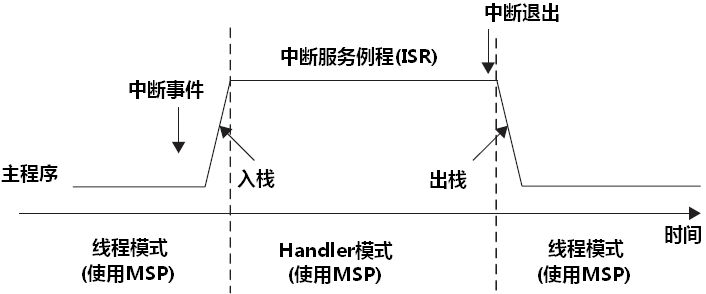


图3.15 CONTROL[1]=0时的堆栈使用情况

当CONTROL[1]=1时，线程模式将不再使用MSP，而改用PSP（handler模式永远使用MSP）。这样做的好处在哪里？原来，在使用OS的环境下，只要OS内核仅在handler模式下执行，用户应用程序仅在用户模式下执行，这种双堆栈机制派上了用场——防止用户程序的堆栈错误破坏OS使用的堆栈。

译注：此时，进入异常时的自动压栈使用的是进程堆栈，进入异常handler后才自动改为MSP，退出异常时切换回PSP，并且从进程堆栈上弹出数据。

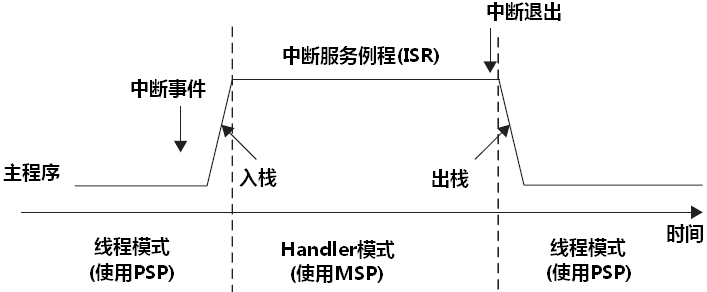


图3.16 CONTROL[1]=1时的堆栈切换情况

在特权级下，可以指定具体的堆栈指针，而不受当前使用堆栈的限制，示例代码如下：

MRS R0, MSP ; 读取主堆栈指针到R0

MSR MSP, R0 ; 写R0的值到主堆栈中

MRS R0, PSP ; 读取进程堆栈指针到R0

MSR PSP, R0 ; 写R0的值到进程堆栈中

通过读取PSP的值，OS就能够获取用户应用程序使用的堆栈，进一步地就知道了在发生异常时，被压入寄存器的内容，而且还可以把其它寄存器进一步压栈（使用STMDB和LDMIA的书写形式）。OS还可以修改PSP，用于实现多任务中的任务上下文切换。

## 3.8 复位序列

在离开复位状态后，CM3做的第一件事就是读取下列两个32位整数的值：

* + 从地址0x0000,0000处取出MSP的初始值。
  + 从地址0x0000,0004处取出PC的初始值——这个值是复位向量，LSB必须是1。然后从这个值所对应的地址处取指。

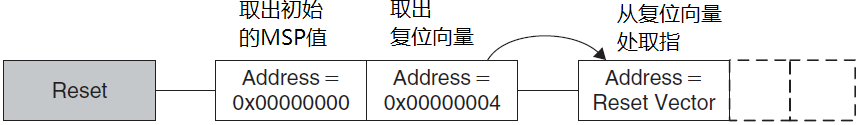


图3.17 复位序列

请注意，这与传统的ARM架构不同——其实也和绝大多数的其它单片机不同。传统的ARM架构总是从0地址开始执行第一条指令。它们的0地址处总是一条跳转指令。**在CM3中，在0地址处提供MSP的初始值，然后紧跟着就是向量表**（向量表在以后还可以被移至其它位置——译注）。**向量表中的数值是32位的地址，而不是跳转指令**。向量表的第一个条目指向复位后应执行的第一条指令。

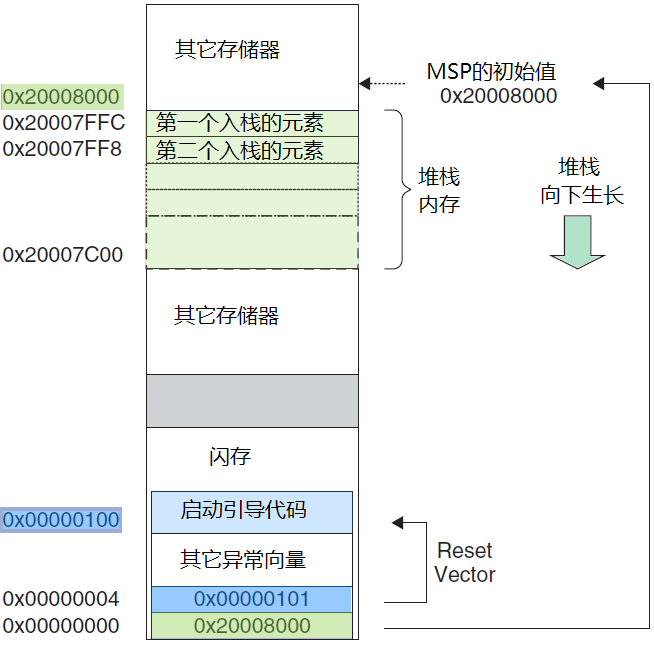


图3.18 初始MSP及PC初始化的一个范例

因为CM3使用的是向下生长的满栈，所以MSP的初始值必须是堆栈内存的末地址加1。举例来说，如果你的堆栈区域在0x20007C00-0x20007FFF之间，那么MSP的初始值就必须是0x20008000。

向量表跟随在MSP的初始值之后——也就是第2个表目。要注意因为CM3是在Thumb态下执行，所以向量表中的每个数值都必须把LSB置1（也就是奇数）。正是因为这个原因，图3.18中使用0x101来表达地址0x100。当0x100处的指令得到执行后，就正式开始了程序的执行。在此之前初始化MSP是必需的，因为可能第1条指令还没来得及执行，就发生了NMI或是其它fault。MSP初始化好后就已经为它们的服务例程准备好了堆栈。

对于不同的开发工具，需要使用不同的格式来设置MSP初值和复位向量——有些则由开发工具自行计算并生成。如果想要获知细节，最快的办法就是参考开发工具提供的一个示例工程。本书的第10章和第20章介绍ARM提供的开发工具，第19章则介绍GCC工具链。