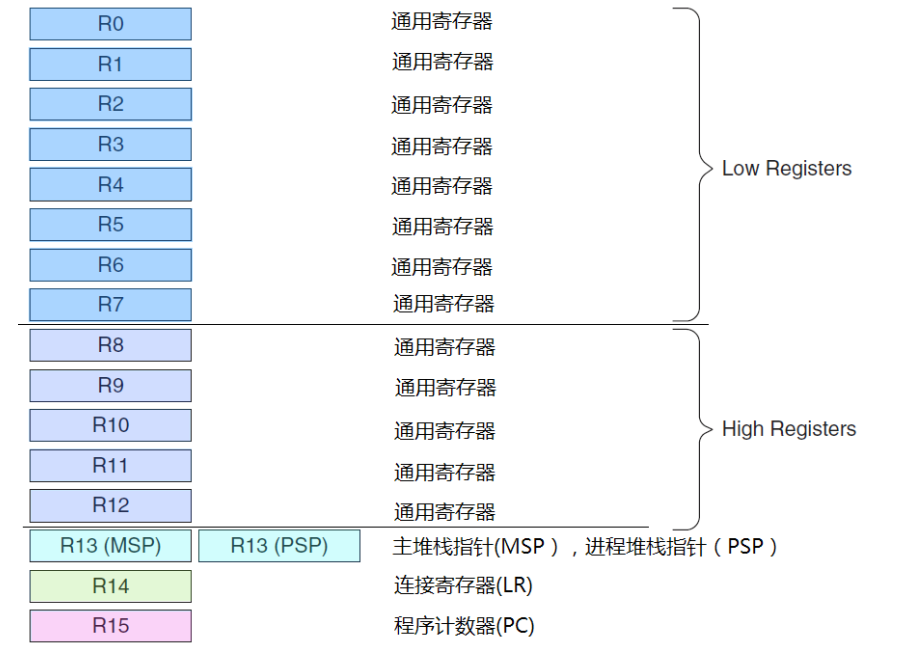
# CH3 CM3基础

## ARM分类：

ARM进一步扩展了它的CPU设计, 内核架构首次从单一款式变成3种款式:

(1) 款式A：设计用于高性能的“开放应用平台”——越来越接近电脑了  
(2) 款式R：用于高端的嵌入式系统，尤其是那些带有实时要求的——又要快又要实时。  
(3) 款式M：用于深度嵌入的，单片机风格的系统中——本书的主角

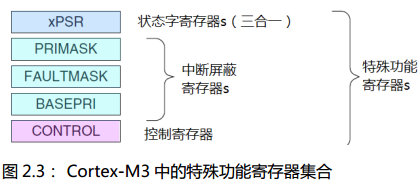


MSP(亦作SP\_main)，这是复位后缺省使用堆栈指针，服务于操作系统内核和异常服务例程；而 PSP(亦作SP\_process)，典型地用于普通的用户线程中。

LR 用于在调用子程序时存储返回地址。

## CM3特殊功能寄存器：

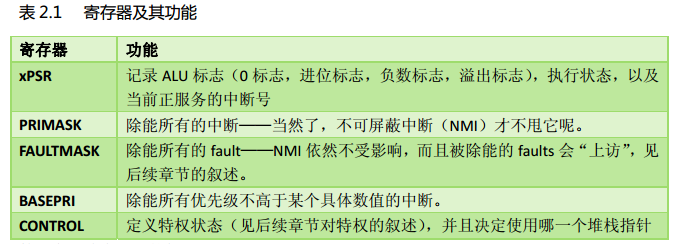
Cortex‐M3 还在内核水平上搭载了若干特殊功能寄存器，包括

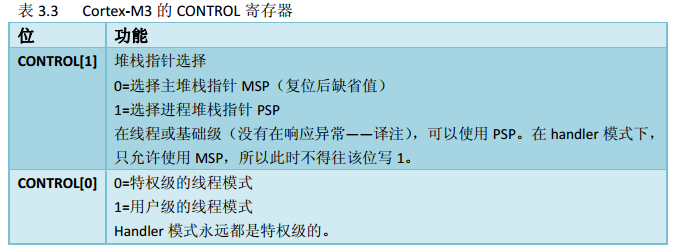


（1）程序状态字寄存器组（ PSRs）

（2）中断屏蔽寄存器组（ PRIMASK, FAULTMASK, BASEPRI，只能在特权级下访问这三个寄存器）

（3）控制寄存器（ CONTROL，只能在特权级下访问这个寄存器，还用于选择当前使用哪个堆栈指针）





※ CONTROL[1]：仅当处于特权级的线程模式下(复位后的状态，见图2.5)，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；

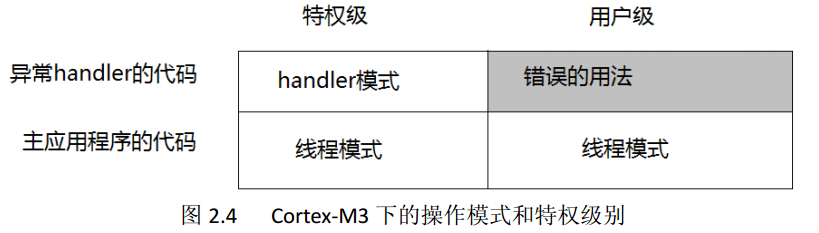
※ CONTROL[0]： 仅当在特权级下操作时才允许写该位。一旦进入了用户级，唯一返回特权级的途径，就是触发一个（软）中断，再由服务例程改写该位;

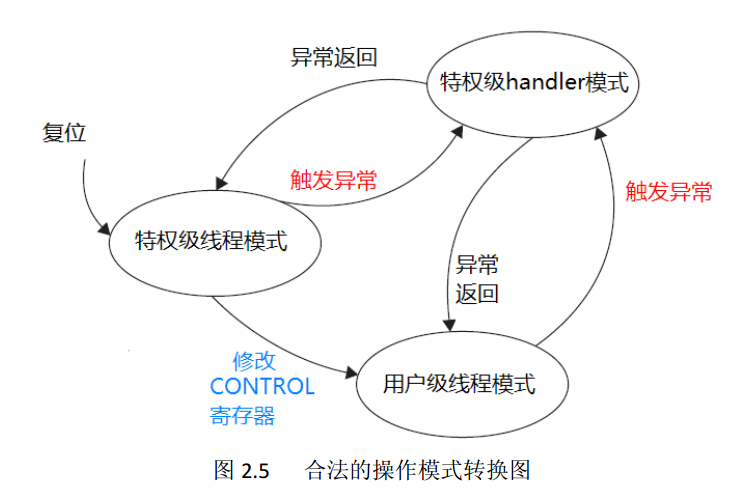
## CM3处理器的模式和级别：

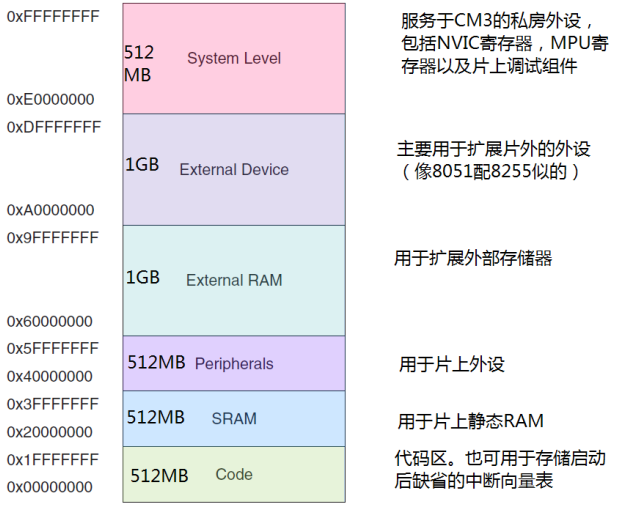
Cortex‐M3 处理器支持两种处理器的操作模式和两级特权操作：

两种操作模式分别为： 处理者模式(handler mode)和线程模式（thread mode），用于区别普通应用程序的代码和异常服务例程的代码（包括中断服务例程的代码）；

Cortex‐M3 的另一个侧面则是特权的分级：特权级和用户级，这可以提供一种存储器访问的保护机制。



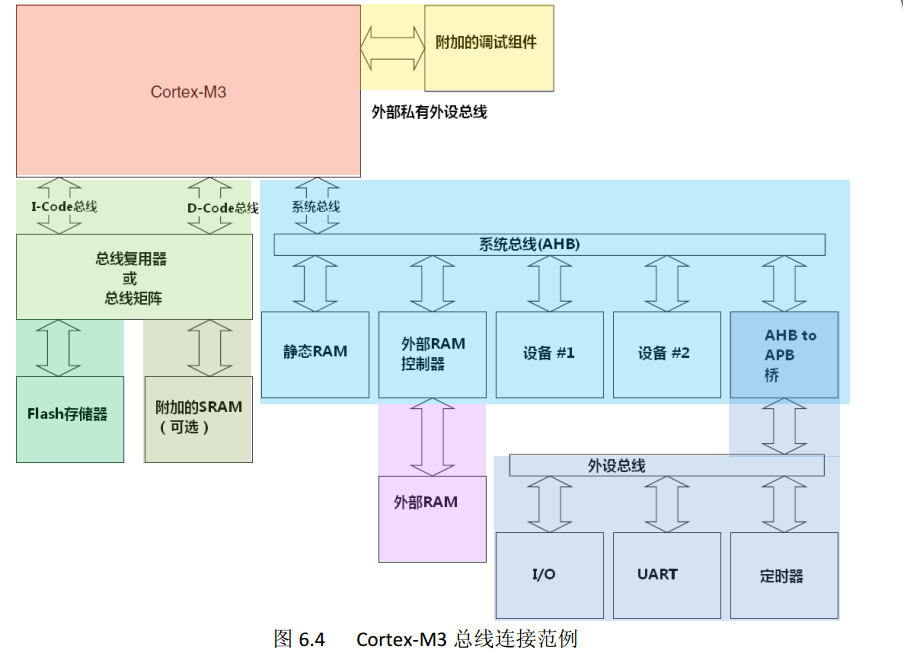




## CM3总线接口：

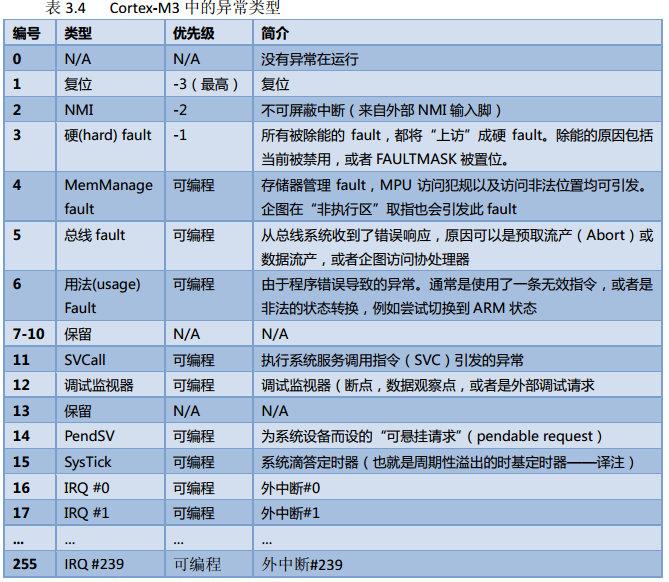
Cortex‐M3 内部有若干个总线接口，以使 CM3 能同时取址和访内（访问内存），它们是：

1. 指令存储区总线（两条, 负责对代码存储区的访问，分别是 I‐Code 总线和 D‐Code， 前者用于取指，后者用于查表等操作）
2. 系统总线( 系统总线用于访问内存和外设，覆盖的区域包括 SRAM，片上外设，片外 RAM，片外扩展设备，以及系统级存储区的部分空间 )
3. 私有外设总线 （主要就是访问调试组件，它们也在系统级存储区）



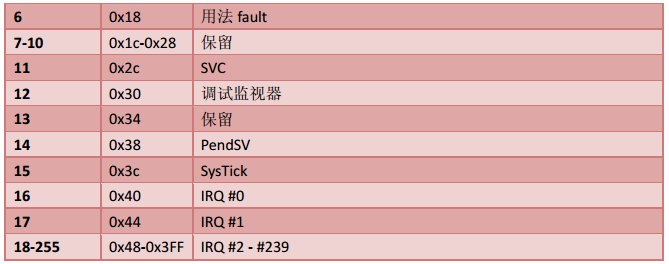
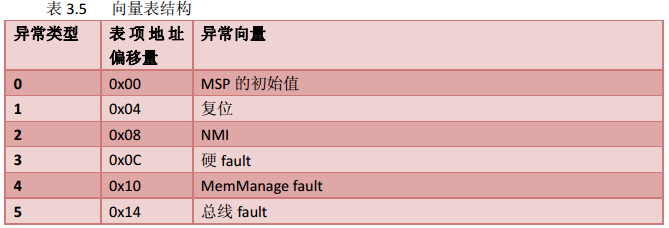
## CM3异常和中断：

Cortex‐M3 支持大量异常，包括 16‐4‐1=11 个系统异常（5个保留），和最多 240 个外部中断（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。



## 向量表：

当一个发生的异常被 CM3 内核接受，对应的异常 handler 就会执行。为了决定 handler 的入口地址， CM3 使用了“向量表查表机制”。



* 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异常 handler 的入口地址；
* 向量表的存储位置是可以设置的，通过 NVIC 中的一个重定位寄存器来指出向量表的地址；

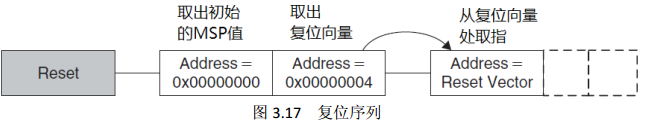
## 栈内存操作：

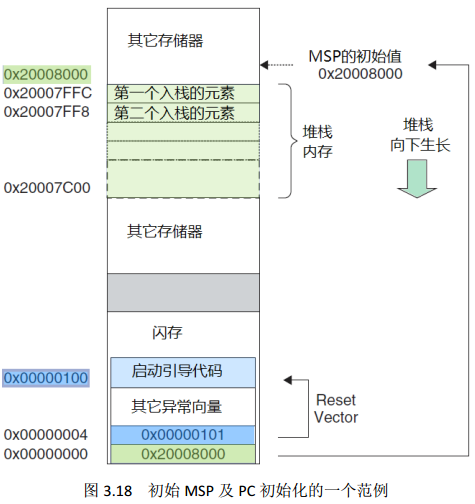
笼统地讲，堆栈操作就是对内存的读写操作，但是其地址由 SP 给出。堆栈的功能就是把寄存器的数据放入内存，以便将来能恢复之——当一个任务或一段子程序执行完毕后恢复。（正常情况下， PUSH 与 POP 必须成对使用，而且参与的寄存器，不论是身份还是先后顺序都必须完全一致。当 PUSH/POP 指令执行时， SP 指针的值也根着自减/自增）

## 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：

1. 从地址 0x0000,0000 处取出 MSP 的初始值。
2. 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（表明CM3是在Thumb态下执行）。然后从这个值所对应的地址处取指。



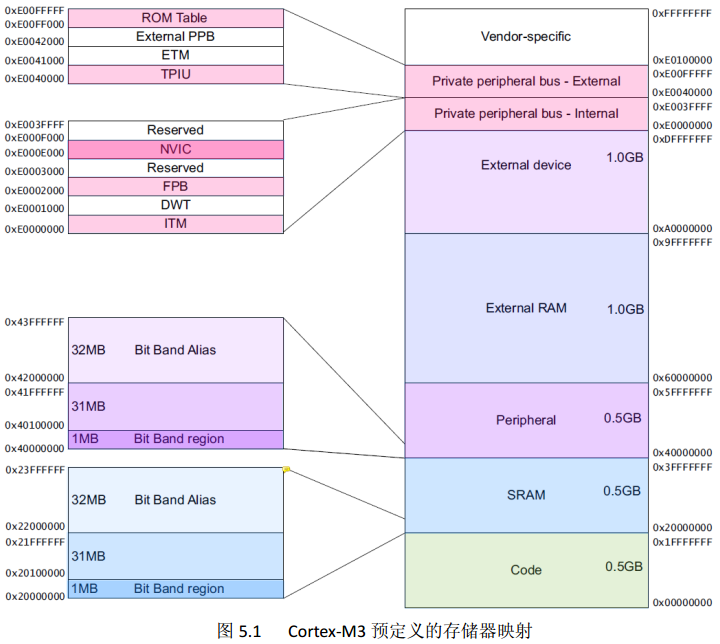


# CH5 存储器系统

## 存储器映射：

CM3 的存储器系统与从传统 ARM 架构的相比，有了如下变化：

1. 它的存储器映射是预定义的， 并且还规定好了哪个位置使用哪条总线;
2. CM3 的存储器系统支持所谓的“位带”(bit‐band)操作。通过它，实现了对单一比特的原子操作(位带操作只适用于数据访问，不适用于取指);
3. CM3 的存储器系统支持非对齐访问和互斥访问;



## 存储器访问属性:

CM3 在定义了存储器映射之外，还为存储器的访问规定了 4 种属性:

(1) 可否缓冲(Bufferable);

(2) 可否缓存(Cacheable);

(3) 可否执行(Executable);

(4) 可否共享(Sharable);

地址空间可以通过另一种方式分为 8 个 512MB 等份：

1. 代码区（ 0x0000\_0000‐ 0x1FFF\_FFFF）: 该区可执行指令，不可缓存, 写操作缓冲;
2. SRAM 区（ 0x2000\_0000 – 0x3FFF\_FFFF）: 该区可执行指令, 写操作缓冲;
3. 片上外设区( 0x4000\_0000 – 0x5FFF\_FFFF ) : 该区不可执行指令，不可缓存；
4. 外部 RAM 区的前半段（ 0x6000\_0000 ‐ 0x7FFF\_FFFF）: 该区是可执行指令, 可缓存;
5. 外部 RAM 区的后半段（ 0x8000\_0000 – 0x9F FF\_FFFF）: 该区是可执行指令, 不可缓存;
6. 外部外设区的前半段( 0xA000\_0000 – 0xBFFF\_FFFF ) : 用于片外外设的寄存器, 也用于多核系统中的共享内存, 该区不可缓冲, 不可执行指令。
7. 外部外设区的后半段( 0xC000\_0000 – 0xDFFF\_FFFF )：目前与前半段的功能完全一致。
8. 系统区( 0xE000\_0000 – 0xFFFF\_FFFF ): 此区是私有外设和供应商指定功能区。此区不可执行代码, 系统区涉及到很多关键部位，因此访问都是严格序列化的（不可缓存，不可缓冲）。而供应商指定功能区则是可以缓存和缓冲的



## 位带操作：

使用普通的加载/存储指令来对单一的比特进行读写。在 CM3中，有两个区中实现了位带。其中一个是 SRAM 区的最低 1MB 范围，第二个则是片内外设区的最低 1MB 范围。位带操作的优越性体现在了可以直接设置GPIO的管脚来控制LED灯的点亮和熄灭，位带操作还能用来简化跳转的判断，如下是新旧操作的对比：

* 读取整个寄存器；
* 掩蔽不需要的位；
* 比较并跳转；

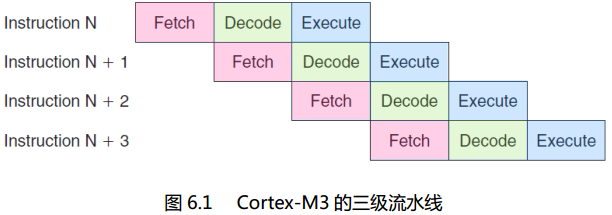
而现在只需要：

* 从位带别名区读取状态位；
* 比较并跳转；

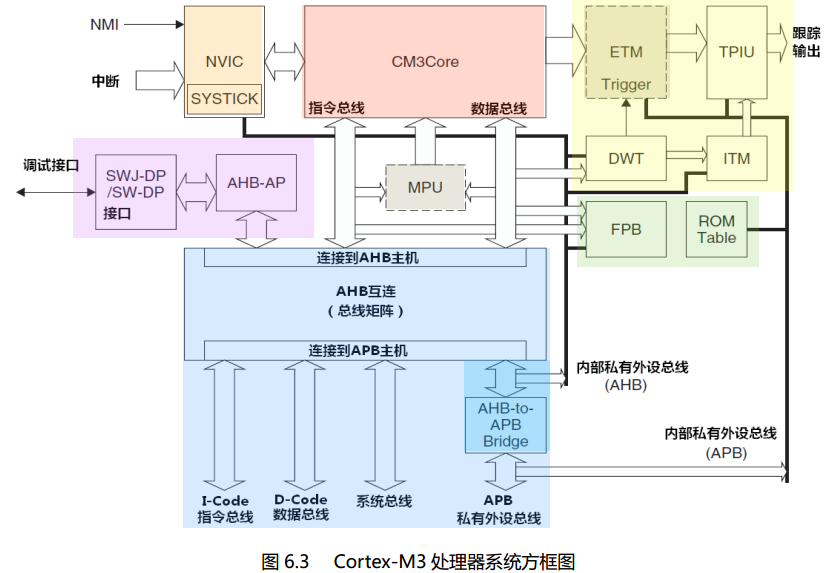
# CH6 实现Cortex-M3的整体风景

## 流水线：

Cortex‐M3 处理器使用一个 3 级流水线。 流水线的 3 级分别是： 取指， 解码和执行。

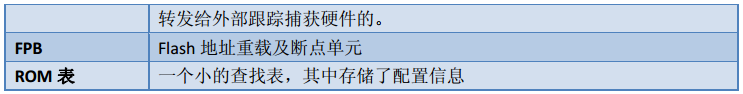
****

## 详细的框图：



虚线框住的 MPU 和 ETM 是可选组件，不一定会包含在每一个 CM3 的 MCU 中。





Cortex‐M3 处理器是以一个“处理器子系统”呈现的，其 CPU 内核本身与 NVIC和一系列调试块都亲密耦合：

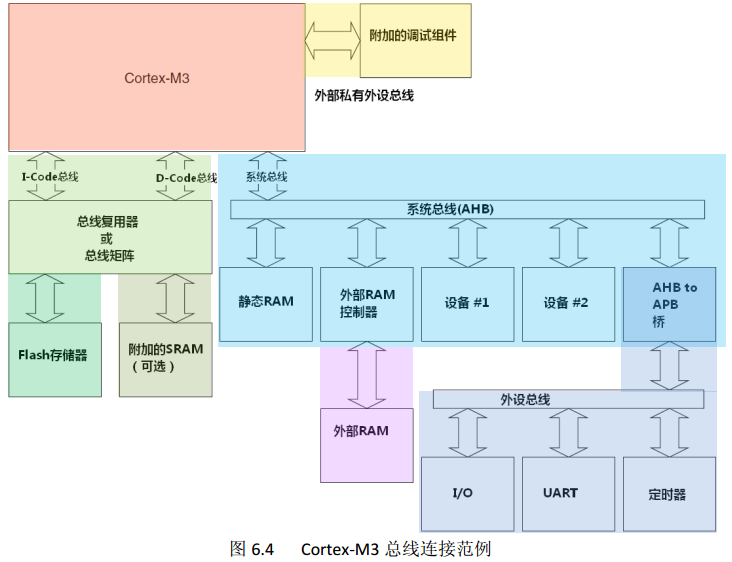
1. CM3Core: Cortex‐M3 处理器的中央处理核心;
2. NVIC(嵌套向量中断控制器): CM3 中内建的中断控制器, 当中断发生时，它会自动取出对应的服务例程入口地址，并直接调用，无需软件判定中断源，缩短了中断延时。
3. SysTick 定时器: 系统滴答定时器是一个非常基本的倒计时定时器，用于在每隔一定的时间产生一个中断，即使是系统在睡眠模式下也能工作。
4. MPU(存储器保护单元): MPU 是一个选配的单元，有些 CM3 芯片可能没有配备此组件。它可以把存储器分成一些 regions，并分别予以保护。(让某些 regions 在用户级下变成只读，从而阻止了一些用户程序破坏关键数据)。
5. BusMatrix: BusMatrix 是 CM3 内部总线系统的核心。它是一个 AHB 互连的网络，通过它可以让数据在不同的总线之间并行传送——只要两个总线主机不试图访问同一块内存区域。
6. AHB to APB: 它是一个总线桥，用于把若干个 APB 设备连接到 CM3 处理器的私有外设总线上（内部的和外部的）。

## Cortex-M3 的总线接口：

1. I-Code 总线: I‐Code 总线是一条基于 AHB‐Lite 总线协议的 32 位总线，负责在 0x0000\_0000 –0x1FFF\_FFFF 之间的取指操作。
2. D-Code 总线: D‐Code 总线也是一条基于 AHB‐Lite 总线协议的 32 位总线，负责在 0x0000\_0000 –0x1FFF\_FFFF 之间的数据访问操作。
3. 系统总线: 系统总线也是一条基于 AHB‐Lite 总线协议的 32 位总线，负责在 0x2000\_0000 –0xDFFF\_FFFF 和 0xE010\_0000 – 0xFFFF\_FFFF 之间的所有数据传送，取指和数据访问都算上。和 D‐Code 总线一样，所有的数据传送都是对齐的。
4. 外部私有外设总线: 系统总线也是一条基于 AHB‐Lite 总线协议的 32 位总线，负责在 0x2000\_0000 –0xDFFF\_FFFF 和 0xE010\_0000 – 0xFFFF\_FFFF 之间的所有数据传送，取指和数据访问都算上。和 D‐Code 总线一样，所有的数据传送都是对齐的。
5. 调试访问端口总线: 调试访问端口总线接口是一条基于“增强型 APB 规格”的 32 位总线，它专用于挂接调试接口，例如 SWJ‐DP 和 SW‐DP。
6. Cortex-M3 的其它接口: 除了总线接口之外， CM3 还有若干个用于其它目的的接口，这些接口的信号都不大可能会引出到引脚上，而只用于连接 SoC 不同的部分，或者干脆就没有使用。



## CM3总线典型的连接方式：



# CH7 异常

## 异常类型:

Cortex‐M3 在内核水平上搭载了一个异常响应系统， 其中，编号为 1－15 的对应系统异常，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。





※ 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。

※ 如果一个发生的异常不能被即刻响应，就称它被“悬起” (pending)( 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disable)

## 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（分别是-3，-2和-1），从而高于所有其它异常。

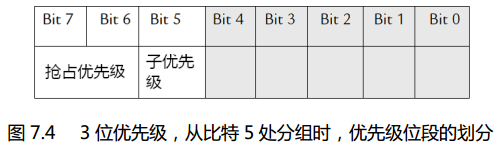
为了使抢占机能变得更可控， CM3 还把 256 级优先级按位分成高低两段，分别是抢占优先级和亚优先级，NVIC 中有一个寄存器是“应用程序中断及复位控制寄存器”（内容见表 7.5），它里面有一个位段名为“优先级组”。该位段的值对每一个优先级可配置的异常都有影响——把其优先级分为个位段： MSB 所在的位段（左边的）对应抢占优先级，而 LSB 所在的位段（右边的）对应亚优先级。





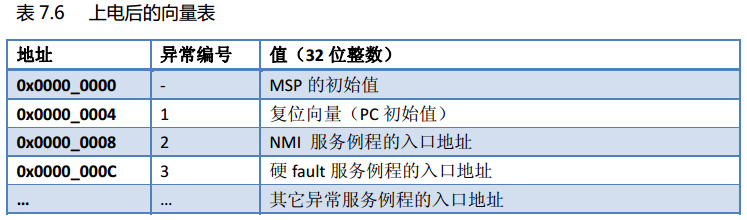
这种优先级分组规定：亚优先级至少是 1 个位。因此抢占优先级最多是 7 个位，造成了最多只有 128 级抢占的现象，但是 CM3 允许从比特 7 处分组，此时所有的位都表达亚优先级，没有任何位表达抢占优先级，因而所有优先级可编程的异常之间就不会发生抢占——相当于在它们之中disable了CM3 的中断嵌套机制。(但是复位， NMI 和硬fault。它们无论何时出现，都立即无条件抢占所有优先级可编程的“平民异常”)

* 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，如下所示。



## 向量表：

当发生了异常并且要响应它时， CM3 需要定位其处理例程的入口地址。这些入口地址存储在所谓的“（异常）向量表”中。缺省情况下， CM3 认为该表位于零地址处，且各向量占用 4 字节，因此每个表项占用 4 字节。（复位后首先读取MSP和复位向量）



如果需要动态地更改向量表，则对于任何器件来说，向量表的起始处都必须包含以下向量：

* 主堆栈指针（ MSP）的初始值
* 复位向量
* NMI
* 硬 fault 服务例程

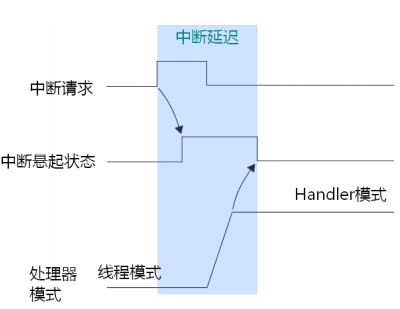
后两者也是必需的，因为有可能在引导过程中发生这两种异常。

可以在 SRAM 中开出一块用于存储向量表。然后在引导完成后，就可以启用内存中的向量表，从而实现向量可动态调整的功能。

## 中断输入及悬起行为：

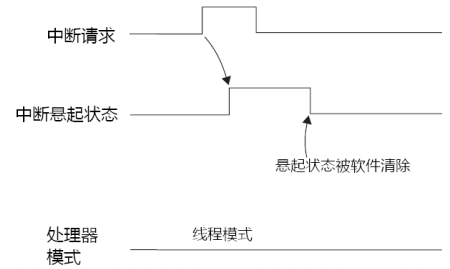
1. 中断悬起：

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。



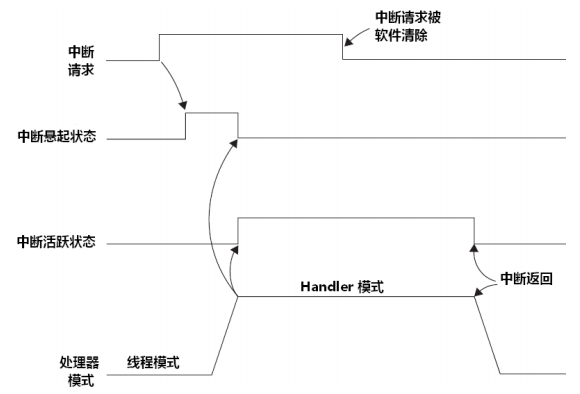
1. 中断在得到处理器响应之前被清除悬起状态：

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。



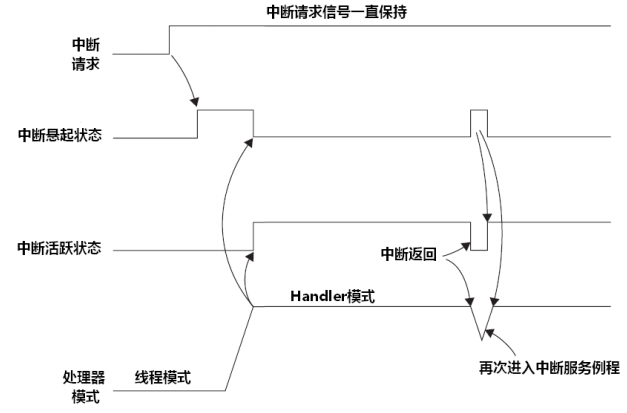
1. 在处理器进入服务例程后对中断活跃状态的设置：

在一个中断活跃后，直到其服务例程执行完毕，并且返回（中断退出）了，才能对该中断的新请求予以响应。



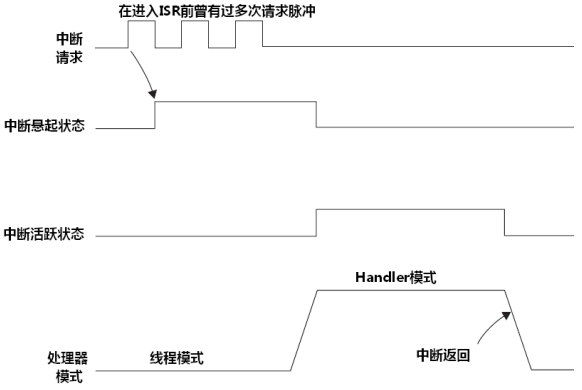
1. 一直维持的中断请求导致服务例程返回后再次悬起该中断：

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。



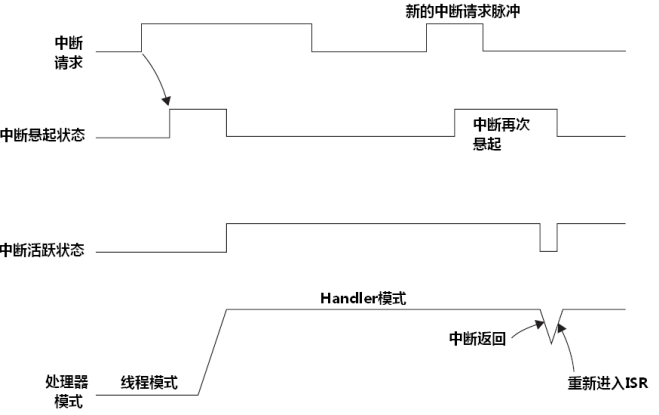
1. 中断请求过快导致一部分请求错失的情况：

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。



1. 在执行 ISR 时中断悬起再次发生：

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。



## Fault 类异常：

1. 总线 faults：当 AHB 接口上正在传送数据时，如果回复了一个错误信号(error response)，则会产生总线 faults。
2. 存储器管理 faults: 存储器管理faults多与MPU有关，其诱因常常是某次访问触犯了MPU设置的保护策略。另外，某些非法访问，例如，在不可执行的存储器区域试图取指，也会触发一个 MemManagefault，而且即使没有 MPU 也会触发。
3. 用法 faults: 发生的场合如下：

* 执行了未定义的指令
* 执行了协处理器指令(Cortex‐M3 不支持协处理器，但是可以通过 fault 异常机制来  
  使用软件模拟协处理器的功能，从而可以方便地在其它 Cortex 处理器间移植)
* 尝试进入 ARM 状态（CM3 不支持 ARM 状态，所以用法 fault 会在切换时产生）
* 无效的中断返回（ LR 中包含了无效/错误的值）
* 使用多重加载/存储指令时，地址没有对齐
* 除数为零
* 任何未对齐的访问

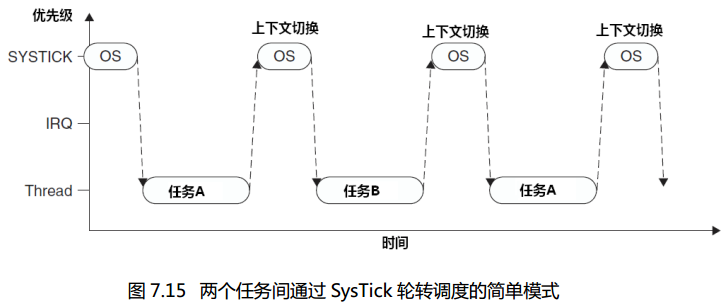
1. 硬 fault: 硬fault 是上文讨论的总线 fault、存储器管理 fault 以及用法 fault 上访的结果。如果这些 fault 的服务例程无法执行，它们就会成为“硬伤”——上访（ escalation）成硬 fault。

## SVC 和 PendSV:

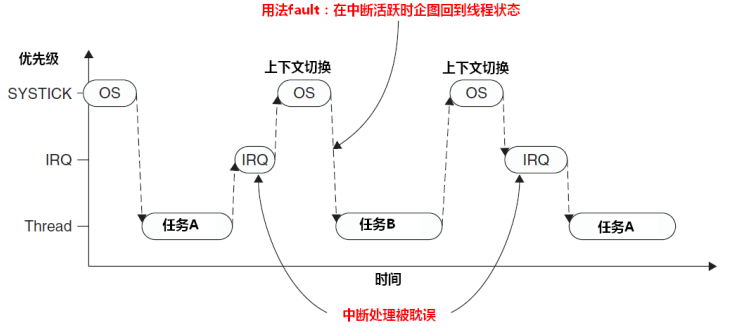
SVC 用于产生系统函数的调用请求。例如，操作系统不让用户程序直接访问硬件，而是通过提供一些系统服务函数，用户程序使用 SVC 发出对系统服务函数的呼叫请求，以这种方法调用它们来间接访问硬件。另一个相关的异常是 PendSV（可悬起的系统调用）。它和 SVC 协同使用。一方面， SVC异常是必须立即得到响应的（若因优先级不比当前正处理的高， 或是其它原因使之无法立即响应， 将上访成硬 fault）, 而PendSV 则不同，它是可以像普通的中断一样被悬起的（不像 SVC 那样会上访）。 OS 可以利用它“缓期执行” 一个异常——直到其它重要的任务完成后才执行动作。

PendSV 的典型使用场合是在上下文切换时，例如， 一个系统中有两个就绪的任务，上下文切换被触发的场合可以是：

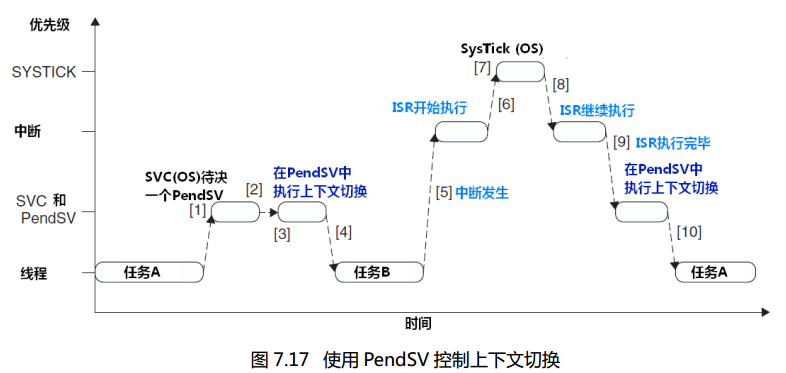
* 执行一个系统调用；
* 系统滴答定时器（ SYSTICK）中断；



但若在产生 SysTick 异常时正在响应一个中断，则SysTick 异常会抢占其 ISR。在这种情况下， OS 不得执行上下文切换，否则将使中断请求被延迟，而且在真实系统中延迟时间还往往不可预知——任何有一丁点实时要求的系统都决不能容忍这种事。因此，如果 OS 在中断活跃时尝试切入线程模式，将触发fault 异常。



PendSV 来完美解决这个问题了。PendSV 异常会自动延迟上下文切换的请求，直到其它的 ISR 都完成了处理后才放行，为实现这个机制，需要把 PendSV 编程为最低优先级的异常。如果 OS 检测到某 IRQ 正在活动并且被 SysTick 抢占，它将悬起一个 PendSV 异常，以便缓期执行上下文切换。



流程如下：

1. 任务 A 呼叫 SVC 来请求任务切换（例如，等待某些工作完成）。

2. OS 接收到请求，做好上下文切换的准备，并且 pend 一个 PendSV 异常。

3. 当 CPU 退出 SVC 后，它立即进入 PendSV，从而执行上下文切换。

4. 当 PendSV 执行完毕后，将返回到任务 B，同时进入线程模式。

5. 发生了一个中断，并且中断服务程序开始执行

6. 在 ISR 执行过程中，发生 SysTick 异常，并且抢占了该 ISR。

7. OS 执行必要的操作，然后 pend 起 PendSV 异常以作好上下文切换的准备。

8. 当 SysTick 退出后，回到先前被抢占的 ISR 中， ISR 继续执行

9. ISR 执行完毕并退出后， PendSV 服务例程开始执行，并且在里面执行上下文切换

10. 当 PendSV 执行完毕后，回到任务 A，同时系统再次进入线程模式。

# CH8 NVIC与中断控制：

## NVIC 概览：

NVIC 的寄存器以存储器映射的方式来访问，除了包含控制寄存器和中断处理的控制逻辑之外，NVIC 还包含了 MPU的控制寄存器、SysTick定时器以及调试控制。NVIC 的访问地址是 0xE000\_E000。 所有 NVIC 的中断控制/状态寄存器都只能在特权级下访问。 不过有一个例外——软件触发中断寄存器可以在用户级下访问以产生软件中断。

## 中断配置基础:

每个外部中断都在 NVIC 的下列寄存器中“挂号”：

1. 使能与除能寄存器
2. 悬起与“解悬”寄存器
3. 优先级寄存器
4. 活动状态寄存器

另外，下列寄存器也对中断处理有重大影响:

(1) 异常掩蔽寄存器（ PRIMASK, FAULTMASK 以及 BASEPRI）

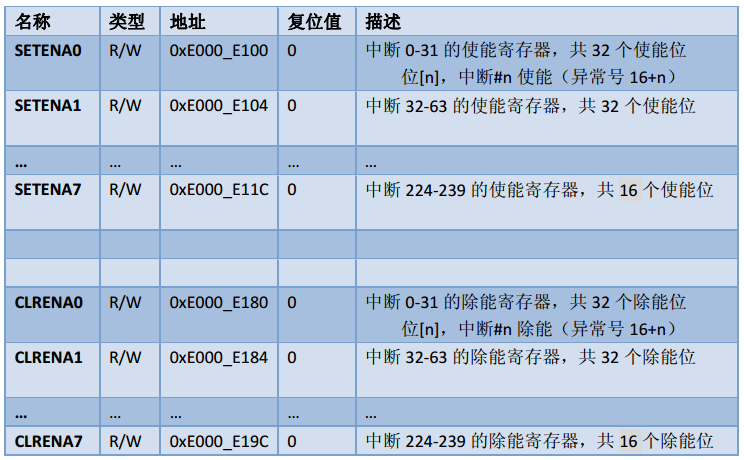
(2) 向量表偏移量寄存器

(3) 软件触发中断寄存器

(4) 优先级分组位段

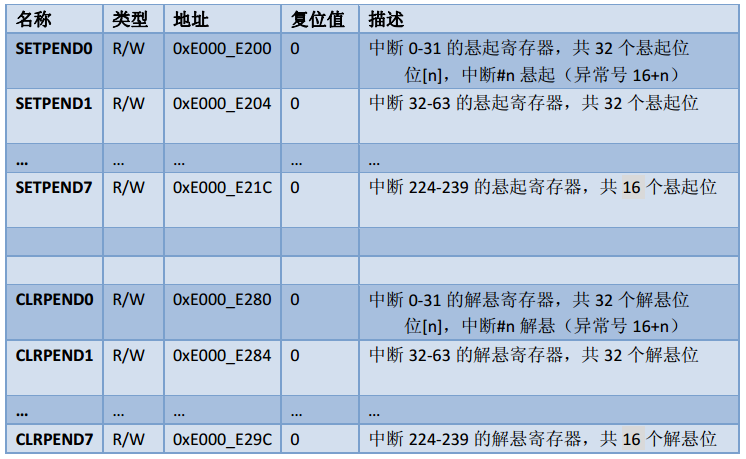
## 中断的使能与除能:

中断的使能与除能分别使用各自的寄存器来控制, CM3 中可以有 240 对使能位／除能位，每个中断拥有一对。这 240 个对子分布在 8 对 32 位寄存器中（最后一对没有用完）。欲使能一个中断，你需要写 1 到对应 SETENA 的位中；欲除能一个中断，你需要写 1 到对应的 CLRENA 位中。



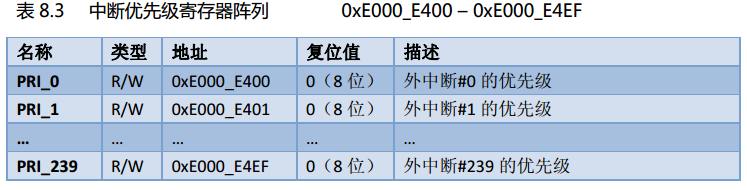
## 中断的悬起与解悬：

如果中断发生时，正在处理同级或高优先级异常，或者被掩蔽，则中断不能立即得到响应。此时中断被悬起。中断的悬起状态可以通过“中断设置悬起寄存器(SETPEND)”和“中断悬起清除寄存器(CLRPEND)”来读取。



## 优先级：

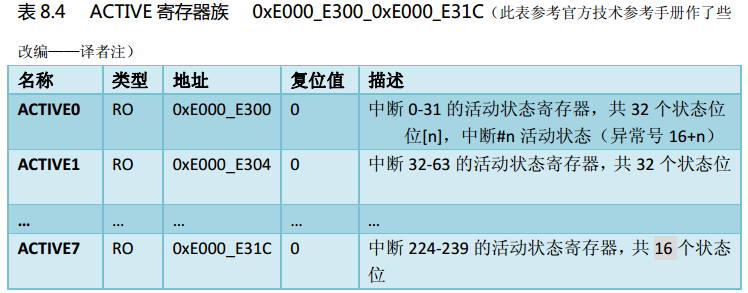
每个外部中断都有一个对应的优先级寄存器，每个寄存器占用 8 位，但是允许最少只使用最高 3 位。 4 个相临的优先级寄存器拼成一个 32 位寄存器。





## 活动状态：

每个外部中断都有一个活动状态位。在处理器执行了其 ISR 的第一条指令后，它的活动位就被置 1，并且直到 ISR 返回时才硬件清零。由于支持嵌套，允许高优先级异常抢占某个ISR。然而，哪怕一个中断被抢占，其活动状态也依然为 1。

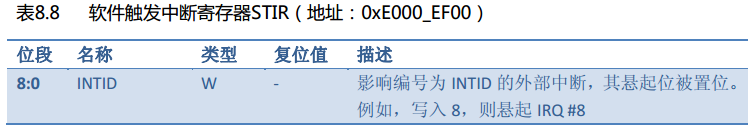


## 中断建立全过程的演示：

1. 当系统启动后，先设置优先级组寄存器。缺省情况下使用组0（ 7位抢占优先级， 1位亚优先级）。
2. 如果需要重定位向量表，先把硬fault和NMI服务例程的入口地址写到新表项所在的地址中。
3. 配置向量表偏移量寄存器，使之指向新的向量表（如果有重定位的话）
4. 为该中断建立中断向量。因为向量表可能已经重定位了，保险起见需要先读取向量表偏移量寄存器的值，再根据该中断在表中的位置，计算出服务例程入口地址应写入的表项，再填写之。如果一直使用ROM中的向量表，则无需此步骤
5. 为该中断设置优先级。
6. 使能该中断

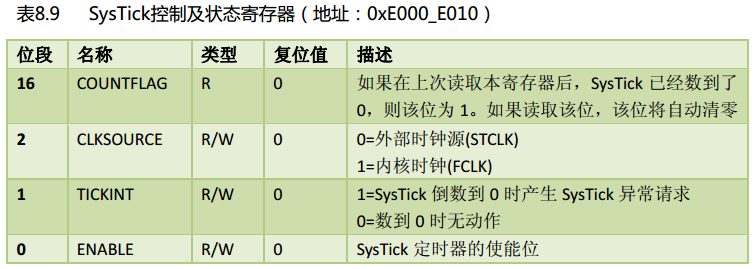
## 软件中断:

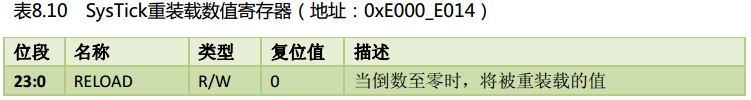
软件中断，包括手工产生的普通中断，能以多种方式产生。最简单的就是使用相应的SETPEND寄存器；而更专业更快捷的作法，则是通过使用软件触发中断寄存器STIR。

****

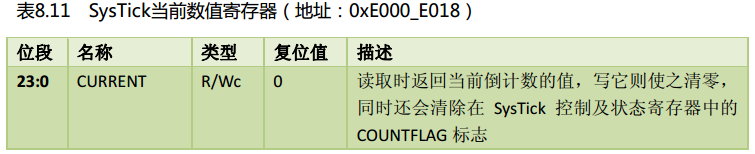
## SysTick定时器：

SysTick定时器被捆绑在NVIC中，用于产生SYSTICK异常（异常号： 15）。SysTick定时器能产生中断， CM3为它专门开出一个异常类型，并且在向量表中有它的一席之地。它使操作系统和其它系统软件在CM3器件间的移植变得简单多了，因为在所有CM3产品间对其处理都是相同的。有4个寄存器控制SysTick定时器：









* SysTick定时器除了能服务于操作系统之外，还能用于其它目的：如作为一个闹铃，用于测量时间等。要注意的是，当处理器在调试期间被喊停（ halt）时，则SysTick定时器亦将暂停运作。

# CH9中断的具体行为

## 中断／异常的响应序列

当CM3开始响应一个中断时，会经过三个过程：

入栈 🡺 取向量 🡺 更新寄存器

(1) 入栈： 把8个寄存器的值压入栈

(2) 取向量：从向量表中找出对应的服务程序入口地址

(3) 更新寄存器 : 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



## 嵌套的中断：

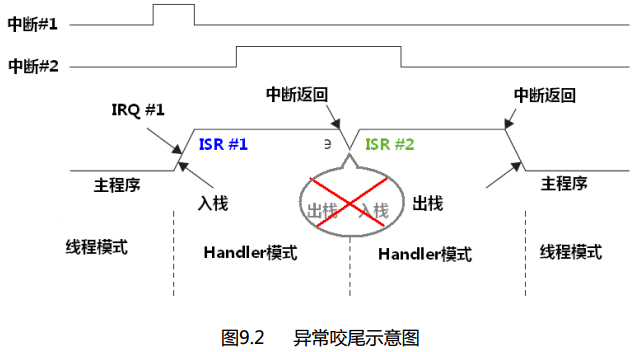
在CM3内核以及NVIC的深处，就已经内建了对中断嵌套的全力支持，根本无需使用用汇编写封皮代码(wrapper code)：

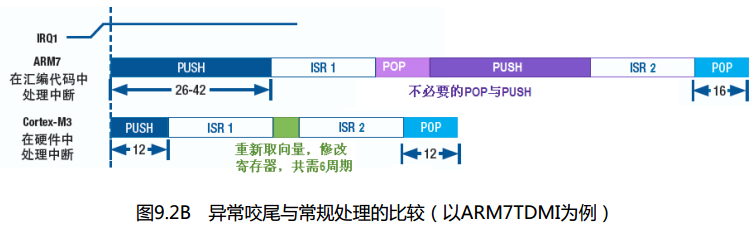
1. NVIC和CM3处理器会为我们排出优先级解码的顺序。
2. 有了自动入栈和出栈，就不用担心在中断发生嵌套时，会使寄存器的数据损毁，从而可以放心地执行服务例程。

* 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。
* 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。

**咬尾中断:**

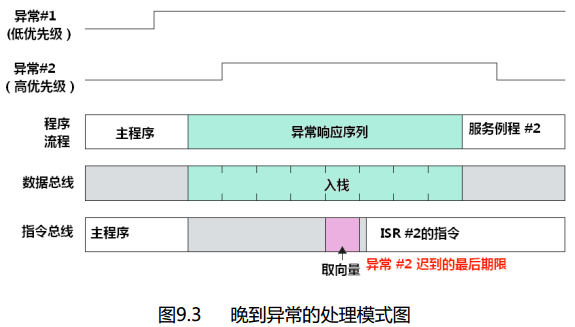
当处理器在响应某异常时，如果又发生其它异常，但它们优先级不够高，则被阻塞, 那么在当前的异常执行返回后，系统处理悬起的异常时，倘若还是先POP然后又把POP出来的内容PUSH回去, 这样就浪费了CPU时间， 所以咬尾中断是不会POP这些寄存器，而是继续使用上一个异常已经PUSH好的成果，从而节省了CPU时间。





**晚到（的高优先级）异常：**

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。



**异常返回值：**

在进入异常服务程序后， LR的值被自动更新为特殊的EXC\_RETURN，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 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

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：

z 应用程序 PSR（ APSR）

z 中断号 PSR（ IPSR）

z 执行 PSR（ EPSR）

===============================================================

表 4.2 16 位数据操作指令

名字 功能

ADC 带进位加法

ADD 加法

AND 按位与（原文为逻辑与，有误——译注）。这里的按位与和 C 的”&”功能相同

ASR 算术右移

BIC 按位清 0（把一个数跟另一个无符号数的反码按位与）

CMN 负向比较（把一个数跟另一个数据的二进制补码相比较）

CMP 比较（比较两个数并且更新标志）

CPY 把一个寄存器的值拷贝到另一个寄存器中

EOR 近位异或

LSL 逻辑左移（如无其它说明，所有移位操作都可以一次移动多格——译注）

LSR 逻辑右移

MOV 寄存器加载数据，既能用于寄存器间的传输，也能用于加载立即数

MUL 乘法

MVN 加载一个数的 NOT 值（取到逻辑反的值）

NEG 取二进制补码

ORR 按位或（原文为逻辑或，有误——译注）

ROR 圆圈右移

SBC 带借位的减法

SUB 减法

TST 测试（执行按位与操作，并且根据结果更新 Z）

REV 在一个 32 位寄存器中反转字节序

REVH 把一个 32 位寄存器分成两个 16 位数，在每个 16 位数中反转字节序

REVSH 把一个 32 位寄存器的低 16 位半字进行字节反转，然后带符号扩展到 32 位

SXTB 带符号扩展一个字节到 32 位

SXTH 带符号扩展一个半字到 32 位

UXTB 无符号扩展一个字节到 32 位

UXTH 无符号扩展一个半字到 32 位

表 4.3 16 位转移指令

名字 功能

B 无条件转移

B<cond> 条件转移

BL 转移并连接。用于呼叫一个子程序，返回地址被存储在 LR 中

CBZ 比较，如果结果为 0 就转移（只能跳到后面的指令——译注）

CBNZ 比较，如果结果非 0 就转移（只能跳到后面的指令——译注）

IT If‐Then

表 4.4 16 位存储器数据传送指令

名字 功能

LDR 从存储器中加载字到一个寄存器中

LDRH 从存储器中加载半字到一个寄存器中

LDRB 从存储器中加载字节到一个寄存器中

LDRSH 从存储器中加载半字，再经过带符号扩展后存储一个寄存器中

LDRSB 从存储器中加载字节，再经过带符号扩展后存储一个寄存器中

STR 把一个寄存器按字存储到存储器中

STRH 把一个寄存器存器的低半字存储到存储器中

STRB 把一个寄存器的低字节存储到存储器中

LDMIA 加载多个字，并且在加载后自增基址寄存器

STMIA 加载多个字，并且在加载后自增基址寄存器

PUSH 压入多个寄存器到栈中

POP 从栈中弹出多个值到寄存器中

表 4.5 其它 16 位指令

名字 功能

SVC 系统服务调用

BKPT 断点指令。如果调试被使能，则进入调试状态（停机）。或者如果调试监视器异常被

使能，则调用一个调试异常，否则调用一个 fault 异常

NOP 无操作

CPSIE 使能 PRIMASK(CPSIE i)/ FAULTMASK(CPSIE f)——清 0 相应的位

CPSID 除能 PRIMASK(CPSID i)/ FAULTMASK(CPSID f)——置位相应的位

表 4.6 32 位数据操作指令

名字 功能

ADC 带进位加法

ADD 加法

ADDW 宽加法（可以加 12 位立即数）

AND 按位与（原文是逻辑与，有误——译注）

ASR 算术右移

BIC 位清零（把一个数按位取反后，与另一个数逻辑与）

BFC 位段清零

BFI 位段插入

CMN 负向比较（把一个数和另一个数的二进制补码比较，并更新标志位）

CMP 比较两个数并更新标志位

CLZ 计算前导零的数目

EOR 按位异或

LSL 逻辑左移

LSR 逻辑右移

MLA 乘加

MLS 乘减

MOVW 把 16 位立即数放到寄存器的底 16 位， 高 16 位清 0

MOV 加载 16 位立即数到寄存器（其实汇编器会产生 MOVW——译注）

MOVT 把 16 位立即数放到寄存器的高 16 位， 低 16 位不影响

MVN 移动一个数的补码

MUL 乘法

ORR 按位或（原文为逻辑或，有误——译注）

ORN 把源操作数按位取反后，再执行按位或（原文为逻辑或，有误——译注）

RBIT 位反转（把一个 32 位整数先用 2 进制表达，再旋转 180 度——译注）

REV 对一个 32 位整数做按字节反转

REVH/REV16 对一个 32 位整数的高低半字都执行字节反转

REVSH 对一个 32 位整数的低半字执行字节反转，再带符号扩展成 32 位数

ROR 圆圈右移

RRX 带进位的逻辑右移一格（最高位用 C 填充，且不影响 C 的值——译注）

SFBX 从一个 32 位整数中提取任意的位段，并且带符号扩展成 32 位整数

SDIV 带符号除法

SMLAL 带符号长乘加（两个带符号的 32 位整数相乘得到 64 位的带符号积，再把积加到另一个带符号 64 位整数中）

SMULL 带符号长乘法（两个带符号的 32 位整数相乘得到 64 位的带符号积）

SSAT 带符号的饱和运算

SBC 带借位的减法

SUB 减法

SUBW 宽减法，可以减 12 位立即数

SXTB 字节带符号扩展到 32 位数

TEQ 测试是否相等（对两个数执行异或，更新标志但不存储结果）

TST 测试（对两个数执行按位与，更新 Z 标志但不存储结果）

UBFX 无符号位段提取

UDIV 无符号除法

UMLAL 无符号长乘加（两个无符号的 32 位整数相乘得到 64 位的无符号积，再把积加到另一

个无符号 64 位整数中）

UMULL 无符号长乘法（两个无符号的 32 位整数相乘得到 64 位的无符号积）

USAT 无符号饱和操作（但是源操作数是带符号的——译注）

UXTB 字节被无符号扩展到 32 位（高 24 位清 0——译注）

UXTH 半字被无符号扩展到 32 位（高 16 位清 0——译注）

表 4.7 32 位存储器数据传送指令

名字 功能

LDR 加载字到寄存器

LDRB 加载字节到寄存器

LDRH 加载半字到寄存器

LDRSH 加载半字到寄存器，再带符号扩展到 32 位

LDM 从一片连续的地址空间中加载多个字到若干寄存器

LDRD 从连续的地址空间加载双字（ 64 位整数）到 2 个寄存器

STR 存储寄存器中的字

STRB 存储寄存器中的低字节

STRH 存储寄存器中的低半字

STM 存储若干寄存器中的字到一片连续的地址空间中

STRD 存储 2 个寄存器组成的双字到连续的地址空间中

PUSH 把若干寄存器的值压入堆栈中

POP 从堆栈中弹出若干的寄存器的值

表 4.8 32 位转移指令

名字 功能

B 无条件转移

BL 转移并连接（呼叫子程序）

TBB 以字节为单位的查表转移。从一个字节数组中选一个 8 位前向跳转地址并转移

TBH 以半字为单位的查表转移。从一个半字数组中选一个 16 位前向跳转的地址并转移

表 4.9 其它 32 位指令

LDREX 加载字到寄存器，并且在内核中标明一段地址进入了互斥访问状态

LDREXH 加载半字到寄存器，并且在内核中标明一段地址进入了互斥访问状态

LDREXB 加载字节到寄存器，并且在内核中标明一段地址进入了互斥访问状态

STREX 检查将要写入的地址是否已进入了互斥访问状态，如果是则存储寄存器的字

STREXH 检查将要写入的地址是否已进入了互斥访问状态，如果是则存储寄存器的半字

STREXB 检查将要写入的地址是否已进入了互斥访问状态，如果是则存储寄存器的字节

CLREX 在本地的处理上清除互斥访问状态的标记（先前由 LDREX/LDREXH/LDREXB 做的标记）

MRS 加载特殊功能寄存器的值到通用寄存器

MSR 存储通用寄存器的值到特殊功能寄存器

NOP 无操作

SEV 发送事件

WFE 休眠并且在发生事件时被唤醒

WFI 休眠并且在发生中断时被唤醒

ISB 指令同步隔离（与流水线和 MPU 等有关——译注）

DSB 数据同步隔离（与流水线、 MPU 和 cache 等有关——译注）

DMB 数据存储隔离（与流水线、 MPU 和 cache 等有关——译注）

0xE0000000 至 0xE00FFFFF

