第13章

# Cortex-M3的其它特性

* SysTick定时器
* 电源管理
* 多处理机通信
* 自复位控制

到了这里，我们已经学完了CM3的绝大多数重要和基础的特性，再加一把劲儿，这章不难，过了以后就到了一个里程碑了。

## 13.1 SysTick定时器

回顾第8章讲述NVIC时，曾走马观花地带过了SysTick定时器。复习一下：SysTick是一个24位的倒计数定时器，当计到0时，将从RELOAD寄存器中自动重装载定时初值。只要不把它在SysTick控制及状态寄存器中的使能位清除，就永不停息。图13.1中小结了SysTick的相关寄存器。

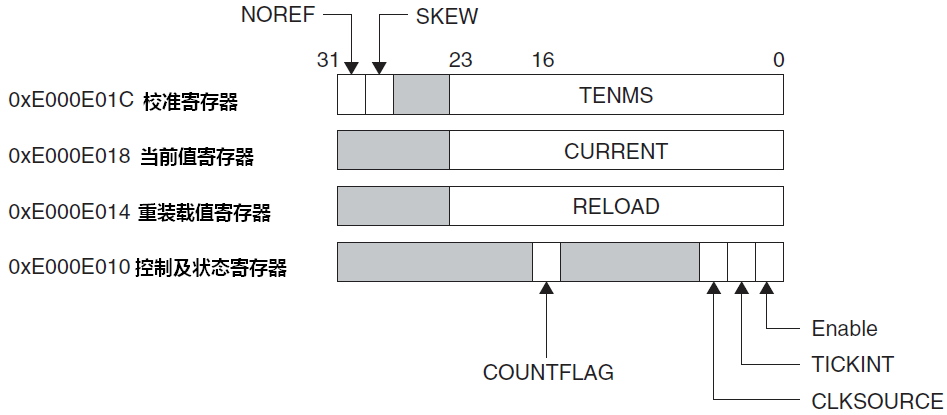


图13.1 SysTick相关寄存器的定义

CM3允许为SysTick提供两个时钟源以供选择。第一个是内核的“自由运行时钟”FCLK。“自由”表现在它不来自系统时钟HCLK，因此在系统时钟停止时FCLK也继续运行。第二个是一个外部的参考时钟。但是使用外部时钟时，因为它在内部是通过FCLK来采样的，因此其周期必须至少是FCLK的两倍（采样定理）。很多情况下芯片厂商都会忽略此外部参考时钟，因此通常不可用。通过检查校准寄存器的位[31](NOREF)，可以判定是否有可用的外部时钟源，而芯片厂商则必须把该引线连接至正确的电平。

当SysTick定时器从1计到0时，它将把COUNTFLAG位置位；而下述方法可以清零之：

读取SysTick控制及状态寄存器（STCSR）

往SysTick当前值寄存器（STCVR）中写任何数据

SysTick的最大使命，就是定期地产生异常请求，作为系统的时基。OS都需要这种“滴答”来推动任务和时间的管理。如欲使能SysTick异常，则把STCSR.TICKINT置位。另外，如果把向量表重定位到了SRAM中，还需要为SysTick异常建立向量，提供其服务例程的入口地址，如下段代码所演示：

; 建立SysTick异常服务例程

MOV R0, #0xF ; 异常号：15

LDR R1, =systick\_handler ; 加载服务例程的入口地址

LDR R2, =0xE000ED08 ; 加载向量表偏移量寄存器的地址

LDR R2, [R2] ; 读取向量表的首地址

STR R1, [R2, R0, LSL #2] ; 写入向量

下面的代码演示启用SysTick的基本程序

; 使能SysTick定时器，并且使能SysTick异常

LDR R0, =0xE000E010 ; 加载STCSR的地址

MOV R1, #0

STR R1, [R0] ; 先停止SysTick，以防意外产生异常请求

LDR R1, =0x3FF ; 让SysTick每1024周期计完一次。因为是从1023数到

; 0，总共数了1024个周期，所以加载值为0x3FF

STR R1, [R0,#4] ; 写入重装载的值

STR R1, [R0,#8] ; 往STCVR中写任意的数，以确保清除COUNTFLAG标志

MOV R1, #0x7 ; 选择FCLK作为时钟源，并使能SysTick及其异常请求

STR R1, [R0] ; 写入数值，开启定时器

除此之外，SysTick定时器还提供了走完10ms所需要的格数(TENMS位段)，作为时间校准的参考信息。在CM3处理器的顶层有一个24位的输入，芯片厂商可以写入一个10ms的加载值，写程序时就可以读取STCR寄存器中的TENMS位段来获取此信息。不一定每个芯片都实现了此功能，因此在使用时还需查阅芯片的数据手册。

SysTick定时器还可以用作闹钟，作为启动一个特定任务的时间依据。例如，如果需要在300周期后执行一段代码，就可以在SysTick异常服务例程中设置执行那段代码的软件标志。使用SysTick时，清零CURRENT再编程RELOAD寄存器，以使它在300周期后产生异常，如下述代码所演示：

LDR r0, =15

LDR r1, =SysTickAlarm ; SysTick异常服务例程为SetupExcpHanler

BL SetupExcpHandler ; 调用前面章节讲到的子程来建立向量

LDR R0, =0xE000E010 ; SysTick寄存器组的基地址

MOV R1, #0 ; 编程前先除能SysTick

STR R1, [R0]

STR R1, [R0,#0x8] ; 清零CURRENT

LDR R1, =(300-12) ; 设置装载值。减去12是为了补偿中延迟

STR R1, [R0,#0x4] ; 写入RELOAD

LDR R4, =SysTickFired ; 在RAM中的一个变量，指示是计时到期

MOV R5, #0 ; 初始为0

STR R5, [R4]

MOV R1, #0x7 ; 使用FCLK，使能SysTick，使能SysTick异常

STR R1, [R0] ; 启动计时

LDR R4, ＝SysTickFired

WaitLoop

LDR R5, [R4] ; 循环查询软件标志

CMP R5, #0

BEQ WaitLoop

... ; SysTickFired在服务例程中被置位，主程序可以继续执行

本例中使用以前讲到的SetupExcpHandler来建立向量表，但注意：必须重定位向量表到RAM中才行。

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

因为计数器是从0开始计数的，所以它会立即把300-12加载入CURRENT。12是中断响应的最短延时，因此减去它用以补偿。但是如果有更高优先级的异常抢占或者阻塞了它，则中断延迟还是会有的。

另外要注意的，减去12只适用于一次性（one shot）的闹钟操作，在这种情况下必须在SysTick服务例程中按停这个SysTick。进一步地，如果其它异常把它延迟得太久，就有可能会使SysTick异常被悬起两次。因此，对于单次处理时，还需要其它一些步骤来消灭二次触发：

SysTickAlarm ; SYSTICK exception handler

PUSH {LR}

LDR R0, =0xE000E010 ; SYSTICK寄存器组的基地址

MOV R1, #0

STR R1, [R0] ; 除能SysTick，因为只使用一次

**LDR R0, =0xE000ED04**

**LDR R1, =0x02000000**  ; 手工清除NVIC中的SysTick悬起位

**STR R1, [R0]**

... ; 执行所需的处理工作

LDR R2, ＝SysTickFired

LDR R1, [R2]

ORR R1, #1

STR R1, [R2] ; 设置软件标志，与主程序同步，以执行任务

POP {PC} ; 异常返回

在服务例程的末尾处，通过设置SysTickFired标志，通知主程序定时已经到期，可以结束循环等待了。

## 13.2 电源管理

不同于以往的处理器，CM3对电源管理的重视，已经上升到处理器内核的水平上。它提供了若两种睡眠模式。在睡眠时，可以停止系统时钟，但可以让FCLK继续走，以允许处理器能被SysTick异常唤醒。这两种睡眠模式依次为：

睡眠：由CM3处理器的SLEEPING信号指示

深度睡眠：由CM3处理器的SLEEPDEEP信号指示

为了判定当前使用的是哪一种睡眠模式，以及其它睡眠时的上下文，需要检视在NVIC的系统控制寄存器，如表13.1所示。要注意，CM3的这两条信号线是给芯片设计者看的，需要芯片设计者配合它们作一系列的处理，因此不同的芯片，响应这两种睡眠模式的方式也是不同的。粗线条的实现可能把它们两个等同处理也说不定。

表13.1 系统控制寄存器（地址：0xE000\_ED10）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 位段 | 名称 | 类型 | 复位值 | 描述 |
| 4 | SEVONPEND | RW | - | 发生异常悬起时将发送事件。在使用WFE指令睡眠后，此位可以使得新悬起的中断把CM3从WFE指令处唤醒。不管这个中断的优先级是否比当前的高，都唤醒。 |
| 3 | 保留 | - | - | - |
| 2 | SLEEPDEEP | R/W | 0 | 当进入睡眠模式时，使能外部的SLEEPDEEP信号，以允许停止系统时钟 |
| 1 | SLEEPONEXIT | R/W | - | 激活“SleepOnExit”功能 |
| 0 | 保留 | - | - | - |

通过执行WFI/WFE指令，请求CM3进入睡眠模式，它们在CM3中的地位就类似于某些处理器的”sleep/slp”指令。WFI表示Wait-For-Interrupt，而WFE表示Wait-For-Event。那么什么可以算是event呢？新来的中断、早先被悬起的中断，或者是通过RXEV信号表示的一个外部事件信号脉冲，都属于event。在处理内部，对事件有一个锁存器，因此过去发生的事件可以用来唤醒将来才执行到的WFE。流程如图13.2所示。

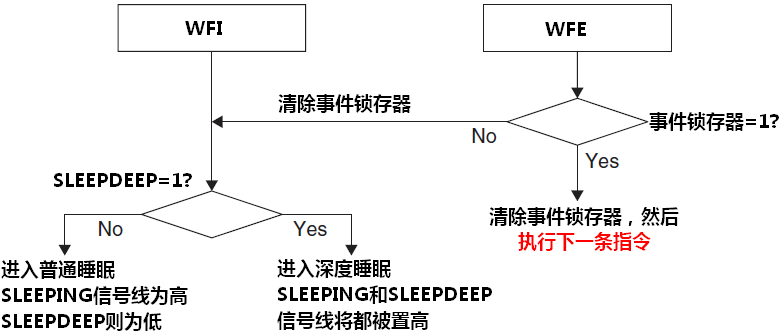


图13.2 进入睡眠模式的序列

当处理器进入睡眠模式时，单片机作如何反应，还取决于芯片的设计。最典型的作法就是把一些外设的时钟停掉以降低功耗。当然，芯片还可以做得更有力，切断一部分功能模块的电源，甚至切断整个芯片的电源并且停止所有的时钟。这是把事情做绝了，只能通过复位来唤醒。为此，芯片厂商可以在单片机上提供一个引脚，并根据它的电平变化来产生此复位信号。另外，芯片厂商还可以在设计时加入少量的SRAM作为后备存储区，该区电力供应不被切断（如STM32），以供应用程序在轮回前，先把今生离别之际的一些重要上下文存入，待到来世再报恩。

WFI/WFE除了进入睡眠的序列不同外，它们的唤醒行为也有所不同。

当从WFI唤醒时，要根据异常系统的游戏规则来决定是否唤醒。只有当该中断的优先级比当前优先级要高（如果是在服务例程中使用WFI），并且比BASEPRI掩蔽的高时，才唤醒处理器并执行ISR。但如果PRIMASK置位，则依然唤醒处理器，然而ISR却不执行了。

WFE则有点区别，不管优先级和掩蔽情况如何，只要SETONPEND置位，它就会不错过任何一个事件，在发生事件时一定把处理器唤醒。至于是否执行ISR，则与WFI的规则相同。

CM3处理器唤醒的具体规则如表13.2A和表13.2B所示。但要注意：这是假设中断的优先级比当前优先级要高的（即没有在异常服务例程中使用WFI/WFE，谁在这里用谁想不开）。

表13.2A WFI的唤醒行为（带“+”的表示执行此动作）

|  |  |  |
| --- | --- | --- |
| 中断优先级 | 唤醒 | 执行ISR |
| PRIMASK=0，且BASEPRI不能掩蔽  PRIMASK=0，且BASEPRI能够掩蔽 | + | + |
| PRIMASK=1，且BASEPRI不能掩蔽  PRIMASK=1，且BASEPRI能够掩蔽 | + |  |

表13.2B WFE的唤醒行为（带“+”的表示执行此动作）

|  |  |  |
| --- | --- | --- |
| 中断优先级 | 唤醒 | 执行ISR |
| PRIMASK=0，SEVONPEND=0，且BASEPRI不能掩蔽  PRIMASK=0，SEVONPEND=0，且BASEPRI能够掩蔽 | + | + |
| PRIMASK=0，SEVONPEND=1，且BASEPRI不能掩蔽  PRIMASK=0，SEVONPEND=1，且BASEPRI能够掩蔽 | +  + | + |
| PRIMASK=1，SEVONPEND=0，且BASEPRI不能掩蔽  PRIMASK=1，SEVONPEND=0，且BASEPRI能够掩蔽 | + |  |
| PRIMASK=1，SEVONPEND=1，且BASEPRI不能掩蔽  PRIMASK=1，SEVONPEND=1，且BASEPRI能够掩蔽 | +  + |  |

译者小结：

* 1. 只有PRIMASK=0时，才执行ISR
  2. 对于WFE，只要SEVONPEND=1，则不管何时发生了什么中断，都一定会唤醒处理器
  3. 不管PRIMASK为何值，只要优先级高到BASEPRI不能掩蔽，就将唤醒处理器
  4. 当PRIMASK=0时，它不会对唤醒产生影响

CM3还有一个“自动睡眠”的机制：SleepOnExit——它可以被编程为从中断服务例程返回后立即睡眠。这样一来，处理器的所有工作就只是响应中断了，其它时间都在睡眠。在真实的应用程序里，通常只有在程序很简单的电池供电设备中，才会用此功能。如欲使用此特性，需要把系统控制寄存器中的SLEEPONEXIT位置位。如图13.3所示。

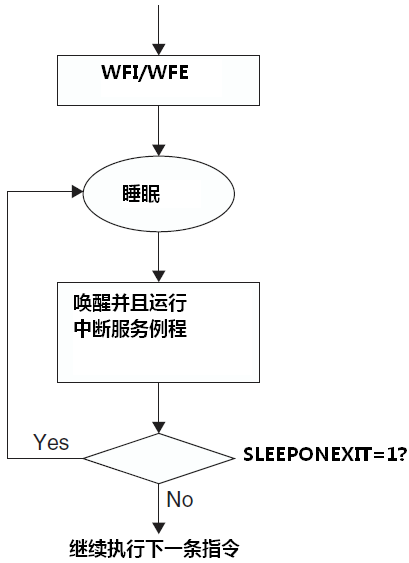


图13.3 SleepOnExit功能演示

## 13.3 多处理机通信

最让人意想不到的就是CM3竟然还支持简单的多核功能！它上面有一个用于处理机之间同步任务的简单通信接口。处理机有一个名为TXEV（Transmit Event）的输出信号，用于发送信号给其它处理机；还有一个名为RXEV（Receive Event）的输入信号，以接收从其它处理机发来的信号。对于一个双核系统来说，事件通信的信号的连接可以如图13.4所示：

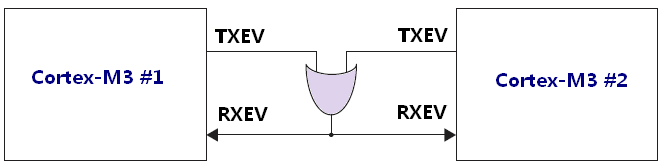


图13.4 双核处理系统间的事件信号连接

如上一小节所述，当处理机因为WFE而睡眠时，可以由外部事件——即RXEV唤醒。CM3提供了SEV指令（Send EVent）。当执行该指令时，当事处理机就会在TXEV上发送一个脉冲，从而可以唤醒另外的睡眠中的处理机，从而实现同步，如图13.5所示。

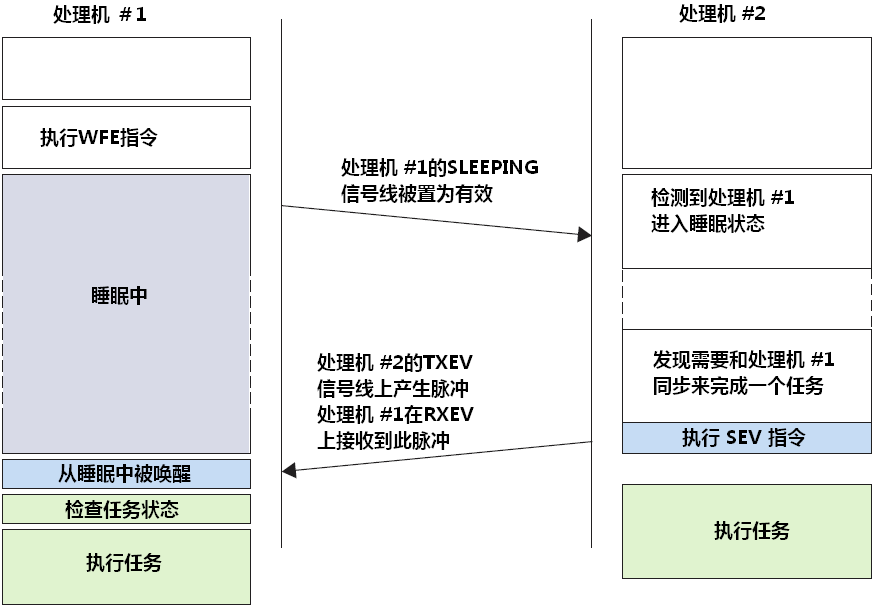


图13.5 双核之间使用事件信号来做同步任务

在使用WFE同步任务时，要明白处理器也以被其它事件唤醒，比如中断和调试事件。所以在被唤醒时，需要先检查是不是由同步事件信号唤醒的。使用WFE同步任务的流程如图13.6所示。

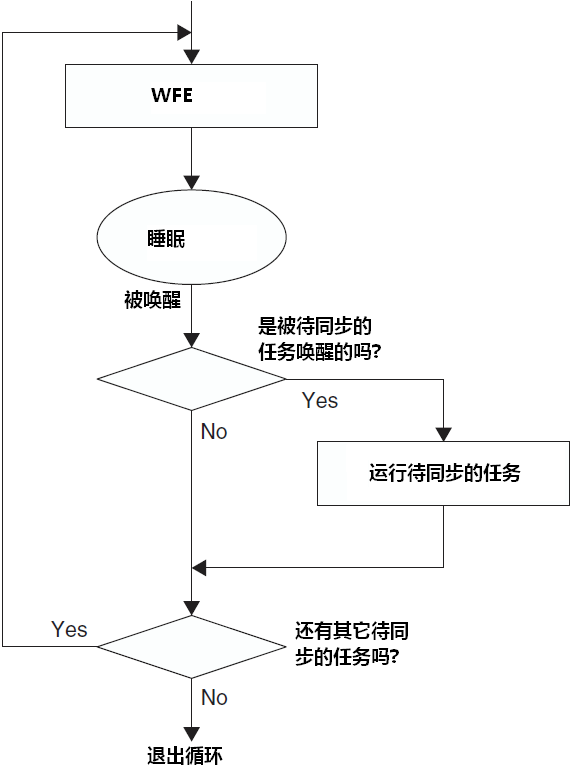


图13.6 使用WFE同步任务模式图

通过使用WFE，我们可以让两个处理机同步地配合完成一个任务（也可能会有少量时钟周期的时差，这取决于器件的实现方式）。上图演示的是两台处理机的情况，事实上处理机的数目并没有限制，但无论如何都必须有一个担当“主机”，用于发送同步事件。

当执行WFE时，它首先检视本地事件锁存器。如果锁存器的值为零，则使内核睡眠；如果发现锁住了先前的事件信号，则清零锁存器，并且取消此次睡眠，继续执行下一条指令。早先发生的异常、执行的SEV指令都可以置位锁存器。所以要注意，如果曾经执行过SEV，则紧挨着的WFE不会使处理器睡眠，只是清除了锁存的值，处理器依然继续执行。

### 13.3.1 多机同步的深入讨论

事实上，同步问题远远要复杂得多。如果只是按图13.6那样单单使用WFE，只能应付小儿科的任务同步问题。在复杂的应用程序中，为正确地同步任务还需要附加的代码。正如上文所提到的，处理器也以被其它事件唤醒，比如中断和调试事件。因此，内部的事件寄存器的当前状态常常是未知的，故而不能保证在执行WFE指令后就一定能进入睡眠。事实上，WFE常常在循环中使用（用于降低系统的功耗），循环体中的代码检查状态，以判定需要同步的任务是否应该在WFE后执行。

这种用法最典型示例就是多核系统中的信号量。在典型的情况下，需要一个系统级的互斥访问监视器，在它的辅助下使用互斥访问指令来实现自旋锁（spin lock，熟悉Linux的读者请笑一个），以一轮一轮地尝试锁住共享的存储器或外设。自旋锁设施由RTOS提供，通常由汇编语言写成。RTOS提供类似spin\_lock()和spin\_unlock()的函数，而任务则可以使用这两个函数来锁住所需的共享资源。

常规的自旋锁代码如下所示：

spin\_lock ; 获取自旋锁的汇编示例代码，r0指向自旋锁变量

MOVS r2, #1 ; r2待会要写入自旋锁变量，表示资源已锁

spin\_lock\_loop

LDREX r1, [r0]

CMP r1, #0

BNE spin\_lock\_loop ; 资源已被锁住，需重试

STREX r1, r2, [r0] ; 使用STREX指令尝试设置Lock\_Variable为1

CMP r1, #0 ; 检查STREX指令的返回值

BNE spin\_lock\_loop ; STREX指令没有成功执行，重试

DMB ; 执行数据存储器隔离，以确保数据已落实到物理内存中。

BX LR ; 返回

在共享资源使用完毕后，需要释放自旋锁：

spin\_unlock ; 释放自旋锁的汇编示例代码，r0指向自旋锁变量

MOVS r1, #0

**~~DMB~~ ;原文DMB在这里使用，疑似不妥**

STR r1, [r0] ; Clear lock

DMB ; 执行数据存储器隔离，以确保数据已落实到物理内存中。

BX LR ; 返回

自旋锁的副作用，就是会当（获取锁的）处理机空闲时使（等待锁的）处理机白白空转，浪费能源。因此，我们在上例的自旋锁中加入WFE/SEV来解决这个问题：一方面，在尝试上锁的函数中，一旦发现已上锁就执行WFE；而在释放锁的函数中（在另一个处理机中执行此函数），释放后执行SEV指令以唤醒所有尝试上锁的处理机。

spin\_lock\_with\_WFE ; 使用WFE配合获取自旋锁的汇编示例代码，r0指向自旋锁变量

MOVS r2, #1 ; r2待会要写入自旋锁变量，表示资源已锁

spin\_lock\_loop

LDREX r1, [r0]

CBNZ r1, lock\_is\_set ; 如果r1!=0，则表示已上锁

STREX r1, r2, [r0] ; 使用STREX指令尝试设置Lock\_Variable为1

CMP r1, #0 ; 检查STREX指令的返回值

BNE spin\_lock\_loop ; STREX指令没有成功执行，重试

DMB ; 执行数据存储器隔离，以确保数据已落实到物理内存中。

BX LR ; 返回

lock\_is\_set

WFE ; 资源已锁。等待使用资源的处理机释放锁后使用SEV发出信号

B spin\_lock\_loop ; 被唤醒，不管是不是被SEV唤醒的，先去尝试上锁

在共享资源使用完毕后，需要释放自旋锁：

在解开自旋锁的函数中，需要使用SEV指令来唤醒其它所有需要该锁的处理机。

spin\_unlock\_with\_SEV ; 释放自旋锁的汇编示例代码，r0指向自旋锁变量

MOVS r1, #0

**~~DMB~~ ;原文DMB在这里使用，疑似不妥**

STR r1, [r0] ; Clear lock

DMB ; 执行数据存储器隔离，以确保数据已落实到物理内存中。

SEV

BX LR ; Return

通过在信号量代码中配合使用事件通信接口，就可以在使用自旋锁尝试获取共享资源时消除不必要的功耗。类似的技术也可以用于创建消息队列等其它任务同步设施。

译者添加：自旋锁不是谁想用谁就能用的，必须分场合。如果是同一个处理机内的多个任务需要某共享资源，且在其它处理机上没有需要此资源的任务，就不得使用带WFE的自旋锁，因为在执行WFE后，该处理机已经睡眠了，无法再执行其它指令，更不要说调度其它任务让它调用spin\_unlock\_with\_SEV了。此时又没有“外力”，因此就很可能要“长眠”了！进一步地，单机场合下不得使用自旋锁。因为自旋锁可能导致死循环：优先级最高的任务如果使用自旋锁未果，则在按优先级调度的RTOS中，如果没有反优先级倒转机制，就会使最高优先级的任务永远死循环，CPU利用率100%，却再也执行不了其它任务！

在大多数CM3系统中，会只使用一个内核。此时，常常是把RXEV脚拉低，或者连接到其它可以产生事件的外设上。

Cortex-M3 r2p0修订版新增

请注意：当使能了SLEEPONEXIT特性时，CM3在异常退出后不经过执行WFI/WFE就会进入睡眠模式。因此当需要执行睡眠时，在正常的使用场合下，要在WFI/WFE指令（得到执行）之前就使能SLEEPONEXIT。

在Cortex-M3修订版2（已于2008年出品）中，又添加了新的特性以支持低功耗。从软件的立场上来看，WFI/WFE依然故我。但在硬件上，修订版的深度睡眠模式则睡得更深：允许送往处理器内核的时钟信号停止。那这么一来怎样唤醒内核呢？原来，修订版2的内核新增了一个独立的单元，称作“唤醒中断控制器”。有了它，处理器内核可以在进入掉电模式时，把处理器状态信息存储到特殊的逻辑小室（cells）中，从而更狠地降低空闲时的功耗。

要使用新的掉电模式，还需要一个外部电源管理单元来配合，后者用于控制上电序列和掉电序列。该单元由芯片供应商提供，在使用掉电特性前可能还要编程它，因此需要参考芯片供应商提供的技术文档。关于掉电特性，还有两点要注意的。首先，是它会关掉送往SysTick定时器的时钟。第2，当连接了一个调试器时，为了使它能够正常地访问调试寄存器，会自动除能这个掉电特性。

## 13.4 自复位控制

CM3允许由软件触发复位序列，用于特殊的调试或维护目的（没事别玩啊）。在CM3中，有两种方法可以执行自我复位。第一种方法，是通过置位NVIC中应用程序中断与复位控制寄存器(AIRCR)的VECTRESET位（位偏移：0），如下所示：

LDR R0, =0xE000ED0C ; NVIC AIRCR address

LDR R1, =0x05FA000**1** ; 置位 VECTRESET位，前面的0x05FA是访问钥匙

STR R1, [R0] ; 触发复位序列

deadloop

B deadloop ; 该死循环保证后面的指令不可能被执行到

这种复位的作用范围覆盖了整个CM3处理器中，除了调试逻辑之外的所有角落，但是它不会影响到CM3处理器外部的任何电路，所以单片机上的各片上外设和其它电路都不受影响。

复位的第二种方法，是置位同一个寄存器中的SYSRESETREQ位。这种复位则会波及整个芯片上的电路：它会使CM3处理器把送往系统复位发生器的请求线置为有效。但是系统复位发生器不是CM3的一部分，而是由芯片厂商实现，因此不同的芯片对此复位的响应也不同。因此，读者需要认真参阅该芯片规格书，明白当发生片内复位时，各外设和功能模块都会回到什么样的初始状态，或者有哪些功能模块不受影响（比如，STM32系列的芯片有后备存储区，该区就被特殊对待）。SYSRESETREQ的使用如下面代码段所演示：

LDR R0, =0xE000ED0C ; NVIC AIRCR address

LDR R1, =0x05FA000**4** ; 置位 SYSRESETREQ，前面的0x05FA是访问钥匙

STR R1, [R0] ; 触发复位序列

deadloop

B deadloop ; 该死循环保证后面的指令不可能被执行到

大多数情况下，复位发生器在响应SYSRESETREQ时，它也会同时把CM3处理器的系统复位信号(SYSRESETn)置为有效。通常，SYSRESETREQ不应复位调试逻辑。

这里有一个要注意的问题：从SYSRESETREQ被置为有效，到复位发生器执行复位命令，往往会有一个延时。在此延时期间，处理器仍然可以响应中断请求。但我们的本意往往是要让此次执行到此为止，不要再做任何其它事情了。所以，最好在发出复位请求前，先把FAULTMASK置位。