**1 常见芯片的电源管理**

**1.1 STM32F4系列**

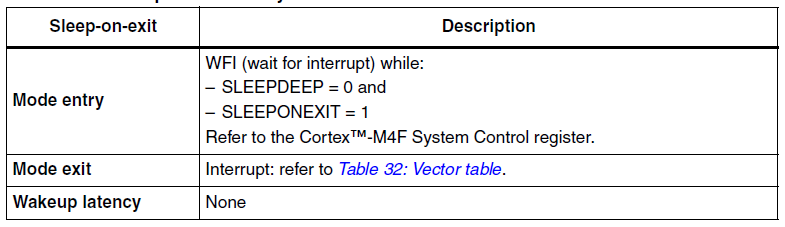
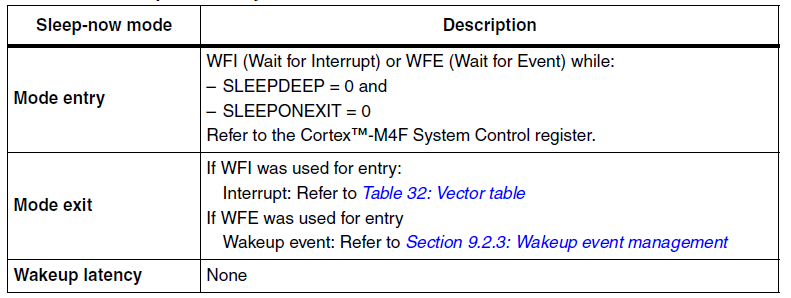
该系列芯片提供了三种低功耗模式：

* 休眠模式
* STOP模式
* Standby模式

**1.1.1 Sleep模式**

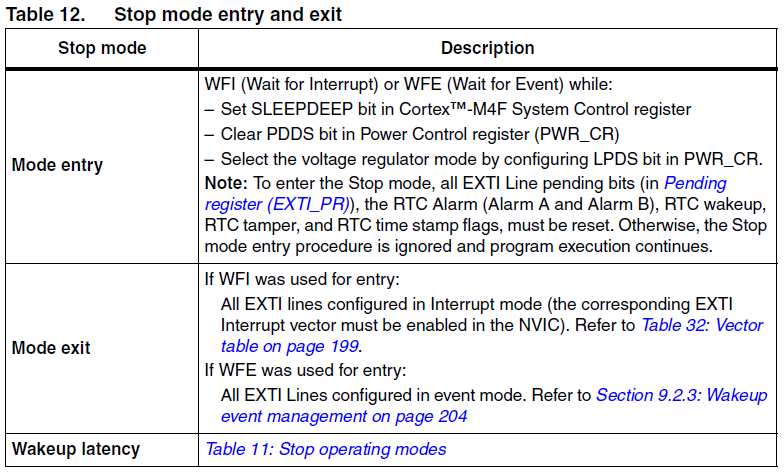
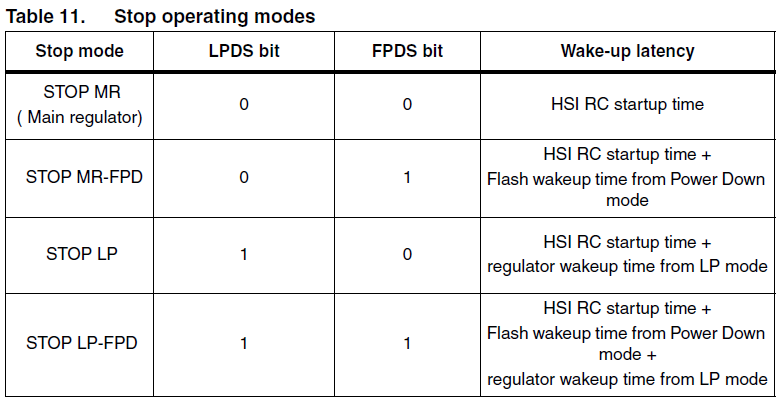
内核（包括MPU）停止工作，外设（定时器、时钟等）仍然在工作，退出睡眠模式后程序从停止处继续执行，该模式类似于人在睡觉，大脑已经停止思考，但心脏脉搏仍在跳动。

sleep模式是通过执行WFI(wait for interrupt)/WFE(Wait for Event)命令来实现的。根据SLEEPONEXIT又分成了Sleep-now和Sleep-on-exit两种情况。



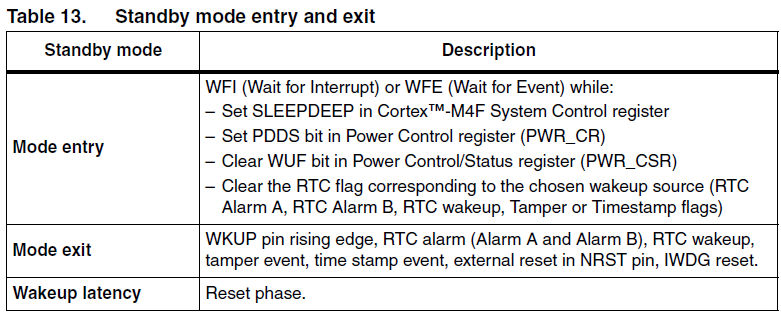
**1.1.2 STOP模式**

内核包括1.2V区域的所有时钟停止运作，PLLs, HSI和HSE RC oscillators被失能，计数器和寄存器的内容不变，并且在该模式下电压和内部FLASH，IWDG，RTC也可以通过软件事先的配置进入低功耗模式，当内部flash进入低功耗模式时，那么在退出该模式后也会有相应的启动延时，故对实时性要求高的系统对是否试FLASH进入低功耗模式应慎重，退出该模式后程序从停止处开始执行。



**1.1.3 Standby模式**

功耗最低状态，在STOP状态的基础上，电压调节器失能，除backup区域，SRAM以及寄存器的值全部丢失，故退出该状态后，系统类似于复位。



**1.2 IMX7D系列**

IMX7D系统芯片提供了三种低功耗模式：

* Wait Mode
* Stop Mode
* Deep Sleep Mode（只针对A7）

LPCG (Low Power Clock Gating),用于软件控制或自动控制SOC上面所有模块的时钟。

进入和退出,Only for M4　Control：

|  |  |  |
| --- | --- | --- |
| Mode | 进入 | 退出 |
| Wait Mode | GPC\_SLPCR[EN\_M4\_FASTWUP\_WAIT\_MODE]  GPC\_LPCR\_M4[CPU\_CLK\_ON\_LPM]are set to ON/OFF  GPC\_LPCR\_M4[LPM]are set to WAIT or STOP | Interrupts |
| Stop Mode | GPC\_SLPCR[EN\_M4\_FASTWUP\_STOP\_MODE]  GPC\_LPCR\_M4[CPU\_CLK\_ON\_LPM]are set to ON/OFF  GPC\_LPCR\_M4[LPM]are set to WAIT or STOP |
| Deep Sleep  Mode | Both A7 and M4 are STOP mode  Both GPC\_SLPCR[EN\_A7\_FASTWUP\_STOP\_MODE] and  GPC\_SLPCR[EN\_M4\_FASTWUP\_STOP\_MODE] are not set.  • GPC\_SLPCR[EN\_DSM] is set |

**2 FreeRTOS低功耗研究**

**2.1 操作系统中的任务调度**

在一个操作系统的实现中，实现上下文的切换有两种情况：

* 执行一个系统调用
* 系统滴答定时器（SYSTICK）中断，轮转需要。

**2.1.1 SVC系统调用**

SVC（系统服务调用，亦简称系统调用）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。因此，当用户程序想要控制特定的硬件时，它就会产生一个SVC 异常，然后操作系统提供的SVC 异常服务例程得到执行，它再调用相关的操作系统函数，后者完成用户程序请求的服务。

这种“提出要求——得到满足”的方式，很好、很强大、很方便、很灵活、很能可持续发展。

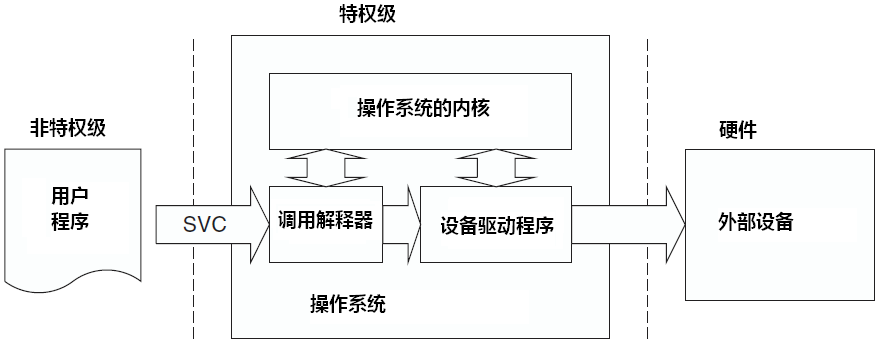
首先，它使用户程序从控制硬件的繁文缛节中解脱出来，而是由OS 负责控制具体的硬件。

第二，OS 的代码可以经过充分的测试，从而能使系统更加健壮和可靠。

第三，它使用户程序无需在特权级下执行，用户程序无需承担因误操作而瘫痪整个系统的风险。

第四，通过SVC 的机制，还让用户程序变得与硬件无关，因此在开发应用程序时无需了解硬件的操作细节，从而简化了开发的难度和繁琐度，并且使应用程序跨硬件平台移植成为可能。

开发应用程序唯一需要知道的就是操作系统提供的应用编程接口（API），并且了解各个请求代号和参数表，然后就可以使用SVC 来提出要求了（事实上，为使用方便，操作系统往往会提供一层封皮，以使系统调用的形式看起来和普通的函数调用一致。各封皮函数会正确使用SVC指令来执行系统调用——译者注）。其实，严格地讲，操作硬件的工作是由设备驱动程序完成的，只是对应用程序来说，它们也是操作系统的一部分。



SVC 异常通过执行”SVC”指令来产生。该指令需要一个立即数，充当系统调用代号。SVC异常服务例程稍后会提取出此代号，从而解释本次调用的具体要求，再调用相应的服务函数。

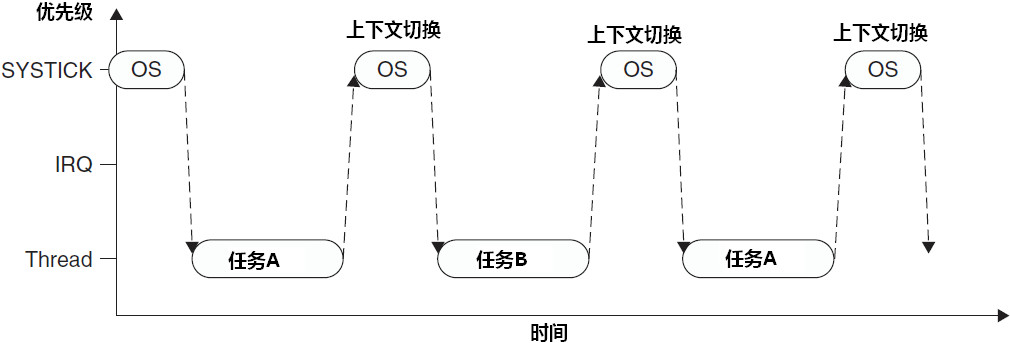
例如，

SVC 0x3 ; 调用3 号系统服务

在SVC 服务例程执行后，上次执行的SVC 指令地址可以根据自动入栈的返回地址计算出。找到了SVC 指令后，就可以读取该SVC 指令的机器码，从机器码中萃取出立即数，就获知了请求执行的功能代号。如果用户程序使用的是PSP，服务例程还需要先执行MRS Rn,PSP 指令来获取应用程序的堆栈指针。

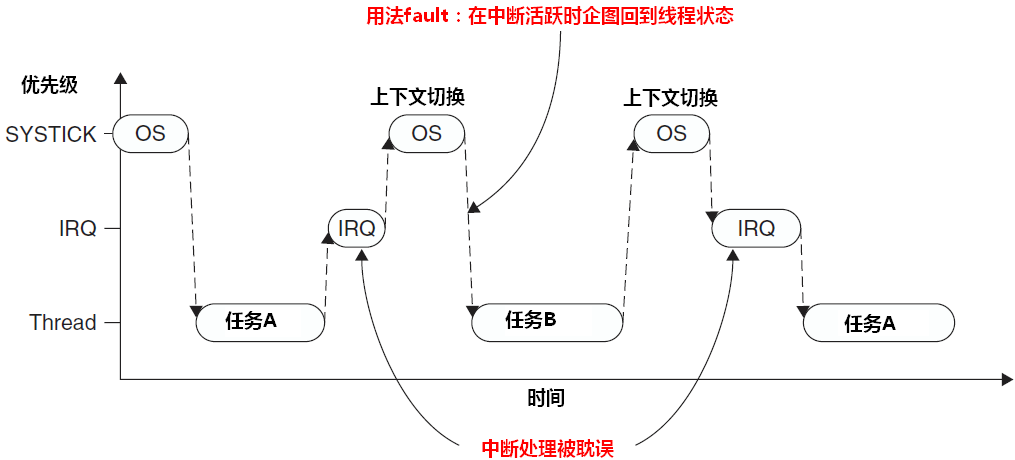
**2.1.2 基于SYSTICK任务切换**

我们以一个简单的例子来辅助理解。在一个操作系统调用中，任务A和任务B处在就绪状态。两个任务将通过SYSTICK简单轮换的模型如下：



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，在CM3 中也是严禁没商量——如果OS 在某中断活跃时尝试切入线程模

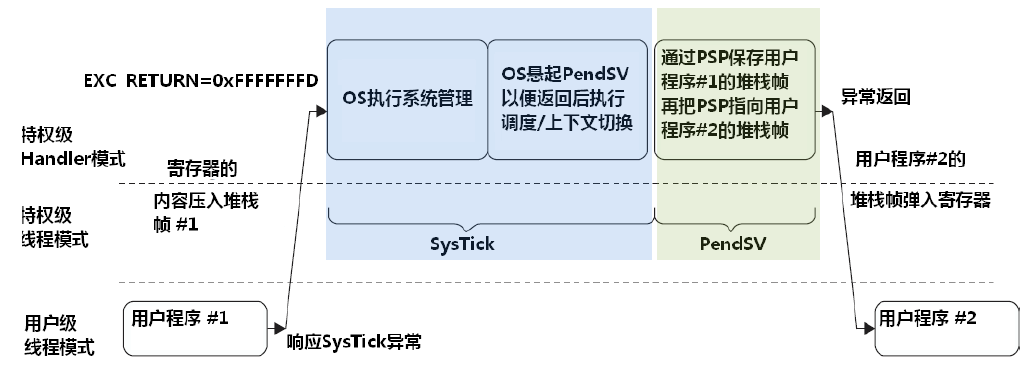
式，将触犯用法fault 异常。整个过程如下图所示：



为解决此问题，早期的OS 大多会检测当前是否有中断在活跃中，只有没有任何中断需要响应时，才执行上下文切换（切换期间无法响应中断）。然而，这种方法的弊端在于，它可以把任务切换动作拖延很久（因为如果抢占了IRQ，则本次SysTick 在执行后不得作上下文切换，只能等待下一次SysTick 异常），尤其是当某中断源的频率和SysTick 异常的频率比较接近时，会发生“共振”。

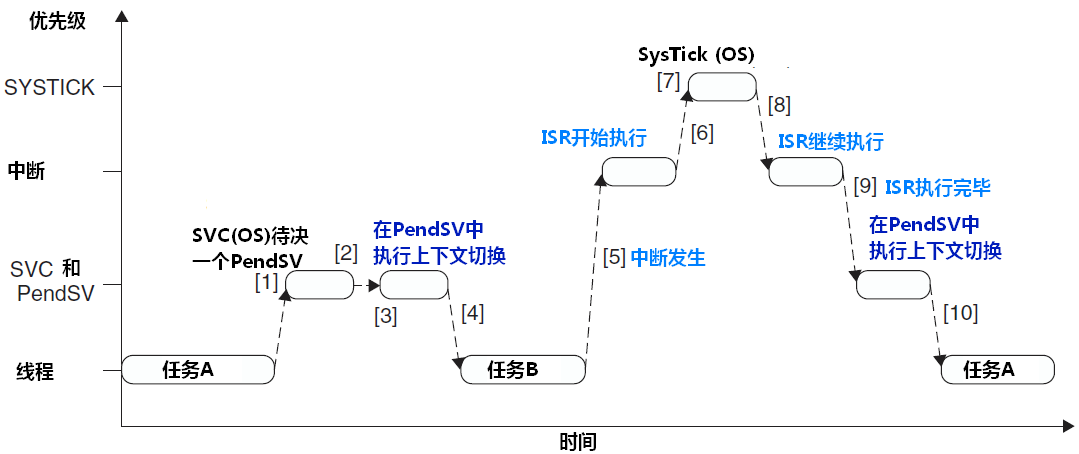
Cortext M中引进了PendSV的概念，很好地解决了这个问题。。PendSV 异常会自动延迟上下文切换的请求，直到其它的ISR 都完成了处理后才放行。为实现这个机制，需要把PendSV 编程为最低优先级的异常。如果OS检测到某IRQ正在活动并且被SysTick抢占，它将悬起一个PendSV异常，以便缓期执行上下文切换。

我们来看看一个SysTick 异常推动简单轮转调度模式图。



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，它周期性把执行点转入操作系统，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。

假设存在任务A和任务B，使用PendSV控制的上下文切换如下图：



个中事件的流水账记录如下：

[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。

在FreeRTOS运行时是不断在不同的任务间进行切换，而驱动这一调度过程是通过系统tick来驱动的，即每产生一次系统tick则检查一下当前正在运行的任务的环境判断是否需要切换任务，即调度，如果需要，则触发PendSV，通过在PendSV中断调用vTaskSwitchContext()函数来实现任务的调度。

在FreeRTOS/source/portable/rvds/arm\_cm4f/Port.c中，有SYSTICK的异常实现函数xPortSysTickHandler（）。

|  |
| --- |
| void xPortSysTickHandler( void )  {  /\* The SysTick runs at the lowest interrupt priority, so when this interrupt  executes all interrupts must be unmasked. There is therefore no need to  save and then restore the interrupt mask value as its value is already  known. \*/  ( void ) portSET\_INTERRUPT\_MASK\_FROM\_ISR();  {  /\* Increment the RTOS tick. \*/  if( xTaskIncrementTick() != pdFALSE )  {  /\* A context switch is required. Context switching is performed in  the PendSV interrupt. Pend the PendSV interrupt. \*/  portNVIC\_INT\_CTRL\_REG = portNVIC\_PENDSVSET\_BIT;  }  }  portCLEAR\_INTERRUPT\_MASK\_FROM\_ISR( 0 );  }  /\*-----------------------------------------------------------\*/ |

另外在同一个文件还有PendSV具体实现函数xPortPendSVHandler()。通过注释可以很好了解整个过程。这里不具体展开咯。

|  |
| --- |
| \_\_asm void xPortPendSVHandler( void )  {  extern uxCriticalNesting;  extern pxCurrentTCB;  extern vTaskSwitchContext;  PRESERVE8  mrs r0, psp  isb  /\* Get the location of the current TCB. \*/  ldr r3, =pxCurrentTCB  ldr r2, [r3]  /\* Is the task using the FPU context? If so, push high vfp registers. \*/  tst r14, #0x10  it eq  vstmdbeq r0!, {s16-s31}  /\* Save the core registers. \*/  stmdb r0!, {r4-r11, r14}  /\* Save the new top of stack into the first member of the TCB. \*/  str r0, [r2]  stmdb sp!, {r3}  mov r0, #configMAX\_SYSCALL\_INTERRUPT\_PRIORITY  msr basepri, r0  **bl vTaskSwitchContext**  mov r0, #0  msr basepri, r0  ldmia sp!, {r3}  /\* The first item in pxCurrentTCB is the task top of stack. \*/  ldr r1, [r3]  ldr r0, [r1]  /\* Pop the core registers. \*/  ldmia r0!, {r4-r11, r14}  /\* Is the task using the FPU context? If so, pop the high vfp registers  too. \*/  tst r14, #0x10  it eq  vldmiaeq r0!, {s16-s31}  msr psp, r0  isb  #ifdef WORKAROUND\_PMU\_CM001 /\* XMC4000 specific errata \*/  #if WORKAROUND\_PMU\_CM001 == 1  push { r14 }  pop { pc }  nop  #endif  #endif  bx r14  nop  nop  } |

**2.2 FreeRTOS的低功耗优化**

**2.2.1 TickLess方案**

**2.2.1.1 Tickless 基本原理**

一句话概括就是实时改变滴答时钟的中断频率。

上小节提到，操作系统都有一个滴答时钟，以一定的频率产生定时器中断，freeRTOS使用systick时钟产生滴答时钟，在操作系统的运行过程中，大部分时间可能在运行空闲任务，也就是这个时候无事可干，那么如果我们能在无事可干的时间让系统进入睡眠或者停止模式，那么系统的功耗是不是就会降下去呢，答案是肯定的，那么我们每次进入空闲任务之后就让系统休眠，直至产生滴答时钟中断被唤醒，在中断服务函数中查看是不是有需要运行的任务运行，但是我们一般的滴答时钟周期设置为10ms，也就是说不管有没有任务运行，必须10ms醒过来检测一次。

但是freeRTOS不满足于这样，他想到了一个更“省电”的方法：原理如下，当我们开启freeRTOS的低功耗模式后，系统会在空闲模式中计算出下一次某一个应用任务开始运行的时间，那么freeRTOS就设置SYSTICKD的重装值，使之恰好在这个时间点产生滴答时钟中断，然后“睡觉”，直至中断发生推出睡眠，从而进行任务的切换，这样系统就只需要在必须醒来的时候醒来进行任务切换，而不必每10ms就醒过来检测一次，那么是不是可以美美的睡上一觉呢  
 问题：如果需要睡眠的时间太长，超过了systick所能产生的最大时钟周期怎么办？  
 答：freeRTOS提出了解决方案，那就是使用普通定时器作为滴答时钟，通过改变其分频数来达到systick不能达到的的时钟周期

**2.2.1.2 Tickless具体实现**

FreeRTOS提供了Tickless的具体实现。默认情况下，是不开启的。需要通过在FreeRTOSConfig.h中将configUSE\_TICKLESS\_IDLE宏开关设置成非0。

这部分是在Idle Task：portTASK\_FUNCTION( prvIdleTask, pvParameters )函数中实现的。源码在FreeRTOS/source/Tasks.c中。

|  |
| --- |
| static portTASK\_FUNCTION( prvIdleTask, pvParameters )  {  /\* Stop warnings. \*/  ( void ) pvParameters;  for( ;; )  {  ......  /\* This conditional compilation should use inequality to 0, not equality  to 1. This is to ensure portSUPPRESS\_TICKS\_AND\_SLEEP() is called when  user defined low power mode implementations require  configUSE\_TICKLESS\_IDLE to be set to a value other than 1. \*/  #if ( configUSE\_TICKLESS\_IDLE != 0 )  {  TickType\_t xExpectedIdleTime;  /\* It is not desirable to suspend then resume the scheduler on  each iteration of the idle task. Therefore, a preliminary  test of the expected idle time is performed without the  scheduler suspended. The result here is not necessarily  valid. \*/  // prvGetExpectedIdleTime()用于计算从现在到下一个任务运行的时间，也就是可以睡大觉的时间  xExpectedIdleTime = prvGetExpectedIdleTime();  if( xExpectedIdleTime >= configEXPECTED\_IDLE\_TIME\_BEFORE\_SLEEP )  {  vTaskSuspendAll();  {  /\* Now the scheduler is suspended, the expected idle  time can be sampled again, and this time its value can  be used. \*/  configASSERT( xNextTaskUnblockTime >= xTickCount );  xExpectedIdleTime = prvGetExpectedIdleTime();  if( xExpectedIdleTime >= configEXPECTED\_IDLE\_TIME\_BEFORE\_SLEEP )  {  // portSUPPRESS\_TICKS\_AND\_SLEEP（）通过xExpectedIdleTime参数调整systick的重装值然后开始“睡大觉”等待下一次唤醒，这个函数本质上就是对于SYSTICK寄存器的操作。  traceLOW\_POWER\_IDLE\_BEGIN();  portSUPPRESS\_TICKS\_AND\_SLEEP( xExpectedIdleTime );  traceLOW\_POWER\_IDLE\_END();  }  else  {  mtCOVERAGE\_TEST\_MARKER();  }  }  ( void ) xTaskResumeAll();  }  else  {  mtCOVERAGE\_TEST\_MARKER();  }  }  #endif /\* configUSE\_TICKLESS\_IDLE \*/  }  } |

仔细研读函数，你还会发现freeRTOS还提供了两个接口函数，用于更加精准的功耗控制：

configPRE\_SLEEP\_PROCESSING(xModifiableIdleTime);

configPOST\_SLEEP\_PROCESSING( xExpectedIdleTime );

这两个函数的位置分别位于\_wfi指令前面和后面，用户可以在进入睡眠之前通过configPRE\_SLEEP\_PROCESSING( xModifiableIdleTime )函数对控制器的外设进行配置，以达到最优的省电效果，退出睡眠模式后通过configPOST\_SLEEP\_PROCESSING（xModifiableIdleTime ）对之前配置的外设进行还原。

我们回头来看下这个函数prvGetExpectedIdleTime（），为什么通过prvGetExpectedIdleTime（）可以取到下一个Task唤醒的时间呢？

其实是因为我们在调用vTaskDelay（）或是vTaskDelayUtil（）的时候，会去创建一个等待Task的列表，这个列表是根据等待的时间来进行组织的，参考prvAddCurrentTaskToDelayedList( const TickType\_t xTimeToWake )这个函数的实现，同时会保存一个全局变量xNextTaskUnblockTime用来标识最近一个唤醒时间间隔。

|  |
| --- |
| #if ( configUSE\_TICKLESS\_IDLE != 0 )  static TickType\_t prvGetExpectedIdleTime( void )  {  TickType\_t xReturn;  if( pxCurrentTCB->uxPriority > tskIDLE\_PRIORITY )  {  xReturn = 0;  }  else if( listCURRENT\_LIST\_LENGTH( &( pxReadyTasksLists[ tskIDLE\_PRIORITY ] ) ) > 1 )  {  /\* There are other idle priority tasks in the ready state. If  time slicing is used then the very next tick interrupt must be  processed. \*/  xReturn = 0;  }  else  {  xReturn = **xNextTaskUnblockTime** - xTickCount;  }  return xReturn;  }  #endif /\* configUSE\_TICKLESS\_IDLE \*/ |

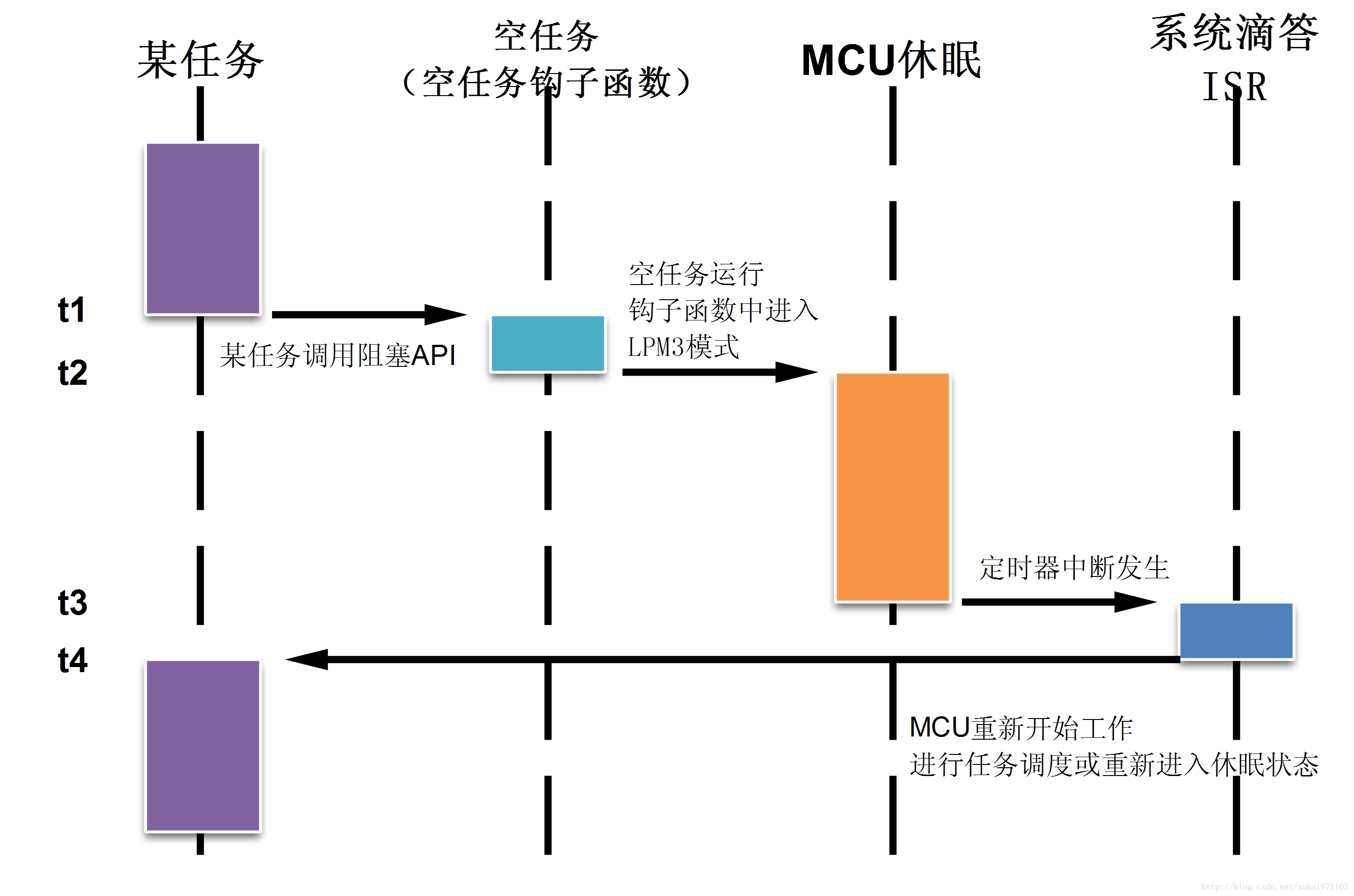
**2.2.1.3 省电效果对比**

在F429开发板上面，我验证了一下freeRTOS低功耗的省电效果，我的工程是建立两个任务，一个任务是让led1灯以500ms亮灭，另一个任务是让led2灯以1000ms亮灭  
 首先是关闭低功耗特性，使程序运行，发现电流在125mA~134mA,注意，单纯的看这个数据没有意义，我们需要和低功耗模式系下进行比较，因为开发板上面还有其他的芯片，功耗也是不可忽略的  
 接下来通过相关配置开启freeRTOS睡眠模式，发现电流在99mA~111mA之间，比正常工作模式减少了25mA左右  
 置位SLEEPDEEP位，使freeRTOS进入stop模式，发现电流在88mA~96mA之间 比正常工作模式减少了40mA左右

**2.2.2 IdleHook方案**

还有一种方案就是在vApplicationIdleHook（）中直接控制处理器进入到低功耗模式。

**2.2.2.1 基本原理**



例如某任务在t1时刻调用阻塞API，例如vTaskDelay，此时任务交出CPU使用权由OS进行任务调度。t2时刻，由于没有其他就绪任务，OS运行空任务，在空任务的最后进入空任务钩子函数，在空任务钩子函数中MCU进入低功耗模式，此时可进入LPM3模式。t3时刻MCU进入低功耗模式之后，MCU停止工作。t4时刻，由于系统滴答时钟中断服务函数中，MCU重新处于活跃状态，并且通过指令退出低功耗模式，此时OS任务调度器再次工作，若此时任务再次处于就绪状态便重新运行该任务。

以上便是如何利用OS实现低功耗的基本方法，但是t3时刻和t4时刻是有反复的。例如，系统滴答时钟ISR发生之后，OS任务调度器中并没有就绪的任务，只能再次运行空任务，通过空任务再次进入低功耗模式，如此反复直到某任务就绪便执行任务代码。

**2.2.2.2代码实现**

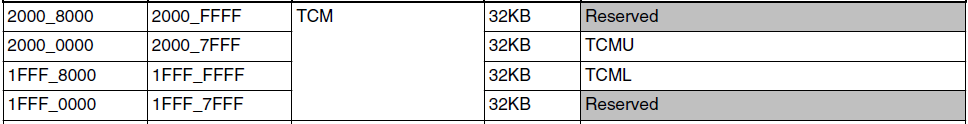
在FreeRTOS中，若需要打开空任务钩子函数，需要在FreeRTOSConfig.h中定义configUSE\_IDLE\_HOOK

#define configUSE\_IDLE\_HOOK 1

在创建任务之前，必须对一些环境、变量进行初始化：

|  |
| --- |
| /\*  \* Low Power Management initialization  \*/  **void LPM\_MCORE\_Init(GPC\_Type \* base)**  {  // Init GPC（**General Power Controller**）  GPC\_Init(base);  // Copy critical function to TCM Space  prepare\_ram\_wfi();  // Init the wakeup interrupt link list  lpm\_init\_wakeup\_interrupt\_list();  // GPT4 Patch, see function implementation for details  GPT\_Patch\_Prepare();  }  /\*!  \* @brief Check if "RAM\_Wfi" is located inside TCM, if not so, copy it to TCM  \*/  **void prepare\_ram\_wfi()**  {  if (((uint32\_t)RAM\_Wfi < 0x1FFF8000) || ((uint32\_t)RAM\_Wfi > 0x20007FFF))  // If no, copy the RAM\_Wfi function to TCM.  runInRAM = (void(\*)(void))RAM\_Function\_Copy((uint8\_t \*)RAM\_Wfi,  (uint8\_t \*)RAM\_Wfi\_End);  else  // If yes, just assign RAM\_Wfi to runInRAM pointer.  runInRAM = RAM\_Wfi;  }  /\*  \* Tool function to copy function area [func\_start, func\_end] to M4 data area  \* which is TCM in 7Dual Project  \*/  **static uint8\_t\* RAM\_Function\_Copy(uint8\_t\* func\_start, uint8\_t\* func\_end)**  {  uint8\_t\* ram\_code\_ptr;  uint32\_t ram\_function\_start;  ram\_function\_start = (uint32\_t)func\_start & ~0x3;  ram\_code\_ptr = pvPortMalloc((uint8\_t\*)func\_end - (uint8\_t\*)ram\_function\_start);  memcpy(ram\_code\_ptr, (uint8\_t\*)ram\_function\_start, (uint32\_t)func\_end -  (uint32\_t)ram\_function\_start);  ram\_code\_ptr = (uint8\_t \*)((uint32\_t)ram\_code\_ptr | ((uint32\_t)func\_start & 0x3));  return ram\_code\_ptr;  } |

这里面需要特别关注函数prepare\_ram\_wfi()。因为，M4的代码需要从TCM中进行load，而在M4的Memory Map关系中，TCM的映射如下：



所以就通过RAM\_Function\_Copy（）完成拷贝。

熟悉FreeRTOS的同学都知道，如果#define configUSE\_IDLE\_HOOK 1，系统进入到Idle模式之后，函数vApplicationIdleHook（）就会被调用。

|  |
| --- |
| /\*!  \* @brief Custom function to be run in idletask  \*/  void vApplicationIdleHook(void)  {  /\* Waiting for Wake up event. \*/  LPM\_MCORE\_WaitForInt();  } |

LPM\_MCORE\_WaitForInt（）中调用WFI（）指令，进入low power mode；

接下来看看low power mode的设定：

|  |
| --- |
| void LPM\_MCORE\_SetPowerStatus(GPC\_Type \* base, LPM\_POWER\_STATUS\_M4 m4\_next\_lpm)  {  uint32\_t next\_lpm = GPC\_LPCR\_M4\_LPM0(0);  switch (m4\_next\_lpm) {  case LPM\_M4\_STATE\_RUN:  next\_lpm = GPC\_LPCR\_M4\_LPM0(0);  break;  case LPM\_M4\_STATE\_WAIT:  next\_lpm = GPC\_LPCR\_M4\_LPM0(1);  break;  case LPM\_M4\_STATE\_STOP:  next\_lpm = GPC\_LPCR\_M4\_LPM0(2);  break;  default:  break;  }  /\*  \* Patch, let GPC-M4 observe the GPR0 interrupt for a period as long  \* as 5 32KHz clock cycle before set it to a Low power status  \*/  if (m4\_next\_lpm != LPM\_M4\_STATE\_RUN)  {  uint32\_t i;  LPM\_MCORE\_RegisterWakeupInterrupt(GPC, GPT4\_IRQn, GPC\_IRQ\_WAKEUP\_ENABLE);  for (i=0; i!=GPC\_SYNC\_DELAY\_CNT; i++)  \_\_NOP();  LPM\_MCORE\_RegisterWakeupInterrupt(GPC, GPT4\_IRQn, GPC\_IRQ\_WAKEUP\_DISABLE);  }  GPC\_SetM4NextLPM(base, next\_lpm);  /\*change lpm state variable\*/  m4\_lpm\_state = m4\_next\_lpm;  } |

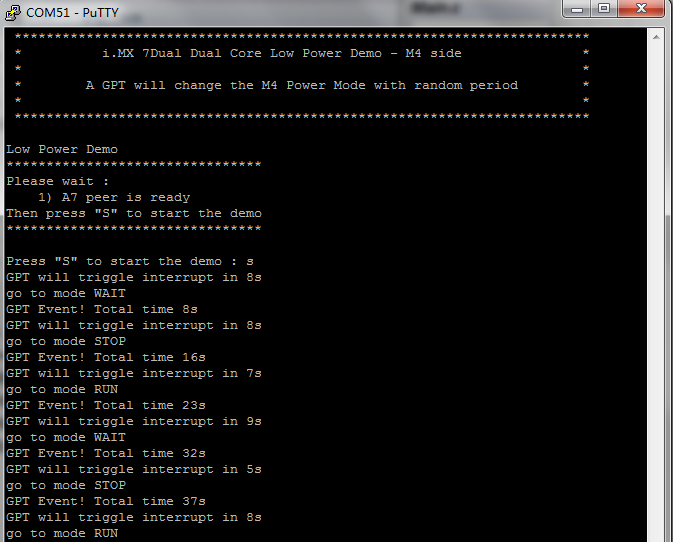
**2.2.2.3 Task设计**

在Task的设计中，首先读取当前的LPM的状态，并由LPM\_MCORE\_SetPowerStatus（）设定下一个状态。

同时，维护一个timer，用于固定时间将M4核从LPM状态中唤醒。

具体实例请参考：FreeRTOS\_BSP\_1.0.1\_iMX7D/examples/imx7d\_sdb\_m4/demo\_apps/low\_power\_imx7d/rand\_wfi。

例程的执行结果如下：



参考文献

【1】基于STM32F429的freeRTOS的低功耗学习，http://bbs.eeworld.com.cn/thread-474584-1-1.html

【2】Cortext M3权威指南

【3】Cortext M3 TechM

【4】如何在FreeRTOS下实现低功耗——MSP430F5438平台，

http://blog.csdn.net/xukai871105/article/details/12751435