RTOS 低功耗设计原理及实现

Tickless Idle Mode （FreeRTOS 下的实现）

一.前言

目前，越来越多的嵌入式产品在开发中使用 RTOS 作为软件平台，同时，开发中对低功耗的要求也越来越高，这篇文档会讨论一下如何在 RTOS 中处理微控制器的低功耗特性。

应用中使用的 RTOS 一般采用基于时间片轮转的抢占式任务调度机制，

一般的低功耗设计思路如下：

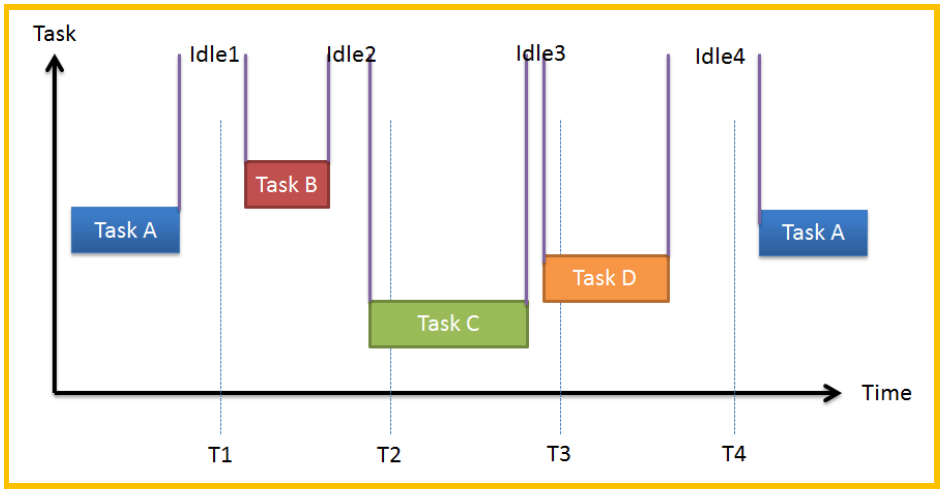
1. 当 Idle 任务运行时，进入低功耗模式；
2. 在适当的条件下，通过中断或者外部事件唤醒 MCU。

但是，从第二点可以看出，每次当 OS 系统定时器产生中断时，也会将 MCU 从低功耗模式中唤醒，而频繁的进入低功耗模式/从低功耗模式中唤醒会使得 MCU 无法进入深度睡眠，对低功耗设计而言也是不合理的。

在 FreeRTOS 中给出了一种低功耗设计模式 ——Tickless Idle Mode，这个方法可以让 MCU 更长时间的处于低功耗模式。

二. Tickless Idle Mode 的原理及实现

1. 情景分析



上图是任务调度示意图，横轴是时间轴，T1，T2，T3，T4 是 RTOS 的时间片基准，有四个任务分别是 TaskA,B,C,D，

Task A: 周期性任务

Task B: 周期性任务

Task C: 突发性任务

Task D: 周期性任务

从图中可以看出在四个任务进行调度之间，会有四次空闲期间（此时 RTOS 会调度 Idle 任务运行，软件设计的目标应该是尽可能使 MCU 在 Idle 任务运行时处于低功耗模式）。

Idle1: Idle 任务运行期间，会产生一次系统时钟滴答，此时会唤醒 MCU，唤醒后 MCU 又会进入低功耗模式，这次唤醒是无意义的。期望使 MCU 在 Idle1 期间一直处于低功耗模式，因此适当调整系统定时器中断使得 T1 时不触发系统时钟中断，中断触发点设置为 Task B 到来时；

Idle2：Task C 在系统滴答到达前唤醒 MCU（外部事件），MCU 可以在 Idle2 中可以一直处于低功耗模式；

Idle3: 与 Idle2 情况相同，但 Idle3 时间很短，如果这个时间很短，那么进入低功耗模式的意义并不大，因此在进入低功耗模式时软件应该添加策略；

Idle4: 与 Idle1 情况相同。

2. Tickless Idle Mode 的软件设计原理

Tickless Idle Mode 的设计思想在于尽可能得在 MCU 空闲时使其进入低功耗模式。从上述情景中可以看出软件设计需要解决的问题有：

a. 合理的进入低功耗模式（避免频繁使 MCU 在低功耗模式和运行模式下进行不必要的切换）；
   RTOS 的系统时钟源于硬件的某个周期性定时器（Cortex-M 系列内核多数采用 SysTick），RTOS 的任务调度器可以预期到下一个周期性任务（或者定时器任务）的触发时间，如上文所述，调整系统时钟定时器中断触发时间，可以避免 RTOS 进入不必要的时间中断，从而更长的时间停留在低功耗模式中，此时 RTOS 的时钟不再是周期的而是动态的（在原有的时钟基准时将不再产生中断，即 Tickless）；
b. 当 MCU 被唤醒时，通过某种方式提供为系统时钟提供补偿。
   MCU 可能被两种情况所唤醒，动态调整过的系统时钟中断或者突发性的外部事件，无论是哪一种情况，都可以通过运行在低功耗模式下的某种定时器来计算出 MCU 处于低功耗模式下的时间，在 MCU 唤醒后对系统时间进行软件补偿；
c. 软件实现时，要根据具体的应用情景和 MCU 低功耗特性来处理问题。
   尤其是 MCU 的低功耗特性，不同 MCU 处于不同的低功耗模式下所能使用的外设（主要是定时器）是不同的，RTOS 的系统时钟可以进行适当的调整。

3. Tickless Idle Mode 的实现

这里以 STM32F407 系列的 MCU 为例，首先需要明确的是 MCU 的低功耗模式，F407 有 3 种低功耗模式，Sleep, Stop, Standby，在 RTOS 平台时，SRAM 和寄存器的数据不应丢失，此外需要一个定时器为 RTOS 提供系统时钟，这里选择 Sleep 模式下进行实现。

Table 24. Low-power mode summary

Mode name	Entry	Wakeup	Effect on 1.2 V domain clocks	Effect on V_{DD} domain clocks	Voltage regulator
Sleep (Sleep now or Sleep-on-exit)	WFI	Any interrupt	CPU CLK OFF no effect on other clocks or analog clock sources	None	ON
	WFE	Wakeup event			
Stop	PDDS and LPDS bits + SLEEPDEEP bit + WFI or WFE	Any EXTI line (configured in the EXTI registers, internal and external lines)	All 1.2 V domain clocks OFF	HSI and HSE oscillators OFF	ON or in low- power mode (depends on *PWR power control register (PWR_CR) for STM32F405xx/07xx and STM32F415xx/17xx* and *PWR power control register (PWR_CR) for STM32F42xxx and STM32F43xxx*)
Standby	PDDS bit + SLEEPDEEP bit + WFI or WFE	WKUP pin rising edge, RTC alarm (Alarm A or Alarm B), RTC Wakeup event, RTC tamper events, RTC time stamp event, external reset in NRST pin, IWDG reset	All 1.2 V domain clocks OFF	HSI and HSE oscillators OFF	OFF

- 使能

```
#define configUSE_TICKLESS_IDLE        1
```

- 空闲任务（RTOS 空闲时自动调用）

```
/* Idle 任务 */
void prvIdleTask( void *pvParameters )
{
	for( ; ; )
	{
		...

		#if ( configUSE_TICKLESS_IDLE != 0 )
		{
			TickType_t xExpectedIdleTime;

			/* 用户策略以决定是否需要进入 Tickless Mode */
			xExpectedIdleTime = prvGetExpectedIdleTime();

			if( xExpectedIdleTime >= configEXPECTED_IDLE_TIME_BEFORE_SLEEP )
			{
				vTaskSuspendAll();		// 挂起调度器
				{
					configASSERT( xNextTaskUnblockTime >= xTickCount );
					xExpectedIdleTime = prvGetExpectedIdleTime();

					if( xExpectedIdleTime >=
					    configEXPECTED_IDLE_TIME_BEFORE_SLEEP )
					{
					 /* 用户函数接口 */
					 /* 1. 进入低功耗模式和如何退出低功耗模式  */
					 /* 2. 系统时间补偿 */
					  portSUPPRESS_TICKS_AND_SLEEP( xExpectedIdleTime );
					}
				}
				(void) xTaskResumeAll();	// 恢复调度器
			}
		}
```

```
            #endif        /* configUSE_TICKLESS_IDLE */
            ...
        }
}
```

- 低功耗模式处理（根据 MCU 的低功耗模式编写代码，代码有点长……）

```
void vPortSuppressTicksAndSleep( portTickType xExpectedIdleTime )
{
      unsigned long ulReloadValue, ulCompleteTickPeriods,
     ulCompletedSysTickDecrements;
      portTickType xModifiableIdleTime;

      /* 最长睡眠时间不可以超过定时器的最大定时值 */
      /* 通过调整定时器的时间基准可以获得更理想的最大定时值 */
      if( xExpectedIdleTime > xMaximumPossibleSuppressedTicks )
      {
            xExpectedIdleTime = xMaximumPossibleSuppressedTicks;
      }

      /* 停止 SysTick */
      portNVIC_SYSTICK_CTRL_REG = portNVIC_SYSTICK_CLK_BIT |
                                  portNVIC_SYSTICK_INT_BIT;

      /* 计算唤醒时的系统时间，用于唤醒后的系统时间补偿 */
      ulReloadValue = portNVIC_SYSTICK_CURRENT_VALUE_REG +
                     ( ulTimerCountsForOneTick * ( xExpectedIdleTime - 1UL ) );
      if( ulReloadValue > ulStoppedTimerCompensation )
      {
            ulReloadValue -= ulStoppedTimerCompensation;
      }

      __disable_interrupt();

      /* 确认下是否可以进入低功耗模式 */
      if( eTaskConfirmSleepModeStatus() == eAbortSleep )
      {
            /* 不可以，重新启动系统定时器 */
            portNVIC_SYSTICK_LOAD_REG = portNVIC_SYSTICK_CURRENT_VALUE_REG;

            portNVIC_SYSTICK_CTRL_REG = portNVIC_SYSTICK_CLK_BIT |
                                        portNVIC_SYSTICK_INT_BIT |
                                        portNVIC_SYSTICK_ENABLE_BIT;

            portNVIC_SYSTICK_LOAD_REG = ulTimerCountsForOneTick - 1UL;

            __enable_interrupt();
      }
      else
      {
           /* 可以进入低功耗模式 */

           /* 保存时间补偿，重启系统定时器 */
            portNVIC_SYSTICK_LOAD_REG = ulReloadValue;
           portNVIC_SYSTICK_CURRENT_VALUE_REG = 0UL;
```

```
portNVIC_SYSTICK_CTRL_REG = portNVIC_SYSTICK_CLK_BIT |
                            portNVIC_SYSTICK_INT_BIT |
                            portNVIC_SYSTICK_ENABLE_BIT;

/* 进入低功耗模式，可以通过 configPRE_SLEEP_PROCESSING 函数进行低功耗模式下
    时钟及外设的配置*/
xModifiableIdleTime = xExpectedIdleTime;
configPRE_SLEEP_PROCESSING( xModifiableIdleTime );
if( xModifiableIdleTime > 0 )
{
      __DSB();
      __WFI();
      __ISB();
}

 /* 退出低功耗模式 */

configPOST_SLEEP_PROCESSING( xExpectedIdleTime );

portNVIC_SYSTICK_CTRL_REG = portNVIC_SYSTICK_CLK_BIT |
                            portNVIC_SYSTICK_INT_BIT;

__disable_interrupt()
__enable_interrupt();


/*唤醒有两种情况：系统定时器或者外部事件（中断）*/
if((portNVIC_SYSTICK_CTRL_REG & portNVIC_SYSTICK_COUNT_FLAG_BIT) != 0)
{
      /* 系统定时器唤醒，时间补偿 */
      unsigned long ulCalculatedLoadValue;


      ulCalculatedLoadValue = ( ulTimerCountsForOneTick - 1UL ) -
            ( ulReloadValue - portNVIC_SYSTICK_CURRENT_VALUE_REG );

      if( ( ulCalculatedLoadValue < ulStoppedTimerCompensation ) ||
          ( ulCalculatedLoadValue > ulTimerCountsForOneTick ) )
      {
            ulCalculatedLoadValue = (ulTimerCountsForOneTick - 1UL);
      }

      portNVIC_SYSTICK_LOAD_REG = ulCalculatedLoadValue;

      ulCompleteTickPeriods = xExpectedIdleTime - 1UL;
}
else
{

      /* 外部事件（中断）唤醒 */
      ulCompletedSysTickDecrements = ( xExpectedIdleTime *
      ulTimerCountsForOneTick ) - portNVIC_SYSTICK_CURRENT_VALUE_REG;

      ulCompleteTickPeriods = ulCompletedSysTickDecrements /
                              ulTimerCountsForOneTick;
```

```
                portNVIC_SYSTICK_LOAD_REG = ( ( ulCompleteTickPeriods + 1 ) *
                      ulTimerCountsForOneTick ) - ulCompletedSysTickDecrements;
            }

            /* 重启 Systick，调整系统定时器中断为正常值 */
            portNVIC_SYSTICK_CURRENT_VALUE_REG = 0UL;
            portENTER_CRITICAL();
            {
                portNVIC_SYSTICK_CTRL_REG = portNVIC_SYSTICK_CLK_BIT |
                                            portNVIC_SYSTICK_INT_BIT |
                                            portNVIC_SYSTICK_ENABLE_BIT;
                vTaskStepTick( ulCompleteTickPeriods );
                portNVIC_SYSTICK_LOAD_REG = ulTimerCountsForOneTick - 1UL;
            }
            portEXIT_CRITICAL();
        }
}
```

4. 写在最后的话

STM32 家族中拥有不同的系列，特别是专为低功耗应用设计的 L 系列，为其设计 RTOS 低功耗特性实现时可以有更多的实现方式（例，某种模式下内核停止运行，此时可以使用外部定时器或者 RTC 来代替 Systick 作为系统定时器）。