STM32Cube高效开发教程（高级篇）

第3章 FreeRTOS的中断管理

王维波
中国石油大学（华东）控制科学与工程学院

STM32Cube高效开发教程（高级篇）

作者：王维波，鄢志丹，王钊

人民邮电出版社

2022年2月出版

如果有读者需要本书课件的PPT版本用于备课，可以给作者发邮件免费获取，并可加入专门的教学和技术交流QQ群

邮箱：wangwb@upc.edu.cn

3.1 FreeRTOS与中断

3.2 任务与中断服务函数

3.3 任务和中断程序设计示例

中断是MCU的硬件特性。STM32使用4个位设置优先级分组策略，优先级数字越低，优先级别越高。每个中断有一个ISR

FreeRTOS在SysTick中断里进行任务调度申请，在PendSV中断里进行任务调度。启用FreeRTOS后，会自动设置NVIC

NVIC | Code generation

Priority Group: 4 bits for pre-emption priority 0 bits for subpriority

☐ Sort by Premption Priority and Sub Priority

Search: Search (Crtl+F)

☐ Show only enabled interrupts

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority	Uses FreeRTOS functions
Non maskable interrupt	☑	0	0	☐
Hard fault interrupt	☑	0	0	☐
Memory management fault	☑	0	0	☐
Pre-fetch fault, memory access fault	☑	0	0	☐
Undefined instruction or illegal state	☑	0	0	☐
System service call via SWI instruction	☑	0	0	☐
Debug monitor	☑	0	0	☐
Pendable request for system service	☑	15	0	☑
System tick timer	☑	15	0	☑
PVD interrupt through EXTI line 16	☐	5	0	☑
Flash global interrupt	☐	5	0	☑
RCC global interrupt	☐	5	0	☑
ADC1, ADC2 and ADC3 global interrupts	☐	5	0	☑
Time base: TIM6 global interrupt, DAC1 and DAC2 ...	☑	0	0	☐
FPU global interrupt	☐	5	0	☑

中断优先级分组会自动设置为4位全部用于抢占优先级，所以抢占优先级编号是0到15。对应于文件FreeRTOSConfig.h中的参数configPRIO_BITS，

```
#define configPRIO_BITS        4
```

设置FreeRTOS的Config参数时，有2个与中断相关的参数

```
Interrupt nesting behaviour configuration
        LIBRARY_LOWEST_INTERRUPT_PRIORITY          15
        LIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY     5
```

- configLIBRARY_LOWEST_INTERRUPT_PRIORITY，表示中断的最低优先级数值。因为中断分组策略是4位全用于抢占优先级，所以这个数值为15

Interrupt nesting behaviour configuration

LIBRARY_LOWEST_INTERRUPT_PRIORITY	15
LIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY	5

- configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY，表示FreeRTOS可管理的最高优先级，默认数值为5。只有在中断优先级数值等于或大于5的中断ISR函数里才可以调用FreeRTOS的中断安全API函数。

configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY绝不允许设置为0，绝对不要在大于此优先级的中断ISR函数里调用FreeRTOS的API函数，即使带“FromISR”的中断安全函数也不可以

在文件FreeRTOSConfig.h中还定义了一个参数，是用于写入寄存器的表示最低优先级（15=0x0F）的数值，定义如下：

```
#define configKERNEL_INTERRUPT_PRIORITY
    ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 -
    configPRIO_BITS) )
```

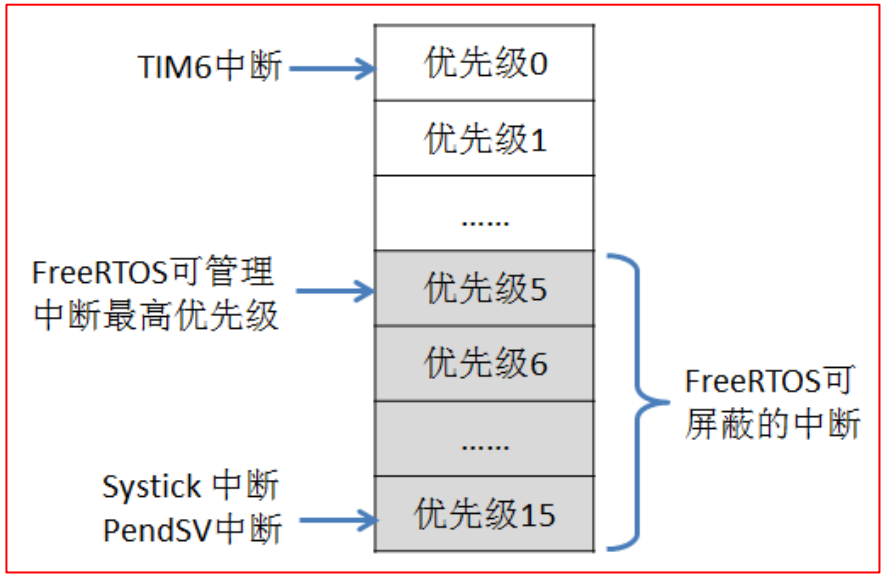
这个宏的值是0xF0，这个宏再用于定义PendSV和SysTick的中断优先级，在文件port.c中的定义如下：

```
#define portNVIC_PENDSV_PRI
            ((( uint32_t) configKERNEL_INTERRUPT_PRIORITY ) << 16UL)
#define portNVIC_SYSTICK_PRI    ((( uint32_t)
            configKERNEL_INTERRUPT_PRIORITY ) << 24UL)
```

这是用于写入寄存器的值，其数值与中断优先级的表示有关。直观的就是图3-1中的设置，PendSV优先级为15，SysTick的优先级为15

configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY=5

- HAL基础时钟（示例中是TIM6）的中断优先级为0
- PendSV和SysTick的中断优先级为15，所以，只有在没有其他中断需要处理的情况下才会发生任务切换



- 中断分为2组，高优先级的一组中断不受FreeRTOS的管理，称为FreeRTOS不可屏蔽中断，低优先级的一组是FreeRTOS可屏蔽中断，可以用函数taskDISABLE_INTERRUPTS()屏蔽这些低优先级中断

3.2 任务与中断服务函数

3.2.1 任务与ISR函数的关系

3.2.2 中断屏蔽和临界代码段

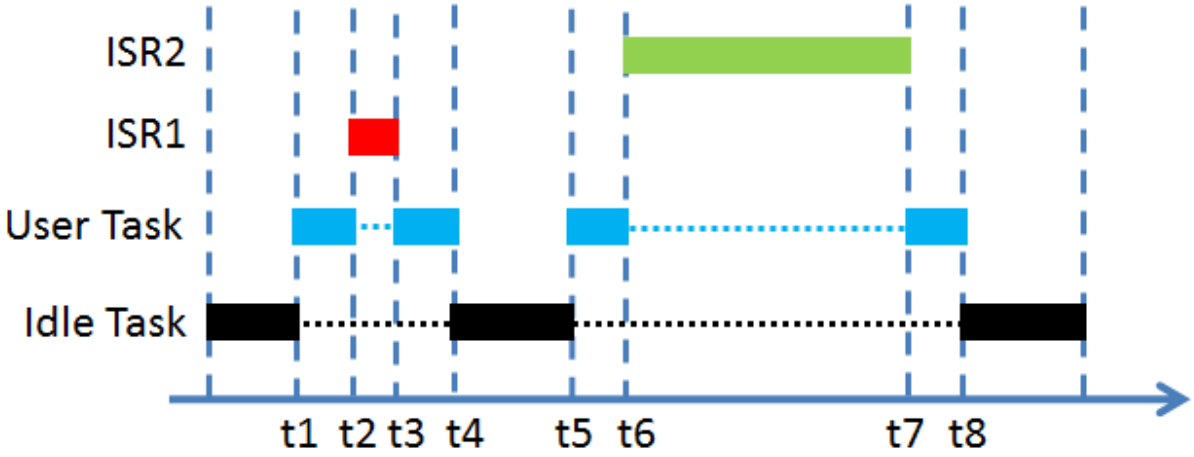
3.2.3 在ISR函数中使用FreeRTOS API函数

3.2.4 中断及其ISR函数设计原则

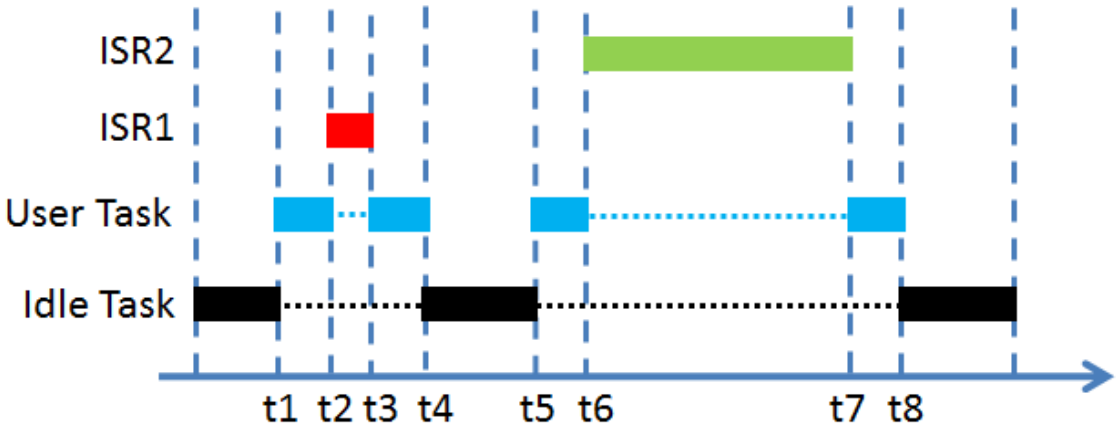
3.2.1 任务与ISR函数的关系

- 中断是MCU的硬件特性，由硬件事件或软件信号引起中断，运行哪个ISR是由硬件决定的。中断的优先级数字越低表示优先级别越高，所以中断的最高优先级为0
- FreeRTOS的任务是一个纯软件的概念，与硬件系统无关。任务的优先级是编程者在软件中赋予的，任务的优先级数字越低表示优先级越低，所以任务的最低优先级为0。FreeRTOS的任务调度算法决定哪个任务处于运行状态
- 任务只有在没有ISR运行的时候才能运行，即使优先级最低的中断也可以抢占高优先级的任务的执行，而任务不能抢占ISR的运行【这一句需要重点解释】

任务函数与中断的ISR运行时序图



- 在t2时刻发生了一个中断1，不管User Task的任务优先级有多高，ISR1函数都会抢占CPU。ISR1执行完成后，User Task才可以继续执行。
- 在t6时刻发生了中断2，ISR2函数同样抢占了CPU。但是ISR2占用CPU的时间比较长，导致User Task执行时间变长，从软件运行响应来说，可能就是软件响应变得迟钝了。



由于ISR函数执行时就无法执行任务函数，所以，如果一个ISR函数执行的时间比较长，任务函数无法及时执行，FreeRTOS也无法进行任务调度，就会导致软件响应变迟钝。

在实际的软件设计中，一般要尽量简化ISR函数的功能，使其尽量少占用CPU的时间。ISR函数一般只负责数据采集后收发，将数据处理的任务放到任务函数里去执行。

3.2.2 中断屏蔽和临界代码段

一个任务函数在执行的时候，可能会被其他高优先级的任务抢占CPU，也可能被任何一个中断的ISR函数抢占CPU。在某些时候，任务的某段代码可能很关键，需要连续执行完，不希望被其他任务或中断打断，这种程序段称为临界段（Critical Section）

```
taskDISABLE_INTERRUPTS()        //屏蔽MCU的部分中断
taskENABLE_INTERRUPTS()         //接触中断屏蔽

taskENTER_CRITICAL()            //开始临界代码段，可以嵌套定义
taskEXIT_CRITICAL()             //结束临界代码段

taskENTER_CRITICAL_FROM_ISR()
taskEXIT_CRITICAL_FROM_ISR( x )
```

3.2.3 在ISR函数中使用FreeRTOS API函数

在中断的ISR里调用普通的API函数可能会存在问题，例如调用一个API函数使一个任务进入阻塞状态，因为ISR执行的时候是不能进行任务调度的。

为此，FreeRTOS的API函数分为两个版本：一个称为“任务级”，也就是普通名称的API函数；另一个称为“中断级”，即带后缀“FromISR”的函数或带后缀“FROM_ISR”的宏函数，中断级API函数也被称为中断安全API函数。

注意，在ISR中绝对不能使用任务级API函数，但是在任务函数中可以使用中断级API函数。而且，在FreeRTOS不能管理的高优先级中断的ISR里，连中断级API函数也不能用。

5.2.4 中断及其ISR程序设计原则

中断的优先级和ISR函数程序设计应该遵循如下的原则：

- 中断分为FreeRTOS不可屏蔽中断和可屏蔽中断，要根据中断的重要性和功能为其设置合适的中断优先级。
- ISR函数的代码应该尽量简短，将处理功能延迟到任务里去实现。
- 在可屏蔽中断的ISR函数里能调用中断级的FreeRTOS API函数，绝对不能调用普通的FreeRTOS API函数。在不可屏蔽中断的ISR函数里，不能调用任何的FreeRTOS API函数。

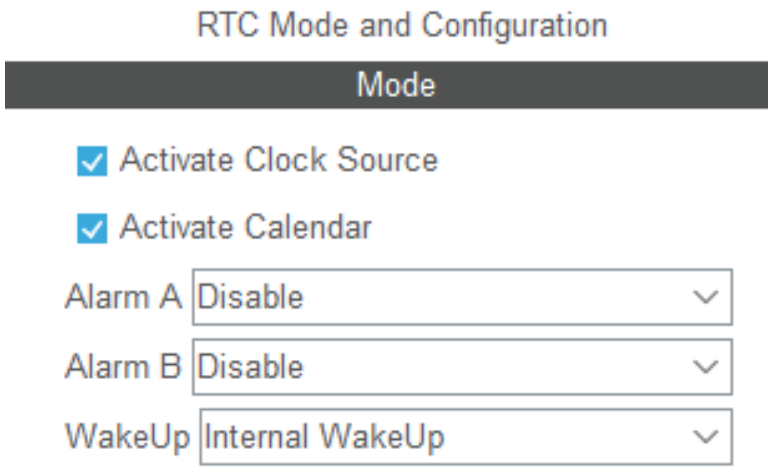
3.3 任务和中断程序设计示例

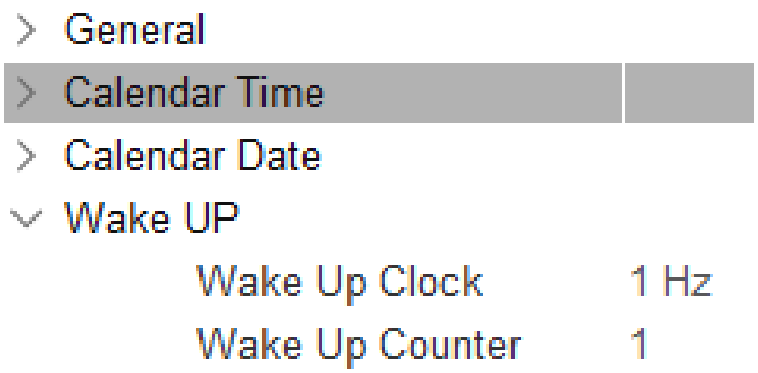
3.3.1 示例功能和CubeMX项目设置

使用RTC的周期唤醒中断，在此中断里读取RTC的当前时间并在LCD上显示，在FreeRTOS中设计一个任务Task_LED1。通过各种参数设置和稍微修改代码，测试和验证任务与重点的特点

（1）RTC的设置

唤醒周期设置为1秒

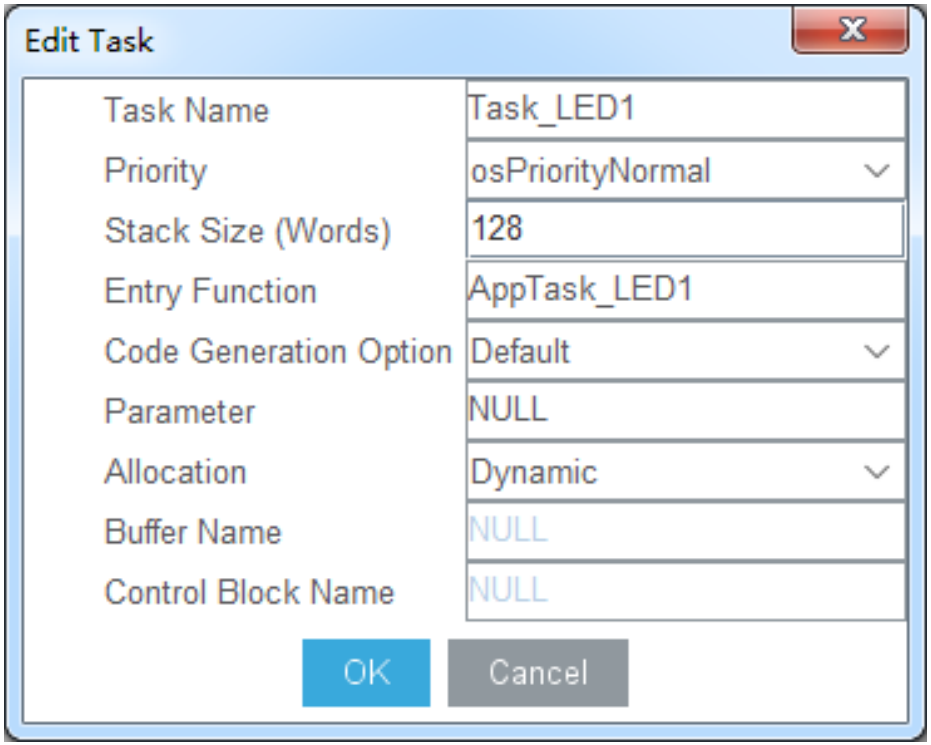




（2）FreeRTOS设置

创建一个任务Task_LED1，任务的参数设置如图。

任务Task_LED1用于使LED1闪烁，所以还需将PF9引脚设置为GPIO_Output。



（3）中断设置

RTC的周期唤醒中断是用户程序需要用到的中断，其优先级是可以设置的。先将其优先级设置为1，也就是FreeRTOS不可屏蔽的中断

NVIC Mode and Configuration

Configuration

NVIC | Code generation

Priority Group: 4 bits for pre-emption priority 0 bits for subpriority

☐ Sort by Premption Priority and Sub Priority

Search: Search (Crtl+F)

☑ Show only enabled interrupts ☑ Force DMA channels Interrupts

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority	Uses FreeRTOS functions
Non maskable interrupt	☑	0	0	☐
Hard fault interrupt	☑	0	0	☐
Memory management fault	☑	5	0	☑
Pre-fetch fault, memory access fault	☑	0	0	☐
Undefined instruction or illegal state	☑	5	0	☑
System service call via SWI instruction	☑	0	0	☐
Debug monitor	☑	0	0	☐
Pendable request for system service	☑	15	0	☑
System tick timer	☑	15	0	☑
RTC wake-up interrupt through EXTI line 22	☑	1	0	☐
Time base: TIM6 global interrupt, DAC1 and DAC2 ...	☑	0	0	☐

3.3.2 基本功能代码

1. 主程序

```
int main(void)
{
        HAL_Init();                 //HAL初始化
        SystemClock_Config();           //系统时钟配置
         /* Initialize all configured peripherals */
        MX_GPIO_Init();
        MX_FSMC_Init();
        MX_RTC_Init();          //RTC初始化

        /* USER CODE BEGIN 2 */
        TFTLCD_Init();    //LCD 初始化
        LCD_ShowStr(10, 10, (uint8_t *)"Demo3_1:Task and ISR");
        /* USER CODE END 2 */

      osKernelInitialize();                     //RTOS内核初始化
      MX_FREERTOS_Init();                       //FreeRTOS对象初始化
      osKernelStart();                                  //启动RTOS内核
      while (1)
      {     }
}
```

2. RTC唤醒中断的处理

在文件rtc.c中重新实现RTC周期唤醒事件的回调函数

```
void HAL_RTCEx_WakeUpTimerEventCallback(RTC_HandleTypeDef *hrtc)
{
        RTC_TimeTypeDef sTime;
        RTC_DateTypeDef sDate;
        if (HAL_RTC_GetTime(hrtc, &sTime,  RTC_FORMAT_BIN) == HAL_OK)
        {
            HAL_RTC_GetDate(hrtc, &sDate,  RTC_FORMAT_BIN);
/* 调用HAL_RTC_GetTime()之后必须调用HAL_RTC_GetDate()以解锁数据，才能连续更
        新日期和时间 */
            uint16_t  xPos=30, yPos=50;
        //显示 时间  mm:ss
            LCD_ShowUintX0(xPos,yPos,sTime.Minutes,2);          //2位数字显示，前端补0
            LCD_ShowChar(LCD_CurX, yPos, ':', 0);
            LCD_ShowUintX0(LCD_CurX,yPos,sTime.Seconds,2);  //2位数字显示，前端补0
        }
        //HAL_Delay(1000);          //在后面测试用到时取消注释
}
```

3. Task_LED1的任务函数实现

任务Task_LED1的功能非常简单，就是使LED1闪烁，循环周期是200ms

```
/*  任务Task_LED1的任务函数    */
void AppTask_LED1(void *argument)
{
  /* USER CODE BEGIN AppTask_LED1 */
  for(;;)
  {
        HAL_GPIO_TogglePin(GPIOF, GPIO_PIN_9);    //LED1闪烁
        vTaskDelay(pdMS_TO_TICKS(200));           //进入阻塞状态
  }
  /* USER CODE END AppTask_LED1 */
}
```

运行时可以发现RTC周期中断和任务的程序都能按期望运行，每隔2秒在LCD上刷新当前时间，LED1也是规律性地快速闪烁。

3.3.3 各种特性的测试

1. 中断的ISR函数长时间占用CPU对任务的影响

对RTC周期唤醒回调函数的代码稍作修改，将最后一行上的延时HAL_Delay(1000)取消注释。

在开发板上运行，出现的效果就是LED1不能像前面那样规律性地快速闪烁，而是闪烁几次后停顿约1000ms，这是因为CPU被ISR函数占用了约1000ms。

即使将RTC周期唤醒中断的优先级修改为15，程序运行的结果也是一样的。

2. 在任务中屏蔽中断

为测试在任务中屏蔽中断的效果，对RTC做如下的修改

- 将RTC周期唤醒中断的优先级设置为7，变成了FreeRTOS可屏蔽中断
- 将RTC周期唤醒的周期设置为1秒
- RTC中断ISR函数代码中最后一行的HAL_Delay(1000)注释掉，ISR函数能快速执行完

将任务Task_LED1的任务函数修改为如下的内容

```
void AppTask_LED1(void *argument)
{
  /* USER CODE BEGIN AppTask_LED1 */
  for(;;)
  {
        taskDISABLE_INTERRUPTS();
//      taskENTER_CRITICAL();        //内部会调用taskDISABLE_INTERRUPTS()
        HAL_GPIO_TogglePin(GPIOF, GPIO_PIN_9);    //LED1闪烁
        HAL_Delay(2000);             //连续运行2000ms，任务处于运行状态
//      taskEXIT_CRITICAL();         //内部会调用taskENABLE_INTERRUPTS()
        taskENABLE_INTERRUPTS();
  }
  /* USER CODE END AppTask_LED1 */
}
```

在开发板测试，会发现LCD上的时间大约2秒钟才变化一次，而不是设置的1秒周期。这是因为在任务函数中屏蔽了中断，而RTC周期唤醒中断优先级为7，被屏蔽了。

情况1：将RTC周期唤醒中断优先级设置为1，其他设置和程序不变

下载后测试会发现，LCD上的时间是每1秒钟刷新一次了。这是因为RTC周期唤醒中断优先级为1，是FreeRTOS不可屏蔽中断。

情况2：将RTC周期唤醒中断优先级重新设置为7，将任务函数中的taskDISABLE_INTERRUPTS()和taskENABLE_INTERRUPTS()相应的替换为taskENTER_CRITICAL()和taskEXIT_CRITICAL()

测试会发现，运行效果是一样的

情况3：RTC周期唤醒中断优先级设置为7，任务函数中的HAL_Delay()函数替换为vTaskDelay()

```
void AppTask_LED1(void *argument)
{
  /* USER CODE BEGIN AppTask_LED1 */
  for(;;)
  {
        taskDISABLE_INTERRUPTS();
        HAL_GPIO_TogglePin(GPIOF, GPIO_PIN_9);    //LED1闪烁
        vTaskDelay(pdMS_TO_TICKS(2000));  //进入阻塞状态
        taskENABLE_INTERRUPTS();
  }
  /* USER CODE END AppTask_LED1 */
}
```

测试会发现，LCD上的时间每1秒刷新一次，任务函数的临界代码段里延时2000ms对RTC的周期唤醒中断响应没有影响，而使用函数HAL_Delay()是有影响的。

这是因为执行函数vTaskDelay()会使当前任务进入阻塞状态，RTOS要进行任务调度。而任务的切换是在PendSV的中断里发生的，所以RTOS必须要打开中断，只要中断被打开，RTC周期唤醒中断的ISR就能及时响应。

练习任务

1. 看教材，练习本章的示例。