

STM32Cube高效开发教程（高级篇）

第2章 FreeRTOS的任务管理

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



第2章 FreeRTOS的任务管理

2.1 任务的一些基本概念

2.2 FreeRTOS的任务调度

2.3 任务管理相关函数

2.4 多任务编程示例一

2.5 任务管理工具函数

2.6 多任务编程示例二

2.1 任务的一些基本概念

2.1.1 多任务基本运行机制

2.1.2 任务的状态

2.1.3 任务的优先级

2.1.4 空闲任务

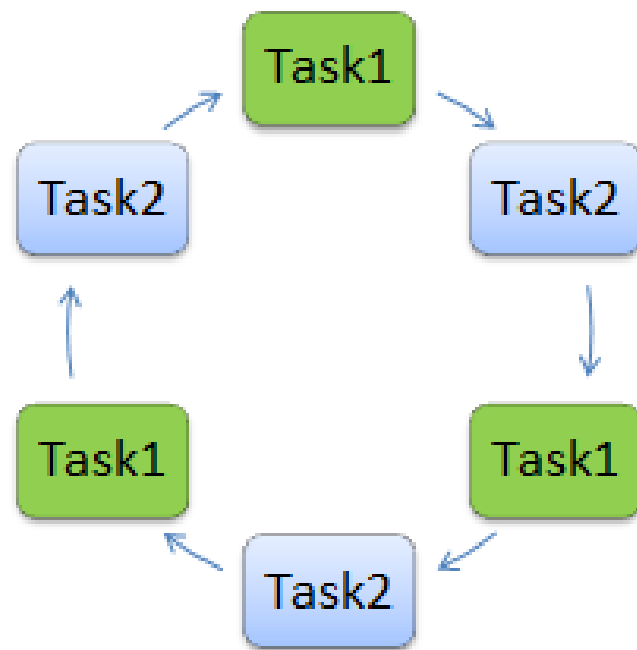
2.1.1 多任务基本运行机制

在单核处理器上，任何时刻只能有一个任务占用CPU并运行。

RTOS的任务调度使得多个任务对CPU实现了分时复用的功能。

在一个时间片内会有一个任务占用CPU并执行，在一个时间片结束时（实际是基础时钟定时器发生中断时）进行任务调度

当多个任务的优先级不同时，FreeRTOS还会使用基于优先级的抢占式任务调度方法

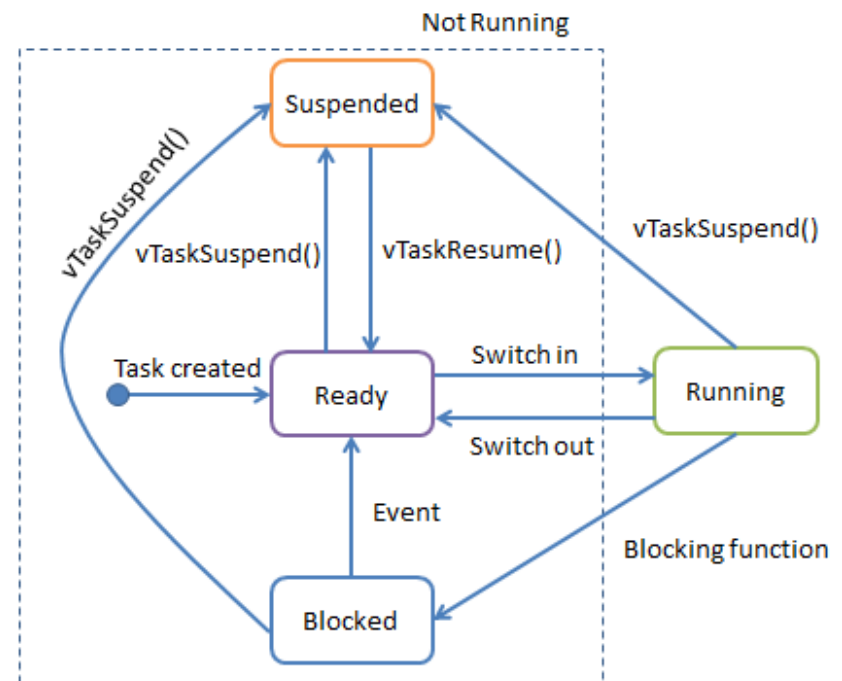


2.1.2 任务的状态

1. 就绪状态 (Ready)

任务被创建之后就处于就绪状态。根据抢占式任务调度的特点，任务调度的结果有以下几种情况：

- 如果当前没有其他处于运行状态的任务，就绪的任务就进入运行状态
- 如果就绪任务的优先级高于或等于当前运行任务的优先级，就绪的任务就进入运行状态
- 如果就绪任务的优先级低于当前运行任务的优先级，就绪的任务继续处于就绪状态



2. 运行状态 (Running)

占有CPU并运行的任务就处于运行状态。处于运行状态的任务在空闲的时候应该让出CPU的使用权。

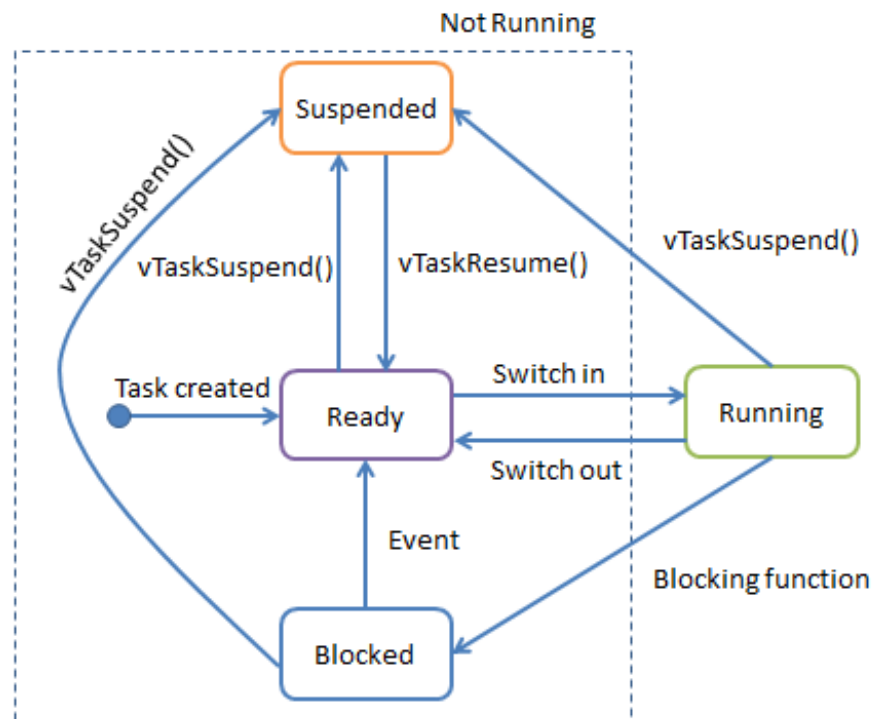
处于运行状态的任务有两种

方法主动让出CPU的使用权

- 执行函数vTaskSuspend()进入挂起状态

- 执行阻塞类函数进入阻塞状态

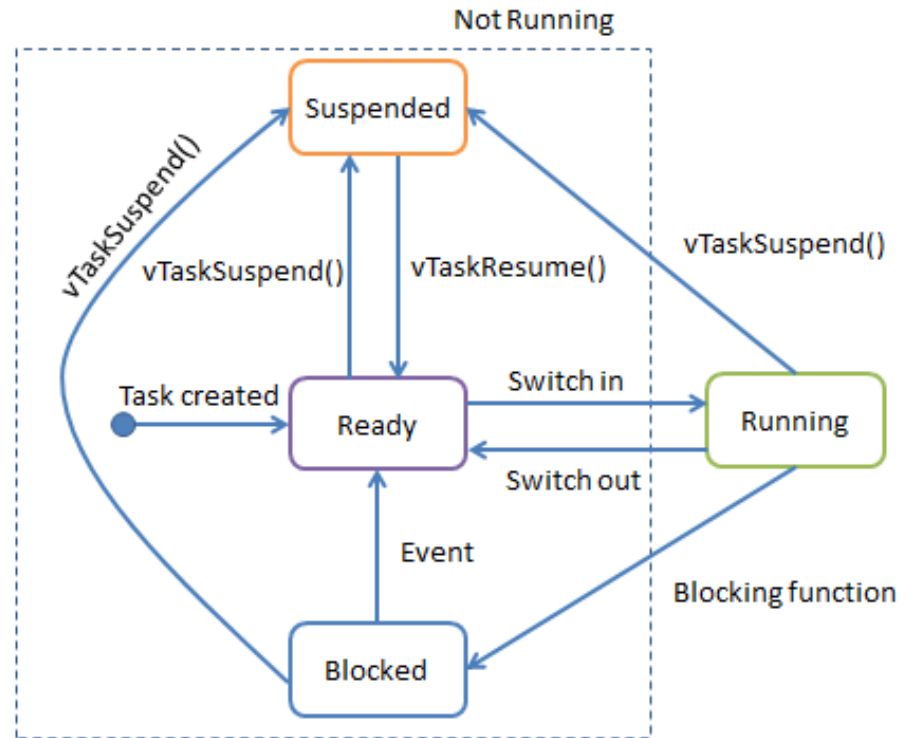
运行的任务交出CPU使用权后，任务调度器可以使其他就绪状态的任务进入运行状态。



3. 阻塞状态 (Blocked)

阻塞状态就是任务暂时让出CPU的使用权，处于一种等待的状态。运行状态的任务可以调用两类函数进入阻塞状态。

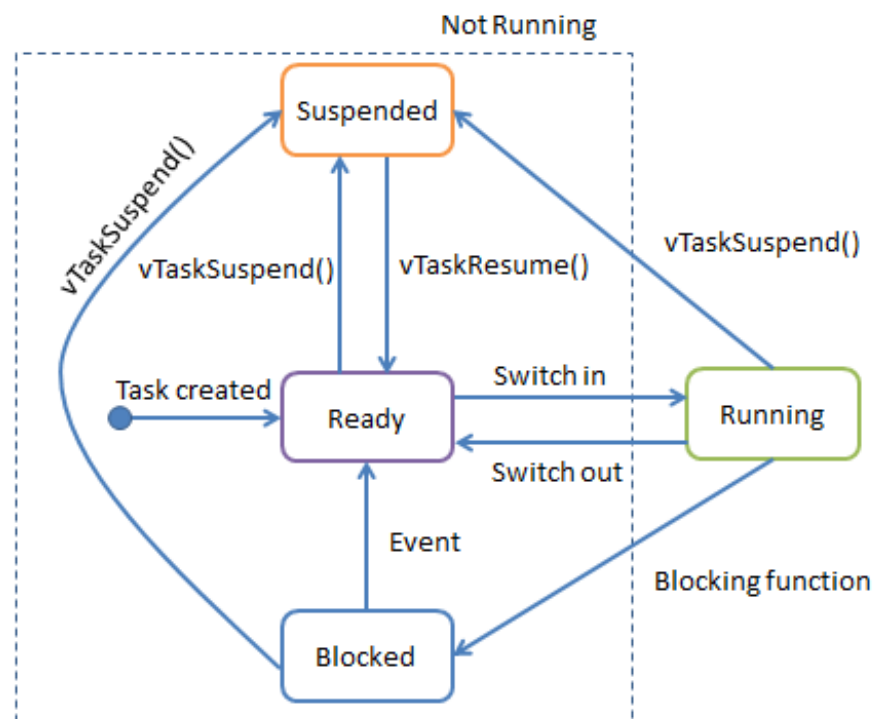
- 时间延迟函数，如 `vTaskDelay()`
- 用于进程间通讯的事件请求函数，如请求信号量的函数 `xSemaphoreTake()`



4. 挂起状态 (Suspended)

挂起状态的任务就是暂停的任务，挂起状态的任务不参与调度器的调度。其他3种状态的任务都可以通过调用函数 `vTaskSuspend()` 进入挂起状态。

挂起后的状态不能自动退出挂起状态，需要在其他任务里调用 `vTaskResume()` 函数使一个挂起的任务变为就绪状态



2.1.3 任务的优先级

- 在FreeRTOS中，每个任务都必须设置一个优先级
- 总的优先级个数由宏configMAX_PRIORITIES定义，缺省值是56
- 优先级数字越低，优先级别越低，所以最低优先级是0，最高优先级是（configMAX_PRIORITIES-1）
- 在创建任务时就必须为任务设置初始的优先级，在任务运行起来后还可以修改优先级别
- 多个任务可以具有相同的优先级

2.1.4 空闲任务

osKernelStart()启动FreeRTOS的任务调度器时，会自动创建一个空闲任务（Idle task），空闲任务的优先级为0。

与空闲任务相关的几个主要配置参数是：

- `configUSE_TICK_HOOK`，是否使用空闲函数的钩子函数，若配置为1，则可以利用空闲任务的钩子函数，系统空闲时做一些处理
- `configIDLE_SHOULD_YIELD`，空闲任务是否对同优先级的用户任务主动让出CPU使用权，这会影响任务调度结果
- `configUSE_TICKLESS_IDLE`，是否在空闲任务时关闭基础时钟，若设置为1，可实现系统的低功耗

2.2 FreeRTOS的任务调度

2.2.1 任务调度方法概述

2.2.2 使用时间片的抢占式调度方法

2.2.3 不使用时间片的抢占式调度方法

2.2.4 合作式任务调度方法

2.2.1 任务调度方法概述

FreeRTOS有两种任务调度算法，**基于优先级的抢占式**（pre-emptive）调度算法和**合作式**调度（co-operative）算法，其中抢占式调度算法又可以使用时间片或不使用时间片。

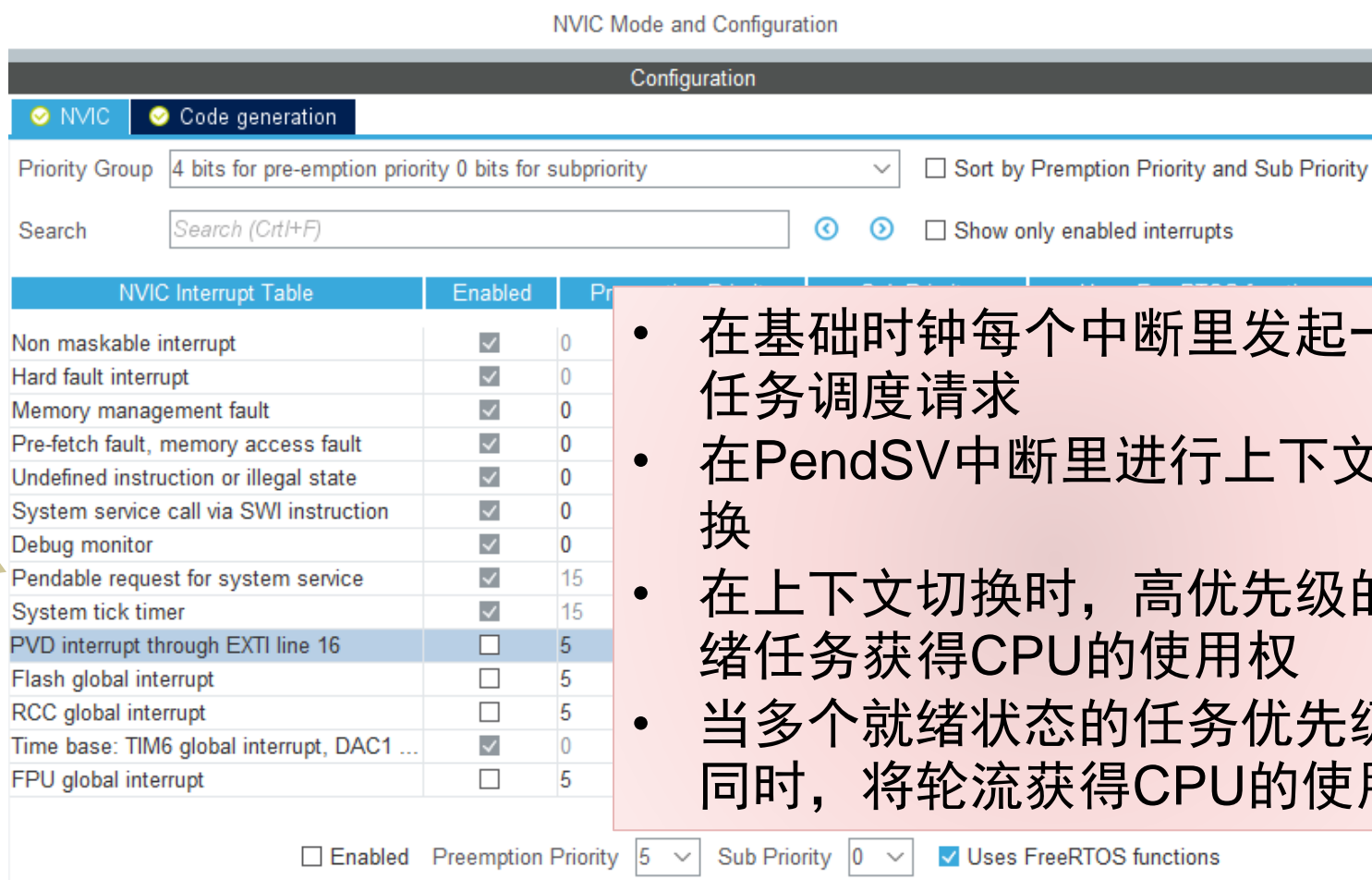
调度方式	宏定义参数	取值	特点
抢占式 （使用时间片）	configUSE_PREEMPTION	1	基于优先级的抢占式任务调度，同优先级任务使用时间片轮流进入运行状态（默认模式）
	configUSE_TIME_SLICING	1	
抢占式 （不使用时间片）	configUSE_PREEMPTION	1	基于优先级的抢占式任务调度，同优先级任务不使用时间片调度
	configUSE_TIME_SLICING	0	
合作式	configUSE_PREEMPTION	0	只有当运行状态的任务进入阻塞状态，或显式地调用要求执行任务调度的函数taskYIELD()，FreeRTOS才会发生任务调度，选择就绪状态的高优先级任务进入运行状态
	configUSE_TIME_SLICING	任意	

2.2.2 使用时间片的抢占式调度方法

FreeRTOS基础时钟的一个定时周期称为一个时间片（time slice），默认值为1ms。当使用时间片时，在基础时钟的每次中断里会要求进行一次上下文切换（context switching），函数xPortSysTickHandler()就是SysTick定时中断的处理函数

```
void xPortSysTickHandler( void )
{
    /* SysTick 中断的抢占优先级是15，优先级最低 */
    portDISABLE_INTERRUPTS();    //禁用所有中断
    {
        if( xTaskIncrementTick() != pdFALSE ) //增加RTOS滴嗒计数器的值
        {
            /* 将PendSV中断的挂起标志位置位，申请进行上下文切换,上下文切换在PendSV中断里处理 */
            portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
        }
    }
    portENABLE_INTERRUPTS();    //使能中断
}
```

这个函数的功能就是将PendSV（Pendable request for system service，系统可挂起服务请求）中断的挂起标志位置位，也就是发起上下文切换的请求，而进行上下文切换是在PendSV的中断服务程序里完成的。



- 在基础时钟每个中断里发起一次任务调度请求
- 在PendSV中断里进行上下文切换
- 在上下文切换时，高优先级的就绪任务获得CPU的使用权
- 当多个就绪状态的任务优先级相同时，将轮流获得CPU的使用权

任务运行时序图

- ◆ 横轴是时间轴
- ◆ 纵轴是系统中运行的任务
- ◆ 垂直方向的虚线表示发生任务切换的时间点
- ◆ 水平方向的实心矩形表示任务占据CPU处于运行状态的时间段
- ◆ 水平方向的虚线表示任务处于就绪状态的时间段
- ◆ 水平方向的空白段表示任务处于阻塞状态或挂起状态的时间段

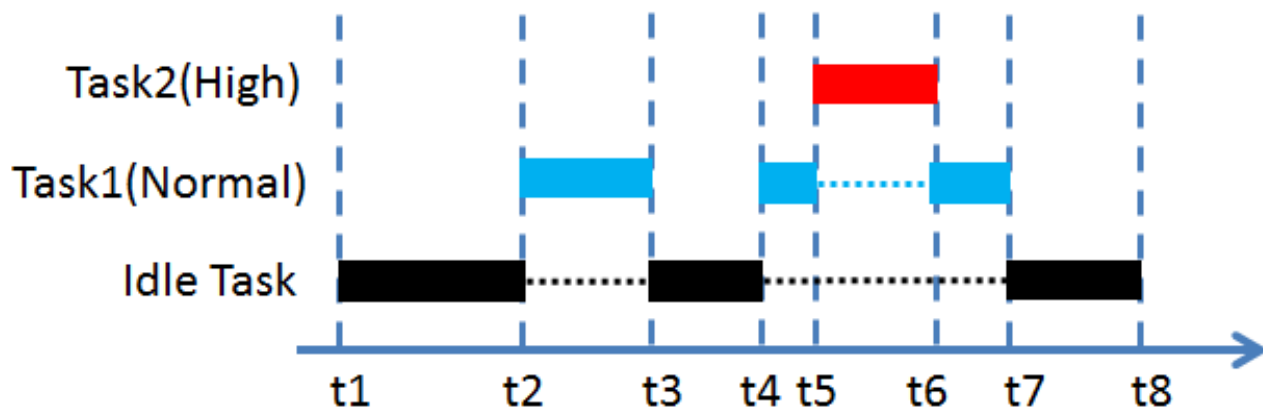
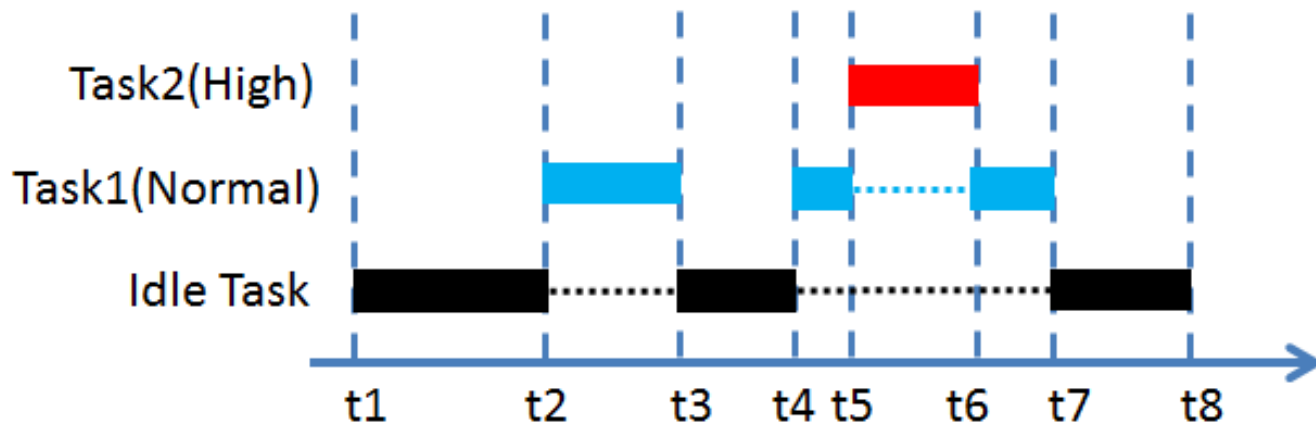
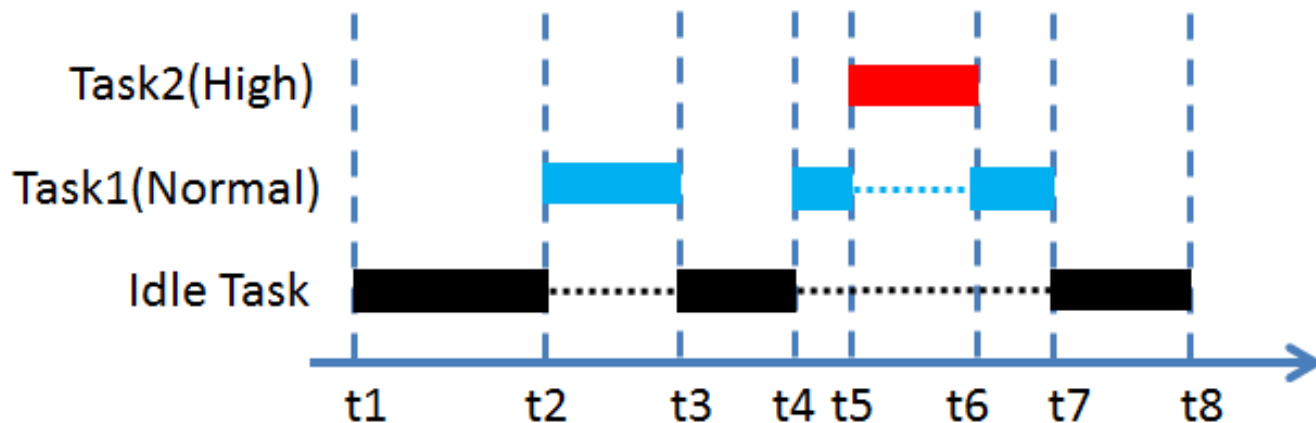


图2-3 任务运行时序图（带时间片的抢占式任务调度方法）



- t1时刻开始是空闲任务在运行，这时候系统里没有其他任务处于就绪状态。
- 在t2时刻进行调度时，Task1抢占CPU开始运行，因为Task1的优先级高于空闲任务。
- 在t3时刻，Task1进入阻塞状态，让出了CPU的使用权，空闲任务又进入运行状态。
- 在t4时刻，Task1又进入运行状态。



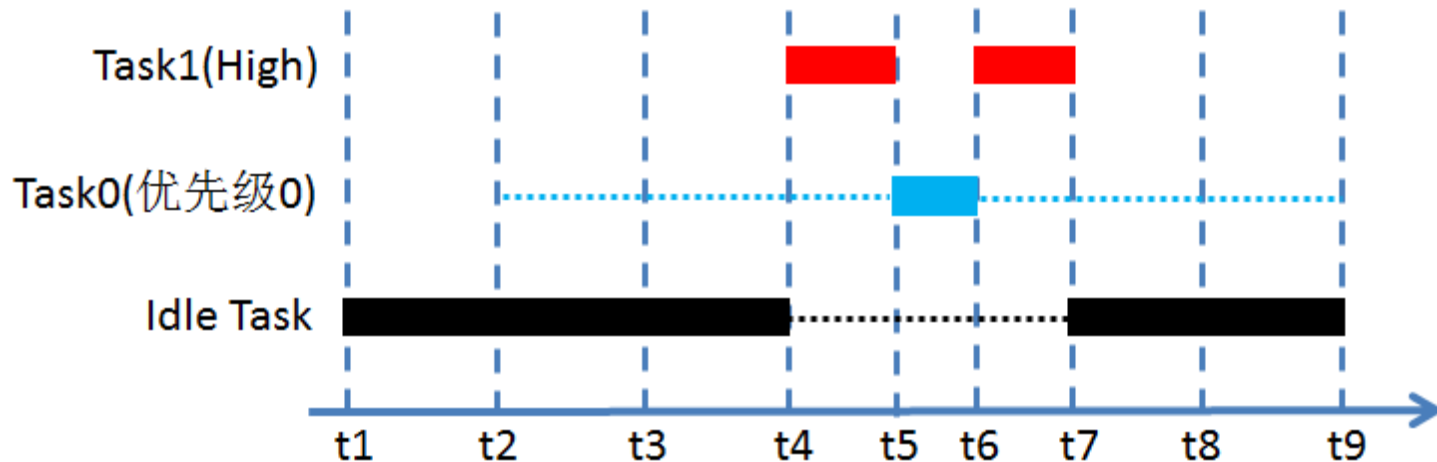
- 在t5时刻，更高优先级的Task2抢占了CPU开始运行，Task1进入就绪状态。
- 在t6时刻，Task2运行后进入阻塞状态，让出CPU使用权，Task1从就绪状态变为运行状态。
- 在t7时刻，Task1进入阻塞状态，主动让出CPU使用权，空闲任务又进入运行状态。

2.2.3 不使用时间片的抢占式调度方法

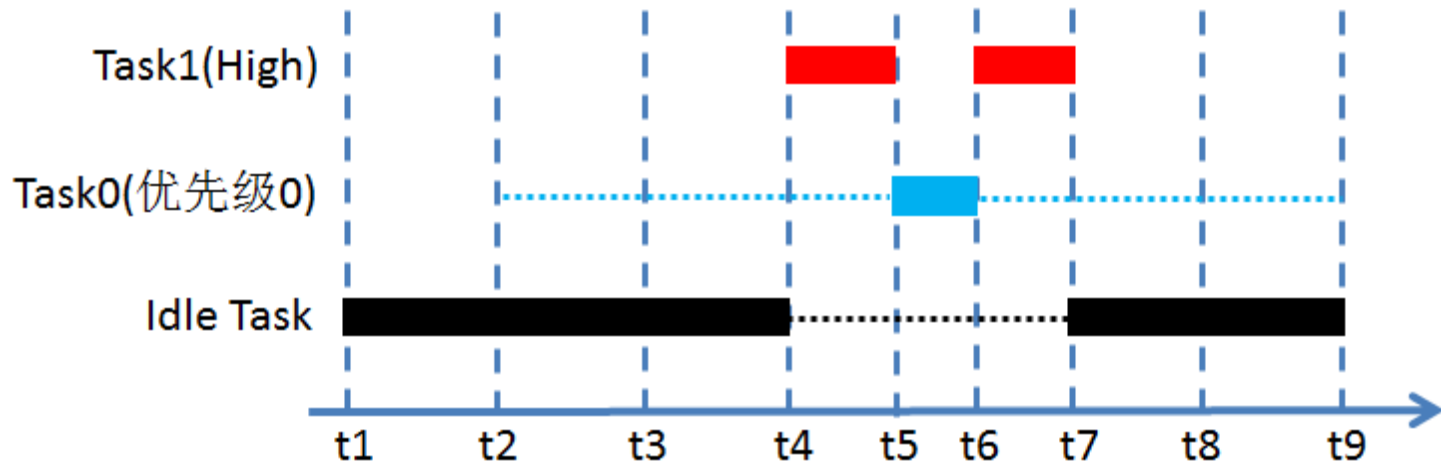
使用时间片的抢占式调度方法在每个SysTick中断里都进行一次上下文切换请求，从而进行任务调度。而不使用时间片的抢占式调度算法只在以下情况下才进行任务调度：

- 有更高级别的任务进入就绪状态时；
- 运行状态的任务进入阻塞状态或挂起状态时。

所以，不使用时间片时，进行上下文切换的频率比使用时间片时低，从而可降低CPU的负担。但是，对于同优先级的任务可能会出现占用CPU时间相差很大的情况。



- 在t1时刻，空闲任务占用CPU，因为系统里没有其他任务处于就绪状态。
- 在t2时刻，Task0进入就绪状态。但是Task0与空闲任务优先级相同，且调度算法不使用时间片，不会让Task0和空闲任务轮流使用CPU，所以Task0就保持就绪状态。
- 在t4时刻，高优先级的Task1抢占CPU。



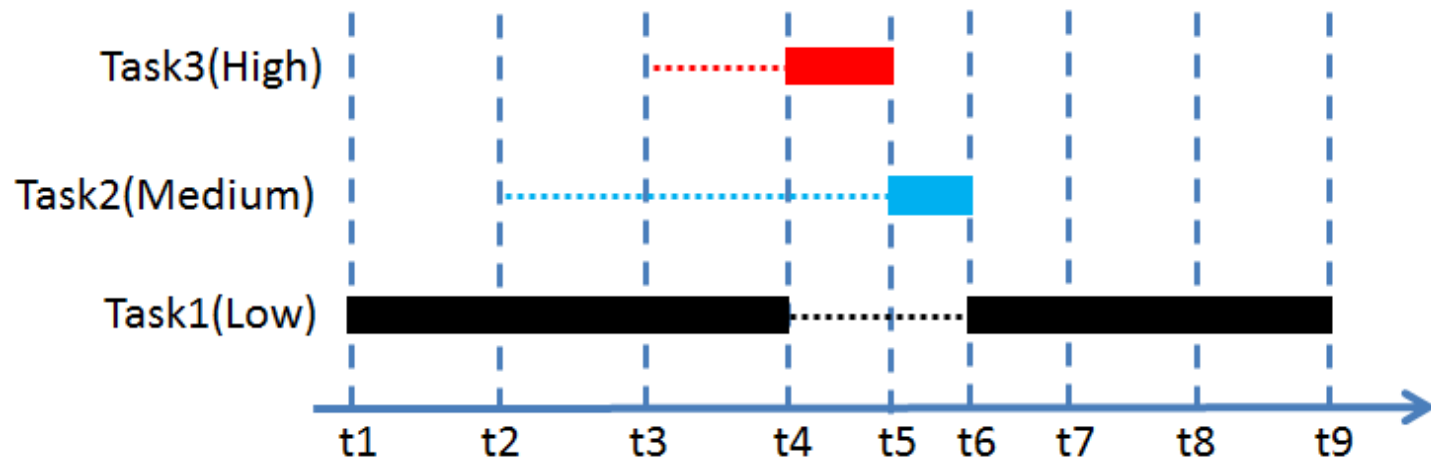
- 在t5时刻，Task1进入阻塞状态，系统进行一次任务调度，Task0获得CPU的使用权。
- 在t6时刻，Task1再次抢占CPU，Task0又进入就绪状态。
- 在t7时刻，Task1进入阻塞状态，系统进行一次任务调度，空闲任务获得CPU使用权。之后没有发生任务调度的机会，所以Task0就一直处于就绪状态。

2.2.4 合作式任务调度方法

使用合作式任务调度方法时，FreeRTOS不主动进行上下文切换，而是运行状态的任务进入阻塞状态，或显式地调用taskYIELD()函数让出CPU使用权时才进行上下文切换。

任务不会发生抢占，所以也不使用时间片。

函数taskYIELD()的作用就是主动申请进行一次上下文切换。



- ◆ 在t1时刻，低优先级的Task1处于运行状态。
- ◆ 在t2时刻，中等优先级的Task2进入就绪状态，但不能抢占CPU。
- ◆ 在t3时刻，高优先级的Task3进入就绪状态，但是也不能抢占CPU。
- ◆ 在t4时刻，Task1调用函数taskYIELD()主动申请进行一次上下文切换，高优先级的Task3获得CPU使用权。
- ◆ 在t5时刻，Task3进入阻塞状态，就绪的Task2获得CPU的使用权。
- ◆ 在t6时刻，Task2进入阻塞状态，Task1又获得CPU使用权。

2.3 任务管理相关函数

2.3.1 相关函数概述

2.3.2 主要函数功能说明

2.3.1 相关函数概述

分组	FreeRTOS函数	函数功能
任务管理	xTaskCreate()	创建一个任务，动态分配内存
	xTaskCreateStatic()	创建一个任务，静态分配内存
	vTaskDelete()	删除当前任务或另一个任务
	vTaskSuspend()	挂起当前任务或另一个任务
	vTaskResume()	恢复另一个挂起任务的运行
调度器管理	vTaskStartScheduler()	开启任务调度器
	vTaskSuspendAll()	挂起调度器，但不禁止中断。调度器被挂起后不会再进行上下文切换
	xTaskResumeAll()	恢复调度器的执行，但是不会解除用函数vTaskSuspend()单独挂起的任务的挂起状态
	vTaskStepTick()	用于在tickless低功耗模式时补足系统时钟计数节拍
延时与调度	vTaskDelay()	当前任务延时指定节拍数，并进入阻塞状态
	vTaskDelayUntil()	当前任务延时到指定的时间，并进入阻塞状态，用于精确延时的周期性任务
	xTaskGetTickCount()	返回基础时钟定时器的当前计数值
	xTaskAbortDelay()	终止另一个任务的延时，使其立刻退出阻塞状态
	taskYIELD()	请求进行一次上下文切换，用于合作式任务调度

文件task.h中还有几个常用的宏函数

//关闭MCU的所有可屏蔽中断

#define taskDISABLE_INTERRUPTS() portDISABLE_INTERRUPTS()

//使能MCU的中断

#define taskENABLE_INTERRUPTS() portENABLE_INTERRUPTS()

这两个宏函数用于关闭和开启MCU的可屏蔽中断，用于界定不受其他中断干扰的代码段。

只能关闭FreeRTOS可管理的中断优先级别，也就是参数configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY定义的最高优先级【下一章详细介绍】。

界定临界代码段

```
//开始临界代码段
#define taskENTER_CRITICAL()    portENTER_CRITICAL()

//结束临界代码段
#define taskEXIT_CRITICAL()     portEXIT_CRITICAL()
```

这两个函数用于界定一个**临界（Critical）代码段**，在临界代码段内，任务不会被更高优先级的任务抢占，以保障代码执行的连续性。

2.3.主要函数功能说明

1. 创建任务

CubeMX导出的代码中使用函数`osThreadNew()`创建任务，它会根据任务的属性自动调用`xTaskCreate()`动态分配内存创建任务，或调用`xTaskCreateStatic()`静态分配内存创建任务

动态分配内存创建任务的函数是`xTaskCreate()`

```
 BaseType_t xTaskCreate( TaskFunction_t pxTaskCode,           //任务函数名称
                        const char * const pcName,           //任务的备注名称
                        const uint16_t usStackDepth,         //栈空间大小，单位：字
                        void * const pvParameters,           //传递给任务函数的参数
                        UBaseType_t uxPriority,               //任务优先级
                        TaskHandle_t * const pxCreatedTask )   //任务的句柄
```

2. 延时函数

延时函数vTaskDelay()用于延时一定节拍数，它会使当前任务进入阻塞状态。

```
void vTaskDelay( const TickType_t xTicksToDelay );
```

参数xTicksToDelay表示基础时钟的节拍数，一般会结合宏函数**pdMS_TO_TICKS()**将一个时间转换为节拍数，然后调用vTaskDelay()，如

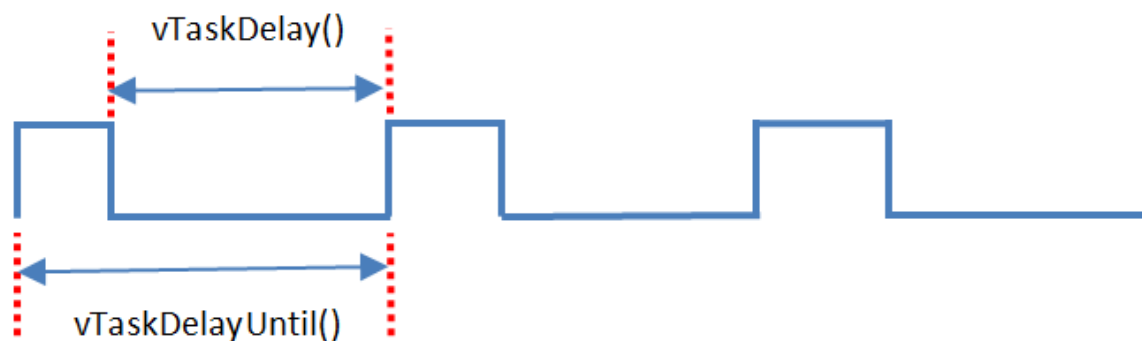
```
vTaskDelay(pdMS_TO_TICKS(500));    //延时500ms
```

3. 绝对延时函数

```
void vTaskDelayUntil( TickType_t * const pxPreviousWakeTime, const  
    TickType_t xTimeIncrement );
```

参数pxPreviousWakeTime表示上次任务唤醒时，基础时钟计数器的值；参数xTimeIncrement表示相对于上次唤醒时刻延时的节拍数。

函数里每次会自动更新pxPreviousWakeTime的值，但是在第一次调用时需要给一个初值



一个周期内，高电平表示任务运行时间，低电平表示阻塞时间，上跳沿表示唤醒时刻，下跳沿表示进入阻塞状态的时刻

```
void AppTask_Function(void *argument)
{
    TickType_t previousWakeTime=xTaskGetTickCount();
    for(;;) /* 死循环 */
    {
        //死循环内的功能代码
        //确保一个循环周期是1000ms
        vTaskDelayUntil(&previousWakeTime, pdMS_TO_TICKS(1000));
    }
}
```

2.4 多任务编程示例一

2.4.1 示例功能与CubeMX项目设置

2.4.2 初始化代码

2.4.3 编写用户功能代码

2.4.1 示例功能与CubeMX项目设置

示例Demo2_1MultiTasks的功能：设计两个任务，在任务1里使LED1闪烁，在任务2里使LED2闪烁。

✓ Config parameters

✓ Include parameters

✓ User Constants

✓ Tasks and Queues

Tasks

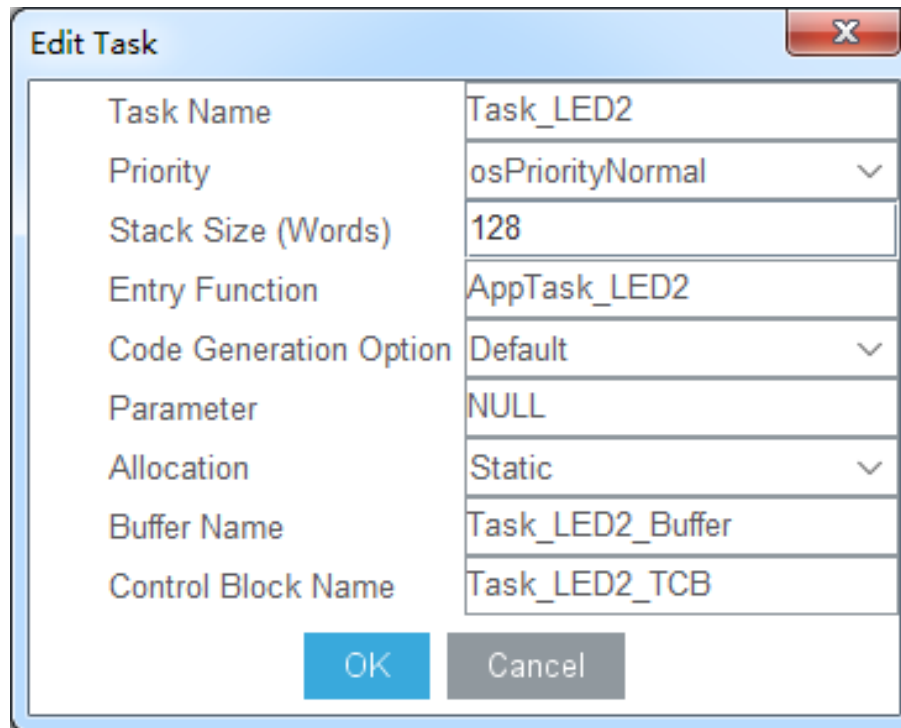
Task Name	Priority	Stack...	Entry Function	Code Ge...	Param...	Allocation	Buffer Name	Control Block Name
Task_LED1	osPriorityNormal	128	AppTask_LED1	Default	NULL	Dynamic	NULL	NULL
Task_LED2	osPriorityNormal	128	AppTask_LED2	Default	NULL	Static	Task_LED2_Buffer	Task_LED2_TCB

Add

Delete

在FreeRTOS中创建两个任务

- 两个任务的优先级都设置为osPriorityNormal
- Task_LED2使用静态分配内存。使用静态分配内存时需要设置作为栈空间的数组名称，以及控制块名称



The image shows a dialog box titled "Edit Task" with a close button (X) in the top right corner. It contains a table with task configuration parameters. The parameters and their values are: Task Name (Task_LED2), Priority (osPriorityNormal), Stack Size (Words) (128), Entry Function (AppTask_LED2), Code Generation Option (Default), Parameter (NULL), Allocation (Static), Buffer Name (Task_LED2_Buffer), and Control Block Name (Task_LED2_TCB). At the bottom, there are "OK" and "Cancel" buttons.

Parameter	Value
Task Name	Task_LED2
Priority	osPriorityNormal
Stack Size (Words)	128
Entry Function	AppTask_LED2
Code Generation Option	Default
Parameter	NULL
Allocation	Static
Buffer Name	Task_LED2_Buffer
Control Block Name	Task_LED2_TCB

以静态分配内存方式创建任务时，需要定义Buffer Name
和Control Block Name

2.4.2 初始化代码

1. 主程序

```
int main(void)
{
    HAL_Init();           //HAL 初始化，复位所有外设，初始化Flash和SysTick
    SystemClock_Config(); //系统时钟配置
    /* 初始化所有已配置外设 */
    MX_GPIO_Init();       //GPIO初始化，LED1和LED2的GPIO初始化

    osKernelInitialize(); //RTOS内核初始化
    MX_FREERTOS_Init();   //FreeRTOS对象初始化函数，在freertos.c中实现
    osKernelStart();      //启动RTOS内核

    /* We should never get here as control is now taken by the scheduler */
    while (1)
    {
    }
}
```

2. 任务的创建

Task_LED1是动态分配内存，定义任务句柄和任务属性，任务属性里无需分配内存。

```
/* 任务 Task_LED1 的定义，动态分配内存方式 */  
osThreadId_t Task_LED1Handle;    //任务Task_LED1的句柄变量  
  
const osThreadAttr_t Task_LED1_attributes = {///任务Task_LED1的属性  
    .name = "Task_LED1",    //任务名称  
    .priority = (osPriority_t) osPriorityNormal, //任务优先级  
    .stack_size = 128 * 4    //栈空间大小， 128*4字节  
};
```

2. 任务的创建

Task_LED2是静态分配内存，需要定义用作栈空间的数组，控制块变量。

```
typedef StaticTask_t osStaticThreadDef_t; //类型符号定义，

/* 任务 Task_LED2 的定义，静态分配内存方式 */
osThreadId_t Task_LED2Handle;    //任务Task_LED2的句柄变量

uint32_t Task_LED2_Buffer[ 128 ]; //任务Task_LED2的栈空间数组

osStaticThreadDef_t Task_LED2_TCB; //任务Task_LED2的任务控制块

const osThreadAttr_t Task_LED2_attributes = { //任务Task_LED2的属性
    .name = "Task_LED2",           //任务名称
    .stack_mem = &Task_LED2_Buffer[0], //栈空间数组
    .stack_size = sizeof(Task_LED2_Buffer), //栈空间大小,字
    .cb_mem = &Task_LED2_TCB,       //任务控制块
    .cb_size = sizeof(Task_LED2_TCB), //任务控制块大小
    .priority = (osPriority_t) osPriorityNormal, //任务优先级
};
```

函数MX_FREERTOS_Init()中创建两个任务，都使用函数osThreadNew()，这个函数内部会自动调用xTaskCreate()或xTaskCreateStatic()

```
void MX_FREERTOS_Init(void)
{
    /* 创建任务Task_LED1 */
    Task_LED1Handle = osThreadNew( AppTask_LED1, NULL,
    &Task_LED1_attributes);

    /* 创建任务 Task_LED2 */
    Task_LED2Handle = osThreadNew( AppTask_LED2, NULL,
    &Task_LED2_attributes);
}
```

2.4.3 编写用户功能代码

为两个任务函数编写代码，并且对任务的属性稍微做些修改，以测试带时间片的抢占式任务调度方法的特点，以及vTaskDelay()和vTaskDelayUntil()等函数的使用方法。

1. 相同优先级的任务的执行

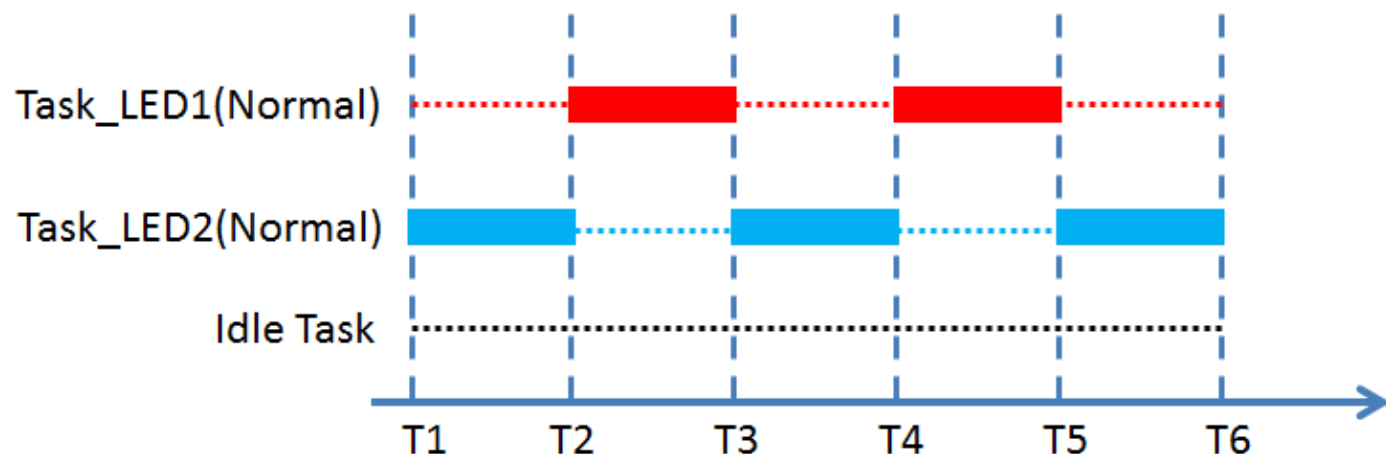
两个任务的优先级相同，分别使LED1和LED2以不同的周期闪烁。

注意，两个任务函数的for循环中都使用延时函数HAL_Delay()，这个延时函数不会使任务进入阻塞状态，而是一直处于连续运行状态。

```
void AppTask_LED1(void *argument)           //任务Task_LED1的入口函数
{
    /* USER CODE BEGIN AppTask_LED1 */
    for(;;)
    {
        HAL_GPIO_TogglePin(GPIOF, GPIO_PIN_9); //PF9=LED1
        HAL_Delay(1000);
    }
    /* USER CODE END AppTask_LED1 */
}
```

```
void AppTask_LED2(void *argument)           //任务Task_LED2的入口函数
{
    /* USER CODE BEGIN AppTask_LED2 */
    for(;;)
    {
        HAL_GPIO_TogglePin(GPIOF, GPIO_PIN_10); //PF10=LED2
        HAL_Delay(500);
    }
    /* USER CODE END AppTask_LED2 */
}
```

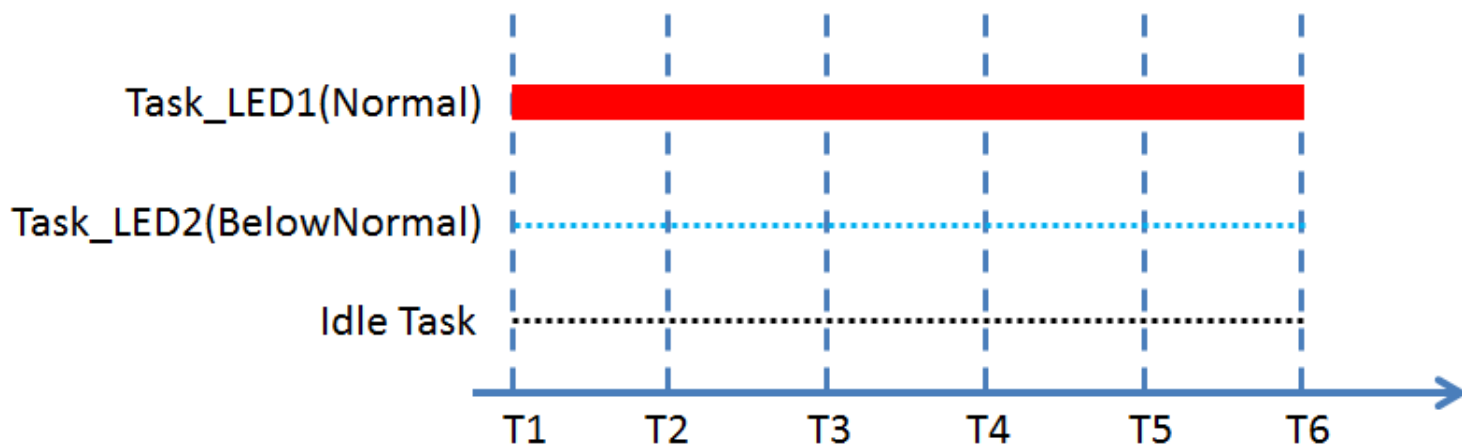

下载到开发板运行，会发现LED1和LED2都能闪烁，两个任务都被执行。



由于两个任务具有相同的优先级，所以调度器使两个任务轮流占用CPU。两个任务都是连续运行的，所以每个任务每次占用CPU的时间是一个嘀嗒时钟周期。空闲任务总是无法获得CPU的使用权。

2. 低优先级任务被“饿死”的情况

修改任务优先级，其他不做任何修改。下载运行，会发现只有LED1闪烁，而LED2不闪烁。



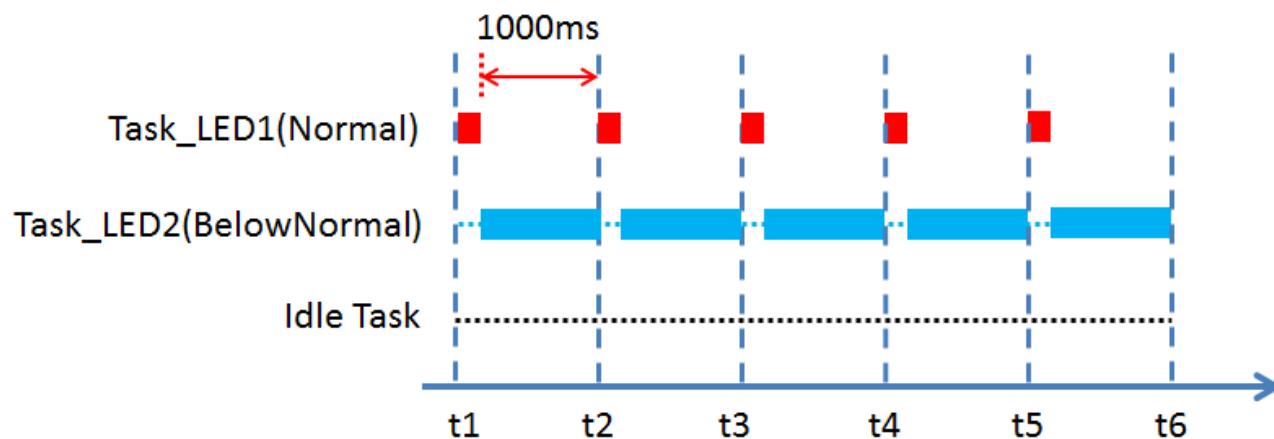
由于Task_LED1具有高优先级，且是连续运行的，不会进入阻塞状态，所以低优先级的任务Task_LED2和空闲任务都无法获得CPU的使用权，它们被“饿死”了。

3. 高优先级任务主动进入阻塞状态

对前面的程序再稍作修改，保留一高一低的优先级设置，将Task_LED1中的延时函数修改为vTaskDelay()，Task_LED2中的延时函数仍然使用HAL_Delay()

```
void AppTask_LED1(void *argument)           //任务Task_LED1的入口函数
{
    /* USER CODE BEGIN AppTask_LED1 */
    for(;;)
    {
        HAL_GPIO_TogglePin(GPIOF, GPIO_PIN_9); //PF9=LED1
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
    /* USER CODE END AppTask_LED1 */
}
```

下载到开发板运行，会发现LED1和LED2都能闪烁，两个任务都可以被执行。



- 任务Task_LED1大部分时间处于阻塞状态
- 在任务Task_LED1处于阻塞状态时，任务Task_LED2可以获得CPU的使用权
- 任务Task_LED1在延时结束后，可以抢占CPU的使用权
- 空闲任务还是无法获得CPU的使用权

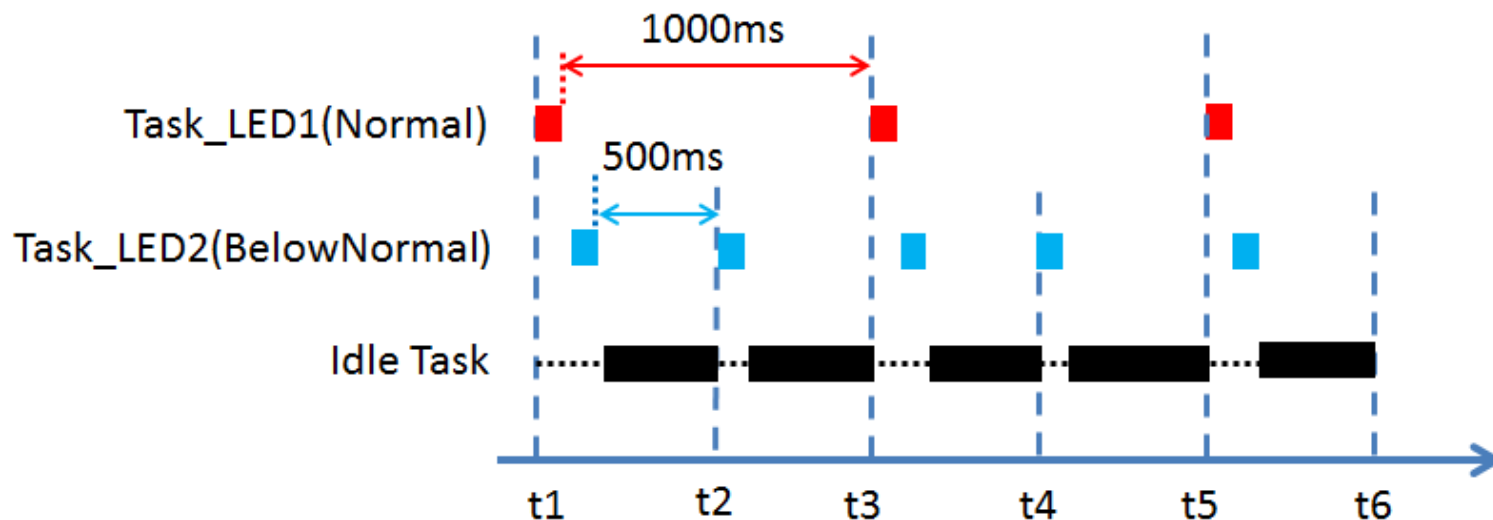
4. 多任务系统一般的任务函数的设计

在使用抢占式任务调度方法时，要根据任务的重要性分配不同的优先级，在任务空闲时要让出CPU的使用权，使其他就绪状态的任务能获得CPU的使用权。

对前面的程序再稍作修改，

- Task_LED2的优先级为osPriorityBelowNormal
- 任务Task_LED2的优先级任然为osPriorityNormal
- 任务函数中都使用vTaskDelay()延时函数

下载运行，会发现LED1和LED2都能闪烁



任务Task_LED1和Task_LED2大部分时间处于阻塞状态，系统的空闲任务获得CPU的使用权。

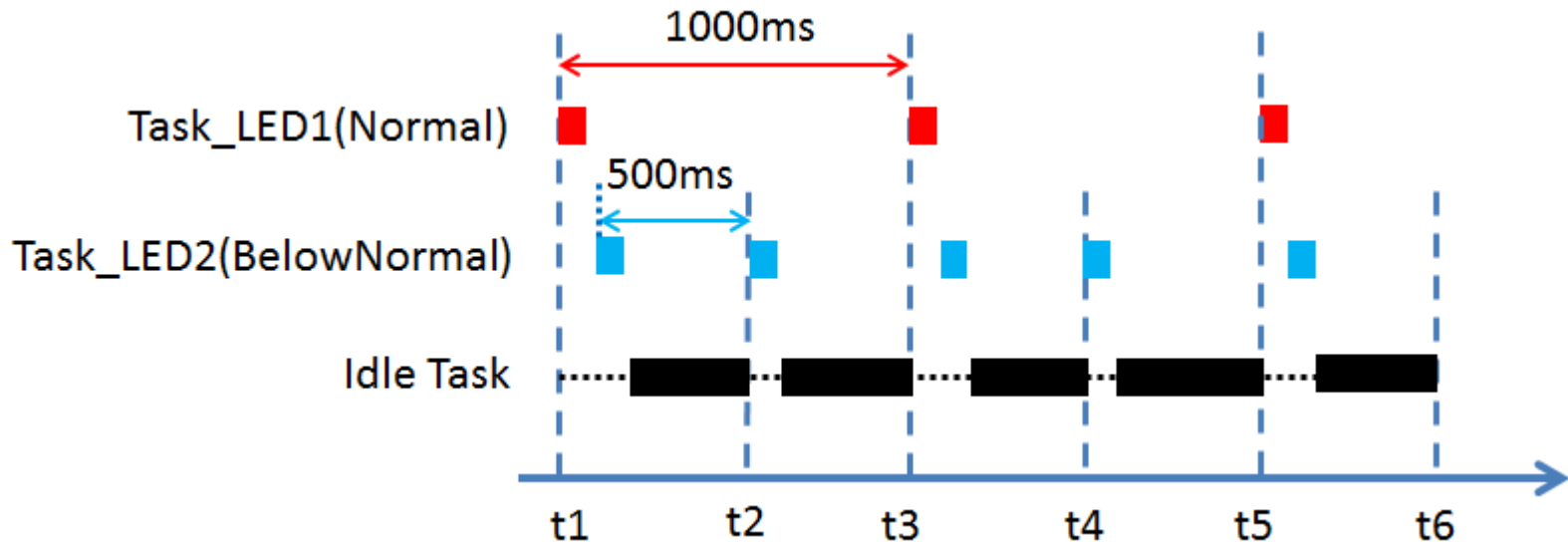
5. 使用vTaskDelayUntil()函数

如果要在任务函数的循环中实现严格的周期性，就应该使用函数vTaskDelayUntil()。对上一步的程序稍做修改，在两个任务的任务函数中使用函数vTaskDelayUntil()。

```
void AppTask_LED1(void *argument)           //任务Task_LED1的入口函数
{
    /* USER CODE BEGIN AppTask_LED1 */
    TickType_t ticks1=pdMS_TO_TICKS(1000); //时间（ms）转换为节拍数
    TickType_t previousWakeTime= xTaskGetTickCount();
    for(;;)
    {
        HAL_GPIO_TogglePin(GPIOF, GPIO_PIN_9); //PF9=LED1
        vTaskDelayUntil(&previousWakeTime, ticks1); //循环周期1000ms
    }
    /* USER CODE END AppTask_LED1 */
}
```

使用函数vTaskDelayUntil()延时的时间是从任务上次被转入运行状态开始的绝对时间。

第一次执行时需要通过xTaskGetTickCount()获取基础时钟计数器的当前值作为previousWakeTime的初值。



2.5 任务管理工具函数

1. 获取任务句柄

FreeRTOS函数	函数功能
xTaskGetCurrentTaskHandle()	获得当前任务的句柄
xTaskGetIdleTaskHandle()	获取空闲任务的句柄
xTaskGetHandle()	根据任务名称返回任务句柄，运行比较慢

2. 获取任务的信息

FreeRTOS函数	函数功能
uxTaskPriorityGet()	获得一个任务的优先级
uxTaskPriorityGetFromISR()	函数uxTaskPriorityGet()的ISR版本
vTaskPrioritySet()	设置一个任务的优先级，可以在运行过程中改变一个任务的优先级
vTaskGetInfo()	返回一个任务的信息，包括状态信息、栈空间信息等
pcTaskGetName()	根据任务句柄返回任务的名称，参数为NULL时返回任务自己的名称
uxTaskGetStackHighWaterMark()	返回一个任务的栈空间的最高水位，即最少可用空间，返回值越小，表明任务的栈空间越容易溢出
eTaskGetState()	返回一个任务的当前运行状态，返回值是枚举类型eTaskState

vTaskGetInfo()用于获取一个任务的信息

```
void vTaskGetInfo( TaskHandle_t xTask,      //任务的句柄
                   TaskStatus_t *pxTaskStatus, //存储任务状态信息的结构体指针
                   BaseType_t xGetFreeStackSpace, //是否返回栈空间高水位值
                   eTaskState eState );        //指定任务的状态
```

- xTask是需要查询的任务的句柄
- pxTaskStatus是存储返回信息的TaskStatus_t结构体指针
- xGetFreeStackSpace表示是否在结构体TaskStatus_t中返回栈空间的高水位值usStackHighWaterMark
- eState用于指定查询信息时的任务状态，虽然结构体TaskStatus_t中有获取任务状态的成员变量，但是不如直接赋值快

结构体TaskStatus_t的定义如下

```
typedef struct xTASK_STATUS
{
    TaskHandle_t xHandle;                /* 任务的句柄 */
    const char *pcTaskName;              /* 任务的名称 */
    UBaseType_t xTaskNumber;             /* 任务的唯一编号 */
    eTaskState eCurrentState;            /* 任务的状态 */
    UBaseType_t uxCurrentPriority;        /* 任务的优先级 */
    UBaseType_t uxBasePriority;           /* 在使用互斥量时，为避免优先级反
    转而继承的优先级，参数configUSE_MUTEXES设置为1时此变量才有意义 */
    uint32_t ulRunTimeCounter;            /* 任务运行的总时间，
    configGENERATE_RUN_TIME_STATS 设置为1时才有意义*/
    StackType_t *pxStackBase;            /* 指向栈空间的低地址 */
    uint16_t usStackHighWaterMark;       /* 栈空间的高水位值，也就是出现过
    的最小剩余栈空间大小 */
} TaskStatus_t;
```

枚举类型eTaskState，这个枚举类型表示了任务的运行、就绪、阻塞、挂起等状态，其定义如下。

```
typedef enum
{
    eRunning = 0,      /* 运行状态 */
    eReady,            /* 就绪状态 */
    eBlocked,           /* 阻塞状态 */
    eSuspended,         /* 挂起状态，或无限等待时间的阻塞状态 */
    eDeleted,           /* 任务被删除，但是其任务控制块还没有被释放 */
    eInvalid            /* 无效状态 */
} eTaskState;
```

3. 内核信息

FreeRTOS函数	函数功能
uxTaskGetNumberOfTasks()	返回内核管理的所有任务的个数，包括就绪的、阻塞的、挂起的任务，也包括虽然删除了，但还没有在空闲任务里释放的任务
vTaskList	创建一个列表，显示所有任务的信息。此函数会禁止所有中断，需要使用sprintf()函数，所以一般只用于调试阶段
uxTaskGetSystemState()	获取系统中所有任务的任务状态，包括每个任务的句柄、任务名称、优先级等信息。
vTaskGetRunTimeStats()	获得每个任务的运行时间统计
xTaskGetTickCount()	返回基础时钟定时器的当前计数值
xTaskGetTickCountFromISR()	函数xTaskGetTickCount()的ISR版本
xTaskGetSchedulerState()	返回任务调度器的运行状态，返回值是宏定义常量

4. 其他函数

FreeRTOS函数	函数功能
vTaskSetApplicationTaskTag()	设置一个任务的标签值，每个任务可以设置一个标签值，保存于任务控制块中
xTaskGetApplicationTaskTag()	获取一个任务的标签值
xTaskCallApplicationTaskHook()	调用任务关联的钩子函数
vTaskSetThreadLocalStoragePointer()	每个任务有一个指针数组，此函数为任务设置一个本地存储指针，指针的用途由用户自己决定，内核不使用此指针数组
pvTaskGetThreadLocalStoragePointer()	获取任务的指针数组中的一个指定序号的指针
vTaskSetTimeOutState()	获取当前的时钟状态，以便作为xTaskCheckForTimeOut()函数的初始条件参数
xTaskCheckForTimeOut()	检查是否超过等待的节拍数，与vTaskSetTimeOutState()函数结合使用，仅用于高级用途

2.6 多任务编程示例二

2.6.1 示例功能与STM32CubeMX项目设置

2.6.2 初始化代码

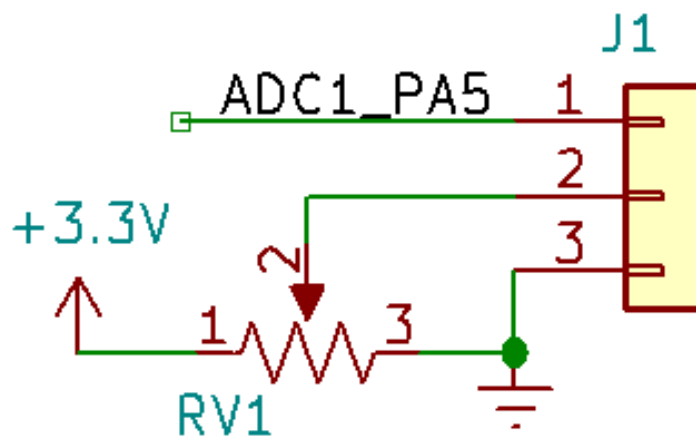
2.6.3 程序功能的实现

2.6.1 示例功能与STM32CubeMX项目设置

创建2个任务，任务Task_ADC通过ADC1的IN5通道定时采集电位器的电压值，并在LCD上显示。任务Task_Info用于测试任务信息统计的一些工具函数，在LCD上显示

在CubeMX中做以下设置

- 导入TFT LCD项目
- 使用ADC1的IN5输入通道，轮询方式采集
- 使用LED1



在FreeRTOS中创建2个任务，两个任务具有不同的优先级，修改了默认的栈空间大小，都是用动态分配内存方式。

Tasks

Task Name	Priority	Stack Size (Words)	Entry Function	Allocation
Task_ADC	osPriorityNormal	256	AppTask_ADC	Dynamic
Task_Info	osPriorityLow	256	AppTask_Info	Dynamic

图中两个任务的栈空间大小并不是随意给出的，是在程序运行过程中通过统计栈空间的最高水位值，然后给出的一个比较安全的值。在运行多任务程序时，有时程序没有任何错误，但是FreeRTOS就是无法进行任务调度，这通常是因为任务的栈空间大小不够引起的，增大栈空间后就可排除错误。要设置合理的栈空间大小，最好运行时统计一下任务的最该水位值。

2.6.2 初始化代码

1. 主程序

```
int main(void)
{
    HAL_Init();    //HAL初始化
    SystemClock_Config(); //系统时钟初始化
    /* 初始化所有已配置外设 */
    MX_GPIO_Init();    //GPIO初始化, LED1引脚PF9初始化
    MX_FSMC_Init(); //FSMC LCD接口初始化
    MX_ADC1_Init(); //ADC1初始化

    /* USER CODE BEGIN 2 */
    TFTLCD_Init(); //LCD 初始化
    LCD_ShowString(10,10,(uint8_t *)"Demo2_2:FreeRTOS Task Utilities");
    /* USER CODE END 2 */

    osKernellInitialize();    //RTOS内核初始
    MX_FREERTOS_Init(); //FreeRTOS对象初始化
    osKernelStart();    //启动RTOS内核

    while (1)
    {
        }
}
```

2. 创建任务

两个任务都是动态分配内存

```
/* 任务Task_ADC的相关定义*/
```

```
osThreadId_t Task_ADCHandle;          //任务Task_ADC的句柄变量  
const osThreadAttr_t Task_ADC_attributes = { //任务Task_ADC的属性  
    .name = "Task_ADC",  
    .priority = (osPriority_t) osPriorityNormal,  
    .stack_size = 256 * 4 //栈存储空间大小, 单位: 字节  
};
```

```
/* 任务Task_Info的相关定义 */
```

```
osThreadId_t Task_InfoHandle;          //任务Task_Info的句柄变量  
const osThreadAttr_t Task_Info_attributes = { //任务Task_Info的属性  
    .name = "Task_Info",  
    .priority = (osPriority_t) osPriorityLow,  
    .stack_size = 256 * 4 //栈存储空间大小, 单位: 字节  
};
```

```
void MX_FREERTOS_Init(void)
```

```
{  
    /* 创建任务 Task_ADC */  
    Task_ADCHandle = osThreadNew(AppTask_ADC, NULL, &Task_ADC_attributes);  
  
    /*创建任务Task_Info */  
    Task_InfoHandle = osThreadNew(AppTask_Info, NULL, &Task_Info_attributes);  
}
```

2.6.3 程序功能的实现

1. 任务Task_ADC的功能实现

程序较长，看源程序

- ADC以轮询方式进行数据采集
- 使用了vTaskDelayUntil()函数，以保证ADC数据采集的周期是严格的500ms

2. 任务Task_Info的功能实现

程序较长，看源程序

(1) 使用函数vTaskGetInfo()获取一个任务的信息

先获取任务句柄，再获取任务信息

```
// TaskHandle_t taskHandle= xTaskGetCurrentTaskHandle(); //获取当前任务句柄
// TaskHandle_t taskHandle= xTaskGetIdleTaskHandle();    //获取空闲任务句柄
// TaskHandle_t taskHandle= xTaskGetHandle("Task_ADC"); //通过任务名称
TaskHandle_t taskHandle= Task_ADCHandle;    //直接使用任务句柄变量

vTaskGetInfo(taskHandle, &taskInfo, getFreeStackSpace, taskState);
```

函数返回的任务信息存储在结构体变量taskInfo里。

(2) 使用函数uxTaskGetStackHighWaterMark ()获取一个任务的高水位值

(3) 内核其他信息的获取

函数uxTaskGetNumberOfTasks()获取当前管理的任务数

函数vTaskList()可以获取管理的任务的列表信息，其返回结果是字符串

(4) 任务的删除

任务的主体是一个无限循环，在任务函数中不允许出现return语句。如果跳出了无限循环，需要在任务函数返回之前执行vTaskDelete(NULL)删除任务自己。

Demo2_2:Task Utilities

Task_ADC: ADC by polling

ADC Value(mV)= 1543

Task_Info: Show task info

Get by vTaskGetInfo()

Task Name= IDLE

Task Number= 3

Task State= 1

Task Priority= 0

High Water Mark= 118

High Water Mark of tasks

Idle Task= 118

Task_ADC= 136

Task_Info= 124

Kernel Info

uxTaskGetNumberOfTasks()= 4

Demo2_2:Task Utilities

Task_ADC: ADC by polling

ADC Value(mV)= 1539

Task_Info: Show task info

Get by vTaskGetInfo()

Task Name= IDLE

Task Number= 3

Task State= 1

Task Priority= 0

High Water Mark= 118

High Water Mark of tasks

Idle Task= 118

Task_ADC= 136

Task_Info= 124

Kernel Info

uxTaskGetNumberOfTasks()= 4

Task_Info is deleted

示例运行时的LCD显示

任务Task_Info执行一段时间后删除了自己

练习任务

1. 看教材，练习本章的示例。