

FreeRTOS学习笔记

本次学习并未深入学完FreeRTOS系统知识，只是学了几个重要和比赛所需的部分！！

1.FreeRTOS是什么？

- A.一款实时操作系统
- B.其内核支持抢占式，合作是和时间片调度
- C.系统小巧、简单、易用，通常占用内核4k-9k字节空间
- D.高移植性，适合C语言编写开发
- E.程序模块化，以任务形式调度，方便控制

话不多说，立马开始，冲冲冲！！

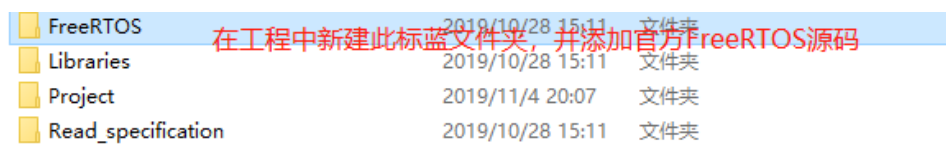
2.FreeRTOS的移植

我是以STM32F427板子为例，学习参考于[STM32F407 FreeRTOS开发手册V1.1.pdf](#)

A.新建工程 (build on Keil 5 5.28)

由于该工程模板应用于学习FreeRTOS，所以越简单越妙，方便移植检查错误。

B.移植FreeRTOS系统源码，向工程中添加相应文件



FreeRTOS	2019/10/28 15:11	文件夹
Libraries	2019/10/28 15:11	文件夹
Project	2019/11/4 20:07	文件夹
Read_specification	2019/10/28 15:11	文件夹

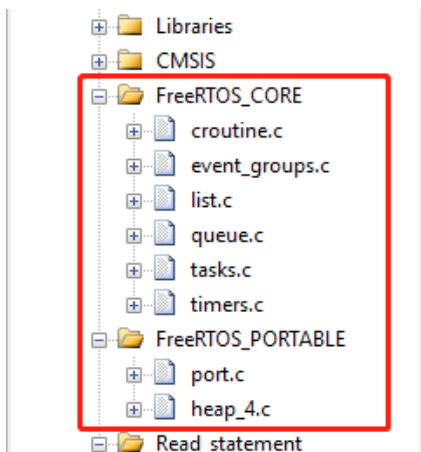
添加好源码后如下图所示

名称	修改日期	类型	大小
include	2019/10/28 15:11	文件夹	
portable	2019/10/28 15:11	文件夹	
croutine.c	2019/5/11 9:24	C Source	13 KB
event_groups.c	2019/5/11 9:24	C Source	27 KB
list.c	2019/5/11 9:24	C Source	9 KB
queue.c	2019/5/11 9:24	C Source	95 KB
readme.txt	2019/2/18 1:38	文本文档	1 KB
stream_buffer.c	2019/5/11 9:24	C Source	43 KB
tasks.c	2019/10/28 22:18	C Source	171 KB
timers.c	2019/5/11 9:24	C Source	40 KB

进入portable，删除一些文件，保留以下三个文件夹和一个readme.txt文件

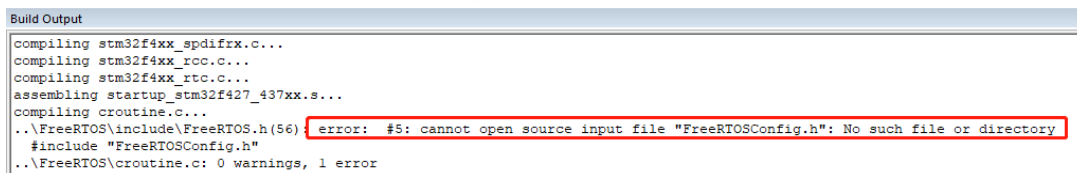


打开Keil 5 工程并添加相应文件夹和文件（基础不再展开）



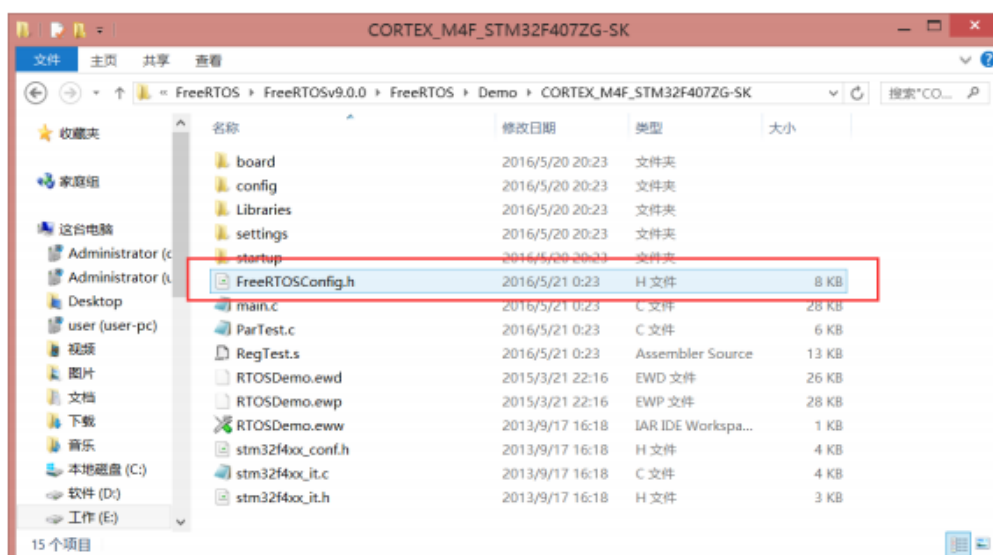
注意：添加头文件路径必须到子文件夹并且编译一下看是否有错误！（答案是肯定的，不过一般不会是你的锅，若有错往下看进行修改！）

接下来你会发现打不开“FreeRTOSConfig.h”的文件（一个FreeRTOS配置文件）



这是因为缺少了这一个头文件在源码里，当然我们可以去源码库里找出来。

显然在CORTEX_M4F_STM32F407ZG-SK文件夹里找到了它



接下来我们将它 Ctrl+C 再打开我们工程里的FreeRTOS的include文件夹中Ctrl + V 即可，如图

此电脑 > 桌面 > FreeRTOS试行 > FreeRTOS > include

名称	修改日期	类型	大小
croutine.h	2019/5/11 9:24	C/C++ Header	26 KB
deprecated_definitions.h	2019/5/11 9:24	C/C++ Header	8 KB
event_groups.h	2019/5/11 9:24	C/C++ Header	30 KB
FreeRTOS.h	2019/5/11 9:24	C/C++ Header	43 KB
FreeRTOSConfig.h	2019/11/1 20:30	C/C++ Header	7 KB
list.h	2019/5/11 9:24	C/C++ Header	19 KB
message_buffer.h	2019/5/13 10:58	C/C++ Header	38 KB
mpu_prototypes.h	2019/5/11 9:24	C/C++ Header	14 KB
mpu_wrappers.h	2019/5/11 9:24	C/C++ Header	10 KB
portable.h	2019/5/11 9:24	C/C++ Header	7 KB
projdefs.h	2019/5/11 9:24	C/C++ Header	6 KB
queue.h	2019/5/11 9:24	C/C++ Header	61 KB
semphr.h	2019/5/11 9:24	C/C++ Header	48 KB
stack_macros.h	2019/5/11 9:24	C/C++ Header	6 KB
StackMacros.h	2019/5/11 9:24	C/C++ Header	7 KB

同样回到工程再编译一次，再次检查存在的错误

```
compiling port.c...
..\FreeRTOS\portable\RVDS\ARM_CM4F\port.c(712): error: #20: identifier "SystemCoreClock" is undefined
    portNVIC_SYSTICK_LOAD_REG = ( configSYSTICK_CLOCK_HZ / configTICK_RATE_HZ ) - 1UL;
..\FreeRTOS\portable\RVDS\ARM_CM4F\port.c: 0 warnings, 1 error
compiling timers.c...
compiling heap_4.c...
".\Objects\RoboMaster.axf" - 1 Error(s), 0 Warning(s).
```

发现没有定义Systemclock，接下来打开“FreeRTOSConfig.h”的文件

发现了原因

```
/* Ensure stdint is only used by the compiler, and not the assembler. */
#if defined ( __ICCARM__ )
    #include <stdint.h>
    extern uint32_t SystemCoreClock;
#endif

/*
FreeRTOS基础配置选项
*/
```

该语句不符合条件忽略编译了

修改为这样

```
/* Ensure stdint is only used by the compiler, and not the assembler. */
#if defined ( __ICCARM__ ) || defined ( __CC_ARM ) || defined ( __GNUC__ )
    #include <stdint.h>
    extern uint32_t SystemCoreClock;
#endif

/*
FreeRTOS基础配置选项
*/
```

再次编译会发现还有一些小错误

```
oboMaster.axf: Error: L6200E: Symbol PendSV_Handler multiply defined (by port.o and stm32f4xx_it.o).
oboMaster.axf: Error: L6200E: Symbol SVC_Handler multiply defined (by port.o and stm32f4xx_it.o).
oboMaster.axf: Error: L6200E: Symbol SysTick_Handler multiply defined (by bsp_delay.o and stm32f4xx_it.o).
information to list image symbols.
information to list load addresses in the image map.
```

这是由于port.c和stm32f4xx_it.c中都定义了这三个中断函数，所以我们需要把stm32f4xx_it.c中的屏蔽掉即可

```

    */
    //void SVC_Handler(void)
    //{
}

/**
 * @brief This function handles PendSV_Handler exception.
 * @param None
 * @retval None
 */
//void PendSV_Handler(void)
//{}

/**
 * @brief This function handles SysTick_Handler.
 * @param None
 * @retval None
 */
//void SysTick_Handler(void)
//{}

```

继续编译，还是有错误

```

linking...
.\Objects\RoboMaster.axf: Error: L6218E: Undefined symbol vApplicationIdleHook (referred from tasks.o).
.\Objects\RoboMaster.axf: Error: L6218E: Undefined symbol vApplicationTickHook (referred from tasks.o).
.\Objects\RoboMaster.axf: Error: L6218E: Undefined symbol vApplicationMallocFailedHook (referred from heap_4.o).
Not enough information to list image symbols.
Not enough information to list load addresses in the image map.
Finished: 2 information, 0 warning and 3 error messages.

```

三个钩子函数我们并没有用到，我们只需在FreeRTOSConfig.h中将其的宏定义由1改为0即可

```

#define configUSE_PREEMPTION 1 //1使用抢占式内核，0不使用
#define configCPU_CLOCK_HZ ( SystemCoreClock ) //CPU频率
#define configTICK_RATE_HZ ( ( TickType_t ) 1000 ) //时钟节拍频率，这里用1000
#define configMAX_PRIORITIES ( 5 ) //可使用最大优先级
#define configMINIMAL_STACK_SIZE ( ( unsigned short ) 130 ) //空闲任务堆栈大小
#define configMAX_TASK_NAME_LEN ( 16 ) //任务名字字符串长度
#define configUSE_16_BIT_TICKS 0 //系统节拍计数器变量类型
#define configIDLE_SHOULD_YIELD 1 //为1时空闲任务放弃CPU
#define configUSE_MUTEXES 1 //为1时使用互斥信号量
#define configQUEUE_REGISTRY_SIZE 8 //不为0时表示启用队列

#define configCHECK_FOR_STACK_OVERFLOW 0 //大于0时启用堆栈溢出检查
//用户必须提供一个栈溢出钩子函数
//此值可以为1或者2，1表示每次检查，2表示每2次检查一次
#define configUSE_RECURSIVE_MUTEXES 1 //为1时使用递归互斥信号量
#define configUSE_MALLOC_FAILED_HOOK 0 //1使用内存申请失败钩子函数
#define configUSE_APPLICATION_TASK_TAG 0 //为1时使用任务应用任务标签
#define configUSE_COUNTING_SEMAPHORES 1 //为1时使用计数信号量

#define configUSE_TIME_SLICING 1
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 1

#define configUSE_IDLE_HOOK 0
#define configUSE_TICK_HOOK 0

```

至此，移植FreeRTOS基本完成!!!

3.FreeRTOS的部分文件修改

A.System中sys.h的修改

将此宏定义由0改为1

```

#include "stm32f4xx.h"

//0,不支持ucos
//1,支持ucos
#define SYSTEM_SUPPORT_OS 1 //定义系统文件夹是否支持ucos

```

B.Basis中bsp_delay.c的修改

我采用了条件编译加入了FreeRTOS所需的代码段并且屏蔽了之前的代码段仿真影响FreeRTOS的运行

```

#elif Delay_Mode == freertos
#include "delay.h"
#include "sys.h"

#if SYSTEM_SUPPORT_OS
    #include "FreeRTOS.h"          //支持FreeRTOS时, 使用
    #include "task.h"
#endif

static u8 fac_us=0;                //us延时倍乘数
static u16 fac_ms=0;               //ms延时倍乘数,在os下,代表每个节拍的ms数
volatile uint32_t sysTickUptime = 0;

extern void xPortSysTickHandler(void);
//systick 中断服务函数,使用 os 时用到
void SysTick_Handler(void)
{
    sysTickUptime++;
    if(xTaskGetSchedulerState() != taskSCHEDULER_NOT_STARTED) //系统已经运行
    {
        xPortSysTickHandler();
    }
}

//初始化延迟函数
//SYSTICK 的时钟固定为 AHB 时钟,基础例程里面 SYSTICK 时钟频率为 AHB/8
//这里为了兼容 FreeRTOS, 所以将 SYSTICK 的时钟频率改为 AHB 的频率!
//SYSCLK:系统时钟频率
void Delay_init(u8 SYSCLK)
{
    u32 reload;
    //SysTick频率为HCLK
    SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK);
    fac_us=SYSCLK;                  //不论是否使用OS,fac_us都需要使用
    reload=SYSCLK;                  //每秒钟的计数次数 单位为M
    reload*=1000000/configTICK_RATE_HZ; //根configTICK_RATE_HZ定溢出时间
    //reload为24位寄存器,最大值:16777216,在168M下,约合0.0998s左右
    fac_ms=1000/configTICK_RATE_HZ; //代表os可以延时的最少单位
    SysTick->CTRL|=SysTick_CTRL_TICKINT_Msk; //开启SYSTICK中断
    SysTick->LOAD=reload;             //每1/delay_ostickspersec秒中断一次
    SysTick->CTRL|=SysTick_CTRL_ENABLE_Msk; //开启SYSTICK
}

//延时nus
//nus:要延时的us数.
//nus:0~204522252(最大值即2^32/fac_us@fac_us=168)

```

```

//nus:要延时的us数.
//nus:0~204522252(最大值即2^32/fac_us@fac_us=168)
void delay_us(u32 nus)
{
    u32 ticks;
    u32 told,tnow,tcnt=0;
    u32 reload=SysTick->LOAD; //LOAD 的值
    ticks=nus*fac_us; //需要的节拍数
    told=SysTick->VAL; //刚进入时的计数器值
    while(1)
    {
        tnow=SysTick->VAL;
        if(tnow!=told)
        {
            //这里注意一下 SYSTICK 是一个递减的计数器就可以了.
            if(tnow<told)tcnt+=told-tnow;
            else tcnt+=reload-tnow+told;
            told=tnow;
            if(tcnt>=ticks)break; //时间超过/等于要延迟的时间,则退出.
        }
    };
} //延时nms
//nms:要延时的ms数
//nms:0~65535
void delay_ms(u32 nms)
{
    if(xTaskGetSchedulerState()!=taskSCHEDULER_NOT_STARTED) //系统已经运行
    {
        if(nms>=fac_ms) //延时的时间大于 os 的最少时间周期
        {
            vTaskDelay(nms/fac_ms); //FreeRTOS 延时
        }
        nms%=fac_ms; //os 已经无法提供这么小的延时了,
        //采用普通方式延时
    }
    delay_us((u32)(nms*1000)); //普通方式延时
}
//延时 nms,不会引起任务调度
//nms:要延时的 ms 数
void delay_xms(u32 nms)
{
    u32 i;
    for(i=0;i<nms;i++) delay_us(1000);
}

#endif

```

C.Basis中bsp_usart.c的修改

下面是示例

usart.c 文件修改也很简单, usart.c 文件有两部分要修改, 一个是添加 FreeRTOS.h 头文件, 默认是添加的 UCOS 中的 includes.h 头文件, 修改以后如下:

```

//如果使用 os,则包括下面的头文件即可.
#if SYSTEM_SUPPORT_OS
#include "FreeRTOS.h" //os 使用
#endif

```

另外一个就是 USART1 的中断服务函数, 在使用 UCOS 的时候进出中断的时候需要添加 OSIntEnter()和 OSIntExit(), 使用 FreeRTOS 的话就不需要了, 所以将这两行代码删除掉, 修改以后如下:

```

void USART1_IRQHandler(void) //串口 1 中断服务程序
{
    u8 Res;
    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
    {
        Res =USART_ReceiveData(USART1); //(USART1->DR); //读取接收到的数据

        if((USART_RX_STA&0x8000)==0)//接收未完成
        {

```

```

        if(USART_RX_STA&0x4000)//接收到了 0x0d
        {
            if(Res!=0x0a)USART_RX_STA=0;//接收错误,重新开始
            else USART_RX_STA|=0x8000; //接收完成了
        }
        else //还没收到 0X0D
        {
            if(Res==0x0d)USART_RX_STA|=0x4000;
            else
            {
                USART_RX_BUF[USART_RX_STA&0X3FFF]=Res ;
                USART_RX_STA++;
                if(USART_RX_STA>(USART_REC_LEN-1))USART_RX_STA=0;
            }
        }
    }
}
}

```

D.在FreeRTOSConfig.h中

```

152 to all Cortex-M ports, and do not rely on any particular library functions. */
153 #define configKERNEL_INTERRUPT_PRIORITY    ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY) )
154 /* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!!
155 See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
156 #define configMAX_SYSCALL_INTERRUPT_PRIORITY    ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 - configLIBRARY_LOWEST_INTERRUPT_PRIORITY) )
157
158 /* Normal assert() semantics without relying on the provision of an assert.h
159 header file. */
160 #define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS(); for( ;; ); }
161
162 /* Definitions that map the FreeRTOS port interrupt handlers to their CMSIS
163 standard names. */
164 #define vPortSVCHandler SVC_Handler
165 #define xPortPendSVHandler PendSV_Handler
166 #define xPortSysTickHandler SysTick_Handler
167
168 #endif /* FREERTOS_CONFIG_H */
169
170

```

屏蔽掉此行代码

完成上述步骤，就可以开始FreeRTOS真正的学习了！

4.FreeRTOS动态任务的创建与删除

动态和静态的区别在于动态会自动为我们分配好内存，而静态任务需要我们自己手动分配内存

A.首先我们要对一些函数进行初始化（例如前面修改过的延时函数）


```

void Function_Init(void)
{
    /***** 中断优先级配置 *****/
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4); //配置系统
    /*****BSP层配置*****/
    Delay_init(168); //配置时钟
    BSP_LED_Config(); //LED初始化
    CAN1_QuickInit(); //CAN1初始化
    BSP_CAN2_Config(); //CAN2初始化
    //BSP_TIM6_Config(89,9999); //TIM6初始化
    //BSP_TIM7_Config(899,9999); //TIM7初始化
    BSP_KEY_Config(); //按键初始化
    BSP_USART2_Config(115200); //串口2初始化
    delay_ms(2000);
    /*****应用模块配置*****/
    // Chassis_Init(); //底盘初始化
    // DR16_receiverInit(); //接收机初始化
    // GM6020_Config(); //GM6020电机初始化
    // Gyroscope_Init(); //陀螺仪配置初始化
    // JudgeSystem_Init(); //裁判系统初始化
}

void Function_Text(void)
{
    /*****上位机收发数据*****/
}

```

一般中断优先级组默认设为组4（4位优先级全为抢占优先级），配置好时钟和按键并在 **Function_Init ()** 中完成初始化

回到**main.c**

```

int main(void)
{
    Function_Init();
    //所有函数初始化参数
    App_Task_Create(); //任务创建函数
    vTaskStartScheduler();
    //开启任务调度
    while(1)
    {
        Function_Text();
        //检测设备收发信息情况
    }
}

```

我在这包装了一下创建函数，FreeRTOS的创建任务函数放在了在这个函数里面

进入这个函数里面

首先认识下**xTaskCreat**，这是一个FreeRTOS提供的任务创建函数，该函数内有**任务的名称**
任务的函数，**任务的堆栈大小**，**传递给任务函数的参数**，**任务优先级**以及**任务句柄**。

即每次创建一个任务都需要这类信息来配置，同时也由此看出宏定义的必要性。

无论新建什么任务第一步都是**设置任务优先级**，**编写完任务句柄**同时**声明任务函数**，三者缺一不可，具体可跳入参考其定义。


```
#include "Task_Creat.h"

/**
 * @file Task_Creat.c
 * @author Wu Guoxi
 * @version FreeRTOS V1.0
 * @date October 28th
 * @brief 任务创建函数
 */
```

```
//任务创建
#define START_TASK_PRIO 11 //任务优先级
TaskHandle_t StartTask_Handler; //任务句柄
void Start_Task(void *pvParameters); //任务函数

/*LED*/
#define TASK_LED_PRIO 1 //任务优先级
TaskHandle_t Task_LED_Handler; //任务句柄

/*KEY*/
#define TASK_KEY_PRIO 2 //任务优先级
TaskHandle_t Task_KEY_Handler; //任务句柄
```

第一步：设置任务优先级
任务句柄
声明任务函数

```
void App_Task_Create(void)
{
    xTaskCreate((TaskFunction_t )Start_Task, //任务函数
               (const char* )"Start_Task", //任务名称
               (uint16_t )STK_SIZE_128, //任务堆栈大小
               (void* )NULL, //传递给任务函数的参数
               (UBaseType_t )START_TASK_PRIO, //任务优先级
               (TaskHandle_t* )&StartTask_Handler); //任务句柄
}
```

第二步：编写用于创建任务的任务函数

注意：任务优先级不能选0或者（32-1）！！

现在通过点灯来检验一下任务是否可行，在创建任务函数的这一任务中（有点绕，可以理解为创建任务也是一个任务）创建一个LED任务。如图所示

```
/*-----*/
//创建 LED 任务
xTaskCreate((TaskFunction_t )Task_LED
            (const char* )"Task_LED",
            (uint16_t )STK_SIZE_128,
            (void* )NULL,
            (UBaseType_t )TASK_LED_PRIO,
            (TaskHandle_t* )&Task_LED_Handler);
/*-----*/
```

红圈所圈起的就是LED的具体运作情况的函数，如下

```
#include "Task_LED.h"

/**
 * @file Task_LED.c
 * @author Wu Guoxi
 * @version FreeRTOS V1.0
 * @date October 28th
 * @brief LED任务函数
 */

void Task_LED(void *pvParameters)
{
    while(1)
    {
        LED_G_ON;
        LED_R_OFF;
        LED_1_TOGGLE;
        LED_3_TOGGLE;
        LED_5_TOGGLE;
        LED_7_TOGGLE;
        vTaskDelay(250);
        LED_R_TOGGLE;
        vTaskDelay(250);
    }
}
```

可以看到这里面有个**vTaskDelay ()** 函数（后面章节再做详细解释），由于FreeRTOS添加的代码段需要放入死循环内，所以加入系统所给的任务延时函数，防止程序卡在死循环里，**一般放在任务结束末尾**。

注：它和**delay_ms**等函数有一个重大区别，前者每一次延时都会调用到FreeRTOS中的其他函数，而**delay_ms**不会调用其他系统函数。总的来说，最好只在程序末尾调用**vTaskDelay ()** 函数，其余时候尽量用**delay_ms**函数，一样能达到精确延时的效果。

B.接下来介绍下**vTaskDelete ()** 函数，显然这是个删除任务的函数，其是以任务句柄为指引对任务进行删除，是将一个任务函数从任务列表删除，释放内存，被删掉的任务函数则不能再被调用!!!

5.FreeRTOS动态任务的挂起与恢复

任务挂起和恢复API函数如下：

函数	描述
vTaskSuspend()	挂起一个任务。
vTaskResume()	恢复一个任务的运行。
xTaskResumeFromISR()	中断服务函数中恢复一个任务的运行。

vTaskSuspend(任务句柄 / NULL)：挂起该任务句柄的函数 / 当前函数

vTaskResume(任务句柄)：恢复该任务句柄的函数

挂起和删除最大的区别就是前者并不删除任务，只是中断任务，暂时不执行该任务，随后还是可以通过恢复函数让任务继续运行。

下面是一个挂起LED任务的按键任务作为例子

```
void Task_KEY(void *pvParameters)
{
    while(1)
    {
        if (KEY_SCAN(GPIOB,GPIO_Pin_2) == SET)
        {
            LED_R_OFF;
            vTaskSuspend(Task_LED_Handler);
        }
        LED_G_TOGGLE;
        vTaskDelay(1000);
    }
}
```

6.FreeRTOS中断配置和临界段

6.1 优先级分组定义

表 4.1.3.1 中 PRIGROUP 就是优先级分组，它把优先级分为两个位段：MSB 所在的位段(左边的)对应抢占优先级，LSB 所在的位段(右边的)对应亚优先级，如表 4.1.3.2 所示。

分组位置	表达抢占优先级的位段	表达亚优先级的位段
0(默认)	[7:1]	[0:0]
1	[7:2]	[1:0]
2	[7:3]	[2:0]
3	[7:4]	[3:0]
4	[7:5]	[4:0]
5	[7:6]	[5:0]
6	[7:7]	[6:0]
7	无	[7:0]

表 4.1.3.2 抢占优先级和亚优先级的表达，位数与分组位置的关系

在看一下 STM32 的优先级分组情况，我们前面说了 STM32 使用了 4 位，因此最多有 5 组优先级分组设置，这 5 个分组在 msic.h 中有定义，如下：

```
#define NVIC_PriorityGroup_0      ((uint32_t)0x700) /*!< 0 bits for pre-emption priority
4 bits for subpriority */
#define NVIC_PriorityGroup_1      ((uint32_t)0x600) /*!< 1 bits for pre-emption priority
3 bits for subpriority */
#define NVIC_PriorityGroup_2      ((uint32_t)0x500) /*!< 2 bits for pre-emption priority
2 bits for subpriority */
#define NVIC_PriorityGroup_3      ((uint32_t)0x400) /*!< 3 bits for pre-emption priority
1 bits for subpriority */
#define NVIC_PriorityGroup_4      ((uint32_t)0x300) /*!< 4 bits for pre-emption priority
0 bits for subpriority */
```

可以看出 STM32 有 5 个分组，但是一定要注意！STM32 中定义的分组 0 对应的值是 7！如果我们选择分组 4，即 NVIC_PriorityGroup_4 的话，那 4 位优先级就都全是抢占优先级了，没有亚优先级，那么就有 0~15 共 16 个优先级。而移植 FreeRTOS 的时候我们配置的就是组 4，如图 4.1.3.2 所示：

```
52
53 int main(void)
54 {
55     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4); //设置系统中断优先级分组4
56     delay_init(168); //初始化延时函数
57     uart_init(115200); //初始化串口
58     LED_Init(); //初始化LED端口
59
60     //创建开始任务
61     xTaskCreate((TaskFunction_t) start_task, //任务函数
62                (const char*) "start_task", //任务名称
63                (uint16_t) START_STK_SIZE, //任务堆栈大小
64                (void*) NULL, //传递给任务函数的参数
65                (UBaseType_t) START_TASK_PRIO, //任务优先级
66                (TaskHandle_t*) &StartTask_Handler); //任务句柄
67     vTaskStartScheduler(); //开启任务调度
68 }
69
```

图 4.1.3.2 优先级分组配置

6.2 优先级设置

每个外部中断都有一个对应的优先级寄存器，每个寄存器占 8 位，因此最大宽度是 8 位，但是最小为 3 位。4 个相临的优先级寄存器拼成一个 32 位寄存器。如前所述，根据优先级组的设置，优先级又可以分为高、低两个位段，分别抢占优先级和亚优先级。STM32 我们已经设置位组 4，所以就只有抢占优先级了。优先级寄存器都可以按字节访问，当然也可以按半字/字来访问，有意义的优先级寄存器数目由芯片厂商来实现，如表 4.1.4.1 和 4.1.4.2 所示：

名称	类型	地址	复位值	描述
PRI_0	R/W	0xE000_E400	0(8 位)	外中断#0 的优先级
PRI_1	R/W	0xE000_E401	0(8 位)	外中断#1 的优先级
⋮	⋮	⋮	⋮	⋮
PRI_239	R/W	0xE000_E4EF	0(8 位)	外中断#239 的优先级

表 4.1.4.1 中断优先级寄存器阵列(地址：0xE000_E400~0xE000_E4EF)

名称	类型	地址	复位值	描述
PRI_4		0xE000_ED18		存储管理 fault 的优先级
PRI_5		0xE000_ED19		总线 fault 的优先级
PRI_6		0xE000_ED1A		用法 fault 的优先级
-	-	0xE000_ED1B	-	-
-	-	0xE000_ED1C	-	-
-	-	0xE000_ED1D	-	-
-	-	0xE000_ED1E	-	-

PRI_11		0xE000_ED1F		SVC 优先级
PRI_12		0xE000_ED20		调试监视器的优先级
-	-	0xE000_ED21	-	-
PRI_14		0xE000_ED22		PendSV 的优先级
PRI_15		0xE000_ED23		SysTick 的优先级

表 4.1.4.2 系统异常优先级阵列(地址：0xE000_ED18~0xE000_ED23)

上面说了，4 个相临的寄存器可以拼成一个 32 位的寄存器，因此地址 0xE000_ED20~0xE000_ED23 这四个寄存器就可以拼接成一个地址为 0xE000_ED20 的 32 位寄存器。这一点很重要！因为 FreeRTOS 在设置 PendSV 和 SysTick 的中断优先级的时候都是直接操作的地址 0xE000_ED20。

6.3用于中断屏蔽的特殊寄存器

A、PRIMASK和FAULTMASK寄存器

在许多应用中，需要暂时屏蔽所有的中断—执行一些对时序要求严格的任务，这个时候就可以使用 PRIMASK 寄存器，PRIMASK 用于禁止除 NMI 和 HardFault 外的所有异常和中断，汇编编程的时候可以使用 CPS(修改处理器状态)指令修改 PRIMASK 寄存器的数值：

```
CPSIE    I;    //清除 PRIMASK(使能中断)
CPSID    I;    //设置 PRIMASK(禁止中断)
```

PRIMASK 寄存器还可以通过 MRS 和 MSR 指令访问，如下：

```
MOVS     R0, #1
MSR      PRIMASK, R0 ;//将 1 写入 PRIMASK 禁止所有中断
```

以及：

```
MOVS     R0, #0
MSR      PRIMASK, R0 ;//将 0 写入 PRIMASK 以使能中断
```

UCOS 中的临界区代码保护就是通过开关中断实现的(UCOSIII 也可以使用禁止任务调度的方法来实现临界区代码保护，这里不讨论这种情况)，而开关中断就是直接操作 PRIMASK 寄存器的，所以在 UCOS 中关闭中断的时候关闭了除复位、NMI 和 HardFault 以外的所有中断！

FAULTMASK 比 PRIMASK 更狠，它可以连 HardFault 都屏蔽掉，使用方法和 PRIMASK 类似，FAULTMASK 会在退出时自动清零。

汇编编程的时候可以利用 CPS 指令修改 FAULTMASK 的当前状态：

```
CPSIE    F ;清除 FAULTMASK
CPSID    F ;设置 FAULTMASK
```

还可以利用 MRS 和 MSR 指令访问 FAULTMASK 寄存器：

```
MOVS     R0, #1
MSR      FAULTMASK, R0 ;//将 1 写入 FAULTMASK 禁止所有中断
```

以及：

```
MOVS     R0, #0
MSR      FAULTMASK, R0 ;//将 0 写入 FAULTMASK 使能中断
```

B、BASEPRI寄存器

PRIMASK 和 FAULTMASK 寄存器太粗暴了，直接关闭除复位、NMI 和 HardFault 以外的其他所有中断，但是在有些场合需要对中断屏蔽进行更细腻的控制，比如只屏蔽优先级低于某一个阈值的中断。那么这个作为阈值的优先级值存储在哪里呢？在 BASEPRI 寄存器中，不过如果向 BASEPRI 写 0 的话就会停止屏蔽中断。比如，我们要屏蔽优先级不高于 0X60 的中断，则可以使用如下汇编编程：

```
MOV      R0, #0X60
MSR      BASEPRI, R0
```

如果需要取消 BASEPRI 对中断的屏蔽，可以使用如下代码：

```
MOV      R0, #0
MSR      BASEPRI, R0
```

注意！FreeRTOS 的开关中断就是操作 BASEPRI 寄存器来实现的！它可以关闭低于某个阈值的中断，高于这个阈值的中断就不会被关闭！

6.4 FreeRTOS开关中断

FreeRTOS 开关中断函数为 portENABLE_INTERRUPTS()和 portDISABLE_INTERRUPTS()，这两个函数其实是宏定义，在 portmacro.h 中有定义，如下：

```
#define portDISABLE_INTERRUPTS()    vPortRaiseBASEPRI()
#define portENABLE_INTERRUPTS()     vPortSetBASEPRI(0)
```

```

static portFORCE_INLINE void vPortSetBASEPRI( uint32_t ulBASEPRI )
{
    __asm
    {
        msr basepri, ulBASEPRI
    }
}

/*-----*/

static portFORCE_INLINE void vPortRaiseBASEPRI( void )
{
    uint32_t ulNewBASEPRI = configMAX_SYSCALL_INTERRUPT_PRIORITY;

    __asm
    {
        msr basepri, ulNewBASEPRI
        dsb
        isb
    }
}

```

函数 `vPortSetBASEPRI()` 是向寄存器 `BASEPRI` 写入一个值，此值作为参数 `ulBASEPRI` 传递进来，`portENABLE_INTERRUPTS()` 是开中断，它传递了个 0 给 `vPortSetBASEPRI()`，根据我们前面讲解 `BASEPRI` 寄存器可知，结果就是开中断。

函数 `vPortRaiseBASEPRI()` 是向寄存器 `BASEPRI` 写入宏 `configMAX_SYSCALL_INTERRUPT_PRIORITY`，那么优先级低于 `configMAX_SYSCALL_INTERRUPT_PRIORITY` 的中断就会被屏蔽！

6.5 临界段代码

临界段代码也叫做临界区，是指那些必须完整运行，不能被打断的代码段，比如有的外设的初始化需要严格的时序，初始化过程中不能被打断。FreeRTOS 在进入临界段代码的时候需要关闭中断，当处理完临界段代码以后再打开中断。FreeRTOS 系统本身就有很多的临界段代码，这些代码都加了临界段代码保护，我们在写自己的用户程序的时候有些地方也需要添加临界段代码保护。

FreeRTOS 与临界段代码保护有关的函数有 4 个：`taskENTER_CRITICAL()`、`taskEXIT_CRITICAL()`、`taskENTER_CRITICAL_FROM_ISR()` 和 `taskEXIT_CRITICAL_FROM_ISR()`，这四个函数其实是宏定义，在 `task.h` 文件中有定义。这四个函数的区别是前两个是任务级的临界段代码保护，后两个是中断级的临界段代码保护。

6.5.1 任务级临界段代码保护

taskENTER_CRITICAL()和 taskEXIT_CRITICAL()是任务级的临界代码保护,一个是进入临界段,一个是退出临界段,这两个函数是成对使用的,这函数的定义如下:

```
#define taskENTER_CRITICAL()    portENTER_CRITICAL()
#define taskEXIT_CRITICAL()    portEXIT_CRITICAL()
```

而 portENTER_CRITICAL()和 portEXIT_CRITICAL()也是宏定义,在文件 portmacro.h 中有定义,如下:

```
#define portENTER_CRITICAL()    vPortEnterCritical()
#define portEXIT_CRITICAL()    vPortExitCritical()
```

```
void vPortEnterCritical( void )
{
    portDISABLE_INTERRUPTS();
    uxCriticalNesting++;

    if( uxCriticalNesting == 1 )
    {
        configASSERT( ( portNVIC_INT_CTRL_REG & portVECTACTIVE_MASK ) == 0 );
    }
}

void vPortExitCritical( void )
{
    configASSERT( uxCriticalNesting );
    uxCriticalNesting--;
    if( uxCriticalNesting == 0 )
    {
        portENABLE_INTERRUPTS();
    }
}
```

可以看出在进入函数 vPortEnterCritical()以后会首先关闭中断,然后给变量 uxCriticalNesting 加一, uxCriticalNesting 是个全局变量,用来记录临界段嵌套次数的。函数 vPortExitCritical()是退出临界段调用的,函数每次将 uxCriticalNesting 减一,只有当 uxCriticalNesting 为 0 的时候才会调用函数 portENABLE_INTERRUPTS()使能中断。这样保证了在有多多个临界段代码的时候不会因为某一个临界段代码的退出而打乱其他临界段的保护,只有所有的临界段代码都退出以后才会使能中断!

任务级临界代码保护使用方法如下:

```
void taskcritical_test(void)
{
    while(1)
    {
        taskENTER_CRITICAL();
        total_num+=0.01f;
        printf("total_num 的值为: %.4f\r\n",total_num);
        taskEXIT_CRITICAL();
        vTaskDelay(1000);
    }
}
```

(1)、进入临界区。

(2)、退出临界区。

(1)和(2)中间的代码就是临界区代码,注意临界区代码一定要精简!因为进入临界区会关闭中断,这样会导致优先级低于 configMAX_SYSCALL_INTERRUPT_PRIORITY 的中断得不到及时的响应!

6.5.2中断级临界段代码保护

函数taskENTER_CRITICAL_FROM_ISR和taskEXIT_CRITICAL_FROM_ISR中断级别临界段代码保护，是用在中断服务程序中的，而且这个中断的优先级一定要低于configMAX_SYSCALL_INTERRUPT_PRIORITY!原因前面已经说了。这两个函数在文件task.h中有如下定义：

```
#define taskENTER_CRITICAL_FROM_ISR() portSET_INTERRUPT_MASK_FROM_ISR()
#define taskEXIT_CRITICAL_FROM_ISR( x ) portCLEAR_INTERRUPT_MASK_FROM_ISR( x )
```

接着找 portSET_INTERRUPT_MASK_FROM_ISR() 和 portCLEAR_INTERRUPT_MASK_FROM_ISR()，这两个在文件 portmacro.h 中有如下定义：

```
#define portSET_INTERRUPT_MASK_FROM_ISR()    ulPortRaiseBASEPRI()
#define portCLEAR_INTERRUPT_MASK_FROM_ISR(x)  vPortSetBASEPRI(x)
```

vPortSetBASEPRI()前面已经讲解了，就是给 BASEPRI 寄存器中写入一个值。
函数 ulPortRaiseBASEPRI()在文件 portmacro.h 中定义的，如下：

```
static portFORCE_INLINE uint32_t ulPortRaiseBASEPRI( void )
{
    uint32_t ulReturn, ulNewBASEPRI = configMAX_SYSCALL_INTERRUPT_PRIORITY;

    __asm
    {
        mrs ulReturn, basepri                (1)
        msr basepri, ulNewBASEPRI            (2)
        dsb
        isb
    }

    return ulReturn;                          (3)
}
```

- (1)、先读出 BASEPRI 的值，保存在 ulReturn 中。
- (2)、将 configMAX_SYSCALL_INTERRUPT_PRIORITY 写入到寄存器 BASEPRI 中。
- (3)、返回 ulReturn，退出临界区代码保护的时候要使用到此值！

中断级临界代码保护使用方法如下：

```
//定时器 3 中断服务函数
void TIM3_IRQHandler(void)
{
    if(TIM_GetITStatus(TIM3,TIM_IT_Update)==SET) //溢出中断
    {
        status_value=taskENTER_CRITICAL_FROM_ISR();    (1)
        total_num+=1;
        printf("float_num 的值为: %d\r\n",total_num);
        taskEXIT_CRITICAL_FROM_ISR(status_value);      (2)
    }
    TIM_ClearITPendingBit(TIM3,TIM_IT_Update); //清除中断标志位
}
```

- (1)、进入临界区。
- (2)、退出临界区。