

FreeRTOS on AT32 MCU

前言

本指导手册描述了如何在AT32F系列MCU上使用FreeRTOS。FreeRTOS是一款开源的嵌入式实时操作系统，目前在各种嵌入式应用中应用广泛。本指导手册包括FreeRTOS系统移植、FreeRTOS内核服务讲解、综合Demo三个方面展开，全方位的讲解了FreeRTOS的使用，并且各个章节配套有对应的例程源程序，极大的方便初学者对AT32F系列MCU和FreeRTOS配合使用的深入理解。

本指导手册也描述了怎样充分的利用FreeRTOS的功能，以及如何搭配AT32的外设实现想要的功能。

使用本指导手册时，需配合配套的例程和FreeRTOS的官方指导手册，以便更深入的理解FreeRTOS。

注：本例程对应的代码是基于雅特力提供的V2.x.x 板级支持包（BSP）而开发，对于其他版本BSP，需要注意使用上的区别。

支持型号列表：

支持型号	AT32 全系列
------	----------

目录

1	FreeRTOS 简介	10
2	在 AT32 上移植 FreeRTOS.....	12
2.1	移植 FreeRTOS	12
2.2	例程介绍	14
3	FreeRTOS 调试方法	18
3.1	系统配置	18
3.2	例程介绍	19
4	FreeRTOS 中断优先级管理	21
4.1	AT32 中断配置	21
4.2	FreeRTOS 中断配置	22
4.3	中断优先级和任务优先级区别	23
4.4	临界段保护	23
4.5	例程介绍	24
5	FreeRTOS 任务管理	29
5.1	裸机与带 RTOS 的区别	29
5.2	FreeRTOS 任务状态	30
5.3	FreeRTOS 空闲任务	31
5.4	FreeRTOS 任务相关函数	31
5.5	例程介绍	34
6	FreeRTOS 任务调度	39
6.1	合作式调度	39
6.2	抢占式调度	39
6.3	时间片式调度	40

6.4	例程介绍	41
7	FreeRTOS 消息队列	47
7.1	消息队列介绍	47
7.2	消息队列相关 API	48
7.3	例程介绍	51
8	FreeRTOS 信号量	58
8.1	什么是信号量	58
8.2	二值信号量	58
8.2.1	二值信号量介绍	58
8.2.2	二值信号量 API	59
8.2.3	例程介绍	61
8.3	计数型信号量	66
8.3.1	计数型信号量介绍	66
8.3.2	计数型信号量 API	66
8.3.3	例程介绍	68
8.4	互斥信号量	72
8.4.1	优先级翻转	72
8.4.2	互斥信号量介绍	73
8.4.3	互斥信号量 API	74
8.4.4	例程介绍	76
8.5	递归互斥信号量	82
8.5.1	递归互斥信号量介绍	82
8.5.2	递归互斥信号量 API	82
8.5.3	例程介绍	84
9	FreeRTOS 事件标志组	89
9.1	事件标志组介绍	89
9.2	事件标志组 API	89
9.3	例程介绍	91

10	FreeRTOS 软件定时器组	98
10.1	软件定时器组介绍	98
10.2	软件定时器组 API	100
10.3	例程介绍	102
11	FreeRTOS 低功耗模式	108
11.1	Tickless 机制介绍	108
11.2	例程介绍	110
12	FreeRTOS 内存管理方式	116
12.1	内存管理方式一	116
12.2	内存管理方式二	117
12.3	内存管理方式三	121
12.4	内存管理方式四	122
12.5	内存管理方式五	127
13	FreeRTOS 流缓存	131
13.1	流缓存介绍	131
13.2	流缓存 API	131
13.3	例程介绍	133
14	FreeRTOS 消息缓存	138
14.1	消息缓存介绍	138
14.2	消息缓存 API	138
14.3	例程介绍	140
15	FreeRTOS 任务通知	146
15.1	任务通知介绍	146
15.2	任务通知 API	146
15.3	例程介绍	147

16	FreeRTOS 综合 Demo 演示	153
16.1	Demo 功能简介.....	153
16.2	例程演示.....	153
17	版本历史	155

表目录

表 1. 中断优先级分组	21
表 2. 临界区 API	24
表 3. 动态创建任务函数原型	32
表 4. 静态创建任务函数原型	32
表 5. 任务删除函数原型	33
表 6. 任务挂起函数原型	33
表 7. 任务解除挂起函数原型	33
表 8. 任务延时函数原型	34
表 9. 消息队列 API	48
表 10. 创建消息队列函数原型	49
表 11. 发送消息函数原型	49
表 12. 发送消息函数原型（中断级）	50
表 13. 接受消息函数原型	50
表 14. 接受消息函数原型（中断级）	50
表 15. 二值信号量 API	59
表 16. 创建互斥信号量函数原型	60
表 17. 删除信号量函数原型	60
表 18. 释放信号量函数原型	60
表 19. 获取信号量函数原型	61
表 20. 计数型信号量 API	66
表 21. 创建二值信号量函数原型	67
表 22. 删除信号量函数原型	67
表 23. 释放信号量函数原型	67
表 24. 获取信号量函数原型	67
表 25. 获取信号量值函数原型	68
表 26. 互斥信号量 API	74
表 27. 创建互斥信号量函数原型	75
表 28. 删除信号量函数原型	75
表 29. 释放信号量函数原型	75
表 30. 获取信号量函数原型	76

表 31. 获取信号量值函数原型	76
表 32. 递归互斥信号量 API.....	82
表 33. 创建递归互斥信号量函数原型	83
表 34. 释放信号量函数原型	83
表 35. 获取信号量函数原型	83
表 36. 事件标志组 API.....	90
表 37. 创建事件标志组函数原型.....	90
表 38. 设置事件标志组函数原型.....	90
表 39. 等待事件标志组函数原型.....	91
表 40. 软件定时器组 API	100
表 41. 创建软件定时器函数原型.....	100
表 42. 启动定时器函数原型	101
表 43. 关闭定时器函数原型	101
表 44. 获取定时器名字函数原型.....	102
表 45. 获取定时器 ID 函数原型.....	102
表 46. 设置定时器 ID 函数原型.....	102
表 47. 流缓存 API	131
表 48. 创建流缓存函数原型	132
表 49. 从流缓存接收数据函数原型	132
表 50. 发送数据到流缓存函数原型	132
表 51. 计数型信号量 API	138
表 52. 创建消息缓存函数原型	139
表 53. 从消息缓存接收数据函数原型	139
表 54. 发送数据到消息缓存函数原型	139
表 55. 任务通知 API.....	146
表 56. 文档版本历史	155

图目录

图 1. FreeRTOS 官网界面	12
图 2. FreeRTOS 源码目录	12
图 3. 工程目录	13
图 4. 头文件添加	13
图 5. 移植例程演示	17
图 6. 调试配置参数	18
图 7. 调试例程演示	20
图 8. AT32 NVIC 配置	22
图 9. AT32 外设中断配置	22
图 10. FreeRTOS 中断配置	22
图 11. FreeRTOS 中断管理例程演示	28
图 12. 裸机运行流程	29
图 13. RTOS 运行流程	30
图 14. 任务状态转换关系	31
图 15. 任务管理例程演示	38
图 16. FreeRTOS 官网合作式调度描述	39
图 17. 时间片调度	41
图 18. 时间片例程演示	46
图 19. 消息队列工作流程	47
图 20. 消息队列例程演示	56
图 21. 消息队列例程演示	57
图 22. 二值信号量框图 1	59
图 23. 二值信号量框图 2	59
图 24. 二值信号量框图 3	59
图 25. 二值信号量例程演示	65
图 26. 二值信号量例程演示	65
图 27. 计数型信号量例程演示	72
图 28. 优先级翻转	73
图 29. 互斥信号量解决优先级翻转	74
图 30. 互斥信号量例程演示	82

图 31. 递归互斥信号量例程演示.....	88
图 32. 事件标志组框图	89
图 33. 事件标志组例程演示	97
图 34. 单次模式 VS 周期性模式	98
图 35. 软件定时器 Daemon Task.....	99
图 36. 软件定时器例程演示	107
图 37. 低功耗模式例程演示	115
图 38. 流缓存例程演示	137
图 39. 消息缓存例程演示.....	145
图 40. 任务通知例程演示.....	152
图 41. 综合例程演示 1	153
图 42. 综合例程演示 2	154

1 FreeRTOS 简介

简介

在嵌入式领域，嵌入式实时操作系统正得到越来越广泛的应用。采用嵌入式操作系统（RTOS）可以更合理、更高效的利用CPU的资源，简化应用软件的设计，缩短系统开发时间，更好的保证系统的实时性和可靠性。

FreeRTOS是一个轻量级的实时操作系统内核。作为一个轻量级的操作系统，功能包括：任务管理、时间管理、信号量、消息队列、内存管理、记录功能、软件定时器、协程等，可基本满足较小系统的需求。由于RTOS需要占用一定系统资源（尤其是RAM资源），只有UCOSII/III、embOS、RTT、FreeRTOS等少数实时操作系统能在小RAM的MCU上运行。相对于UCOSII/III和embOS等商用操作系统，FreeRTOS是免费开源的操作系统，具有源码公开、可移植、可剪裁、调度策略灵活等特点，可以方便的移植到MCU上运行。

功能和特点

FreeRTOS主要功能和特点如下：

- 用户可配置内核功能
- 多平台的支持
- 提供一个高层次的信任代码的完整性
- 目标代码小、简单易用
- 遵循MISRA-C标准编程规范
- 强大的执行跟踪功能
- 堆栈溢出检测
- 没有限制的任务数量和优先级
- 多个任务可以分配相同的优先权
- 队列、二进制信号量、计数信号量、递归通信和同步的任务
- 优先级继承
- 免费开源的源代码

系统功能

作为一个轻量级的操作系统，功能包括：任务管理、时间管理、信号量、消息队列、内存管理、记录功能、软件定时器、协程等，可基本满足较小系统的需求。FreeRTOS内核支持优先级调度算法，不同任务可根据重要程度的不同被赋予一定的优先级，CPU总是让处于任务就绪、优先级最高的任务运行。FreeRTOS同样支持时间片轮转调度算法，系统允许不同的任务处于同一优先级下，在没有更高优先级任务就绪的情况下，同一优先级的任务共享系统资源。

FreeRTOS内核可根据需要设置为可剥夺型内核和不可剥夺型内核。当FreeRTOS配置成可剥夺型内核时，处于就绪态的高优先级任务能剥夺低优先级任务的CPU使用权，这样提高了系统的实时性；当FreeRTOS配置成不可剥夺型内核时，处于就绪态的高优先级任务只能等当前任务主动释放CPU使用权才能获得运行，这样提高了系统的运行效率。

结论

在嵌入式领域FreeRTOS是不多的同时具备实时性、开源性、可靠性、易用性、多平台支持等特点的嵌入式操作系统，FreeRTOS已发展到支持包含X86、Xilinx、Altera等多达30种硬件平台，其广泛的应用场景已经越来越受业界人事的关注。

版本发布

当前最新版本

FreeRTOS V10.2.1 2019-05-13

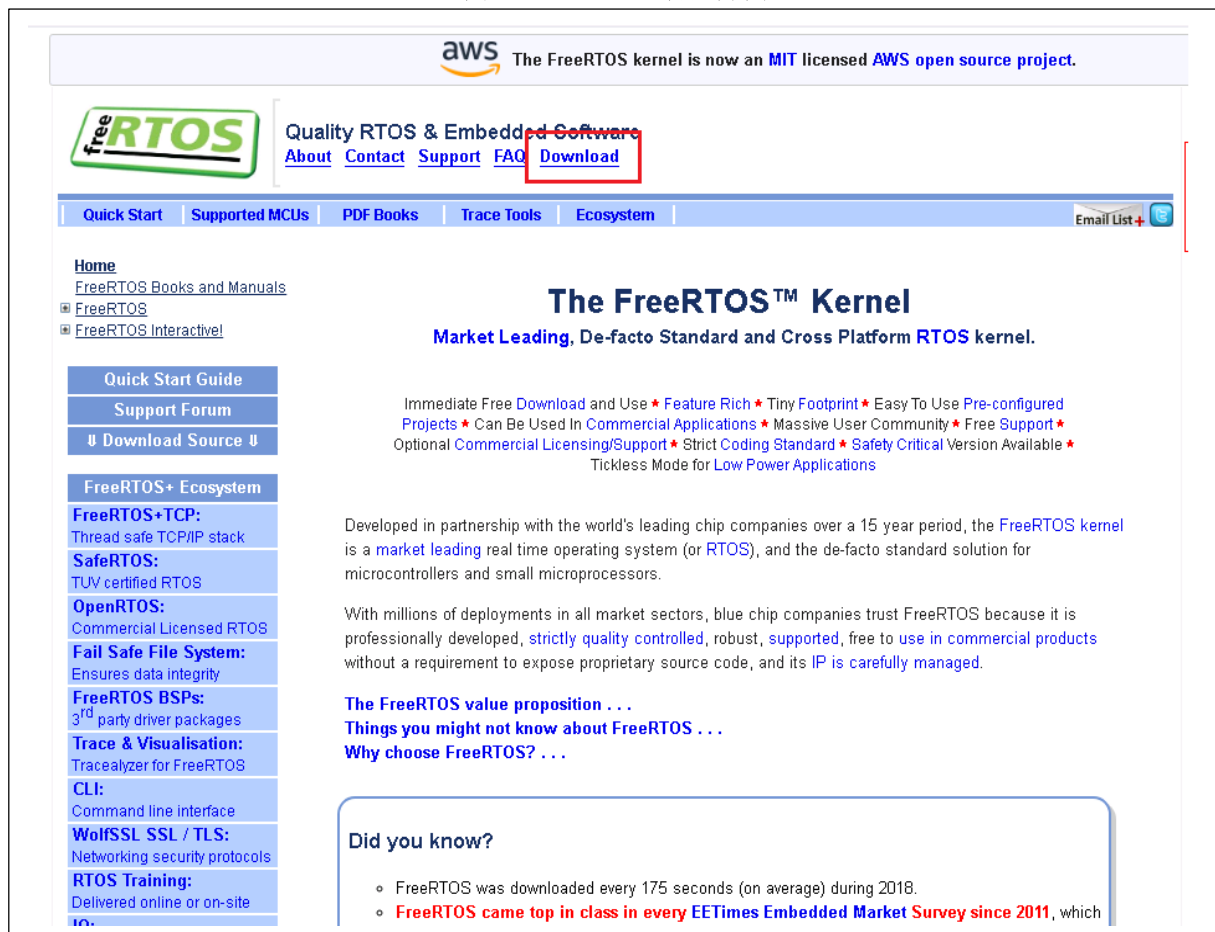
2 在 AT32 上移植 FreeRTOS

本节将介绍如何将FreeRTOS官网下载的源码包移植到AT32F系列MCU上运行，硬件平台为雅特力提供的AT-START Board。

2.1 移植 FreeRTOS

第一步：首先需要到FreeRTOS官方网站下载源码包，官方网站地址为：www.freertos.org。打开后网站界面如下：

图 1. FreeRTOS 官网界面



点击上方的Download即可下载最新版本的FreeRTOS源码包，本指导手册下载的FreeRTOS版本号为当前最新版本：V10.2.1。

第二步：将FreeRTOS源码包文件中的middlewares文件夹复制到工程中去，并添加到工程目录中，如下所示：

图 2. FreeRTOS 源码目录

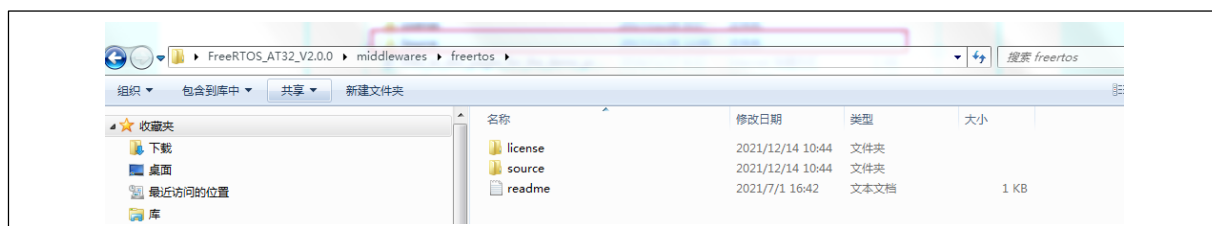
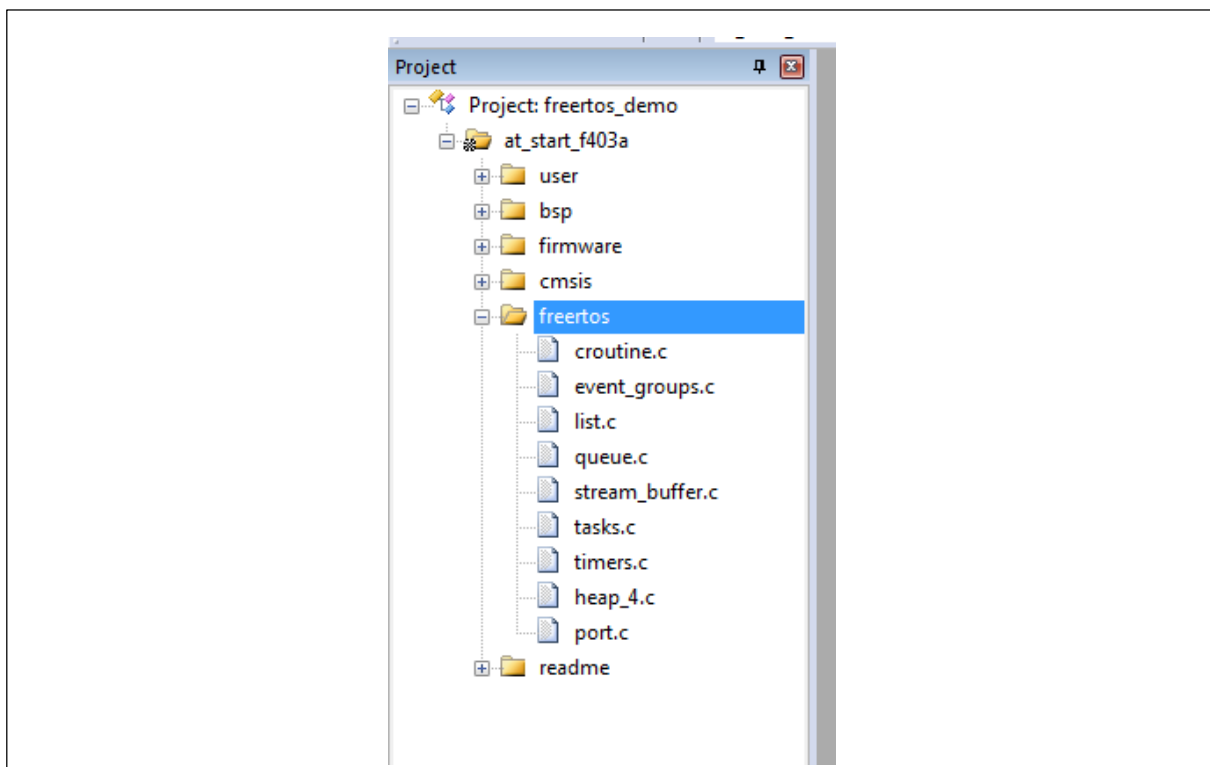


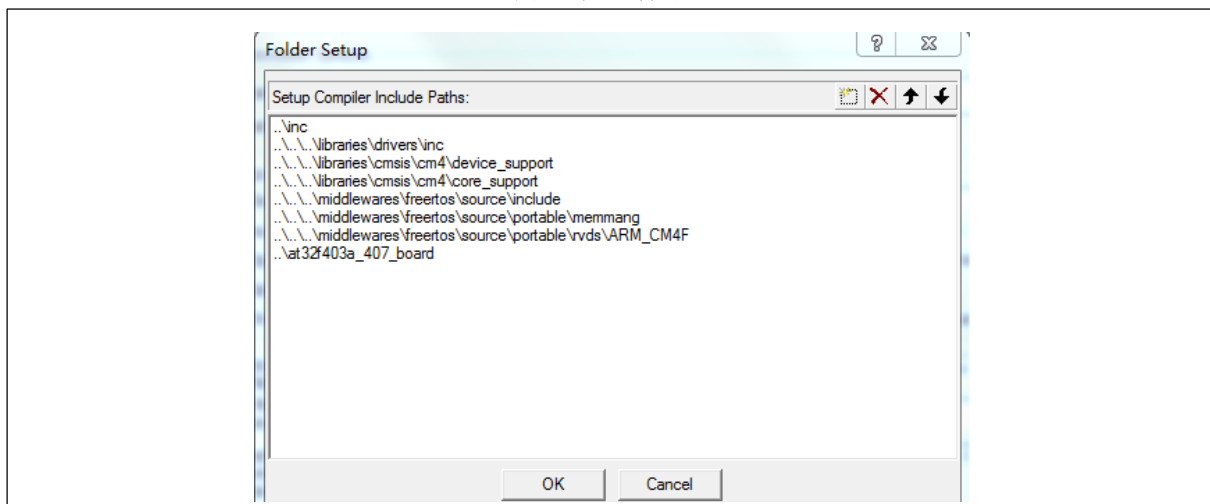
图 3. 工程目录



图片中的文件都是在source这个目录下的，其中heap_4.c是在Source\portable\MemMang目录下；port.c是在Source\portable\RVDS\ARM_CM4F下。

第三步：添加头文件路径到MDK中，如下图所示：

图 4. 头文件添加



头文件路径就是上面图片中的3个文件，添加即可。

第四步：下面来编译一次，发现会报错，内容为找不到FreeRTOSConfig.h这个头文件，其实这是FreeRTOS系统配置的一个文件，可以自行创建，也可以从源码包中的Demo中找一个。这里可以参考配套的例程，就不用自行添加了。后面对FreeRTOS熟悉后可以根据系统的需求配置一个FreeRTOSConfig.h文件。

通过以上4步后再编译一次，发现没有错误也没有警告了。下面就来编写一个任务调度的程序检查是

否移植成功。

2.2 例程介绍

工程名: **Porting_FreeRTOS_AT32**

程序源码:

```
#include "FreeRTOS.h"
#include "task.h"

/* 开始任务优先级 */
#define START_TASK_PRIO      1
/* 开始任务堆栈大小 */
#define START_STK_SIZE      128
/* 开始任务任务句柄 */
TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* LED0 任务优先级 */
#define LED0_TASK_PRIO      4
/* LED0 任务堆栈大小 */
#define LED0_STK_SIZE      128
/* LED0 任务任务句柄 */
TaskHandle_t LED0Task_Handler;
/* LED0 任务入口函数 */
void led0_task(void *pvParameters);

/* LED1 任务优先级 */
#define LED1_TASK_PRIO      3
/* LED1 任务堆栈大小 */
#define LED1_STK_SIZE      128
/* LED1 任务任务句柄 */
TaskHandle_t LED1Task_Handler;
/* LED1 任务入口函数 */
void led1_task(void *pvParameters);

/* 浮点运算任务优先级 */
#define FLOAT_TASK_PRIO      2
/* 浮点运算任务堆栈大小 */
#define FLOAT_STK_SIZE      256
/* 浮点运算任务句柄 */
TaskHandle_t FLOATTask_Handler;
/* 浮点运算入口函数*/
void float_task(void *pvParameters);

int main(void)
```

```
{
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

    system_clock_config();

    /* init usart1 */
    uart_print_init(115200);

    at32_led_init(LED2);
    at32_led_init(LED3);
    at32_led_init(LED4);

    /* 创建开始任务 */
    xTaskCreate((TaskFunction_t) start_task,
                (const char* ) "start_task",
                (uint16_t ) START_STK_SIZE,
                (void* ) NULL,
                (UBaseType_t ) START_TASK_PRIO,
                (TaskHandle_t* ) &StartTask_Handler);

    /* 开启任务调度器 */
    vTaskStartScheduler();
}

/* 开始任务函数 */
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();
    /* 创建 LED0 任务 */
    xTaskCreate((TaskFunction_t) led0_task,
                (const char* ) "led0_task",
                (uint16_t ) LED0_STK_SIZE,
                (void* ) NULL,
                (UBaseType_t ) LED0_TASK_PRIO,
                (TaskHandle_t* ) &LED0Task_Handler);

    /* 创建 LED1 任务 */
    xTaskCreate((TaskFunction_t) led1_task,
                (const char* ) "led1_task",
                (uint16_t ) LED1_STK_SIZE,
                (void* ) NULL,
                (UBaseType_t ) LED1_TASK_PRIO,
                (TaskHandle_t* ) &LED1Task_Handler);

    /* 创建浮点运算任务 */
    xTaskCreate((TaskFunction_t) float_task,
                (const char* ) "float_task",
```

```

        (uint16_t      )FLOAT_STK_SIZE,
        (void*         )NULL,
        (UBaseType_t   )FLOAT_TASK_PRIO,
        (TaskHandle_t* )&FLOATTask_Handler);

/* 删除开始任务 */
vTaskDelete(StartTask_Handler);

/* 退出临界区 */
taskEXIT_CRITICAL();
}

/* LED0 任务函数 */
void led0_task(void *pvParameters)
{
    while(1)
    {
        at32_led_toggle(LED3);
        printf("LED3 Toggle\r\n");
        vTaskDelay(1000);
    }
}

/* LED1 开始任务 */
void led1_task(void *pvParameters)
{
    while(1)
    {
        at32_led_toggle(LED2);
        printf("LED2 Toggle\r\n");
        vTaskDelay(500);
    }
}

/* 浮点运算任务函数 */
void float_task(void *pvParameters)
{
    static float float_num=0.00;
    while(1)
    {
        float_num+=0.01f;
        printf("float_num = %.4f\r\n",float_num);
        vTaskDelay(1000);
    }
}

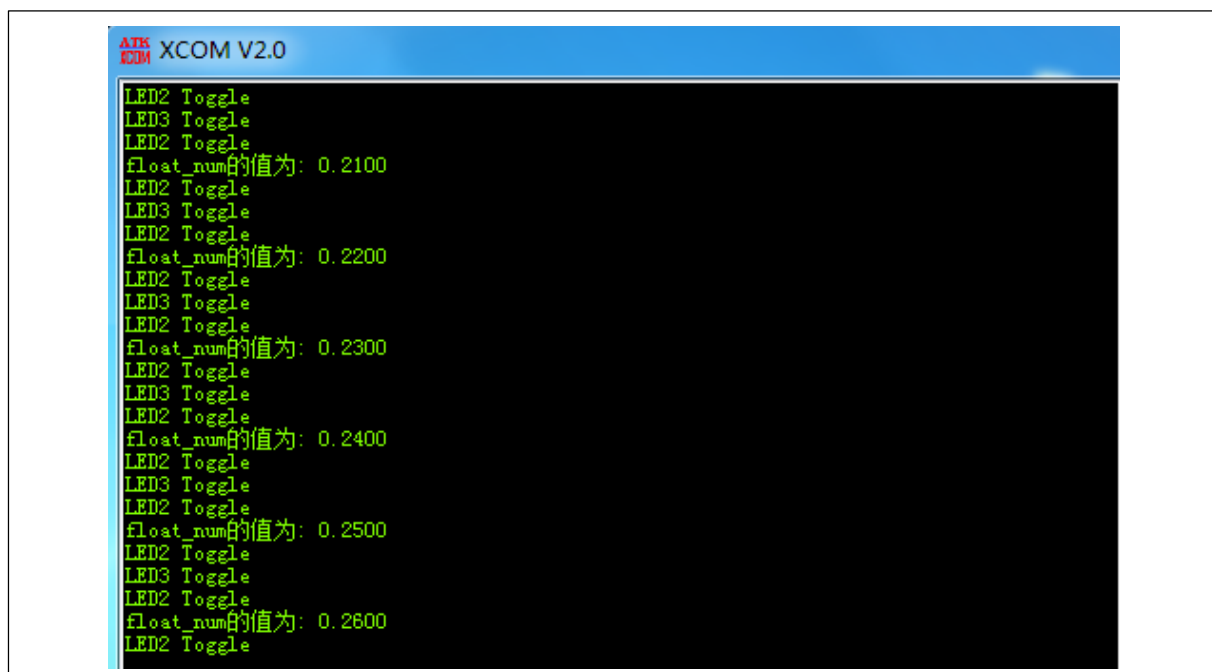
```

以上就是移植例程的源程序，可以看到程序中创建了三个任务，任务LED0、任务LED1、浮点运算任

务。可以看到与普通的裸机（无RTOS）程序不同，每个任务函数里都是一个死循环。三个任务会在任务调度器的调度下得到运行。

编译并下载到目标板可查看运行结果，可以看到目标板上的LED等会交替闪烁，说明任务之间在切换运行，也可通过串口打印查看结果，来看看串口打印的效果：

图 5. 移植例程演示



```

ATK XCOM V2.0
LED2 Toggle
LED3 Toggle
LED2 Toggle
float_num的值为: 0.2100
LED2 Toggle
LED3 Toggle
LED2 Toggle
float_num的值为: 0.2200
LED2 Toggle
LED3 Toggle
LED2 Toggle
float_num的值为: 0.2300
LED2 Toggle
LED3 Toggle
LED2 Toggle
float_num的值为: 0.2400
LED2 Toggle
LED3 Toggle
LED2 Toggle
float_num的值为: 0.2500
LED2 Toggle
LED3 Toggle
LED2 Toggle
float_num的值为: 0.2600
LED2 Toggle

```

以上就是如何将FreeRTOS移植到AT32系列MCU上运行的相关内容，本节有配套的工程源码，后续章节也都采用这种方式，可配合使用。

3 FreeRTOS 调试方法

本节介绍FreeRTOS的调试方法，FreeRTOS内核自带了一些API函数，用户通过配置后就可以使用这些方便的接口函数来获取当前系统中任务的运行情况。通过这种调试手段在开发程序时也能非常方便和高效的查找出问题。

3.1 系统配置

想使用FreeRTOS的调试功能，需要对系统做一些配置，首先要打开相应的宏定义，具体宏定义如下：

图 6. 调试配置参数

```
110 /* 调试时使用,在工程完成时需要屏蔽掉,因为会增加系统开销 */
111 #define configUSE_TRACE_FACILITY 1 //为1启用可视化跟踪调试
112 #define configGENERATE_RUN_TIME_STATS 1 //为1时启用运行时间统计功能
113 #define configUSE_STATS_FORMATTING_FUNCTIONS 1 //与宏configUSE_TRACE_FACILITY同时为1时会编译下面3个函数
114 //prvWriteNameToBuffer(),vTaskList(),
115 //vTaskGetRunTimeStats()
116 #define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() (debug_timerTick = 0)
117 #define portGET_RUN_TIME_COUNTER_VALUE() debug_timerTick
```

上图的配置参数的位置在FreeRTOSConfig.h文件里面。其中的参数debug_timerTick是为了给系统运行时间统计任务提供的时间基准，其具体定义在timer.c文件内，工程中打开了一个硬件定时器（TIMER2），并开启了溢出中断（1ms一次），在中断内对debug_timerTick进行加1操作。

具体程序如下：

```
#include "timer.h"

volatile uint32_t debug_timerTick;//任务运行时间统计使用
extern uint32_t systemcoreclock;

void TIMER_Init(void)
{
    /* 使能 TMR2 时钟 */
    crm_periph_clock_enable(CRM_TMR2_PERIPH_CLOCK, TRUE);
    crm_periph_clock_enable(CRM_TMR3_PERIPH_CLOCK, TRUE);
    crm_periph_clock_enable(CRM_TMR4_PERIPH_CLOCK, TRUE);
    crm_periph_clock_enable(CRM_TMR5_PERIPH_CLOCK, TRUE);

    /* 初始化 TMR2 */
    tmr_base_init(TMR2, systemcoreclock/10000, 10);
    tmr_cnt_dir_set(TMR2, TMR_COUNT_UP);

    tmr_interrupt_enable(TMR2, TMR_OVF_INT, TRUE);

    nvic_irq_enable(TMR2_GLOBAL_IRQn, 2, 0);

    /* 使能 TMR2 */
    tmr_counter_enable(TMR2, TRUE);
}
```

```
void TMR2_GLOBAL_IRQHandler(void)
{
    if(tmr_flag_get(TMR2, TMR_OVF_FLAG) != RESET)
    {
        debug_timerTick++;
        tmr_flag_clear(TMR2, TMR_OVF_FLAG);
    }
}
```

以上就是系统的相关配置内容。

3.2 例程介绍

工程名：02Debug_FreeRTOS

程序源码：

```
/* 调试任务优先级 */
#define Debug_TASK_PRIO      2
/* 调试任务堆栈大小 */
#define Debug_STK_SIZE      512
/* 调试任务任务句柄 */
TaskHandle_t DebugTask_Handler;
/* 调试任务入口函数 */
void debug_task(void *pvParameters);

//创建调试任务
xTaskCreate((TaskFunction_t)debug_task,
            (const char*   )"Debug_task",
            (uint16_t      )Debug_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )Debug_TASK_PRIO,
            (TaskHandle_t* )&DebugTask_Handler);

/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次调试信息 */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/-----*\r\n");
            printf("Task          Status          priority    Remaining_Stack    Num\r\n");
            vTaskList((char *)&buff);
            printf("%s\r\n",buff);
        }
    }
}
```

```
printf("/*-----*/\r\n");
printf("Task      Runing_Num      Usage_Rate\r\n");
vTaskGetRunTimeStats((char *)&buff);
printf("%s\r\n",buff);
}
vTaskDelay(10);
}
}
```

本例程是在移植FreeRTOS例程代码上增加一个任务得到，所以只粘贴了增加部分的程序，具体细节可参考该例程工程。

例程用到了vTaskList()和vTaskGetRunTimeStats()这两个系统API函数。vTaskList()用于获取系统中所有任务的任务名、任务状态、优先级、剩余栈空间、任务序号，并将其内容保存进传入参数的数组内，然后可调用打印函数将其输出到指定串口；vTaskGetRunTimeStats()函数则是用于获取系统中所有任务的任务名、运行时间计数、使用率，并将其内容保存进传入参数的数组内，然后可调用打印函数将其输出到指定串口。通过这两个API函数，就可以清楚的掌握系统的状态，对开发程序起到了很大的帮助。

编译并下载程序到目标板，运行查看打印结果为：

图 7. 调试例程演示



从打印结果可以看到当前系统的所有信息，包括有哪些任务、任务状态、使用率等。这里对几个可能引起疑惑的点做一些介绍，任务状态：X为运行态、R为就绪态、B为阻塞态、D为删除态、S为挂起态；任务序号为创建的先后顺序；运行计数的单位为之前配置的硬件定时器的溢出时间。

4 FreeRTOS 中断优先级管理

本节介绍FreeRTOS中断优先级管理相关的内容，该部分内容很重要，对于理解FreeRTOS和编程都有很大的帮助，理解这部分是为了能编写出实时性强的软件的第一步。

4.1 AT32 中断配置

在介绍AT32中断配置之前，需要了解NVIC的作用。NVIC的全称是Nested vectored interrupt controller，即嵌套向量中断控制器。对于M4内核的MCU（AT32采用先进的Cortex-M4内核），每个中断的优先级都是用寄存器中的8位来设置的，8位的话就可以设置 $2^8 = 256$ 级中断。实际中用不了这么多，所以芯片厂商根据自己生产的芯片做出了调整。比如AT的AT32F系列只使用了这个8位中的高四位[7:4]，低四位取零，这样 $2^4=16$ ，只能表示16级中断嵌套。对于这个NVIC，有个重要的知识点就是优先级分组，抢占优先级和子优先级，下面就以AT32为例进行介绍，AT32F系列只使用了这个8位寄存器的高四位[7:4]。

表 1. 中断优先级分组

优先级分组	抢占优先级	子优先级	高4位使用情况描述
NVIC_PriorityGroup_0	0级抢占优先级	0-15级子优先级	0bit用于抢占优先级 4bit全用于子优先级
NVIC_PriorityGroup_1	0-1级抢占优先级	0-7级子优先级	1bit用于抢占优先级 3bit用于子优先级
NVIC_PriorityGroup_2	0-3级抢占优先级	0-3级子优先级	2bit用于抢占优先级 2bit用于子优先级
NVIC_PriorityGroup_3	0-7级抢占优先级	0-1级子优先级	3bit用于抢占优先级 1bit用于子优先级
NVIC_PriorityGroup_4	0-15级抢占优先级	0级子优先级	4bit全用于抢占优先级 0bit用于子优先级

从上面的表格可以看出，AT32支持5种优先级分组，系统上电复位后，默认使用的是优先级分组0，也就是没有抢占式优先级，只有子优先级，关于这个抢占优先级和这个子优先级有几点一定要说清楚：

- 1) 具有高抢占式优先级的中断可以在具有低抢占式优先级的中断服务程序执行过程中被响应，即中断嵌套，或者说高抢占式优先级的中断可以抢占低抢占式优先级的中断的执行。
- 2) 在抢占式优先级相同的情况下，有几个子优先级不同的中断同时到来，那么高子优先级的中断优先级被响应。
- 3) 在抢占式优先级相同的情况下，如果有低子优先级中断正在执行，高子优先级的中断要等待已被响应的低子优先级中断执行结束后才能得到响应，即子优先级不支持中断嵌套。
- 4) Reset、NMI、Hard Fault 优先级为负数，高于普通中断优先级，且优先级不可配置。

对于初学者还有一个比较纠结的问题就是系统中断（比如：PendSV，SVC，SysTick）是不是一定比外部中断（比如SPI，USART）要高，答案：不是的，它们是在同一个NVIC下面设置的。

在使用FreeRTOS的时候，本指导手册使用NVIC_PriorityGroup_4，这也是FreeRTOS官方推荐的中断管理方式。

在使用AT32库函数配置NVIC时，只需调用一个库函数即可，如下：

图 8. AT32 NVIC 配置

```
int main(void)
{
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
```

此中断优先级分组方式只有抢占优先级，无子优先级，所以在配置AT32外设中断时要注意一下，需设置子优先级为0。具体如下图：

图 9. AT32 外设中断配置

```
tmr_interrupt_enable(TMR2, TMR_OVF_INT, TRUE);

nvic_irq_enable(TMR2_GLOBAL_IRQn, 2, 0);
```

AT32中断主要就是这些，理解到这些就足够了。

4.2 FreeRTOS 中断配置

FreeRTOS的中断配置是在FreeRTOSConfig.h中完成的，内容如下：

图 10. FreeRTOS 中断配置

```
129 /* Cortex-M specific definitions. */
130 #ifndef __NVIC_PRIO_BITS
131 /* __NVIC_PRIO_BITS will be specified when CMSIS is being used. */
132 #define configPRIO_BITS    __NVIC_PRIO_BITS
133 #else
134 #define configPRIO_BITS    4 /* 15 priority levels */
135 #endif
136
137 /* The lowest interrupt priority that can be used in a call to a "set priority"
138 function. */
139 #define configLIBRARY_LOWEST_INTERRUPT_PRIORITY    0x0f /*采用优先级分组4,只有16级抢占优先级,这是最低优先级给PendSV和SysTick用
140
141 /* The highest interrupt priority that can be used by any interrupt service
142 routine that makes calls to interrupt safe FreeRTOS API functions. DO NOT CALL
143 INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER
144 PRIORITY THAN THIS! (higher priorities are lower numeric values. */
145 #define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY    0x03 /*中断优先级为0/1/2/3的中断不能由FreeRTOS管理
146
147 /* Interrupt priorities used by the kernel port layer itself. These are generic
148 to all Cortex-M ports, and do not rely on any particular library functions. */
149 #define configKERNEL_INTERRUPT_PRIORITY    ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
150 /* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!!
151 See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
152 #define configMAX_SYSCALL_INTERRUPT_PRIORITY    ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 - configPRIO_BITS) )
```

1) #define configPRIO_BITS 4

此宏定义用于配置AT32的8位优先级设置寄存器实际使用的位数。AT32都是使用的4位。另外注意一点，这里使用了一个条件编译，用户可以选择将条件编译删掉，直接定义一个#define configPRIO_BITS 4即可。使用条件编译的好处就是方便与系统统一。__NVIC_PRIO_BITS在AT32标准库的头文件at32f403a_407.h中有定义。如果用户在FreeRTOSConfig.h文件里面包含了这个标准库的头文件，那么就会执行条件编译选项：

2) #define configPRIO_BITS __NVIC_PRIO_BITS

```
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY 0x0F
```

此宏定义是用来配置FreeRTOS用到的SysTick中断和PendSV中断的优先级。在NVIC分组设置为4的情况下，此宏定义的范围就是0-15，即专门配置抢占优先级。这里配置为0x0F，SysTick和PendSV都是配置成了最低优先级，实际使用中也建议这样配置。

3) #define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 0x03

此宏定义比较重要，定义了受FreeRTOS管理的最高优先级中断。简单的说就是允许用户在这个

中断服务程序里面调用FreeRTOS的API的最高优先级。设置NVIC的优先级分组为4的情况下。配置configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY为0x03表示用户可以在抢占式优先级为3到15的中断里面调用FreeRTOS的API函数，抢占式优先级为0到2的中断里面是不允许调用的。

4) #define configKERNEL_INTERRUPT_PRIORITY

宏定义configLIBRARY_LOWEST_INTERRUPT_PRIORITY的数值经过4bit偏移后得到一个8bit的优先级数值，即宏定义configKERNEL_INTERRUPT_PRIORITY的数值。这个8bit的数值才可以实际赋值给相应中断的优先级寄存器。也许会有疑问了，为什么前面NVIC配置的时候不是8bit的方式进行配置？这是因为AT32的库函数NVIC_Init()已经为我们做好了。这里的宏定义数值是供PendSV和SysTick中断进行优先级配置的。比如：我们这里配置宏定义configLIBRARY_LOWEST_INTERRUPT_PRIORITY是0x0F，经过4bit偏移后就是0xF0，即SysTick和PendSV的中断优先级就是240。

5) #define configMAX_SYSCALL_INTERRUPT_PRIORITY

宏定义configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY的数值经过4bit偏移后得到一个8bit的优先级数值，即宏定义configMAX_SYSCALL_INTERRUPT_PRIORITY的数值。这个数值是赋值给寄存器basepri使用的，8bit的数值才可以实际赋值给相应中断的优先级寄存器。这里的宏定义数值赋给寄存器basepri后就可以实现全局的开关中断操作了。比如：我们这里配置宏定义configLIBRARY_LOWEST_INTERRUPT_PRIORITY是0x01，经过4bit偏移后就是0x10，即16。调用了FreeRTOS的关中断后，所有优先级数值大于等于16的中断都会被关闭。优先级数值小于16的中断不会被关闭，对寄存器basepri寄存器赋值0，那么被关闭的中断会被打开。

通过以上的配置就将FreeRTOS的中断配置完成了，到此FreeRTOS可管理大于3的优先级的外设（开关中断）。为什么不让FreeRTOS管理所有的中断，这样做是有目的的，及提高系统的实时性，当关闭某中断后那么该中断到来后就会延迟中断的响应时间，降低实时性。在一些非常紧急的任务是不允许这种情况发生的，FreeRTOS就很好的解决了这个问题。

4.3 中断优先级和任务优先级区别

开始接触FreeRTOS也容易在这两个概念上面出现问题。简单的说，这两个之间没有任何关系，不管中断的优先级是多少，中断的优先级永远高于任何任务的优先级，即任务在执行的过程中，中断来了就开始执行中断服务程序。

另外对于AT32F403，AT32F413和AT32F415来说，中断优先级的数值越小，优先级越高。而FreeRTOS的任务优先级是，任务优先级数值越小，任务优先级越低。

4.4 临界段保护

临界段代码也叫做临界区（critical code region），是指那些必须完整运行不能被打断的代码段。比如一些外设的初始化，需要遵循严格的时序要求，初始化过程中不能被打断。FreeRTOS采用的方式就是在进入临界区代码的时候关闭中断，退出临界区的时候打开中断。FreeRTOS系统本身就有许多临界区代码，这些程序段都加了临界区保护，在写应用程序的时候也有很多地方需要用到临界区保护。

与临界段代码保护的函数有4个，如下：

表 2. 临界区 API

函数	功能
taskENTER_CRITICAL()	进入临界区，关中断
taskEXIT_CRITICAL()	退出临界区，开中断
taskENTER_CRITICAL_FROM_ISR()	进入临界区，关中断（ISR 中使用）
taskEXIT_CRITICAL_FROM_ISR(x)	退出临界区，开中断（ISR 中使用）

可以很清楚的看到上面4个函数两两一对，前面两个是用在普通任务中的，而后面两个则是用在中断程序中的。

普通进入临界区的函数和中断内使用进入临界区的函数所实现的功能都是一样的，只是函数的实现方式不同。普通进入临界区的函数使用C语言实现并在进入函数的内部采用一个全局变量实现临界区的嵌套，而中断内使用进入临界区的函数则是用汇编语言实现，并且通过保存和恢复寄存器basepri的数值就可以实现嵌套。汇编语言执行效率高于C语言，然而在中断内需要高的运行效率，两者的差异就在此。

临界区代码的保护是很重要的概念，不单在RTOS中，在Linux、Window中都存在。所以理解这部分对于理解操作系统是很有帮助的。

4.5 例程介绍

工程名：03Interrupt_FreeRTOS

程序源码：

```
#include "FreeRTOS.h"
#include "task.h"

/* 开始任务优先级 */
#define START_TASK_PRIO    1
/* 开始任务堆栈大小 */
#define START_STK_SIZE     128
/* 开始任务任务句柄 */
TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* LED0 任务优先级 */
#define LED0_TASK_PRIO    4
/* LED0 任务堆栈大小 */
#define LED0_STK_SIZE     128
/* LED0 任务任务句柄 */
TaskHandle_t LED0Task_Handler;
/* LED0 任务入口函数 */
void led0_task(void *pvParameters);

/* LED1 任务优先级 */
```



```
#define LED1_TASK_PRIO      3
/* LED1 任务堆栈大小 */
#define LED1_STK_SIZE      128
/* LED1 任务任务句柄 */
TaskHandle_t LED1Task_Handler;
/* LED1 任务入口函数 */
void led1_task(void *pvParameters);

/* 调试任务优先级 */
#define Debug_TASK_PRIO    2
/* 调试任务堆栈大小 */
#define Debug_STK_SIZE     512
/* 调试任务任务句柄 */
TaskHandle_t DebugTask_Handler;
/* 调试任务入口函数 */
void debug_task(void *pvParameters);

int main(void)
{
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

    system_clock_config();

    at32_led_init(LED2);
    at32_led_init(LED3);
    at32_led_init(LED4);

    at32_button_init();

    /* init usart1 */
    uart_print_init(115200);

    TIMER_Init();
    /* 创建开始任务 */
    xTaskCreate((TaskFunction_t)start_task,
                (const char*   )"start_task",
                (uint16_t      )START_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t   )START_TASK_PRIO,
                (TaskHandle_t*  *)&StartTask_Handler);

    /* 开启任务调度器 */
    vTaskStartScheduler();
}

/* 开始任务函数 */
```

```
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();
    /* 创建 LED0 任务 */
    xTaskCreate((TaskFunction_t)led0_task,
                (const char*   )"led0_task",
                (uint16_t      )LED0_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t    )LED0_TASK_PRIO,
                (TaskHandle_t*  )&LED0Task_Handler);
    /* 创建 LED1 任务 */
    xTaskCreate((TaskFunction_t)led1_task,
                (const char*   )"led1_task",
                (uint16_t      )LED1_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t    )LED1_TASK_PRIO,
                (TaskHandle_t*  )&LED1Task_Handler);
    /* 创建调试任务*/
    xTaskCreate((TaskFunction_t)debug_task,
                (const char*   )"Debug_task",
                (uint16_t      )Debug_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t    )Debug_TASK_PRIO,
                (TaskHandle_t*  )&DebugTask_Handler);

    /* 删除开始任务 */
    vTaskDelete(StartTask_Handler);
    /* 退出临界区 */
    taskEXIT_CRITICAL();
}

/* LED0 任务函数 */
void led0_task(void *pvParameters)
{
    while(1)
    {
        /* 进入临界区 */
        taskENTER_CRITICAL();
        at32_led_toggle(LED3);
        printf("进入临界区\r\n");
        /* 使用 TMR5 的硬件中断模拟普通延时,这里能使用 vTaskDelay()会产生任务调度 */
        while(tmr_flag_get(TMR5,TMR_OVF_FLAG)==RESET);
        tmr_flag_clear(TMR5,TMR_OVF_FLAG);
        printf("退出临界区\r\n");
        /* 退出临界区 */
    }
}
```

```

taskEXIT_CRITICAL();
vTaskDelay(1000);
}
}

/* LED1 任务函数 */
void led1_task(void *pvParameters)
{
    while(1)
    {
        at32_led_toggle(LED2);
        vTaskDelay(500);
    }
}

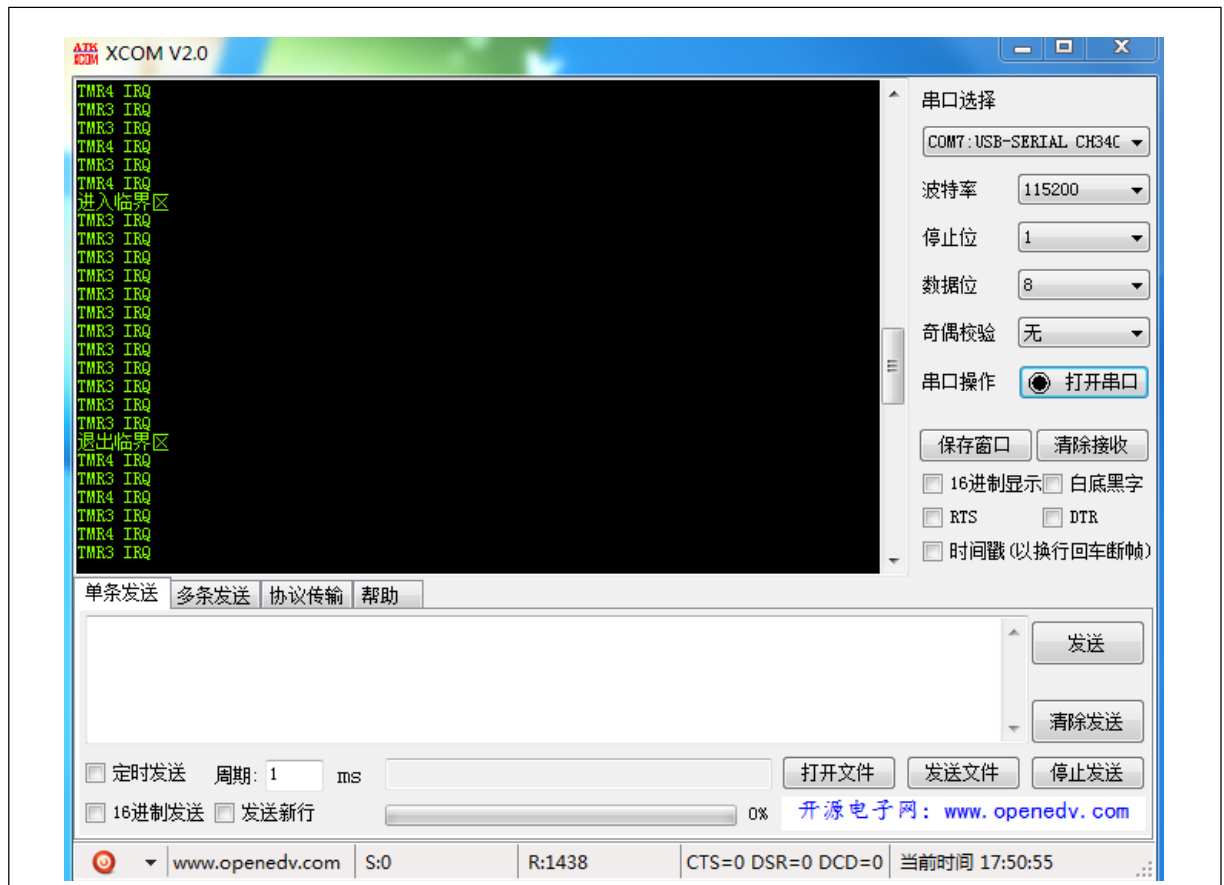
/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次任务信息 */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/*-----*/\r\n");
            printf("Task          Status          priority      Remaining_Stack    Num\r\n");
            vTaskList((char *)&buff);
            printf("%s\r\n",buff);
            printf("/*-----*/\r\n");
            printf("Task          Runing_Num      Usage_Rate\r\n");
            vTaskGetRunTimeStats((char *)&buff);
            printf("%s\r\n",buff);
        }
        vTaskDelay(10);
    }
}

```

以上就是中断优先级管理的例程配套程序，程序中创建了两个硬件定时器（TMR3和TMR4），TMR3和TMR4会定时产生中断，而TMR4的中断优先级为7，可被FreeRTOS的进入临界区程序管理，而TMR3的优先级为2，不可被FreeRTOS的进入临界区程序管理。

编译并下载程序到目标板，运行结果如下：

图 11. FreeRTOS 中断管理例程演示



从打印结果可见当进入临界区后，TMR4的中断不再产生，说明被FreeRTOS内核禁止了中断；退出临界区后TMR4中断正常，说明FreeRTOS重新开启了中断。

5 FreeRTOS 任务管理

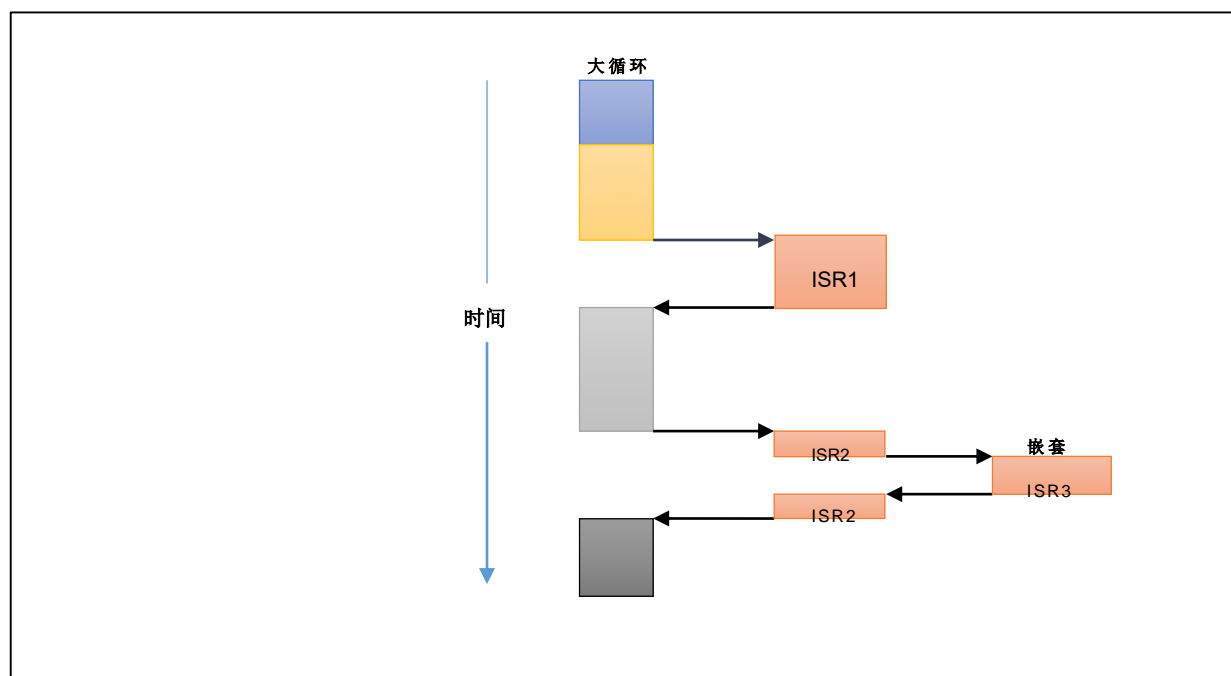
本节介绍FreeRTOS任务管理相关的内容，这部分是FreeRTOS的核心，可以说任务管理对于操作系统就如同大脑对于人一样。它是管理操作系统运行的根本，掌握了这部分内容那么FreeRTOS的后续学习就变得简单了。

5.1 裸机与带 RTOS 的区别

裸机系统

对于裸机系统，通常就是一个大循环顺序执行，每一部分的操作都不是实时性的。为了解决这个问题，可以引入中断，AT32 MCU内部就带有很多中断，紧急事件可以放到中断内执行，这就是所谓的前后台系统。这个大循环就是后台系统，中断处理就是前台系统。其运行流程如下图所示：

图 12. 裸机运行流程



从图中可以清晰的看出前后台系统的处理流程，程序主要是在大循环里面运行，但是随时可能会被打断从而进入中断程序内运行。中断内的程序的实时性较高，但是其他的部分实时性就会变得极差，这也是引入操作系统的重要原因。

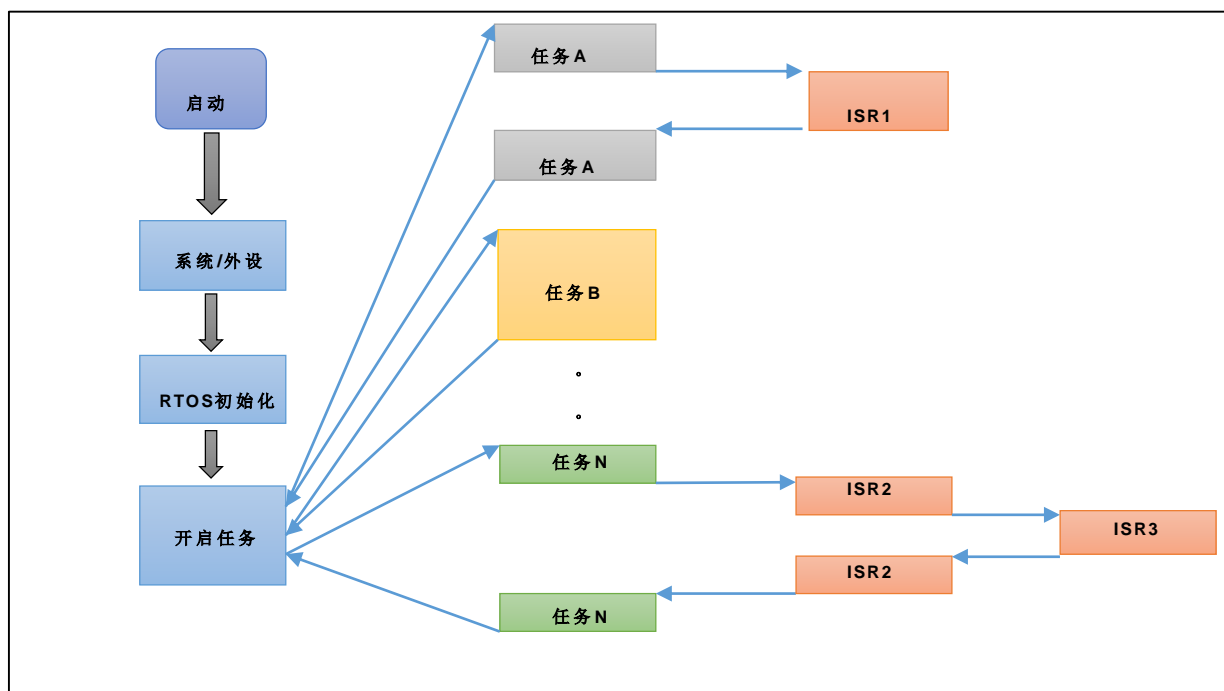
RTOS运行流程

在使用了RTOS后，系统运行的整体流程就会发生变化，系统中的每个任务都是一个循环，这一点在前面的例程代码中也体现出来了，在没有这一章节的介绍的话可能大家会犯迷糊，为什么都有while循环，程序是如何跑出来的呢？这就要关系到RTOS的任务调度器了。

多任务系统或者说RTOS的实现，重点就在这个调度器上，而调度器的作用就是使用相关的调度算法来决定当前需要执行的任务。创建了任务并完成OS初始化后，就可以通过调度器来决定任务A，任务B和任务C的运行，从而实现多任务系统。另外需要注意的是，这里所说的多任务系统同一时刻只能有一个任务可以运行，只是通过调度器的决策，看起来像所有任务同时运行一样。比如现在系统有A/B/C三个任务，每个任务运行10MS，那么给人的直观感受就是所有任务在同时运行。

下面是RTOS运行流程简化图，可以帮助理解其运行机制：

图 13. RTOS 运行流程



从上图可以看出，当开启任务调度器后，系统中的任务就会开始轮询运行，这里的轮询运行相比于裸机的优势在于任务调度器只会让每个任务运行一段时间，而不会像裸机那样直到运行完毕为止，所以这里的实时性就比裸机好。而且RTOS还提供诸如信号量、消息队列功能，这些功能使系统更加的灵活，后续章节会讲到，这里知道便可。

5.2 FreeRTOS 任务状态

讲解FreeRTOS的任务管理，那自然要知道FreeRTOS的所有任务状态以及这些状态间的转换关系，下面就来看看这部分的知识吧。

FreeRTOS的任务状态包括：

1) Running-运行态

当任务处于实际运行状态被称之为运行态，即CPU的使用权被这个任务占用。

2) Ready-就绪态

处于就绪态的任务是指那些能够运行（没有被阻塞和挂起），但是当前没有运行的任务，因为同优先级或更高优先级的任务正在运行。

3) Blocked-阻塞态

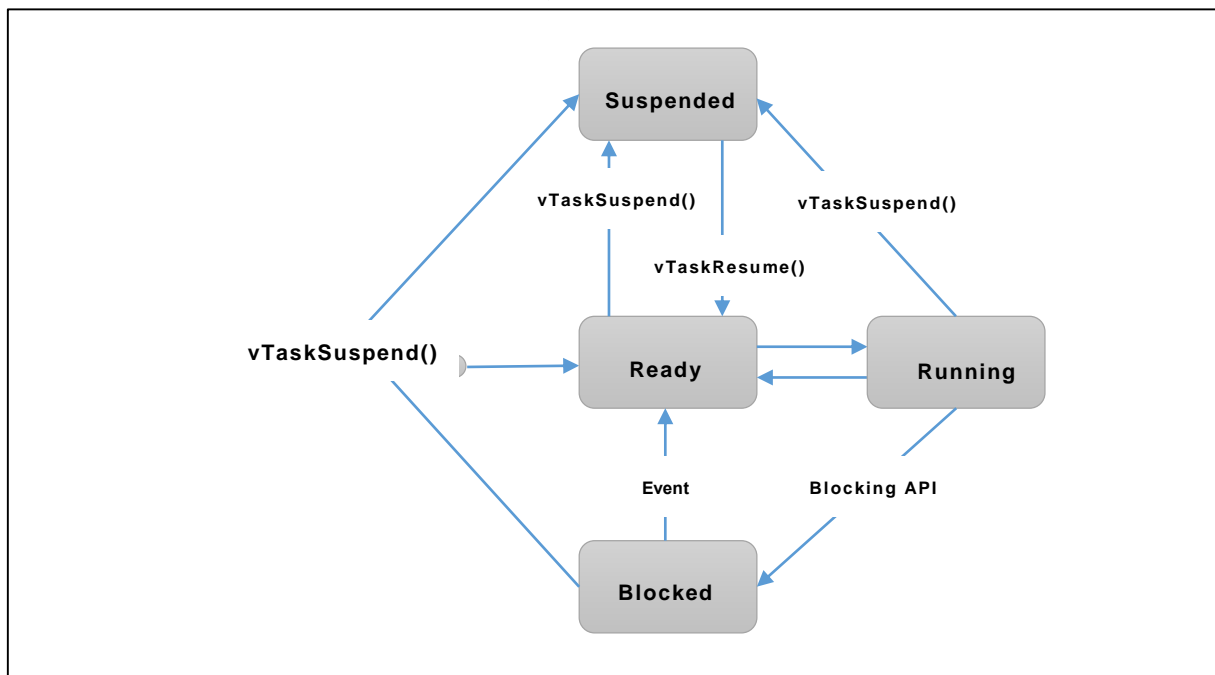
由于等待信号量，消息队列，事件标志组等而处于的状态被称之为阻塞态，另外任务调用延迟函数也会处于阻塞态。

4) Suspended-挂起态

类似阻塞态，通过调用函数vTaskSuspend()对指定任务进行挂起，挂起后这个任务将不被执行，只有调用函数xTaskResume()才可以将这个任务从挂起态恢复。

下面就是FreeRTOS的各个任务的转换关系图，通过这个图就能对任务的运行过程有一个初步的了解，后续章节会逐一细致的讲解每个任务状态。

图 14. 任务状态转换关系



从上图可知，当任务创建后首先进入就绪态（Ready），之后进入运行态（Running），在运行态期间如果调用了进入阻塞态的API函数后就会进入阻塞态（Blocked），在运行态如果调用了进入挂起态的API就会进入挂起态（Suspended）；阻塞态的任务在接收到事件后又会让任务进入就绪态，从而达到运行的结果；处于挂起态的任务也可以调用相应的API在就绪态和运行态之间切换；当然处于阻塞态的任务也可以调用API进入挂起态。

5.3 FreeRTOS 空闲任务

几乎所有的小型 RTOS 中都会有一个空闲任务，空闲任务属于系统任务，是必须要执行的，用户程序不能将其关闭。不光小型系统中有空闲任务，大型的系统里面也有的，比如我们使用的Window系统等。

在前面的FreeRTOS调试方法章节，细心的读者或许已经注意到为什么有一个叫IDLE task的任务占用了CPU百分之九十多的使用权，但是这个任务并没有自己手动创建。原因就是那个空闲任务是系统自己创建的，每当系统没有其他任务要运行时，空闲任务就开始运行了。可能读者会问了，这个空闲任务有什么作用呢？首先一个RTOS每时每刻都需要有任务运行，其次这个空闲任务还可以来做其他的一些工作，例如进入低功耗等（因为当系统运行到空闲任务说明没有任务需要运行了，这时是进入低功耗的最佳时机）。后面的章节还会用到空闲任务，到时再详细讲解。

5.4 FreeRTOS 任务相关函数

下面来看看FreeRTOS常用的任务API函数，这些函数是使用最多的，其他的一些API函数可自行查看FreeRTOS的官方手册进行学习使用。

任务创建函数

任务创建函数分为动态创建任务和静态创建任务，其API的函数名和参数会有些许不同。

xTaskCreate();

描述：

动态创建任务。

原型：

表 3. 动态创建任务函数原型

<pre> BaseType_t xTaskCreate(TaskFunction_t pvTaskCode, const char * const pcName, unsigned short usStackDepth, void *pvParameters, UBaseType_t uxPriority, TaskHandle_t *pxCreatedTask); </pre>	
参数	描述
pvTaskCode	任务函数
pcName	任务名
usStackDepth	任务栈大小
pvParameters	任务函数参数
uxPriority	任务优先级
pxCreatedTask	任务句柄

返回值：

任务创建成功与否的标志。

xTaskCreateStatic();

描述：

静态创建任务。

原型：

表 4. 静态创建任务函数原型

<pre> TaskHandle_t xTaskCreateStatic(TaskFunction_t pvTaskCode, const char * const pcName, uint32_t ulStackDepth, void *pvParameters, UBaseType_t uxPriority, StackType_t * const puxStackBuffer, StaticTask_t * const pxTaskBuffer); </pre>	
参数	描述
pvTaskCode	任务函数
pcName	任务名
ulStackDepth	任务栈大小
pvParameters	任务函数参数
uxPriority	任务优先级
puxStackBuffer	任务堆栈内存
pxTaskBuffer	任务控制块内存

返回值：

任务创建成功与否的标志。

动态创建任务和静态创建任务的区别在于是否需要手动提供任务控制块和任务堆栈的内存。动态创建任务是创建函数内部自动分配需要的内存，而静态创建任务就需要手动分配了。

任务删除

vTaskDelete();

描述:

删除指定的任务。

原型:

表 5. 任务删除函数原型

void vTaskDelete(TaskHandle_t pxTask);	
参数	描述
pxTask	要删除任务的任务句柄

返回值:

无

任务挂起

vTaskSuspend();

描述:

让指定的任务进入挂起态

原型:

表 6. 任务挂起函数原型

void vTaskSuspend(TaskHandle_t pxTaskToSuspend);	
参数	描述
pxTaskToSuspend	要挂起任务的任务句柄

返回值:

无

任务挂起解除

vTaskResume();

描述:

让指定任务解除挂起。

原型:

表 7. 任务解除挂起函数原型

void vTaskResume(TaskHandle_t pxTaskToResume);	
参数	描述
pxTaskToSuspend	要挂起任务的任务句柄

返回值:

无

延时函数

vTaskDelay();

描述:

使任务从运行态切换到阻塞态。

原型:

表 8. 任务延时函数原型

void vTaskDelay(TickType_t xTicksToDelay);	
参数	描述
xTicksToDelay	延时的时间，单位为 Tick

返回值:

无

以上是几个任务相关的API函数，FreeRTOS提供的任务相关的API很多，这里没办法一一列举，如果需要查看FreeRTOS官方提供的API手册即可。

5.5 例程介绍

工程名: **04TaskManagement_FreeRTOS**

程序源码:

```
#include "FreeRTOS.h"
#include "task.h"

/* 开始任务优先级 */
#define START_TASK_PRIO      1
/* 开始任务堆栈大小 */
#define START_STK_SIZE      128
/* 开始任务任务句柄 */
TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* LED0 任务优先级 */
#define LED0_TASK_PRIO      3
/* LED0 任务堆栈大小 */
#define LED0_STK_SIZE      128
/* LED0 任务任务句柄 */
TaskHandle_t LED0Task_Handler;
/* LED0 任务入口函数 */
void led0_task(void *pvParameters);

/* LED1 任务优先级 */
#define LED1_TASK_PRIO      3
/* LED1 任务堆栈大小 */
#define LED1_STK_SIZE      128
```

```

/* LED1 任务任务句柄 */
TaskHandle_t LED1Task_Handler;
/* LED1 任务入口函数 */
void led1_task(void *pvParameters);

/* 打印任务优先级 */
#define Printf_TASK_PRIO      2
/* 打印任务堆栈大小 */
#define Printf_STK_SIZE      128
/* 打印任务句柄 */
TaskHandle_t PrintfTask_Handler;
/* 打印任务入口函数 */
void Printf_task(void *pvParameters);

/* 调试任务优先级 */
#define Debug_TASK_PRIO      2
/* 调试任务堆栈大小 */
#define Debug_STK_SIZE      512
/* 调试任务任务句柄 */
TaskHandle_t DebugTask_Handler;
/* 调试任务入口函数 */
void debug_task(void *pvParameters);

int main(void)
{
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

    system_clock_config();

    at32_led_init(LED2);
    at32_led_init(LED3);
    at32_led_init(LED4);

    at32_button_init();

    /* init usart1 */
    uart_print_init(115200);

    TIMER_Init();
    /* 创建开始任务 */
    xTaskCreate((TaskFunction_t)start_task,
                (const char* )"start_task",
                (uint16_t)START_STK_SIZE,
                (void*)NULL,
                (UBaseType_t)START_TASK_PRIO,

```

```

        (TaskHandle_t*) &StartTask_Handler);

/* 开启任务调度器 */
vTaskStartScheduler();
}

/* 开始任务函数 */
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();
    /* 创建 LED0 任务 */
    xTaskCreate((TaskFunction_t) led0_task,
                (const char*) "led0_task",
                (uint16_t) LED0_STK_SIZE,
                (void*) NULL,
                (UBaseType_t) LED0_TASK_PRIO,
                (TaskHandle_t*) &LED0Task_Handler);
    /* 创建 LED1 任务 */
    xTaskCreate((TaskFunction_t) led1_task,
                (const char*) "led1_task",
                (uint16_t) LED1_STK_SIZE,
                (void*) NULL,
                (UBaseType_t) LED1_TASK_PRIO,
                (TaskHandle_t*) &LED1Task_Handler);
    /* 创建调试任务 */
    xTaskCreate((TaskFunction_t) debug_task,
                (const char*) "Debug_task",
                (uint16_t) Debug_STK_SIZE,
                (void*) NULL,
                (UBaseType_t) Debug_TASK_PRIO,
                (TaskHandle_t*) &DebugTask_Handler);
    /* 创建打印任务 */
    xTaskCreate((TaskFunction_t) Printf_task,
                (const char*) "printf_task",
                (uint16_t) Printf_STK_SIZE,
                (void*) NULL,
                (UBaseType_t) Printf_TASK_PRIO,
                (TaskHandle_t*) &PrintfTask_Handler);

    /* 删除开始任务 */
    vTaskDelete(StartTask_Handler);
    /* 退出临界区 */
    taskEXIT_CRITICAL();
}

/* LED0 任务函数 */

```

```

void led0_task(void *pvParameters)
{
    while(1)
    {
        at32_led_toggle(LED3);
        printf("LED3 Toggle!\r\n");
        /* 挂起打印任务 */
        vTaskSuspend(PrintfTask_Handler);
        printf("Suspend printf task!\r\n");
        vTaskDelay(1000);
        printf("Resume printf task!\r\n");
        /* 解除打印任务的挂起状态 */
        vTaskResume(PrintfTask_Handler);
        vTaskDelay(1000);
    }
}

/* LED1 任务函数 */
void led1_task(void *pvParameters)
{
    while(1)
    {
        at32_led_toggle(LED2);
        printf("LED2 Toggle!\r\n");
        vTaskDelay(100);
    }
}

/* 打印任务函数 */
void Printf_task(void *pvParameters)
{
    while(1)
    {
        printf("printf task!\r\n");
    }
}

/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次调试信息 */
        if(at32_button_press() == USER_BUTTON)
        {

```

```

printf("/-----*\r\n");
printf("Task          Status          priority      Remaining_Stack    Num\r\n");
vTaskList((char *)&buff);
printf("%s\r\n",buff);
printf("/-----*\r\n");
printf("Task          Runing_Num      Usage_Rate\r\n");
vTaskGetRunTimeStats((char *)&buff);
printf("%s\r\n",buff);
}
vTaskDelay(10);
}
}

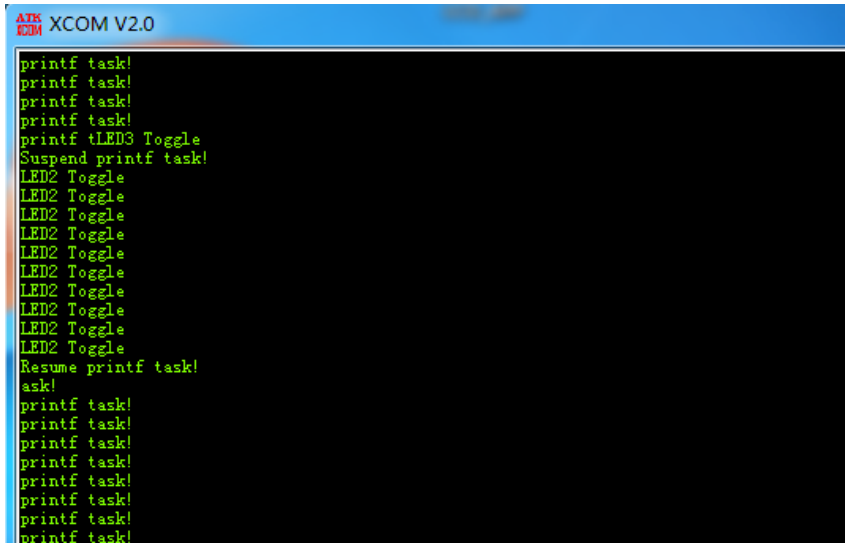
```

以上就是FreeRTOS任务管理的例程源码，程序中创建了4个任务，两个LED灯任务、一个打印任务和一个任务运行信息输出任务。在第一个LED任务中调用vTaskSuspend()函数将打印任务挂起，再调用vTaskDelay()延迟一段时间调用vTaskResume()将打印任务从挂起态切回就绪态。

程序的开始还创建了一个开始任务，在开始任务中再创建上面4个任务。可以看到开始任务只运行了一次之后就调用vTaskDelete()函数将自身删除了。所以开始任务的作用仅仅是创建其他4个任务。

编译并下载程序到目标板，运行效果通过串口打印出来，如下：

图 15. 任务管理例程演示



```

ATK XCOM V2.0
printf task!
printf task!
printf task!
printf task!
printf tLED3 Toggle
Suspend printf task!
LED2 Toggle
LED2 Toggle
LED2 Toggle
LED2 Toggle
LED2 Toggle
LED2 Toggle
LED2 Toggle
LED2 Toggle
LED2 Toggle
LED2 Toggle
LED2 Toggle
LED2 Toggle
LED2 Toggle
LED2 Toggle
LED2 Toggle
Resume printf task!
ask!
printf task!
printf task!
printf task!
printf task!
printf task!
printf task!
printf task!
printf task!

```

从打印结果可以看到，当输出了“LED3 Toggle”后，马上输出“Suspend printf task”，这时打印任务就被挂起了，不再执行；直到输出“Resume printf task”后，打印任务重新回到就绪态，这时就会得到CPU使用权，从而开始运行，打印出“printf task”。整个输出流程是完全符合程序设计思路的，从而验证了上面做讲的API函数。

6 FreeRTOS 任务调度

本节将介绍FreeRTOS的任务调度的三种方式，分别是合作式、抢占式、时间片式。这部分内容是FreeRTOS的核心，想要精通的话可能会花一些时间，学习时可查看FreeRTOS源码进行深刻的理解。

6.1 合作式调度

从FreeRTOS官网可以找到如下图中关于合作式调度的描述：

图 16. FreeRTOS 官网合作式调度描述

The screenshot displays the FreeRTOS website's 'Task Summary' section. On the left, there is a sidebar with links to various resources like 'FreeRTOS BSPs', 'Trace & Visualisation', and 'FreeRTOS+ Lab Projects'. The main content area is titled 'Task Summary' and lists several characteristics of FreeRTOS tasks using smiley face icons. A red-bordered box highlights the 'Characteristics of a 'Co-routine'' section, which includes a note about their limited use and a list of three differences: stack usage, scheduling and priorities, and macro implementation. To the right of the text, a vertical column of logos for various hardware partners is shown, including NXP, MediaTek, Renesas, RISC, SiFi, ST, Augmentex, Texas Instruments, Wittenstein, and Xilinx.

Task Summary

- Simple.
- No restrictions on use.
- Supports full preemption.
- Fully prioritised.
- Each task maintains its own stack resulting in higher RAM usage.
- Re-entrancy must be carefully considered if using preemption.

Characteristics of a 'Co-routine'

Note: Co-routines were implemented for use on very small devices, but are very rarely used in the field these days. For that reason, while there are no plans to remove co-routines from the code, there are also no plans to develop them further.

Co-routines are conceptually similar to tasks but have the following fundamental differences (elaborated further on the [co-routine documentation page](#)):

- Stack usage**
All the co-routines within an application share a single stack. This greatly reduces the amount of RAM required compared to a similar application written using tasks.
- Scheduling and priorities**
Co-routines use prioritised cooperative scheduling with respect to other co-routines, but can be included in an application that uses preemptive tasks.
- Macro implementation**

主要意思为合作式调度主要用于资源很紧张的设备上，现在已经很少使用了。处于这些原因，后期FreeRTOS官方将不再维护此模块，但是会保留。既然FreeRTOS官方都已经放弃了这种调度方式，那对于学习的意义就不大了，所以本使用指南就不对这种调度方式做讲解了，感兴趣的读者可以自行研究。

6.2 抢占式调度

相比合作式调度，抢占式调度就显得格外重要了。都知道，RTOS是一种实时性很强的操作系统，而实时性就是靠抢占式来实现的。

学习抢占式调度之前，需要了解什么是调度器。简单来说，调度器就是使用相关的调度算法来决定当前需要执行的任务。所有的调度器有一个共同的特性：

1. 调度器可以区分就绪态任务和挂起任务（由于延迟，信号量等待，邮箱等待，事件组等待等原因

而使得任务被挂起)。

2. 调度器可以选择就绪态中的一个任务，然后激活它（通过执行这个任务）。当前正在执行的任务是运行态的任务。
3. 不同调度器之间最大的区别就是如何分配就绪态任务间的完成时间。

嵌入式实时操作系统的核心就是调度器和任务切换，调度器的核心就是调度算法。任务切换的实现在不同的嵌入式实时操作系统中区别不大，基本相同的硬件内核架构，任务切换也是相似的。调度算法就有些区别了。

抢占式调度就是调度器算法的一种。在实际的应用中，不同的任务需要不同的响应时间。例如，在一个应用中需要使用电机，键盘和LCD显示。电机比键盘和LCD需要更快速的响应，如果我们使用合作式调度器或者时间片调度，那么电机将无法得到及时的响应，这时抢占式调度是必须的。如果使用了抢占式调度，最高优先级的任务一旦就绪，总能得到CPU的控制权。比如，当一个运行着的任务被其它高优先级的任务抢占，当前任务的CPU使用权就被剥夺了，或者说被挂起了，那个高优先级的任务立刻得到了CPU的控制权并运行。又比如，如果中断服务程序使一个高优先级的任务进入就绪态，中断完成时，被中断的低优先级任务被挂起，优先级高的那个任务开始运行。

使用抢占式调度器，使得最高优先级的任务什么时候可以得到CPU的控制权并运行是可知的，同时使得任务级响应时间得以最优化。总的来说，学习抢占式调度要掌握的最关键点：每个任务都被分配了不同的优先级，抢占式调度器会获得就绪列表中优先级最高的任务，并运行这个任务。

可以举一个例子来形象的看看抢占式调度是如何工作的：

1. 系统初始化，然后开启任务调度器。此时执行的最高优先级的任务Task1，Task1会一直运行直到遇到系统阻塞式的API函数，比如延迟，事件标志等待，信号量等待，Task1任务会被挂起，也就是释放CPU的执行权，让低优先级的任务得到执行。
2. FreeRTOS操作系统继续执行任务就绪列表中下一个最高优先级的任务Task2，Task2执行过程中有两种情况：
 - a) Task1由于延迟时间到，接收到信号量消息等方面的原因，使得Task1从挂起状态恢复到就绪态，在抢占式调度器的作用下，Task2的执行会被Task1抢占。
 - b) Task2会一直运行直到遇到系统阻塞式的API函数，比如延迟，事件标志等待，信号量等待，Task2任务会被挂起，继而执行就绪列表中下一个最高优先级的任务。
3. 如果用户创建了多个任务并且采用抢占式调度器的话，基本都是按照上面两条来执行。根据抢占式调度器，当前的任务要么被高优先级任务抢占，要么通过调用阻塞式API来释放CPU使用权让低优先级任务执行，没有用户任务执行时就执行空闲任务。

6.3 时间片式调度

在小型的嵌入式RTOS中，最常用的的时间片调度算法就是Round-robin调度算法。这种调度算法可以用于抢占式或者合作式的多任务中。另外，时间片调度适合用于不要求任务实时响应的情况。最常见的运用就是将时间片调度算法和抢占式调度算法配合使用。实时性要求不高的任务可以将其设定相同的优先级并且优先级设置的低些，实时性要求高的设置较高的优先级并搭配消息队列、信号量等使用。

实现Round-robin调度算法需要给同优先级的任务分配一个专门的列表，用于记录当前就绪的任务，

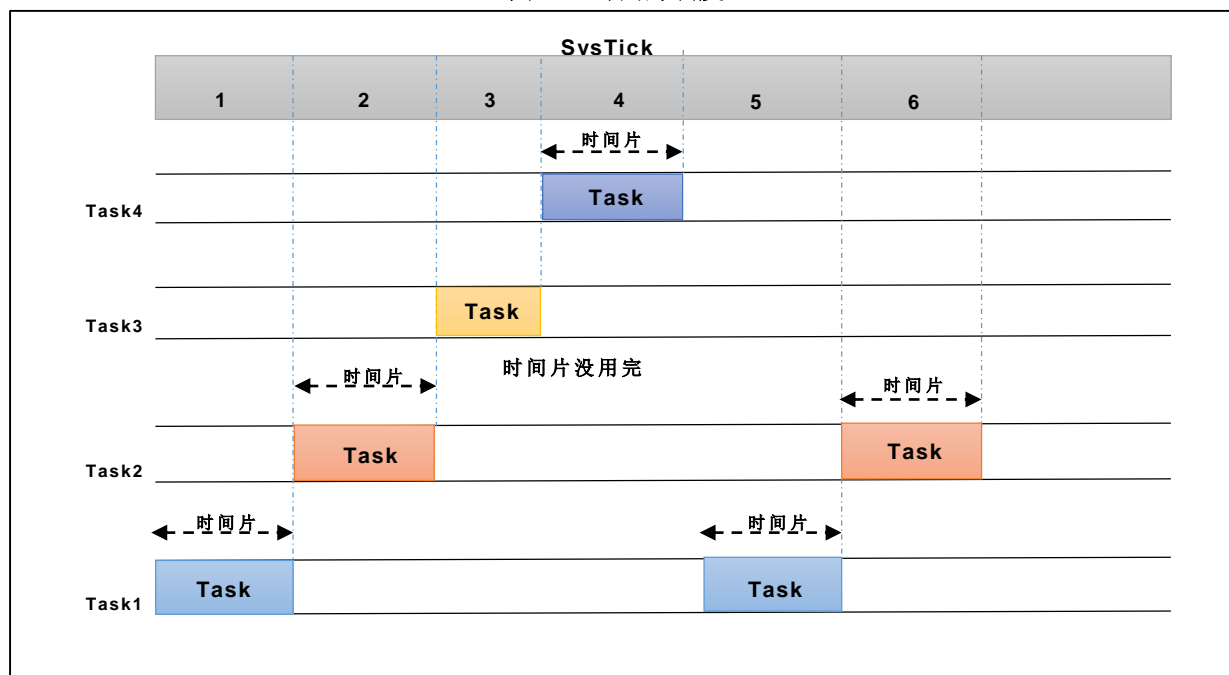
并为每个任务分配一个时间片（也就是需要运行的时间长度，时间片用完了就进行任务切换）。在FreeRTOS中只有同优先级任务才会使用时间片调度，另外还需要用户在FreeRTOSConfig.h文件中使能宏定义：`#define configUSE_TIME_SLICING 1` 默认情况下，此宏定义已经在FreeRTOS.h文件里面使能了，用户可以在FreeRTOSConfig.h文件中再单独使能。

我们可以举一个例子来形象的看看时间片调度是如何工作的：

运行条件：

1. 这里仅对时间片调度进行说明。
2. 创建4个同优先级任务Task1，Task2，Task3和Task4。
3. 每个任务分配的时间片大小是1个系统时钟节拍。

图 17. 时间片调度



根据上图的运行描述：

1. 先运行任务Task1，运行够1个系统时钟节拍后，通过时间片调度切换到任务Task2。
2. 任务Task2运行够1个系统时钟节拍后，通过时间片调度切换到任务Task3。
3. 任务Task3在运行期间调用了阻塞式API函数，调用函数时，虽然1个系统时钟节拍的时间片大小还没有用完，此时依然会通过时间片调度切换到下一个任务Task4。（注意，没有用完的时间片不会再使用，下次任务Task3得到执行还是按照1个系统时钟节拍运行）
4. 任务Task4运行够1个系统时钟节拍后，通过时间片调度切换到任务Task1。

以上就是对FreeRTOS任务调度方式的讲解，下面我们通过一个例程来实际操作一下。本节配套例程采用时间片调度方式。

6.4 例程介绍

工程名：05Time_Slicing_FreeRTOS

程序源码：

```
#include "FreeRTOS.h"
#include "task.h"

/* 开始任务优先级*/
#define START_TASK_PRIO      1
/* 开始任务堆栈大小 */
#define START_STK_SIZE      128
/* 开始任务任务句柄 */
TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* 任务 1 优先级 */
#define TASK1_PRIO      5
/* 任务 1 堆栈大小 */
#define TASK1_STK_SIZE      128
/* 任务 1 任务句柄 */
TaskHandle_t TASK1_Handler;
/* 任务 1 入口函数 */
void task1(void *pvParameters);

/* 任务 2 优先级 */
#define TASK2_PRIO      5
/* 任务 2 堆栈大小 */
#define TASK2_STK_SIZE      128
/* 任务 2 任务句柄 */
TaskHandle_t TASK2_Handler;
/* 任务 2 入口函数 */
void task2(void *pvParameters);

/* 任务 3 优先级 */
#define TASK3_PRIO      5
/* 任务 3 堆栈大小 */
#define TASK3_STK_SIZE      128
/* 任务 3 任务句柄 */
TaskHandle_t TASK3_Handler;
/* 任务 3 入口函数 */
void task3(void *pvParameters);

/* 调试任务优先级 */
#define Debug_TASK_PRIO      6
/* 调试任务堆栈大小 */
#define Debug_STK_SIZE      512
/* 调试任务任务句柄 */
TaskHandle_t DebugTask_Handler;
```

```

/* 调试任务入口函数 */
void debug_task(void *pvParameters);

int main(void)
{
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

    system_clock_config();

    at32_led_init(LED2);
    at32_led_init(LED3);
    at32_led_init(LED4);

    at32_button_init();

    /* init usart1 */
    uart_print_init(115200);

    TIMER_Init();
    /* 创建开始任务 */
    xTaskCreate((TaskFunction_t)start_task,
                (const char*   )"start_task",
                (uint16_t      )START_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t   )START_TASK_PRIO,
                (TaskHandle_t*  )&StartTask_Handler);

    /* 开启任务调度器 */
    vTaskStartScheduler();
}

/* 开始任务函数 */
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();
    /* 创建 task1 任务 */
    xTaskCreate((TaskFunction_t)task1,
                (const char*   )"task1",
                (uint16_t      )TASK1_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t   )TASK1_PRIO,
                (TaskHandle_t*  )&TASK1_Handler);

    /* 创建 task2 任务 */
    xTaskCreate((TaskFunction_t)task2,
                (const char*   )"task2",

```

```

        (uint16_t      )TASK2_STK_SIZE,
        (void*         )NULL,
        (UBaseType_t   )TASK2_PRIO,
        (TaskHandle_t* )&TASK2_Handler);

/* 创建 task3 任务 */
xTaskCreate((TaskFunction_t)task3,
            (const char*    )"task3",
            (uint16_t      )TASK3_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )TASK3_PRIO,
            (TaskHandle_t* )&TASK3_Handler);

/* 创建调试任务 */
xTaskCreate((TaskFunction_t)debug_task,
            (const char*    )"Debug_task",
            (uint16_t      )Debug_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )Debug_TASK_PRIO,
            (TaskHandle_t* )&DebugTask_Handler);

/* 删除开始任务 */
vTaskDelete(StartTask_Handler);

/* 退出临界区 */
taskEXIT_CRITICAL();
}

/* task1 任务函数 */
void task1(void *pvParameters)
{
    while(1)
    {
        printf("task1 running\r\n");
    }
}

/* task2 任务函数 */
void task2(void *pvParameters)
{
    while(1)
    {
        printf("task2 running\r\n");
    }
}

/* task3 任务函数 */
void task3(void *pvParameters)
{
    while(1)
    {
        printf("task3 running\r\n");
    }
}

```

```

    }
}

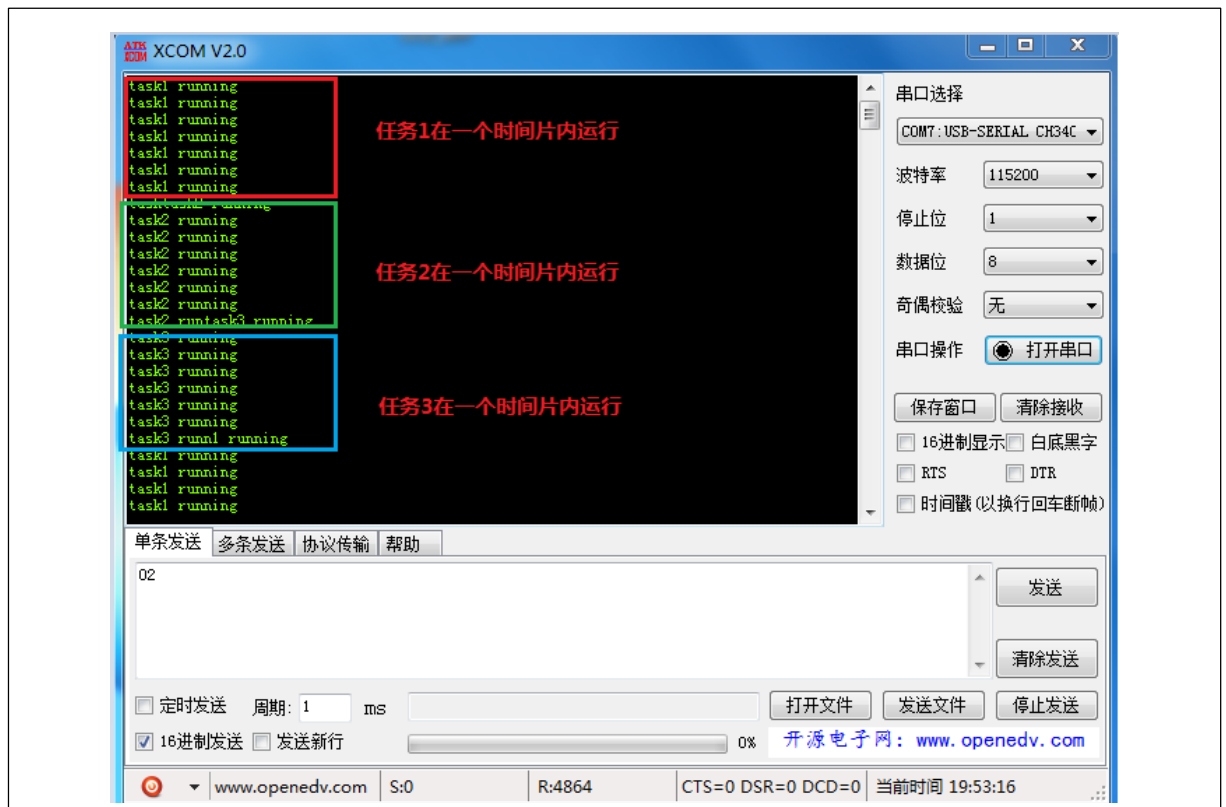
/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次调试信息 */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/-----*\r\n");
            printf("Task          Status          priority      Remaining_Stack    Num\r\n");
            vTaskList((char *)&buff);
            printf("%s\r\n",buff);
            printf("/-----*\r\n");
            printf("Task      Runing_Num      Usage_Rate\r\n");
            vTaskGetRunTimeStats((char *)&buff);
            printf("%s\r\n",buff);
        }
        vTaskDelay(10);
    }
}

```

此例程比较简单，一共4个任务。其中一个任务为打印当前系统信息，通过USER按键即可打印出来；其余三个任务都是打印一个字符串用来提示当前正在运行。需要注意的是这三个任务的任务函数内都没有会引起任务切换的函数，所以这三个任务是按照时间片调度方式运行的。从打印出来的信息也可以看出。

编译并下载程序到目标板，运行后打印出的信息如下图所示：

图 18. 时间片例程演示



从打印就可以直观的感受这三个任务是按照时间片来运行了，当一个时间片用完会自动切换到下一个任务开始运行。

7 FreeRTOS 消息队列

本节将介绍FreeRTOS的消息队列的相关使用，本节的内容很重要。消息队列不仅在实际应用中使用广泛，而且FreeRTOS一些其他的内核服务也是基于消息队列实现的。

7.1 消息队列介绍

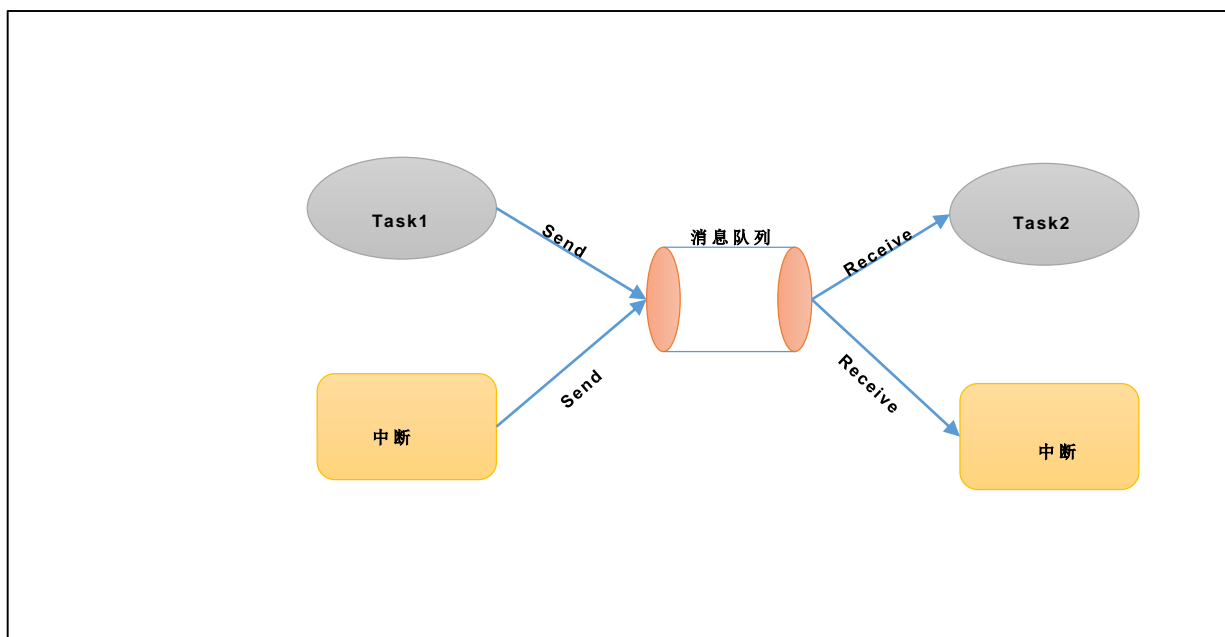
消息队列就是通过RTOS内核提供的服务，任务或中断服务子程序可以将一个消息放入到队列；同样，一个或者多个任务可以通过RTOS内核服务从队列中得到消息。通常，先进入消息队列的消息先传给任务，也就是说，任务先得到的是最先进入到消息队列的消息，即先进先出的原则（FIFO），FreeRTOS的消息队列支持FIFO和LIFO两种数据存取方式。

也许有不理解的初学者会问采用消息队列多麻烦，搞个全局数组不是更简单，其实不然。在裸机编程时，使用全局数组的确比较方便，但是在加上RTOS后就是另一种情况了。相比消息队列，使用全局数组主要有如下四个问题：

1. 使用消息队列可以让RTOS内核有效地管理任务，而全局数组是无法做到的，任务的超时等机制需要用户自己去实现。
2. 使用了全局数组就要防止多任务的访问冲突，而使用消息队列则处理好了这个问题，用户无需担心。
3. 使用消息队列可以有效地解决中断服务程序与任务之间消息传递的问题。
4. FIFO机制更有利于数据的处理。

FreeRTOS的消息可以是任务传给任务，也可以是中断传给任务。在开始消息传递之前要创建一个消息队列，创建过后这个消息队列就由FreeRTOS内核管理，发送的消息会被放进这个消息队里内，想要获取消息的任务或中断就要从这个消息队列里取消息，这样就实现了消息的互相传递。其大体过程如下图所示：

图 19. 消息队列工作流程



从上图看起来消息队列的过程并不复杂。在实际应用中，中断方式的消息机制要注意以下几点问题：

1. 中断函数的执行时间越短越好，防止其它低于这个中断优先级的异常不能得到及时响应。

2. 实际应用中，建议不要在中断中实现消息处理，用户可以在中断服务程序里面发送消息通知任务，在任务中实现消息处理，这样可以有效地保证中断服务程序的实时响应。同时此任务也需要设置为高优先级，以便退出中断函数后任务可以得到及时执行。
3. 中断服务程序中一定要调用专用于中断的消息队列函数，即以FromISR结尾的函数。
需要注意的是在操作系统中实现中断服务程序与裸机是不同的。主要有以下几点需要注意：
 1. 如果FreeRTOS工程的中断函数中没有调用FreeRTOS的消息队列API函数，与裸机编程是一样的。
 2. 如果FreeRTOS工程的中断函数中调用了FreeRTOS的消息队列的API函数，退出的时候要检测是否有高优先级任务就绪，如果有就绪的，需要在退出中断后进行任务切换，这点与裸机编程稍有区别。
 3. 强烈建议在使用AT32芯片时将中断优先级分组设置为组4，即NVIC_PriorityGroup_4。
 4. 用户要在FreeRTOS多任务开启前就设置好优先级分组，一旦设置好切记不可再修改。

7.2 消息队列相关 API

下面来看一下消息队列相关的API函数，想要熟练使用FreeRTOS的消息队列功能，就需要了解相关API函数的功能。

表 9. 消息队列 API

消息队列 API 函数	
API	描述
vQueueAddToRegistry()	给指定消息队列添加名字并增加到消息队列注册表
xQueueAddToSet()	将消息队列添加到队列集合
xQueueCreate()	创建一个消息队列
xQueueCreateSet()	创建队列集合
xQueueCreateStatic()	采用静态方式创建一个消息队列
vQueueDelete()	删除一个消息队列
char *pcQueueGetName()	查询消息队列的名字
xQueueIsQueueEmptyFromISR()	在中断内查询消息队列是否为空
xQueueIsQueueFullFromISR()	在中断内查询消息队列是否满了
uxQueueMessagesWaiting()	查询消息队列中的消息数量
uxQueueMessagesWaitingFromISR()	在中断内查询消息队列中的消息数量
xQueueOverwrite()	不管消息队列是否满了，都会将数据写进去
xQueueOverwriteFromISR()	不管消息队列是否满了，都会将数据写进去（中断内调用）
xQueuePeek()	从消息队列中获取消息，但是不会将此消息从消息队列中移除
xQueuePeekFromISR()	从消息队列中获取消息，但是不会将此消息从消息队列中移除（中断内调用）
xQueueReceive()	从消息队列中获取消息
xQueueReceiveFromISR()	从消息队列中获取消息（中断内调用）
xQueueRemoveFromSet()	从消息队列集合中移除一个消息队列
xQueueReset()	复位消息队列（清空）
xQueueSelectFromSet()	从消息队列集合中选择一个消息队列
xQueueSelectFromSetFromISR()	从消息队列集合中选择一个消息队列（中断内调用）
xQueueSend()	发送一个消息到消息队列的后面

xQueueSendToBack()	
xQueueSendToFront()	发送一个消息到消息队列的前面
xQueueSendFromISR()	发送一个消息到消息队列的后面（中断内调用）
xQueueSendToBackFromISR()	
xQueueSendToFrontFromISR()	发送一个消息到消息队列的前面（中断内调用）
uxQueueSpacesAvailable()	查询消息队列中剩余的空间

以上就是所有的消息队列相关的API函数，这里只是将其都罗列出来，后续不可能对其一一讲解，只会挑几个常用的和配套例程中使用到的作讲解。其他API函数若想详细了解，可查看FreeRTOS官方配套的API指导手册。

下面来看几个常用的API函数的原型：

xQueueCreate();

描述：

创建一个消息队列。

原型：

表 10. 创建消息队列函数原型

QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);	
参数	描述
uxQueueLength	创建消息队列的长度
uxItemSize	每个消息的大小

返回值：

创建的消息队列的句柄。

xQueueSend();

描述：

向指定消息队列发送一个消息。

原型：

表 11. 发送消息函数原型

BaseType_t xQueueSend(QueueHandle_t xQueue, const void * pvltemToQueue, TickType_t xTicksToWait);	
参数	描述
xQueue	接受消息的消息队列
pvltemToQueue	发送的消息
xTicksToWait	超时等待时间

返回值：

发送消息成功与否的标志。

xQueueSendFromISR();

描述：

在中断处理函数中，发送消息的函数。

原型：

表 12. 发送消息函数原型（中断级）

BaseType_t xQueueSendFromISR(QueueHandle_t xQueue, const void *pvItemToQueue, BaseType_t *pxHigherPriorityTaskWoken);	
参数	描述
xQueue	接受消息的消息队列
pvItemToQueue	发送的消息
pxHigherPriorityTaskWoken	出中断前是否进行任务切换标志

返回值：

发送消息成功与否的标志。

xQueueReceive();

描述：

从指定消息队列内接受消息。

原型：

表 13. 接受消息函数原型

BaseType_t xQueueReceive(QueueHandle_t xQueue, void *pvBuffer, TickType_t xTicksToWait);	
参数	描述
xQueue	指定的消息队列
pvBuffer	用于保存消息的 Buffer
xTicksToWait	超时等待时间

返回值：

接受消息成功与否标志。

xQueueReceiveFromISR();

描述：

在中断处理函数中从指定消息队列内接受消息。

原型：

表 14. 接受消息函数原型（中断级）

BaseType_t xQueueReceiveFromISR(QueueHandle_t xQueue, void *pvBuffer, BaseType_t *pxHigherPriorityTaskWoken);	
参数	描述
xQueue	指定的消息队列
pvBuffer	用于保存消息的 Buffer
pxHigherPriorityTaskWoken	出中断前是否进行任务切换标志

返回值:

接受消息成功与否标志。

以上对几个API做了详细的讲解，本节配套例程中会用到这几个函数，通过例程可以更直观的感受消息队列的使用方式。

7.3 例程介绍

工程名: **06Message_Queue_FreeRTOS**

程序源码:

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* 开始任务优先级*/
#define START_TASK_PRIO      1
/* 开始任务堆栈大小 */
#define START_STK_SIZE      128
/* 开始任务任务句柄 */
TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* 消息处理任务优先级 */
#define Process_Message_TASK_PRIO      1
/* 消息处理任务堆栈大小 */
#define Process_Message_STK_SIZE      256
/* 消息处理任务任务句柄 */
TaskHandle_t Process_MessageTask_Handler;
/* 消息处理任务入口函数 */
void Process_Message_task(void *pvParameters);

/* 消息接受任务优先级 */
#define Receive_Message_TASK_PRIO      3
/* 消息接受任务堆栈大小 */
#define Receive_Message_STK_SIZE      256
/* 消息接受任务任务句柄 */
TaskHandle_t Receive_MessageTask_Handler;
/* 消息接受任务入口函数 */
void Receive_Message_task(void *pvParameters);

/* 调试任务优先级 */
#define Debug_TASK_PRIO      3
/* 调试任务堆栈大小 */
```

```
#define Debug_STK_SIZE      512

/* 调试任务任务句柄 */
TaskHandle_t DebugTask_Handler;

/* 调试任务入口函数 */
void debug_task(void *pvParameters);

/* 定义一个消息结构体 */
typedef struct A_Message
{
    char ucMessageID;
    u8 ucData;
} AMessage;

/* 定义一个消息队列 */
QueueHandle_t AT_xQueue;

/* 消息队列长度和消息大小 */
#define QUEUE_LENGTH 5
#define QUEUE_ITEM_SIZE sizeof( AMessage )

int main(void)
{
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

    system_clock_config();

    at32_led_init(LED2);
    at32_led_init(LED3);
    at32_led_init(LED4);

    at32_button_init();

    /* init usart1 */
    uart_print_init(115200);

    /* 创建开始任务 */
    xTaskCreate((TaskFunction_t)start_task,
                (const char*  )"start_task",
                (uint16_t      )START_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t    )START_TASK_PRIO,
                (TaskHandle_t*  )&StartTask_Handler);

    /* 开启任务调度器 */
    vTaskStartScheduler();
}
```

```

/* 开始任务函数 */
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();
    /* 创建消息队列 */
    AT_xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );
    if(AT_xQueue == NULL)
    {
        /* 消息队列创建失败 */
        while(1);
    }
    /* 必须在创建消息队列之后再初始化定时器 */
    TIMER_Init();
    /* 创建消息接受任务 */
    xTaskCreate((TaskFunction_t)Receive_Message_task,
                (const char*   )"Receive_Message_task",
                (uint16_t       )Receive_Message_STK_SIZE,
                (void*          )NULL,
                (UBaseType_t    )Receive_Message_TASK_PRIO,
                (TaskHandle_t*   )&Receive_MessageTask_Handler);
    /* 创建消息处理任务 */
    xTaskCreate((TaskFunction_t)Process_Message_task,
                (const char*   )"Process_Message_task",
                (uint16_t       )Process_Message_STK_SIZE,
                (void*          )NULL,
                (UBaseType_t    )Process_Message_TASK_PRIO,
                (TaskHandle_t*   )&Process_MessageTask_Handler);
    /* 创建调试任务 */
    xTaskCreate((TaskFunction_t)debug_task,
                (const char*   )"Debug_task",
                (uint16_t       )Debug_STK_SIZE,
                (void*          )NULL,
                (UBaseType_t    )Debug_TASK_PRIO,
                (TaskHandle_t*   )&DebugTask_Handler);
    /* 删除开始任务 */
    vTaskDelete(StartTask_Handler);
    /* 退出临界区 */
    taskEXIT_CRITICAL();
}
/* 消息接受任务函数 */
void Receive_Message_task(void *pvParameters)
{
    AMessage Message1;
    while(1)

```

```
{
    switch(GetUartData())
    {
        case NODATA:
            break;
        case 0x01:
            Message1.ucMessageID = 'a';
            Message1.ucData = 0x01;
            /* 接受 Toggle LED2 的消息 */
            xQueueSend(AT_xQueue,&Message1,10);
            break;
        case 0x02:
            Message1.ucMessageID = 'b';
            Message1.ucData = 0x02;
            /* 接受 Toggle LED3 的消息 */
            xQueueSend(AT_xQueue,&Message1,10);
            break;
        case 0x03:
            Message1.ucMessageID = 'c';
            Message1.ucData = 0x03;
            /* 接受 Toggle LED4 的消息 */
            xQueueSend(AT_xQueue,&Message1,10);
            break;
        default:
            break;
    }
    vTaskDelay(100);
}
}
/* 消息处理任务函数 */
void Process_Message_task(void *pvParameters)
{
    AMessage Message2;
    while(1)
    {
        /* 接受消息 */
        if(xQueueReceive(AT_xQueue, &Message2, portMAX_DELAY) != pdPASS)
        {
            printf("no message\r\n");
        }
        else
        {
            /* 判断消息类型并做出相应动作 */
            if((Message2.ucData == 0x01)&&(Message2.ucMessageID == 'a'))
                at32_led_toggle(LED2);
        }
    }
}
```

```

        if((Message2.ucData == 0x02)&&(Message2.ucMessageID == 'b'))
            at32_led_toggle(LED3);
        if((Message2.ucData == 0x03)&&(Message2.ucMessageID == 'c'))
            at32_led_toggle(LED4);
        if((Message2.ucData == 0x04)&&(Message2.ucMessageID == 'd'))
            printf("Timer interrupt\r\n");
    }
}
}
/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次调试信息 */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/*-----*/\r\n");
            printf("Task          Status          priority      Remaining_Stack      Num\r\n");
            vTaskList((char *)&buff);
            printf("%s\r\n",buff);
            printf("/*-----*/\r\n");
            printf("Task          Runing_Num      Usage_Rate\r\n");
            vTaskGetRunTimeStats((char *)&buff);
            printf("%s\r\n",buff);
        }
        vTaskDelay(10);
    }
}

void TMR3_GLOBAL_IRQHandler(void)
{
    AMessage Message3;
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    if(tmr_flag_get(TMR3,TMR_OVF_FLAG)==SET)
    {
        Message3.ucMessageID = 'd';
        Message3.ucData = 0x04;
        /* 发送定时器中断消息到消息队列 */
        xQueueSendFromISR(AT_xQueue,&Message3,&xHigherPriorityTaskWoken);
        /* 判断是否需要任务切换 */
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
        tmr_flag_clear(TMR3,TMR_OVF_FLAG);
    }
}

```

```
}
```

以上是消息队列配套例程的源程序。程序中主要有3个任务会运行（开始任务这里不讨论），第一个是调试信息打印任务，这个任务在检测到目标板的USER按键按下后就会打印一次当前系统的任务信息情况，包括有任务名、任务状态、任务剩余堆栈大小、任务序号、任务占用的CPU时间等。第二个任务是消息接受任务，这个任务会定期去串口缓存区内读取数据，如果串口缓存区内有数据就会读取回来然后分析数据，分析数据后就会将分析的结果发送到消息队列中。第三个任务是消息处理任务，当消息队列中有数据的话这个任务就会运行，并从消息队列中接受消息，然后根据不同的消息可以反转目标板的LED2/3/4。程序中还开启了一个硬件定时器，当定时器产生溢出中断时就会向消息队列里发送消息，处理消息的任务接收到此消息后就会打印出“Timer interrupt”。

注：在运行此例程时，需保证硬件环境正确。要保证目标板的串口1的管脚正确连接到PC（可使用USB转串口工具）。

编译并下载程序到目标板，运行效果如下：

图 20. 消息队列例程演示



从上图中的打印信息可以看到，定时器中断发送消息到消息队列正常，并且当按键按下后也可输出任务信息。

下面来看如何通过上位机发送消息从而控制目标板的LED灯。

图 21. 消息队列例程演示



通过发送0x01、0x02、0x03就可以分别控制LED2/3/4了。需要注意的是要勾选以16进制发送。

8 FreeRTOS 信号量

本节将介绍FreeRTOS的信号量相关的内容，包括二值信号量、计数型信号量、互斥信号量、递归互斥信号量。此部分内容为任务间通信和资源共享的重要手段之一，所以掌握是很有必要的。

8.1 什么是信号量

信号量（semaphores）的发明可以追述到20世纪60年代中期。使用信号量的最初目的是为了给共享资源建立一个标志，该标志表示该共享资源被占用情况。这样，当一个任务在访问共享资源之前，就可以先对这个标志进行查询，从而在了解资源被占用的情况之后，再来决定自己的行为。

在实际生活中，信号量其实无处不在，例如停车场，假如停车场有一百个停车位，那么在没有车辆驶入的情况下信号量就是100，当驶入一辆汽车信号量减1，驶出一辆汽车信号量就加1。当驶入的汽车为100的时候，信号量就会为0，这个时候如果还有汽车准备驶入停车场就无法再获取到信号量了从而禁止驶入。直到有汽车驶出停车场后信号量才会被释放，这个时候之前想要驶入停车场的汽车就会获得刚刚释放的信号量从而准许进入停车场。

在实际应用中使用信号量主要是实现以下两个功能：

1. 两个任务之间或者中断函数跟任务之间的同步功能，其实就是共享资源为1的时候。
2. 多个共享资源的管理，就像上面举的停车场的例子。

针对这两种功能，FreeRTOS分别提供了二值信号量和计数信号量，其中二值信号量可以理解成计数信号量的一种特殊形式，即初始化为仅有一个资源可以使用，只不过FreeRTOS对这两种都提供了API函数，而像RTX，uCOS-II/III是仅提供了一个信号量功能，设置不同的初始值就可以分别实现二值信号量和计数信号量。当然，FreeRTOS使用计数信号量也能够实现同样的效果。

8.2 二值信号量

8.2.1 二值信号量介绍

二值信号量通常用于互斥访问或同步，二值信号量和互斥信号量很相似，但是还是存在细微的差别，主要在于互斥信号量有优先级继承机制，而二值信号量没有（该内容后续会讲到）。因此二值信号量更适合做同步使用（任务与任务间同步或任务和中断间同步），而互斥信号量适合用于简单的互斥访问。

和队列一样，信号量API函数也允许设置一个阻塞时间，阻塞时间是指当任务获取信号量的时候由于信号量无效而进入阻塞态的最大时间节拍数，即如果等待信号量的时间到达了设定的阻塞时间则任务会放弃等待该信号量。如果同时有多个任务等待同一信号量，则最高优先级的任务会优先获取信号量从而解除阻塞状态。

其实二值信号量就是一个队列项为1的队列。这个特殊的队列要么是满的，要么是空的，这样正好就是二值了。任务和中断不需要知道这个队列中消息的内容，只需要知道是空的还是满的即可。只要知道了是空的还是满的就可以利用信号量机制实现任务和任务、任务和中断间的同步。

下面来看一看FreeRTOS官网提供的一个二值信号量的简易框图：

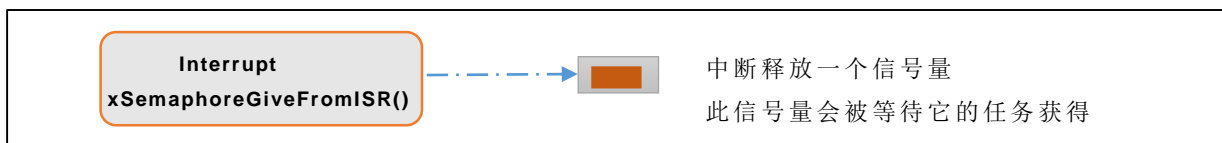
1. 任务想获取信号量，但是由于没有信号量，所以进入阻塞状态。

图 22. 二值信号量框图 1



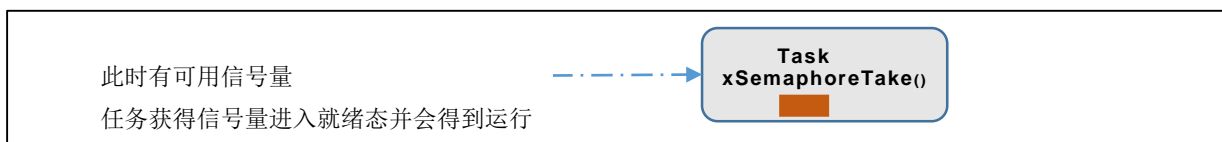
2. 某中断应用程序内释放了一个信号量。

图 23. 二值信号量框图 2



3. 等待信号量的任务可以获得信号量，并从阻塞状态进入到运行态。

图 24. 二值信号量框图 3



以上三个步骤为二值信号量的运行过程，其中红色实心矩形代表二值信号量，可以看到二值信号量必须被释放了才会被另一个任务获得，如果需要该二值信号量的任务无法获得就会进入阻塞态，直到此二值信号量被释放并获得。

8.2.2 二值信号量 API

下面来看看二值信号量的相关API函数，FreeRTOS为二值信号量专门开放了一些API函数供用户调用，使用二值信号量的功能就是调用API的过程，用户只需在应用程序中正确的调用这些API函数接口就可以了。

表 15. 二值信号量 API

二值信号量 API 函数	
API	描述
xSemaphoreCreateBinary()	创建一个二值信号量
xSemaphoreCreateBinaryStatic()	静态创建一个二值信号量
vSemaphoreDelete()	删除一个信号量
xSemaphoreGive()	释放一个信号量
xSemaphoreGiveFromISR()	内释放一个信号量（中断级）
xSemaphoreTake()	获取一个信号量
xSemaphoreTakeFromISR()	获取一个信号量（中断级）

以上就是二值信号量相关的API函数，其实就是二值信号量的创建、删除、释放、获取相关的API函数，使用起来比较简单。

注： 以上的释放、获取、删除相关的API在二值信号量、计数型信号量、递归信号量都是通用的。来看看二值信号量的常用API的原型。

xSemaphoreCreateBinary ();

描述:

创建一个二值信号量。

原型:

表 16. 创建互斥信号量函数原型

SemaphoreHandle_t xSemaphoreCreateBinary(void);	
参数	描述
Void	无

返回值:

创建的二值信号量的句柄。

vSemaphoreDelete ();

描述:

删除信号量。

原型:

表 17. 删除信号量函数原型

void vSemaphoreDelete(SemaphoreHandle_t xSemaphore);	
参数	描述
xSemaphore	要删除的信号量的句柄

返回值:

无。

xSemaphoreGive ();

描述:

释放信号量。

原型:

表 18. 释放信号量函数原型

BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore);	
参数	描述
xSemaphore	要释放的信号量的句柄

返回值:

释放成功与否标志。

注: *xSemaphoreGiveFromISR()* 函数为中断级的信号量释放API。

xSemaphoreTake ();

描述:

获取信号量。

原型:

表 19. 获取信号量函数原型

BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);	
参数	描述
xSemaphore	要获取的信号量的句柄
xTicksToWait	超时等待时间

返回值:

获取成功与否标志。

注: *xSemaphoreTakeFromISR ()*函数为中断级的信号量获取API。
以上是二值信号量常用的几个API原型介绍，下面来看看本小节配套的例程。

8.2.3 例程介绍

工程名: **07Binary_Semaphore_FreeRTOS**

程序源码:

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"

/* 开始任务优先级*/
#define START_TASK_PRIO      1
/* 开始任务堆栈大小 */
#define START_STK_SIZE      128
/* 开始任务任务句柄 */
TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* 消息处理任务优先级 */
#define Process_Binary_Semaphore_TASK_PRIO      1
/* 消息处理任务堆栈大小 */
#define Process_Binary_Semaphore_STK_SIZE      256
/* 消息处理任务任务句柄 */
TaskHandle_t Process_Binary_Semaphore_Task_Handler;
/* 消息处理任务入口函数 */
void Process_Binary_Semaphore_task(void *pvParameters);

/* 调试任务优先级 */
#define Debug_TASK_PRIO      3
/* 调试任务堆栈大小 */
#define Debug_STK_SIZE      512
/* 调试任务任务句柄 */
TaskHandle_t DebugTask_Handler;
/* 调试任务入口函数 */
void debug_task(void *pvParameters);
```

```
/* 定义信号量 */
SemaphoreHandle_t AT_xSemaphore;

/* 定义一个变量，接收二值到信号量后对其加 1 操作 */
int Take_Semaphore_Counter = 0;

int main(void)
{
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

    system_clock_config();

    at32_led_init(LED2);
    at32_led_init(LED3);
    at32_led_init(LED4);

    at32_button_init();

    /* init usart1 */
    uart_print_init(115200);

    /* 创建开始任务 */
    xTaskCreate((TaskFunction_t)start_task,
                (const char* )"start_task",
                (uint16_t )START_STK_SIZE,
                (void* )NULL,
                (UBaseType_t )START_TASK_PRIO,
                (TaskHandle_t* )&StartTask_Handler);

    /* 开启任务调度器 */
    vTaskStartScheduler();
}

/* 开始任务函数 */
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();

    /* 创建二值信号量 */
    AT_xSemaphore = xSemaphoreCreateBinary();

    if( AT_xSemaphore == NULL )
    {
        /* 创建二值信号量失败 */
    }
}
```

```

while(1);
}

/* 必须在创建消息队列之后再初始化定时器 */
TIMER_Init();

/* 创建消息处理任务 */
xTaskCreate((TaskFunction_t)Process_Binary_Semaphore_task,
            (const char*   )"Semaphore_task",
            (uint16_t      )Process_Binary_Semaphore_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )Process_Binary_Semaphore_TASK_PRIO,
            (TaskHandle_t* )&Process_Binary_Semaphore_Task_Handler);

/* 创建调试任务 */
xTaskCreate((TaskFunction_t)debug_task,
            (const char*   )"Debug_task",
            (uint16_t      )Debug_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )Debug_TASK_PRIO,
            (TaskHandle_t* )&DebugTask_Handler);

/* 删除开始任务 */
vTaskDelete(StartTask_Handler);

/* 退出临界区 */
taskEXIT_CRITICAL();
}

/* 消息处理任务函数 */
void Process_Binary_Semaphore_task(void *pvParameters)
{
    while(1)
    {
        /* 等待接受二值信号量 */
        if( xSemaphoreTake( AT_xSemaphore, portMAX_DELAY ) == pdTRUE )
        {
            /* 加 1 操作 */
            Take_Semaphore_Counter++;
            printf("The %dth semaphore is received.\r\n",Take_Semaphore_Counter);
        }
    }
}

/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];

```

```

while(1)
{
    /* 按下按键打印一次调试信息 */
    if(at32_button_press() == USER_BUTTON)
    {
        printf("/-----*\r\n");
        printf("Task          Status          priority      Remaining_Stack      Num\r\n");
        vTaskList((char *)&buff);
        printf("%s\r\n",buff);
        printf("/-----*\r\n");
        printf("Task          Runing_Num      Usage_Rate\r\n");
        vTaskGetRunTimeStats((char *)&buff);
        printf("%s\r\n",buff);
    }
    vTaskDelay(10);
}

void TMR3_GLOBAL_IRQHandler(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    if(tmr_flag_get(TMR3,TMR_OVF_FLAG)==SET)
    {
        printf("Send the semaphore for the %dth time.\r\n",Take_Semaphore_Counter+1);
        /* 发送一个信号量 */
        xSemaphoreGiveFromISR( AT_xSemaphore, &xHigherPriorityTaskWoken );

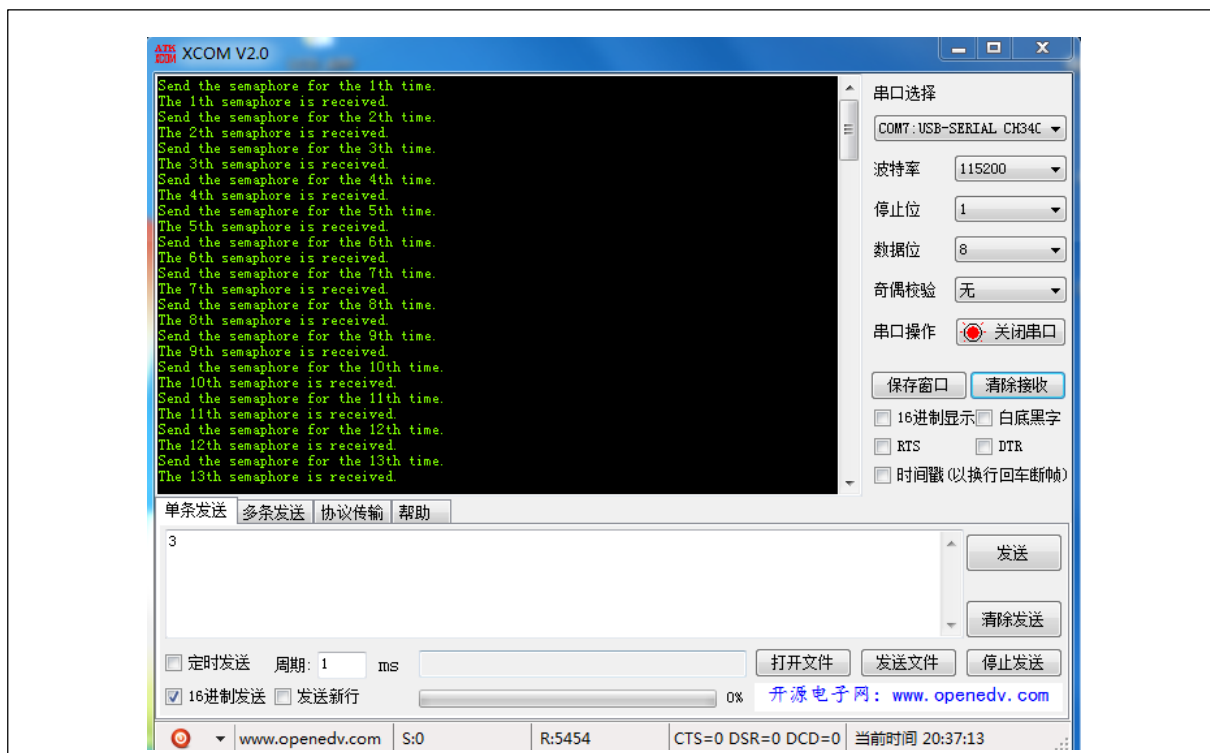
        /* 判断是否需要任务切换 */
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
        tmr_flag_clear(TMR3,TMR_OVF_FLAG);
    }
}

```

以上就是二值信号量部分配套的例程源码。例程中主要实现的任务与中断之间的同步。当发生定时器中断时，在中断处理程序中发送一个二值信号量，这时等待该二值信号量的任务就会从阻塞态转入到就绪态从而得到运行；当任务运行并消耗掉这个二值信号量后，任务又会进入到阻塞态等待这个二值信号量。程序同时提供了调试信息打印的任务，只要按下目标板的**USER**按键就能通过串口打印出当前系统中所有任务的信息。

编译并下载程序到目标板，运行效果如下：

图 25. 二值信号量例程演示



上图就是通过串口打印出来的运行结果，可以看到每当定时器中断函数中释放一次二值信号量，那么等待此二值信号量的任务就会运行一次。

图 26. 二值信号量例程演示



上图为按下USER按键后，打印出来的任务系统中各个任务的信息。具体代码细节可查看配套的工程。

8.3 计数型信号量

8.3.1 计数型信号量介绍

本节来介绍计数型信号量，也可将其称为数值信号量。前面介绍的二值信号量只能为0或者1，那么计数型信号量就是可以大于1的信号量。信号量的本质就是队列，只是不用关注队列中存了什么消息，只需关心队列是否满即可。计数型信号量通常有以下两个应用场景：

1. 事件计数

在这种使用场景下，每次事件发生的时候就在事件处理函数中释放信号量（增加信号量的计数值），其他任务事件会获取信号值，获取一次就在任务事件处理函数中对信号量减1操作。这种场合下创建的计数信号量的初始值为0。

2. 资源管理

在这种使用场景下，信号量代表当前资源的可用数量，例如前面停车场的例子中描述的那样，停车场剩余的车位就是当前可用资源数量（信号量）。任务事件获取一次资源后信号量的值就减1操作（车辆驶入停车场），任务事件释放一次资源后信号量加1操作（车辆驶出停车场）。在这个场合中信号量的初始值为资源的总数量，例如有100个车位，那么信号量初始值为100。

8.3.2 计数型信号量 API

下面来看看计数型信号量的API函数：

表 20. 计数型信号量 API

互斥信号量 API 函数	
API	描述
xSemaphoreCreateCounting()	创建一个计数型信号量
xSemaphoreCreateCountingStatic()	静态创建一个计数型信号量
vSemaphoreDelete()	删除一个信号量
xSemaphoreGive()	释放一个信号量
xSemaphoreGiveFromISR()	内释放一个信号量（中断级）
xSemaphoreTake()	获取一个信号量
xSemaphoreTakeFromISR()	获取一个信号量（中断级）
uxSemaphoreGetCount()	获取当前信号量的值

以上就是计数型信号量相关的API函数，其实就是计数型信号量的创建、删除、释放、获取相关的API函数，使用起来比较简单。

注： 以上的释放、获取、删除相关的API在二值信号量、计数型信号量、递归信号量都是通用的。来看看计数型信号量的常用API的原型。

xSemaphoreCreateCounting ();

描述：

创建一个计数型信号量。

原型：

表 21. 创建二值信号量函数原型

SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount, UBaseType_t uxInitialCount);	
参数	描述
uxMaxCount	计数型信号量允许的最大值
uxInitialCount	计数型信号量初始化值

返回值:

创建的计数型信号量的句柄。

vSemaphoreDelete ();

描述:

删除信号量。

原型:

表 22. 删除信号量函数原型

void vSemaphoreDelete(SemaphoreHandle_t xSemaphore);	
参数	描述
xSemaphore	要删除的信号量的句柄

返回值:

无。

xSemaphoreGive ();

描述:

释放信号量。

原型:

表 23. 释放信号量函数原型

BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore);	
参数	描述
xSemaphore	要释放的信号量的句柄

返回值:

释放成功与否标志。

注: xSemaphoreGiveFromISR()函数为中断级的信号量释放API。

xSemaphoreTake ();

描述:

获取信号量。

原型:

表 24. 获取信号量函数原型

BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);	
参数	描述

xSemaphore	要获取的信号量的句柄
xTicksToWait	超时等待时间

返回值:

获取成功与否标志。

注: *xSemaphoreTakeFromISR ()*函数为中断级的信号量获取API。

uxSemaphoreGetCount ();

描述:

获取信号量值。

原型:

表 25. 获取信号量值函数原型

UBaseType_t uxSemaphoreGetCount(SemaphoreHandle_t xSemaphore);	
参数	描述
xSemaphore	要获取的信号量的句柄

返回值:

获取信号量的数目。

以上是计数型信号量常用的几个API原型介绍，下面来看看本小节配套的例程。

8.3.3 例程介绍

工程名: **08Counting_Semaphore_FreeRTOS**

程序源码:

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"

/* 开始任务优先级*/
#define START_TASK_PRIO      1
/* 开始任务堆栈大小 */
#define START_STK_SIZE       128
/* 开始任务任务句柄 */
TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* 消息处理任务优先级 */
#define Process_Counting_Semaphore_TASK_PRIO      1
/* 消息处理任务堆栈大小 */
#define Process_Counting_Semaphore_STK_SIZE       256
/* 消息处理任务任务句柄 */
TaskHandle_t Process_Counting_Semaphore_Task_Handler;
/* 消息处理任务入口函数 */
```

```

void Process_Counting_Semaphore_task(void *pvParameters);

/* 调试任务优先级 */
#define Debug_TASK_PRIO      3
/* 调试任务堆栈大小 */
#define Debug_STK_SIZE      512
/* 调试任务任务句柄 */
TaskHandle_t DebugTask_Handler;
/* 调试任务入口函数 */
void debug_task(void *pvParameters);

/* 定义信号量 */
SemaphoreHandle_t AT_xSemaphore;
/* 信号量初始值 */
uint8_t Semaphore_Initail_Vaule = 10;
/* 信号量最大值 */
uint8_t Semaphore_Max_Vaule = 10;

/* 剩余信号量数目 */
int Remaining_Semaphore = 0;

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);
    at32_button_init();
    /* init usart1 */
    uart_print_init(115200);
    /* 创建开始任务 */
    xTaskCreate((TaskFunction_t)start_task,
                (const char* )"start_task",
                (uint16_t )START_STK_SIZE,
                (void* )NULL,
                (UBaseType_t )START_TASK_PRIO,
                (TaskHandle_t* )&StartTask_Handler);
    /* 打开调度器 */
    vTaskStartScheduler();
}

/* 开始任务函数 */
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();

    /* 创建计数型信号量 */

```

```

AT_xSemaphore =
xSemaphoreCreateCounting(Semaphore_Max_Vaule, Semaphore_Initail_Vaule);

if( AT_xSemaphore == NULL )
{
    /* 创建计数型信号量失败 */
    while(1);
}

/* 必须在创建消息队列之后再初始化定时器 */
TIMER_Init();

/* 创建消息处理任务 */
xTaskCreate((TaskFunction_t)Process_Counting_Semaphore_task,
            (const char*   )"Semaphore_task",
            (uint16_t      )Process_Counting_Semaphore_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t    )Process_Counting_Semaphore_TASK_PRIO,
            (TaskHandle_t*  )&Process_Counting_Semaphore_Task_Handler);

/* 创建调试任务 */
xTaskCreate((TaskFunction_t)debug_task,
            (const char*   )"Debug_task",
            (uint16_t      )Debug_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t    )Debug_TASK_PRIO,
            (TaskHandle_t*  )&DebugTask_Handler);

/* 删除开始任务 */
vTaskDelete(StartTask_Handler);

/* 退出临界区 */
taskEXIT_CRITICAL();
}

/* 消息处理任务函数 */
void Process_Counting_Semaphore_task(void *pvParameters)
{
    while(1)
    {
        /* 等待接受计数型信号量 */
        if( xSemaphoreTake( AT_xSemaphore, portMAX_DELAY ) == pdTRUE )
        {
            printf("Semaphore is received.\r\n");
            /* 获取剩余信号量数目 */
            Remaining_Semaphore = uxSemaphoreGetCount(AT_xSemaphore);
        }
    }
}

```

```

        printf("%d semaphore are left.\r\n", Remaining_Semaphore);
    }
}

/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次任务信息 */
        if(AT32_BUTTON_Press() == BUTTON_WAKEUP)
        {
            printf("/*-----*\r\n");
            printf("Task      Status  priority  Remaining_Stack  Num\r\n");
            vTaskList((char *)&buff);
            printf("%s\r\n", buff);
            printf("/*-----*\r\n");
            printf("Task          Runing_Num      Usage_Rate\r\n");
            vTaskGetRunTimeStats((char *)&buff);
            printf("%s\r\n", buff);
        }
        vTaskDelay(10);
    }
}

/* TMR3 中断函数 */
void TMR3_GLOBAL_IRQHandler(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    if(TMR_GetFlagStatus(TMR3, TMR_FLAG_Update)==SET)
    {
        printf("TMR3 IRQ release a semaphore.\r\n");
        /* 发送一个信号量 */
        xSemaphoreGiveFromISR( AT_xSemaphore,  &xHigherPriorityTaskWoken );

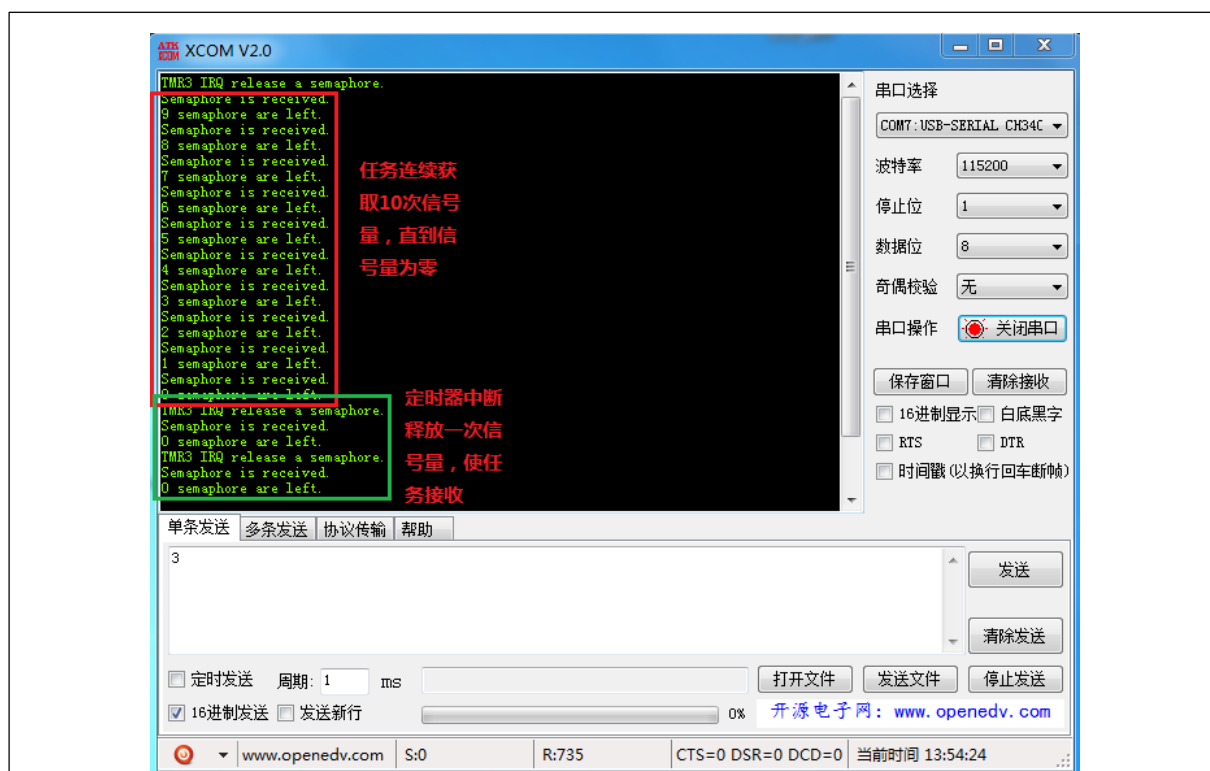
        /* 判断是否需要任务切换 */
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

        /* 清除 TMR3 中断标志 */
        TMR_ClearFlag(TMR3, TMR_FLAG_Update);
    }
}

```

以上就是计数型信号量相关的程序，程序开始会创建一个计数型信号量，并将最大信号量和初始信号量都设置成10。然后创建一个获取信号量的任务，任务函数内不断获取信号量，最开始信号量有10个，那么任务会连续获取10次信号量，等到将信号量全部获取完之后，任务进入阻塞态等待信号量。程序中会初始化一个硬件定时器，在定时器的中断处理函数中会释放一个信号量，从而等待该信号量的任务就会从阻塞态转入就绪态并最终得到运行。当任务获取到硬件定时器释放的信号量之后发现没有信号量可获取了，又转入阻塞态等待该信号量，如此循环。本例程可通过打印信息查看运行结果。编译并下载程序到目标板，运行效果如下：

图 27. 计数型信号量例程演示



上图就是计数型信号量例程的演示效果，可以看到运行结果和程序逻辑相符，任务在消耗完所有信号量后便进入阻塞态等待再次有可获取的信号量。

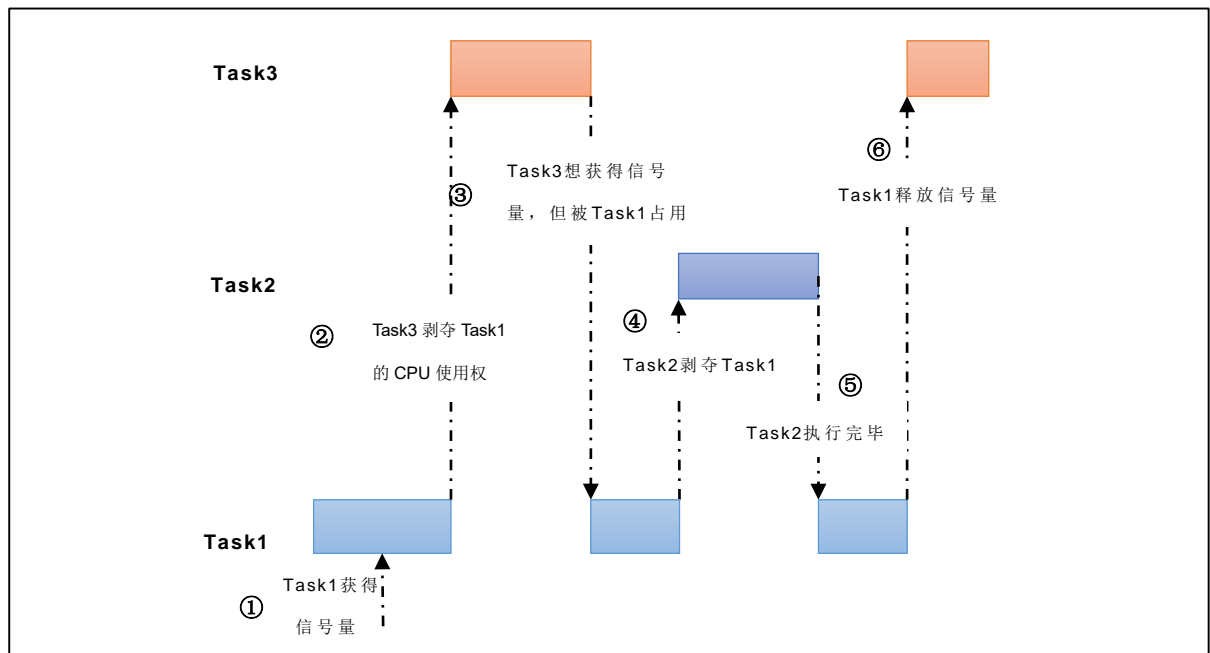
8.4 互斥信号量

8.4.1 优先级翻转

在使用二值信号量的时候很有可能会碰到优先级翻转的问题，优先级翻转在可剥夺型内核是很常见的问题。但是在实时系统中是不允许出现这种情况的，这会打乱系统的预期执行顺序，导致低优先级的任务在高优先级任务之前运行这违背了实时性的初衷。

下面来分析下出现优先级翻转的情况：

图 28. 优先级翻转



上图体现的优先级翻转的一种情况：

- 1) 此时Task1正在运行，并获得二值信号量；
- 2) 此时Task3转为就绪态，优先级最高，剥夺Task1的CPU使用权；但是二值信号量还是被Task1所持有；
- 3) 此时Task3想要得到二值信号量，但是Task1并未释放信号量，所以被迫让出CPU使用权，进入阻塞态等待信号量释放；
- 4) 此时Task1运行，但Task2转入到就绪态，由于Task2优先级高于Task1所以CPU使用权归Task2所有；
- 5) Task2执行完毕释放CPU使用权，此时由于Task3在等待二值信号量，所以不能得到运行，Task1获得CPU使用权。此时就发生了优先级翻转，本身Task3的优先级最高，但是由于要等待信号量其优先级降低到了与Task1一样了，此时由于Task2的执行，使情况更加糟糕了。

以上的优先级翻转的现象，在一些实时性要求极高的应用中，往往是致命的，不可挽救的。所以就引入了互斥信号量的概念，互斥信号量可以回避这个问题。

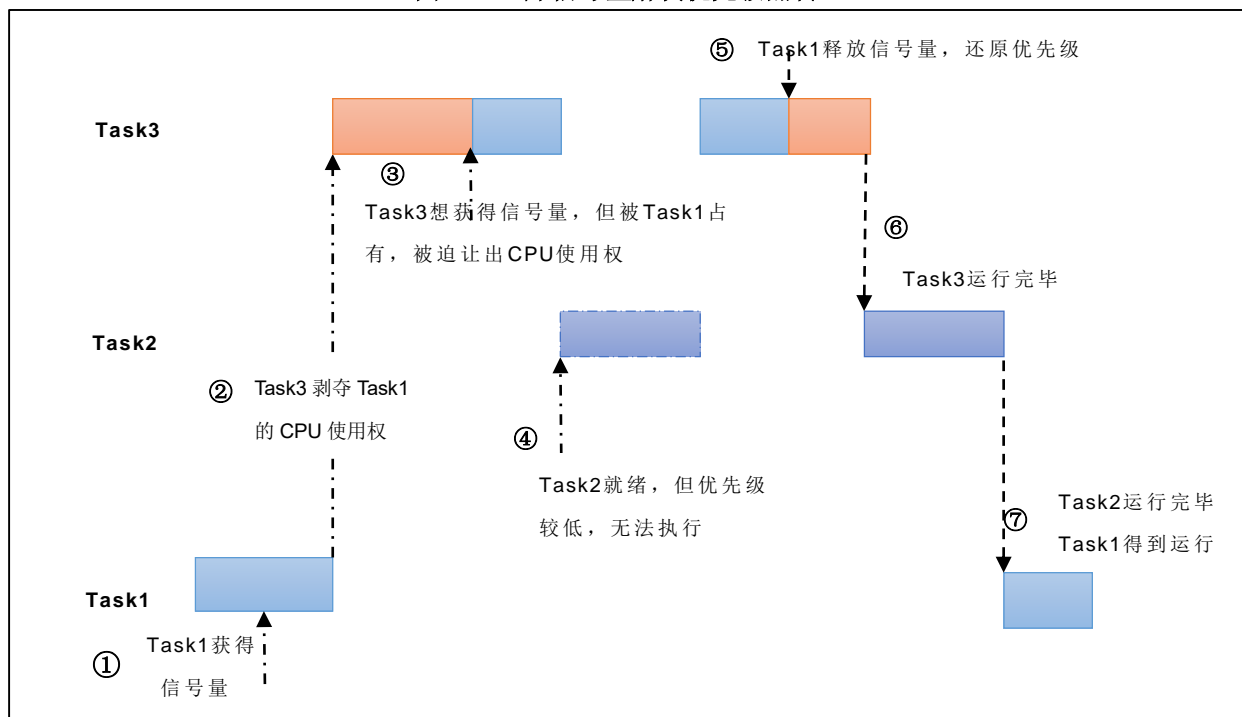
8.4.2 互斥信号量介绍

从互斥信号量这个名字就知道是为了资源互斥访问而设计的，它和二值信号量都有资源互斥访问的功能，只是二值信号量可能会发生前面所讲的优先级翻转的问题，而互斥信号量则采取了措施尽量回避这个问题。互斥信号量可回避优先级翻转问题的原因为：当一个高优先级任务想要获取某互斥信号量，但是该信号量被某低优先级的任务所持有，此时高优先级的任务就会进入阻塞态，在进入阻塞态之前此高优先级的任务会将持有互斥信号量的低优先级的任务的优先级提高到和高优先级任务相同的优先级。

这样持有互斥信号量的低优先级的任务就不会被其他中间等级的优先级的任务所抢占CPU的使用权，尽可能的缩短了高优先级任务的响应时间。可以发现，其实高优先级的任务的响应速度还是受到了影响，不会马上就抢占低优先级任务的CPU使用权和获得互斥信号量，但是此措施已经将响应的延迟降到尽可能低了，基本满足了实时系统的需求。

具体细节请看下图：

图 29. 互斥信号量解决优先级翻转



上图反映了互斥信号量的工作过程：

- 1) Task1 得要 CPU 使用权，并获得互斥信号量；
- 2) Task3 剥夺了 Task1 的 CPU 使用权（Task3 优先级最高），所以 Task3 开始运行；
- 3) Task3 在运行到某阶段想要获取互斥信号量，但发现此时该信号量被 Task1 持有，无法得到互斥信号量，所以 Task3 将 Task1 的优先级提高到和自己相同，即 Task3 让出 CPU 使用权后 Task1 马上开始运行；
- 4) 此时 Task2 从阻塞态变为就绪态，想要获得 CPU 使用权，但是此时原本比自己优先级低的 Task1 的优先级已经高于自己，无奈无法得到 CPU 使用权，只能等待，Task1 继续运行；
- 5) Task1 运行时释放了互斥信号量，释放时内核就将 Task1 的优先级还原到之前的等级（低于 Task2），Task3 获得互斥信号量开始运行；
- 6) Task3 运行结束，释放 CPU 使用权，Task2 开始运行；
- 7) 等到 Task2 运行完毕，系统中无更高优先级任务，所以 Task1 将运行剩余部分。

以上就是互斥信号量的工作过程。其尽可能避免了二值信号量的响应延时间，使系统的实时性更加出色。

注：互斥信号量不能用于中断处理程序内。

8.4.3 互斥信号量 API

下面来看看互斥信号量的 API 函数：

表 26. 互斥信号量 API

互斥信号量 API 函数	
API	描述
<code>xSemaphoreCreateMutex()</code>	创建一个互斥信号量

xSemaphoreCreateMutexStatic()	静态创建一个互斥信号量
vSemaphoreDelete()	删除一个信号量
xSemaphoreGive()	释放一个信号量
xSemaphoreTake()	获取一个信号量
xSemaphoreGetMutexHolder()	获取持有互斥信号量的任务句柄

以上就是互斥信号量相关的API函数，其实就是互斥信号量的创建、删除、释放、获取相关的API函数，使用起来比较简单。

注： 以上的释放、获取、删除相关的API在二值信号量、计数型信号量、递归信号量都是通用的。
来看看互斥信号量的常用API的原型。

xSemaphoreCreateMutex();

描述：

创建一个互斥信号量。

原型：

表 27. 创建互斥信号量函数原型

SemaphoreHandle_t xSemaphoreCreateMutex(void);	
参数	描述
无	无

返回值：

创建的互斥信号量的句柄。

vSemaphoreDelete ();

描述：

删除信号量。

原型：

表 28. 删除信号量函数原型

void vSemaphoreDelete(SemaphoreHandle_t xSemaphore);	
参数	描述
xSemaphore	要删除的信号量的句柄

返回值：

无。

xSemaphoreGive ();

描述：

释放信号量。

原型：

表 29. 释放信号量函数原型

BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore);	
参数	描述
xSemaphore	要释放的信号量的句柄

返回值:

释放成功与否标志。

xSemaphoreTake ();

描述:

获取信号量。

原型:

表 30. 获取信号量函数原型

BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);	
参数	描述
xSemaphore	要获取的信号量的句柄
xTicksToWait	超时等待时间

返回值:

获取成功与否标志。

uxSemaphoreGetCount ();

描述:

获取信号量值。

原型:

表 31. 获取信号量值函数原型

UBaseType_t uxSemaphoreGetCount(SemaphoreHandle_t xSemaphore);	
参数	描述
xSemaphore	要获取的信号量的句柄

返回值:

获取信号量的数目。

以上是互斥信号量常用的几个API原型介绍，下面来看看本小节配套的例程。

8.4.4 例程介绍

工程名: **09Mutex_Semaphore_FreeRTOS**

程序源码:

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"

/* 开始任务优先级*/
#define START_TASK_PRIO      1
/* 开始任务堆栈大小 */
#define START_STK_SIZE      128
/* 开始任务任务句柄 */
```

```

TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* 高优先级任务优先级 */
#define High_Priority_TASK_PRIO      5
/* 高优先级任务堆栈大小 */
#define High_Priority_STK_SIZE      256
/* 高优先级任务任务句柄 */
TaskHandle_t High_Priority_Task_Handler;
/* 高优先级任务入口函数 */
void High_Priority_task(void *pvParameters);

/* 中优先级任务优先级 */
#define Middle_Priority_TASK_PRIO      4
/* 中优先级任务堆栈大小 */
#define Middle_Priority_STK_SIZE      256
/* 中优先级任务任务句柄 */
TaskHandle_t Middle_Priority_Task_Handler;
/* 中优先级任务入口函数 */
void Middle_Priority_task(void *pvParameters);

/* 低优先级任务优先级 */
#define Low_Priority_TASK_PRIO      3
/* 低优先级任务堆栈大小 */
#define Low_Priority_STK_SIZE      256
/* 低优先级任务任务句柄 */
TaskHandle_t Low_Priority_Task_Handler;
/* 低优先级任务入口函数 */
void Low_Priority_task(void *pvParameters);

/* 调试任务优先级 */
#define Debug_TASK_PRIO      3
/* 调试任务堆栈大小 */
#define Debug_STK_SIZE      512
/* 调试任务任务句柄 */
TaskHandle_t DebugTask_Handler;
/* 调试任务入口函数 */
void debug_task(void *pvParameters);

/* 定义信号量 */
SemaphoreHandle_t AT_xSemaphore;

int main(void)
{

```

```

nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

system_clock_config();

at32_led_init(LED2);
at32_led_init(LED3);
at32_led_init(LED4);

at32_button_init();

/* init usart1 */
uart_print_init(115200);

/* 创建开始任务 */
xTaskCreate((TaskFunction_t)start_task,
            (const char*   )"start_task",
            (uint16_t      )START_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )START_TASK_PRIO,
            (TaskHandle_t* )&StartTask_Handler);

/* 开启任务调度器 */
vTaskStartScheduler();
}
/* 开始任务函数 */
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();

    /* 创建互斥信号量 */
    AT_xSemaphore = xSemaphoreCreateMutex();

    if( AT_xSemaphore == NULL )
    {
        /* 创建互斥信号量失败 */
        while(1);
    }

    /* 必须在创建互斥信号量之后再初始化定时器 */
    TIMER_Init();

    /* 创建高优先级任务 */
    xTaskCreate((TaskFunction_t)High_Priority_task,
                (const char*   )"High_task",
                (uint16_t      )High_Priority_STK_SIZE,

```

```

        (void*      )NULL,
        (UBaseType_t  )High_Priority_TASK_PRIO,
        (TaskHandle_t* )&High_Priority_Task_Handler);
/* 创建中优先级任务 */
xTaskCreate((TaskFunction_t )Middle_Priority_task,
            (const char*    )"Middle_task",
            (uint16_t       )Middle_Priority_STK_SIZE,
            (void*          )NULL,
            (UBaseType_t    )Middle_Priority_TASK_PRIO,
            (TaskHandle_t*  )&Middle_Priority_Task_Handler);
/* 创建低优先级任务 */
xTaskCreate((TaskFunction_t )Low_Priority_task,
            (const char*    )"Low_task",
            (uint16_t       )Low_Priority_STK_SIZE,
            (void*          )NULL,
            (UBaseType_t    )Low_Priority_TASK_PRIO,
            (TaskHandle_t*  )&Low_Priority_Task_Handler);
/* 创建调试任务 */
xTaskCreate((TaskFunction_t )debug_task,
            (const char*    )"Debug_task",
            (uint16_t       )Debug_STK_SIZE,
            (void*          )NULL,
            (UBaseType_t    )Debug_TASK_PRIO,
            (TaskHandle_t*  )&DebugTask_Handler);
/* 删除开始任务 */
vTaskDelete(StartTask_Handler);
/* 退出临界区 */
taskEXIT_CRITICAL();
}

/* 高优先级任务函数 */
void High_Priority_task(void *pvParameters)
{
    while(1)
    {
        /* 延时,不让高优先级任务获得互斥信号量 */
        vTaskDelay(10);

        if(xSemaphoreTake( AT_xSemaphore, portMAX_DELAY ) == pdTRUE)
        {
            printf("High priority task received a mutex semaphore.\r\n");
        }
    }
}

```

```

}

void Middle_Priority_task(void *pvParameters)
{
    while(1)
    {
        printf("Middle priority task is running.\r\n");
        printf("Change Middle priority task to Block state.\r\n");
        vTaskSuspend(Middle_Priority_Task_Handler);
        vTaskDelay(10);
    }
}

void Low_Priority_task(void *pvParameters)
{
    while(1)
    {
        /* 使最低优先级任务获得互斥信号量 */
        if(xSemaphoreTake( AT_xSemaphore, portMAX_DELAY ) == pdTRUE)
        {
            printf("Low priority task received a mutex semaphore.\r\n");
            printf("Change Middle priority task to Ready state.\r\n");
            /* 使中等优先级任务进入就绪态 */
            vTaskResume(Middle_Priority_Task_Handler);
            /* 使用定时器中断标志置起模拟普通延时 */
            while(tmr_flag_get(TMR3,TMR_OVF_FLAG)==RESET);
            tmr_flag_clear(TMR3,TMR_OVF_FLAG);

            printf("Low priority task release a mutex semaphore.\r\n");
            xSemaphoreGive( AT_xSemaphore );

        }
    }
}

/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次调试信息 */
        if(at32_button_press() == USER_BUTTON)
        {

```



```

printf("/-----*\r\n");
printf("Task          Status          priority      Remaining_Stack    Num\r\n");
vTaskList((char *)&buff);
printf("%s\r\n",buff);
printf("/-----*\r\n");
printf("Task          Runing_Num      Usage_Rate\r\n");
vTaskGetRunTimeStats((char *)&buff);
printf("%s\r\n",buff);
}
vTaskDelay(10);
}
}

```

以上程序模拟出了采用互斥信号量有效的避免了优先级翻转的问题。程序有3个任务会运行，其任务优先级有大小排列。程序会依次创建互斥信号量、3个任务，为了避免高优先级任务最开始就获得互斥信号量，便采用延时的方式使任务发生切换，即会运行中等优先级的任务，中等优先级的任务会在自己的任务函数里挂起自身进入阻塞态，然后就进入低优先级任务中运行了，低优先级的任务会获取互斥信号量，然后释放中等优先级的任务到就绪态，此时如果没有使用互斥信号量的话，中等优先级的任务会马上运行，但是由于使用了互斥信号量，所以中等优先级的任务不能得到运行，等到最低优先级的任务延时到并释放互斥信号量的时候，之前由于等待互斥信号量而阻塞的最高优先级任务会马上得到运行。高优先级任务会得到互斥信号量，然后中等优先级任务会运行一次，然后阻塞自己。之后由于低优先级任务无法获得互斥信号量，最高优先级也无法获得，所以这三个任务都不会再运行了。

编译并下载程序到目标板，运行效果如下：

图 30. 互斥信号量例程演示



从打印信息看来，完全符合程序的设计。此例程将互斥信号量的优点完全体现了出来，这在实时性要求较高的场合下是很重要的方法。

8.5 递归互斥信号量

8.5.1 递归互斥信号量介绍

递归互斥信号量可以看作是一个特殊的互斥信号量，已经获取了互斥信号量的任务就不能再次获取这个互斥信号量了，而递归互斥信号量则不同，在同一个任务中，此任务可以多次获得互斥递归信号量。需要注意的是，一旦一个任务获得了递归互斥信号量，那么其他任务便不能获得此互斥信号量，只能本身再次获取此递归互斥信号量。

设想一个简单的应用场景，如果一个任务获取了一个互斥信号量去访问一块共享资源，那么这块共享资源归这个任务所有；在这个任务的函数内再想去获取这块共享资源做其他处理就要再次获取互斥信号量，此时由于无法获取，导致任务死锁，再也无法得到运行。这种情况就需要递归互斥信号量了，它可以很好的解决这个问题。

注：递归互斥信号量不可用于中断处理函数内。

8.5.2 递归互斥信号量 API

下面来看看递归互斥信号量的API函数：

表 32. 递归互斥信号量 API

递归互斥信号量 API 函数	
API	描述
xSemaphoreCreateRecursiveMutex()	创建一个递归互斥信号量
xSemaphoreCreateRecursiveMutexStatic()	静态创建一个递归互斥信号量

vSemaphoreDelete()	删除一个信号量
xSemaphoreGiveRecursive()	释放一个递归互斥信号量
xSemaphoreTakeRecursive()	获取一个递归互斥信号量

以上就是递归互斥信号量相关的API函数，其实就是递归互斥信号量的创建、删除、释放、获取相关的API函数，使用起来比较简单。

来看看互斥信号量的常用API的原型。

xSemaphoreCreateRecursiveMutex();

描述:

创建一个互斥信号量。

原型:

表 33. 创建递归互斥信号量函数原型

SemaphoreHandle_t xSemaphoreCreateRecursiveMutex(void);	
参数	描述
无	无

返回值:

创建的递归互斥信号量的句柄。

xSemaphoreGiveRecursive();

描述:

释放递归互斥信号量。

原型:

表 34. 释放信号量函数原型

BaseType_t xSemaphoreGiveRecursive(SemaphoreHandle_t xMutex);	
参数	描述
xMutex	要释放的递归互斥信号量的句柄

返回值:

释放成功与否标志

xSemaphoreTakeRecursive();

描述:

获取递归互斥信号量。

原型:

表 35. 获取信号量函数原型

BaseType_t xSemaphoreTakeRecursive(SemaphoreHandle_t xMutex, TickType_t xTicksToWait);	
参数	描述
xMutex	要获取的递归互斥信号量的句柄
xTicksToWait	超时等待时间

返回值:

获取成功与否标志。

8.5.3 例程介绍

工程名: **10Recursive_Mutex_Semaphore_FreeRTOS**

程序源码:

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"

/* 开始任务优先级*/
#define START_TASK_PRIO      1
/* 开始任务堆栈大小 */
#define START_STK_SIZE      128
/* 开始任务任务句柄 */
TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* 普通任务优先级 */
#define Ordinary_TASK_PRIO      3
/* 普通任务堆栈大小 */
#define Ordinary_STK_SIZE      256
/* 普通任务任务句柄 */
TaskHandle_t Ordinary_Task_Handler;
/* 普通任务入口函数 */
void Ordinary_task(void *pvParameters);

/* 递归任务优先级 */
#define Recursive_TASK_PRIO      3
/* 递归任务堆栈大小 */
#define Recursive_STK_SIZE      256
/* 递归任务任务句柄 */
TaskHandle_t Recursive_Task_Handler;
/* 递归任务入口函数 */
void Recursive_task(void *pvParameters);

/* 调试任务优先级 */
#define Debug_TASK_PRIO      4
/* 调试任务堆栈大小 */
#define Debug_STK_SIZE      512
/* 调试任务任务句柄 */
TaskHandle_t DebugTask_Handler;
/* 调试任务入口函数 */
```

```

void debug_task(void *pvParameters);

/* 定义信号量 */
SemaphoreHandle_t AT_xSemaphore;

int main(void)
{
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

    system_clock_config();

    at32_led_init(LED2);
    at32_led_init(LED3);
    at32_led_init(LED4);

    at32_button_init();

    /* init usart1 */
    uart_print_init(115200);

    /* 创建开始任务 */
    xTaskCreate((TaskFunction_t)start_task,
                (const char*   )"start_task",
                (uint16_t      )START_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t    )START_TASK_PRIO,
                (TaskHandle_t*  *)&StartTask_Handler);

    /* 开启任务调度器 */
    vTaskStartScheduler();
}

/* 开始任务函数 */
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();

    /* 创建递归互斥信号量 */
    AT_xSemaphore = xSemaphoreCreateRecursiveMutex();

    if (AT_xSemaphore == NULL )
    {
        /* 创建递归互斥信号量失败 */
        while(1);
    }
}

```

```

/* 必须在创建递归互斥信号量之后再初始化定时器 */
TIMER_Init();

/* 创建递归任务 */
xTaskCreate((TaskFunction_t)Recursive_task,
            (const char*   )"Recursive_task",
            (uint16_t      )Recursive_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )Recursive_TASK_PRIO,
            (TaskHandle_t*  )&Recursive_Task_Handler);

/* 创建普通任务 */
xTaskCreate((TaskFunction_t)Ordinary_task,
            (const char*   )"Ordinary_task",
            (uint16_t      )Ordinary_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )Ordinary_TASK_PRIO,
            (TaskHandle_t*  )&Ordinary_Task_Handler);

/* 创建调试任务 */
xTaskCreate((TaskFunction_t)debug_task,
            (const char*   )"Debug_task",
            (uint16_t      )Debug_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )Debug_TASK_PRIO,
            (TaskHandle_t*  )&DebugTask_Handler);

/* 删除开始任务 */
vTaskDelete(StartTask_Handler);

/* 退出临界区 */
taskEXIT_CRITICAL();
}

/* 普通任务函数 */
void Ordinary_task(void *pvParameters)
{
    while(1)
    {
        vTaskDelay(5);

        if(xSemaphoreTakeRecursive( AT_xSemaphore, 5 ) == pdTRUE)
        {
            printf("Ordinary task received the recursive mutex semaphore.\r\n");
        }
        else
        {
            printf("Ordinary task didn't get the recursive mutex semaphore.\r\n");
        }
    }
}

```

```

    }

}

}
/* 递归任务函数 */
void Recursive_task(void *pvParameters)
{
    while(1)
    {
        if( AT_xSemaphore != NULL )
        {
            if( xSemaphoreTakeRecursive( AT_xSemaphore, portMAX_DELAY ) == pdTRUE )
            {
                printf("Get the recursive mutex semaphore for the first time.\r\n");
                if(xSemaphoreTakeRecursive( AT_xSemaphore, portMAX_DELAY ) == pdTRUE )
                {
                    printf("Get the recursive mutex semaphore for the second time.\r\n");
                }
                if(xSemaphoreTakeRecursive( AT_xSemaphore, portMAX_DELAY ) == pdTRUE )
                {
                    printf("Get the recursive mutex semaphore for the third time.\r\n");
                }
            }
            vTaskDelay(5);
            xSemaphoreGiveRecursive( AT_xSemaphore );
            printf("Release the recursive mutex semaphore for the first time.\r\n");
            xSemaphoreGiveRecursive( AT_xSemaphore );
            printf("Release the recursive mutex semaphore for the second time.\r\n");
            xSemaphoreGiveRecursive( AT_xSemaphore );
            printf("Release the recursive mutex semaphore for the third time.\r\n");
        }
        vTaskDelay(100);
    }
}
/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次调试信息 */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/*-----*/\r\n");
            printf("Task          Status          priority    Remaining_Stack    Num\r\n");

```

```

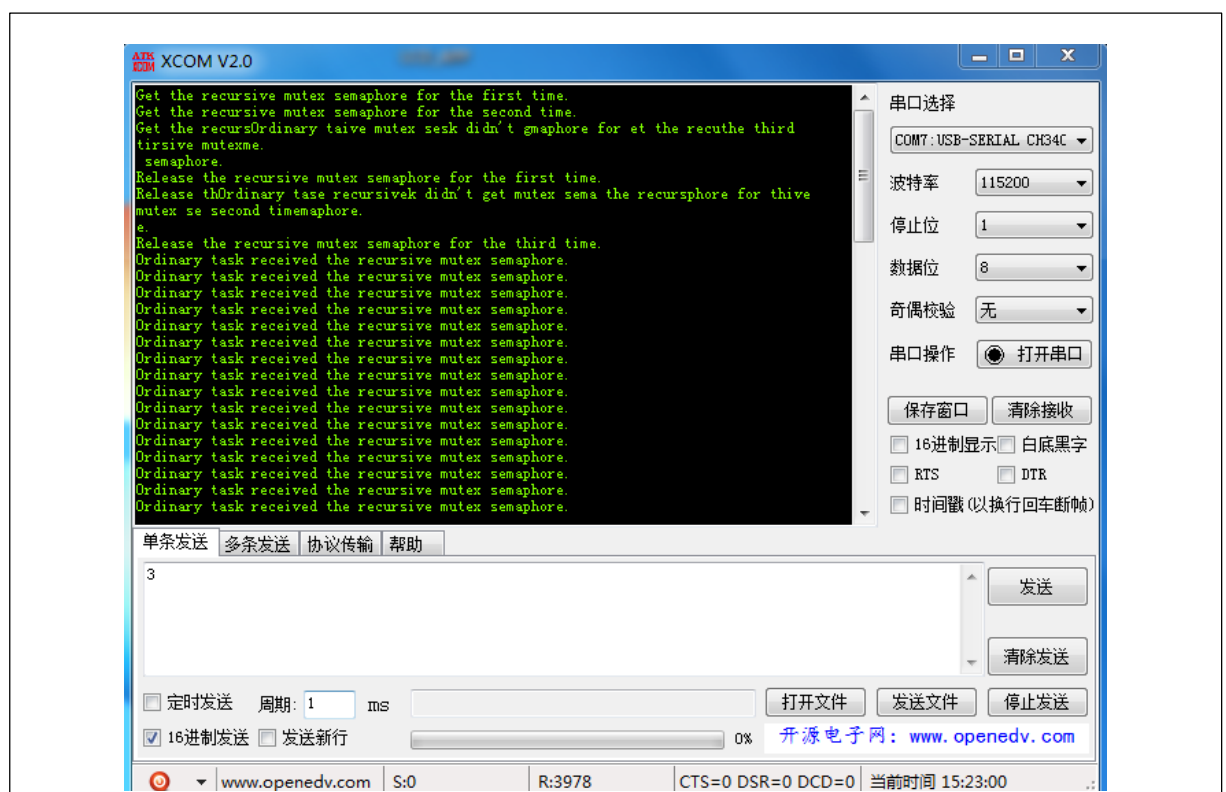
vTaskList((char *)&buff);
printf("%s\r\n",buff);
printf("/-----*\r\n");
printf("Task      Runing_Num      Usage_Rate\r\n");
vTaskGetRunTimeStats((char *)&buff);
printf("%s\r\n",buff);
}
vTaskDelay(10);
}
}

```

以上就是递归互斥信号量配套例程的源程序，程序会建立一个递归互斥信号量，然后建立两个任务，Recursive_task内部会递归的获取三次递归互斥信号量，在没有释放3次递归互斥信号量前，Ordinary_task是无法获取到递归互斥信号量的，只有当Recursive_task全部释放之前获得的3次递归互斥信号量后Ordinary_task才会获得。

编译并下载程序到目标板，运行效果如下：

图 31. 递归互斥信号量例程演示



从打印结果可以看到，首先Recursive_task会连续获取三次递归互斥信号量，再次期间Ordinary_task也想获得，但是由于已经被Recursive_task所占有，所以获取不会成功，等到Recursive_task再连续释放三次递归互斥信号量后，Ordinary_task便获得了递归互斥信号量，此现象正好符合程序设计思路和递归互斥信号量的特性。

注：打印中出现的错乱现象，属于Recursive_task和Ordinary_task任务调度所引起了，属于正常现象。

9 FreeRTOS 事件标志组

本节将介绍FreeRTOS事件标志组相关的内容，事件标志组和消息队列、信号量一样，都是FreeRTOS内核提供的一种内核服务。事件标志组在任务同步中应用相当广泛。

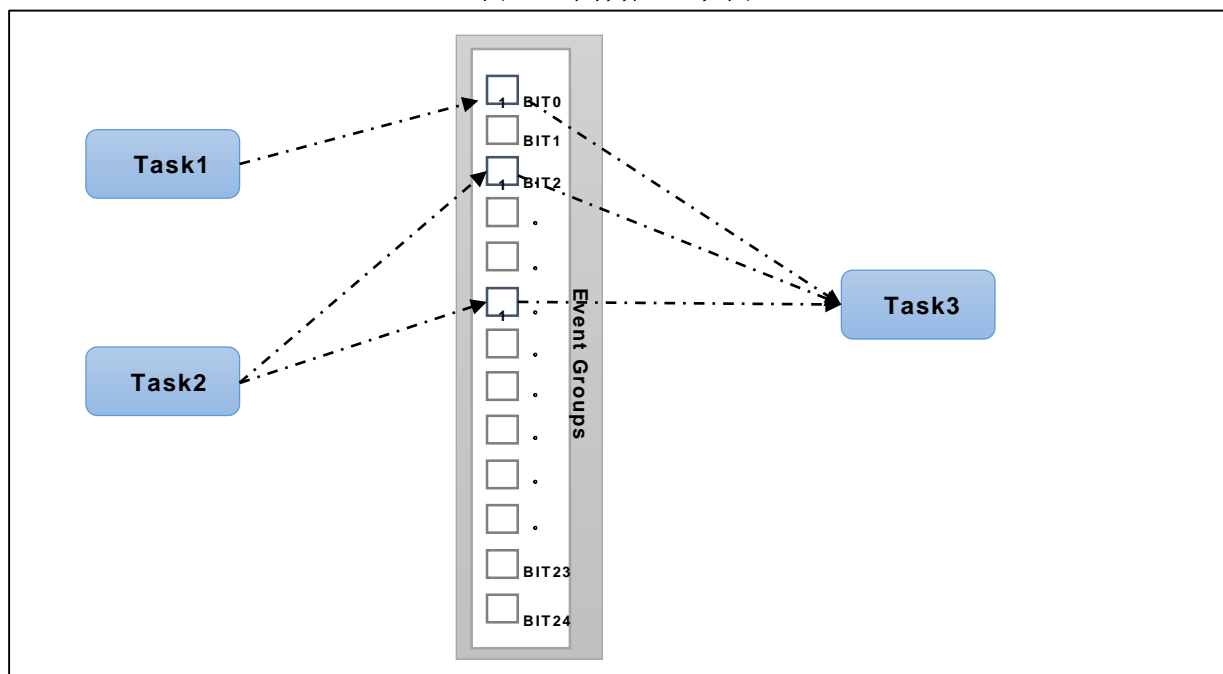
9.1 事件标志组介绍

事件标志组是FreeRTOS内核提供的一种服务，是实现多任务同步的有效机制之一。事件标志组的本质是内核管理一个变量，任务通过设置这个变量的不同BIT位，达到同步的效果。比如某个任务需要等到这个变量的BIT0、BIT1都被置1时才能执行后续的任务代码，那么就需要有任务将这两个BIT位设置为1，只要这两个BIT被设置，那么等待着两个BIT的任务就会马上得到执行，这样就实现了任务间的同步。

也许有不理解的初学者会觉得采用事件标志组多麻烦，用全局变量不是更简单？其实不然，在裸机编程时，使用全局变量的确比较方便，但是在加上RTOS后就是另一种情况了。使用全局变量相比事件标志组主要有如下三个问题：

- 1) 使用事件标志组可以让RTOS内核有效地管理任务，而全局变量是无法做到的，任务的超时等机制需要用户自己去实现。
- 2) 使用了全局变量就要防止多任务的访问冲突，而使用事件标志组则处理好了这个问题，用户无需担心。
- 3) 使用事件标志组可以有效地解决中断服务程序和任务之间的同步问题。

图 32. 事件标志组框图



上图中体现的事件标志组的一个工作流程，Task1和Task2会去设置时间标志组中的对应BIT位，当满足Task3的需求后，Task3会得到此事件标志组并清除事件标志组对应的BIT位。

9.2 事件标志组 API

下面来看看事件标志组相关的API函数：

表 36. 事件标志组 API

事件标志组 API 函数	
API	描述
xEventGroupClearBits()	清除事件标志组中的 BIT 位
xEventGroupClearBitsFromISR()	清除事件标志组中的 BIT 位（中断级）
xEventGroupCreate()	创建一个事件标志组
xEventGroupCreateStatic()	静态创建一个事件标志组
vEventGroupDelete()	删除一个事件标志组
xEventGroupGetBits()	获取当前时间标志组的值
xEventGroupGetBitsFromISR()	获取当前时间标志组的值（中断级）
xEventGroupSetBits()	设置事件标志组的 BIT 位
xEventGroupSetBitsFromISR()	设置事件标志组的 BIT 位（中断级）
xEventGroupSync()	设置事件标志组的 BIT 位并等待其他 BIT 位被设置（同步）
xEventGroupWaitBits()	等待事件标志组对应 BIT 位被设置

以上就是事件标志组对应的API函数，不算很多，使用起来也比较容易。主要来看几个常用API的原型：

xEventGroupCreate();

描述：

创建一个事件标志组。

原型：

表 37. 创建事件标志组函数原型

EventGroupHandle_t xEventGroupCreate(void);	
参数	描述
无	无

返回值：

创建的事件标志组的句柄。

xEventGroupSetBits();

描述：

设置一个事件标志组的对应BIT位。

原型：

表 38. 设置事件标志组函数原型

EventBits_t xEventGroupSetBits(EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet);	
参数	描述
xEventGroup	事件标志组句柄
uxBitsToSet	需要设置的 BIT 位

返回值:

事件标志组的值。

xEventGroupWaitBits();

描述:

等待事件标志组的对应BIT位。

原型:

表 39. 等待事件标志组函数原型

EventBits_t xEventGroupWaitBits(const EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToWaitFor, const BaseType_t xClearOnExit, const BaseType_t xWaitForAllBits, TickType_t xTicksToWait);	
参数	描述
xEventGroup	事件标志组句柄
uxBitsToWaitFor	需要等待的 BIT 位
xClearOnExit	是否在返回时清除所有等待的 BIT 位
xWaitForAllBits	等待的 BIT 位是与关系还是或关系
xWaitForAllBits	等待超时时间

返回值:

事件标志组的值。

9.3 例程介绍

工程名: 11Event_Groups_FreeRTOS

程序源码:

```
#include "FreeRTOS.h"
#include "event_groups.h"

/* 开始任务优先级*/
#define START_TASK_PRIO      1
/* 开始任务堆栈大小 */
#define START_STK_SIZE      128
/* 开始任务任务句柄 */
TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* 事件标志组任务 1 优先级 */
#define EventGroup1_TASK_PRIO      3
```

```

/* 事件标志组任务 1 堆栈大小 */
#define EventGroup1_STK_SIZE      256
/* 事件标志组任务 1 任务句柄 */
TaskHandle_t EventGroup1Task_Handler;
/* 事件标志组任务 1 入口函数 */
void EventGroup1_task(void *pvParameters);

/* 事件标志组任务 2 优先级 */
#define EventGroup2_TASK_PRIO      3
/* 事件标志组任务 2 堆栈大小 */
#define EventGroup2_STK_SIZE      256
/* 事件标志组任务 2 任务句柄 */
TaskHandle_t EventGroup2Task_Handler;
/* 事件标志组任务 2 入口函数 */
void EventGroup2_task(void *pvParameters);

/* 事件标志组接收任务优先级 */
#define EventGroup_Receive_TASK_PRIO      4
/* 事件标志组接收任务堆栈大小 */
#define EventGroup_Receive_STK_SIZE      256
/* 事件标志组接收任务句柄 */
TaskHandle_t EventGroup_ReceiveTask_Handler;
/* 事件标志组接收任务入口函数 */
void EventGroup_Receive_task(void *pvParameters);

/* 调试任务优先级 */
#define Debug_TASK_PRIO      5
/* 调试任务堆栈大小 */
#define Debug_STK_SIZE      512
/* 调试任务任务句柄 */
TaskHandle_t DebugTask_Handler;
/* 调试任务入口函数 */
void debug_task(void *pvParameters);

/* 任务所发送的 BIT */
#define TASK_0_BIT ( 1 << 0 )
#define TASK_1_BIT ( 1 << 1 )
#define TASK_2_BIT ( 1 << 2 )

/* 任务同步标志 */
#define ALL_SYNC_BITS ( TASK_0_BIT | TASK_1_BIT | TASK_2_BIT )

/* 定义一个事件标志组 */
EventGroupHandle_t AT_xEventBits;

```

```
int main(void)
{
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

    system_clock_config();

    at32_led_init(LED2);
    at32_led_init(LED3);
    at32_led_init(LED4);

    at32_button_init();

    /* init usart1 */
    uart_print_init(115200);

    /* 创建开始任务 */
    xTaskCreate((TaskFunction_t)start_task,
                (const char*   )"start_task",
                (uint16_t      )START_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t   )START_TASK_PRIO,
                (TaskHandle_t*  *)&StartTask_Handler);

    /* 开启任务调度器 */
    vTaskStartScheduler();
}

/* 开始任务函数 */
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();

    /* 创建一个事件标志组 */
    AT_xEventBits = xEventGroupCreate();

    if(AT_xEventBits == NULL)
    {
        /* 创建失败 */
        while(1);
    }

    /* 必须在创建消息队列之后再初始化定时器 */
    TIMER_Init();

    /* 创建事件标志组任务 1 */
    xTaskCreate((TaskFunction_t)EventGroup1_task,
                (const char*   )"EventGroup1_task",
```

```

        (uint16_t      )EventGroup1_STK_SIZE,
        (void*         )NULL,
        (UBaseType_t   )EventGroup1_TASK_PRIO,
        (TaskHandle_t* )&EventGroup1Task_Handler);

/* 创建事件标志组任务 2 */
xTaskCreate((TaskFunction_t)EventGroup2_task,
            (const char*    )"EventGroup2_task",
            (uint16_t      )EventGroup2_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )EventGroup2_TASK_PRIO,
            (TaskHandle_t* )&EventGroup2Task_Handler);

/* 创建事件标志组接收任务 */
xTaskCreate((TaskFunction_t)EventGroup_Receive_task,
            (const char*    )"EventReceive_task",
            (uint16_t      )EventGroup_Receive_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )EventGroup_Receive_TASK_PRIO,
            (TaskHandle_t* )&EventGroup_ReceiveTask_Handler);

/* 创建调试任务 */
xTaskCreate((TaskFunction_t)debug_task,
            (const char*    )"Debug_task",
            (uint16_t      )Debug_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )Debug_TASK_PRIO,
            (TaskHandle_t* )&DebugTask_Handler);

/* 删除开始任务 */
vTaskDelete(StartTask_Handler);

/* 退出临界区 */
taskEXIT_CRITICAL();
}

/* 事件标志组任务 1 任务函数 */
void EventGroup1_task(void *pvParameters)
{
    EventBits_t uxBits;

    while(1)
    {
        /* 设置时间标志组的 BIT0&BIT2 */
        uxBits = xEventGroupSetBits( AT_xEventBits, TASK_0_BIT | TASK_2_BIT );
        /* LED2 ON */
        at32_led_on(LED2);
        vTaskDelay(1000);
        if( ( uxBits & ( TASK_0_BIT | TASK_2_BIT ) ) == ( TASK_0_BIT | TASK_2_BIT ) )
        {
            printf("Both bit 0 and bit 2 remained set when the function returned.\r\n");
        }
    }
}

```

```

    }
    else if( ( uxBits & TASK_0_BIT ) != 0 )
    {
        printf("Bit 0 remained set when the function returned, but bit 4 was cleared.\r\n");
    }
    else if( ( uxBits & TASK_2_BIT ) != 0 )
    {
        printf("Bit 4 remained set when the function returned, but bit 0 was cleared.\r\n");
    }
    else
    {
        printf("Neither bit 0 nor bit 4 remained set.\r\n");
    }
}
}
/* 事件标志组任务 2 任务函数 */
void EventGroup2_task(void *pvParameters)
{
    EventBits_t uxBits;

    while(1)
    {
        vTaskDelay(1500);
        /* 设置时间标志组的 BIT1 */
        uxBits = xEventGroupSetBits( AT_xEventBits, TASK_1_BIT );
        /* LED3 ON */
        at32_led_on(LED3);

        if( ( uxBits & TASK_1_BIT ) == TASK_1_BIT )
        {
            printf("Bit 1 remained set when the function returned.\r\n");
        }
        else
        {
            printf(" Bit 1 was cleared when the function returned.\r\n");
        }
    }
}

/* 事件标志组接收任务函数 */
void EventGroup_Receive_task(void *pvParameters)
{
    EventBits_t uxBits;

```

```

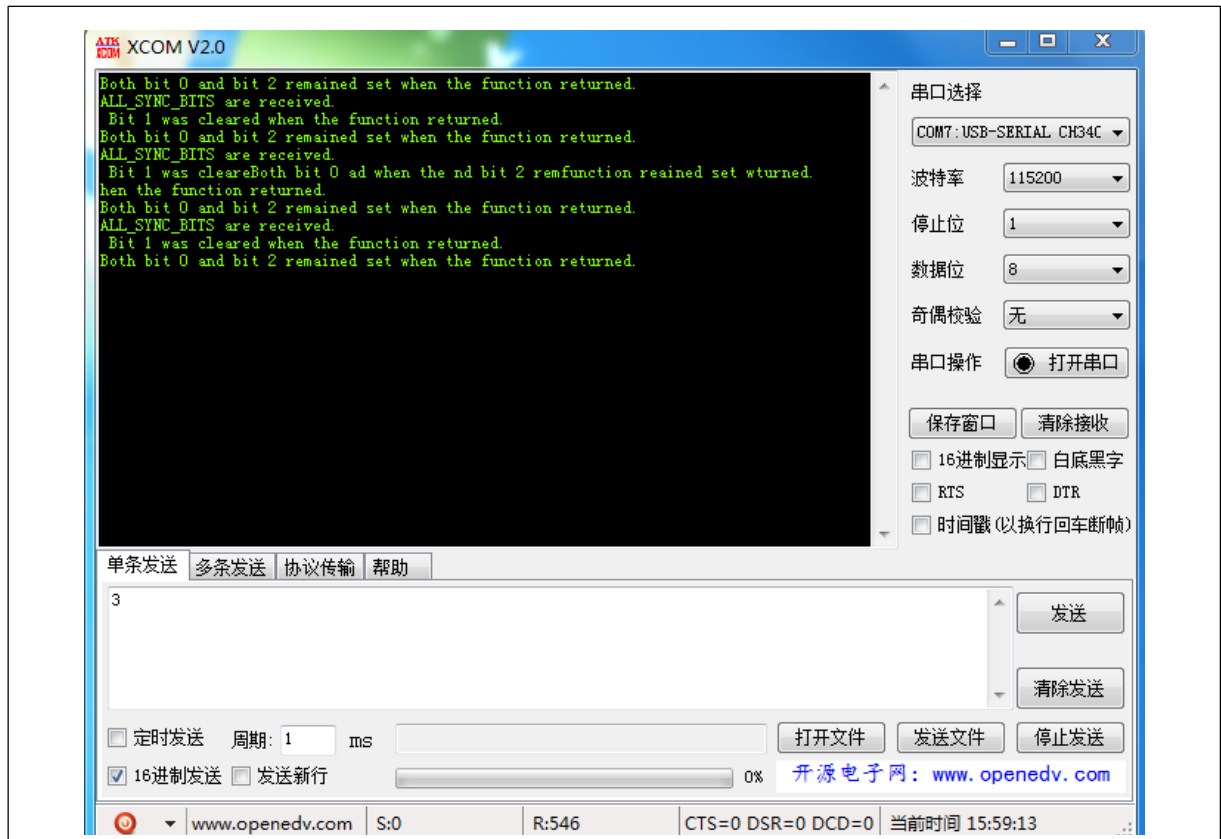
while(1)
{
    /* 接收 ALL_SYNC_BITS,同步任务 */
    uxBits = xEventGroupWaitBits( AT_xEventBits, ALL_SYNC_BITS, pdTRUE, pdTRUE, portMAX_DELAY );
    /* LED4 ON */
    at32_led_on(LED4);
    printf("ALL_SYNC_BITS are received.\r\n");
}
}
/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次调试信息 */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/*-----*/\r\n");
            printf("Task          Status          priority      Remaining_Stack      Num\r\n");
            vTaskList((char *)&buff);
            printf("%s\r\n",buff);
            printf("/*-----*/\r\n");
            printf("Task          Runing_Num      Usage_Rate\r\n");
            vTaskGetRunTimeStats((char *)&buff);
            printf("%s\r\n",buff);
        }
        vTaskDelay(10);
    }
}

```

以上就是事件标志组例程的源程序，程序中定义了一个事件标志组，EventGroup1_task和EventGroup2_task会通过调用xEventGroupSetBits这个API来设置不同的BIT标志位，另外一个任务EventReceive_task会等待它们设置的BIT位，当所有BIT位都满足条件后，EventReceive_task任务就会从阻塞态转到就绪态并得到运行，此例程实现了多个任务间的同步效果。另外此例程也保留了按下USER按键打印出系统任务信息的功能。

编译并下载程序到目标板，运行效果如下：

图 33. 事件标志组例程演示



从上图打印信息可以看到EventGroup1_task任务首先设置BIT0和BIT2，但是马上获取事件标志组的任务为什么就得到运行了呢？明明EventGroup2_task都还没有设置BIT1。其实并不是所看到的这样，EventGroup2_task已经运行了并设置了BIT2，只是在设置了BIT2后内核马上切换到了获取事件标志组的EventReceive_task任务了，等到EventReceive_task运行之后EventGroup2_task才会得到运行。这也就是为什么切换回来打印的提示信息是“返回的值BIT1为零”，因为是EventReceive_task将其清零的。

10 FreeRTOS 软件定时器组

本节将介绍FreeRTOS的软件定时器组相关的内容，软件定时器是内核提供的一种服务，因为可以创建多个软件定时器，所以称其为软件定时器组。

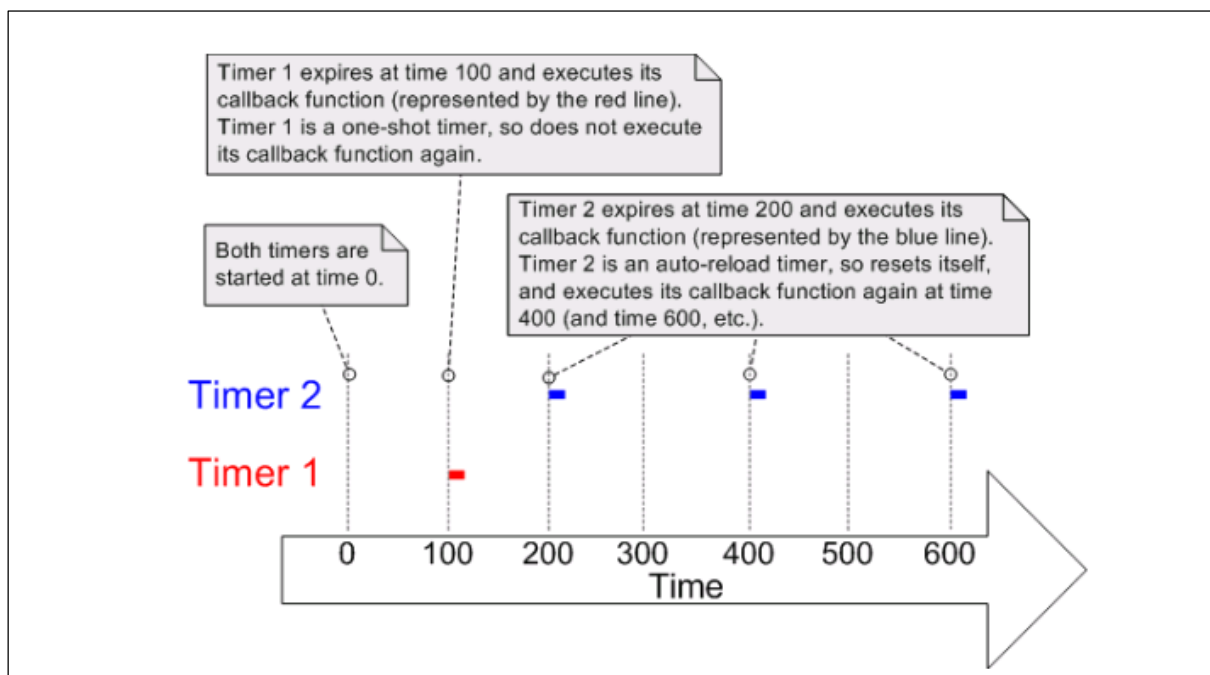
10.1 软件定时器组介绍

FreeRTOS软件定时器组的时基是基于系统时钟节拍实现的，之所以叫软件定时器是因为它的实现不需要使用任何硬件定时器，而且可以创建很多个，综合这些因素，这个功能就被称之为软件定时器组。

既然是定时器，那么它实现的功能与硬件定时器也是类似的。在硬件定时器中，我们是在定时器中断中实现需要的功能，而使用软件定时器时，我们是在创建软件定时器时指定软件定时器的回调函数，在回调函数中实现相应的功能。

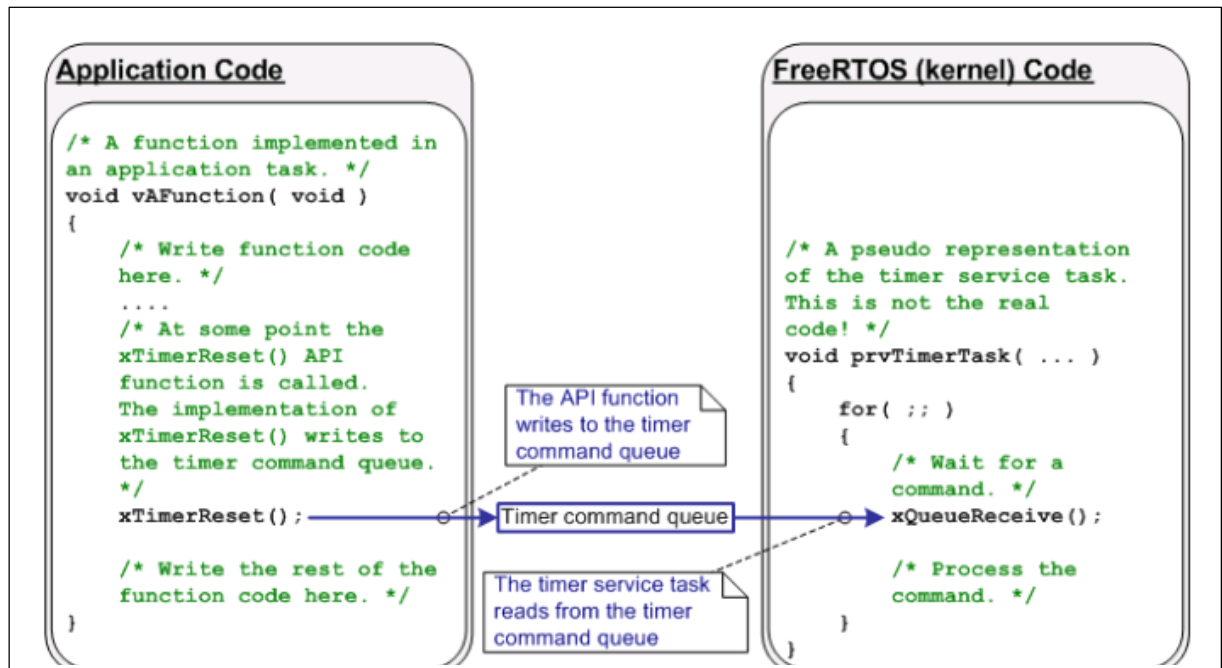
FreeRTOS提供的软件定时器支持单次模式和周期性模式，单次模式就是用户创建了定时器并启动了定时器后，定时时间到将不再重新执行，这就是单次模式软件定时器的含义。周期模式就是此定时器会按照设置的时间周期重复去执行，这就是周期模式软件定时器的含义。另外就是单次模式或者周期模式的定时时间到后会调用定时器的回调函数，用户可以回调函数中加入需要执行的工程代码。下图是FreeRTOS官网里单次模式和周期性模式的对比图：

图 34. 单次模式 VS 周期性模式



FreeRTOS为软件定时器专门创建了一个任务，可以称其为软件定时器的守护进程或后台进程（Daemon Task）。这个任务在系统使能了软件定时器组的功能后系统自动创建。FreeRTOS官网有关于此任务的简图，如下：

图 35. 软件定时器 Daemon Task



上图左边为用户应用程序，右边为内核管理软件定时器的简化程序；可看到他们之间是通过消息队列实现通信功能的，当应用程序调用相关的软件定时器的API函数后，就会向软件定时器的消息队列里发送消息，然后管理软件定时器的任务就会从消息队列中获取到消息做出相应的动作。

要使用FreeRTOS提供的软件定时器，必须先设定以下宏定义：

```
#define configUSE_TIMERS 1
```

此宏定义为使能软件定时器组，要使用软件定时器，此宏必须定义为1；

```
#define configTIMER_TASK_PRIORITY 10
```

此宏定义为内核为软件定时器组创建的任务的优先级，优先级越高响应回调函数的实时性越好，定时器也会更精准；此宏定义用户根据自己的实际需求设置即可；

```
#define configTIMER_QUEUE_LENGTH 15
```

此宏定义为软件定时器组的消息队列的长度，用户根据系统中创建的软件定时器个数适当定义即可；

```
#define configTIMER_TASK_STACK_DEPTH 128
```

此宏定义为软件定时器管理任务的堆栈大小，用户根据实际情况，合理分配即可。

通过以上宏定义的设置，FreeRTOS系统便可提供软件定时器组服务供用户使用了。

10.2 软件定时器组 API

下面来看一下与软件定时器组相关的API函数：

表 40. 软件定时器组 API

软件定时器组 API 函数	
API	描述
xTimerChangePeriod()	改变定时器周期
xTimerChangePeriodFromISR()	改变定时器周期（中断级）
xTimerCreate()	创建一个定时器
xTimerCreateStatic()	静态创建一个定时器
xTimerDelete()	删除一个定时器
xTimerGetExpiryTime()	获取定时器的溢出时间
pcTimerGetName()	获取定时器的名字
xTimerGetPeriod()	获取定时器的周期
xTimerGetTimerDaemonTaskHandle()	获取定时器管理任务的句柄
pvTimerGetTimerID()	获取定时器的 ID
xTimerIsTimerActive()	获取定时器的运行情况
xTimerPendFunctionCall()	定时器发送消息队列是否成功
xTimerPendFunctionCallFromISR()	定时器发送消息队列是否成功（中断级）
xTimerReset()	复位一个定时器
xTimerResetFromISR()	复位一个定时器（中断级）
vTimerSetTimerID()	设置定时器 ID
xTimerStart()	启动定时器
xTimerStartFromISR()	启动定时器（中断级）
xTimerStop()	停止定时器
xTimerStopFromISR()	停止定时器（中断级）

以上是软件定时器组相关的API函数，下面来看看常用的几个API函数原型：

xTimerCreate();

描述：

创建一个定时器。

原型：

表 41. 创建软件定时器函数原型

TimerHandle_t xTimerCreate(const char *pcTimerName, const TickType_t xTimerPeriod, const UBaseType_t uxAutoReload, void * const pvTimerID, TimerCallbackFunction_t pxCallbackFunction);	
参数	描述
pcTimerName	定时器名字
xTimerPeriod	定时器周期

uxAutoReload	定时器模式
pvTimerID	定时器 ID
pxCallbackFunction	定时器回调函数

返回值:

创建的软件定时器句柄。

xTimerStart();

描述:

启动定时器。

原型:

表 42. 启动定时器函数原型

BaseType_t xTimerStart(TimerHandle_t xTimer, TickType_t xTicksToWait);	
参数	描述
xTimer	待启动的定时器句柄
xTicksToWait	等待超时时间

返回值:

启动是否成功标志。

xTimerStop();

描述:

关闭定时器。

原型:

表 43. 关闭定时器函数原型

BaseType_t xTimerStop(TimerHandle_t xTimer, TickType_t xTicksToWait);	
参数	描述
xTimer	待关闭的定时器句柄
xTicksToWait	等待超时时间

返回值:

关闭是否成功标志。

pcTimerGetName();

描述:

获取定时器名字。

原型:

表 44. 获取定时器名字函数原型

const char * pcTimerGetName(TimerHandle_t xTimer);	
参数	描述
xTimer	定时器句柄

返回值:

定时器名字指针。

pvTimerGetTimerID();

描述:

获取定时器ID。

原型:

表 45. 获取定时器 ID 函数原型

void *pvTimerGetTimerID(TimerHandle_t xTimer);	
参数	描述
xTimer	定时器句柄

返回值:

定时器ID。

vTimerSetTimerID();

描述:

设置定时器ID。

原型:

表 46. 设置定时器 ID 函数原型

void vTimerSetTimerID(TimerHandle_t xTimer, void *pvNewID);	
参数	描述
xTimer	定时器句柄
pvNewID	定时器 ID

返回值:

无。

以上为软件定时器常用API函数原型，其他API可自行查阅FreeRTOS官方文档学习即可。

10.3 例程介绍

工程名: 12Software_Timer_FreeRTOS

程序源码:

```
#include "FreeRTOS.h"
```

```
#include "task.h"

/* 开始任务优先级*/
#define START_TASK_PRIO      1
/* 开始任务堆栈大小 */
#define START_STK_SIZE      128
/* 开始任务任务句柄 */
TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* 软件定时器创建任务优先级 */
#define SoftwareTimer_Create_TASK_PRIO      4
/* 软件定时器创建任务堆栈大小 */
#define SoftwareTimer_Create_STK_SIZE      256
/* 软件定时器创建任务句柄 */
TaskHandle_t SoftwareTimer_CreateTask_Handler;
/* 软件定时器创建任务入口函数 */
void SoftwareTimer_Create_task(void *pvParameters);

/* 调试任务优先级 */
#define Debug_TASK_PRIO      5
/* 调试任务堆栈大小 */
#define Debug_STK_SIZE      512
/* 调试任务任务句柄 */
TaskHandle_t DebugTask_Handler;
/* 调试任务入口函数 */
void debug_task(void *pvParameters);

/* 需创建的软件定时器个数 */
#define NUM_TIMERS 5
/* 定义软件定时器数组 */
TimerHandle_t AT_xTimers[ NUM_TIMERS ];
/* 软件定时器名字 */
char *Timer_Name[] = {"Timer1", "Timer2", "Timer3", "Timer4", "Timer5"};
/* 定时器回调函数 */
void AT_vTimerCallback( TimerHandle_t xTimer );

int main(void)
{
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

    system_clock_config();

    at32_led_init(LED2);
```

```

at32_led_init(LED3);
at32_led_init(LED4);

at32_button_init();

/* init usart1 */
uart_print_init(115200);

/* 创建开始任务 */
xTaskCreate((TaskFunction_t)start_task,
            (const char*   )"start_task",
            (uint16_t      )START_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )START_TASK_PRIO,
            (TaskHandle_t* )&StartTask_Handler);

/* 开启任务调度器 */
vTaskStartScheduler();
}

/* 开始任务函数 */
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();

    /* 必须在创建消息队列之后再初始化定时器 */
    TIMER_Init();

    /* 创建软件定时器创建任务 */
    xTaskCreate((TaskFunction_t)SoftwareTimer_Create_task,
                (const char*   )"SoftwareTimer_task",
                (uint16_t      )SoftwareTimer_Create_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t   )SoftwareTimer_Create_TASK_PRIO,
                (TaskHandle_t* )&SoftwareTimer_CreateTask_Handler);

    /* 创建调试任务 */
    xTaskCreate((TaskFunction_t)debug_task,
                (const char*   )"Debug_task",
                (uint16_t      )Debug_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t   )Debug_TASK_PRIO,
                (TaskHandle_t* )&DebugTask_Handler);

    /* 删除开始任务 */
    vTaskDelete(StartTask_Handler);

    /* 退出临界区 */
    taskEXIT_CRITICAL();
}

```



```

}

/* 软件定时器创建任务函数 */
void SoftwareTimer_Create_task(void *pvParameters)
{
    int i;
    /* 进入临界区 */
    taskENTER_CRITICAL();
    for( i=0; i<NUM_TIMERS; i++ )
    {
        /* 创建 5 个软件定时器,循环模式 */
        AT_xTimers[ i ] = xTimerCreate( Timer_Name[i],
                                         ( 1000 * i ) + 1000,
                                         pdTRUE,
                                         ( void * ) 0,
                                         AT_vTimerCallback );

        if( AT_xTimers[ i ] == NULL )
        {
            /* 创建失败 */
            while(1);
        }
        else
        {
            if( xTimerStart( AT_xTimers[ i ], portMAX_DELAY ) != pdPASS )
            {
                /* 启动定时器失败 */
                while(1);
            }
        }
    }

    /* 删除创建软件定时器任务 */
    vTaskDelete(SoftwareTimer_CreateTask_Handler);
    /* 退出临界区 */
    taskEXIT_CRITICAL();
}

/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次调试信息 */
        if(at32_button_press() == USER_BUTTON)
        {

```

```

        printf("/-----*\r\n");
        printf("Task          Status          priority      Remaining_Stack      Num\r\n");
        vTaskList((char *)&buff);
        printf("%s\r\n",buff);
        printf("/-----*\r\n");
        printf("Task          Runing_Num      Usage_Rate\r\n");
        vTaskGetRunTimeStats((char *)&buff);
        printf("%s\r\n",buff);
    }
    vTaskDelay(10);
}
}

/* 软件定时器回调函数 */
void AT_vTimerCallback( TimerHandle_t xTimer )
{
    const uint32_t ulMaxExpiryCountBeforeStopping = 10;
    uint32_t ulCount;
    /* 获取软件定时器 ID */
    ulCount = ( uint32_t ) pvTimerGetTimerID( xTimer );

    ulCount++;

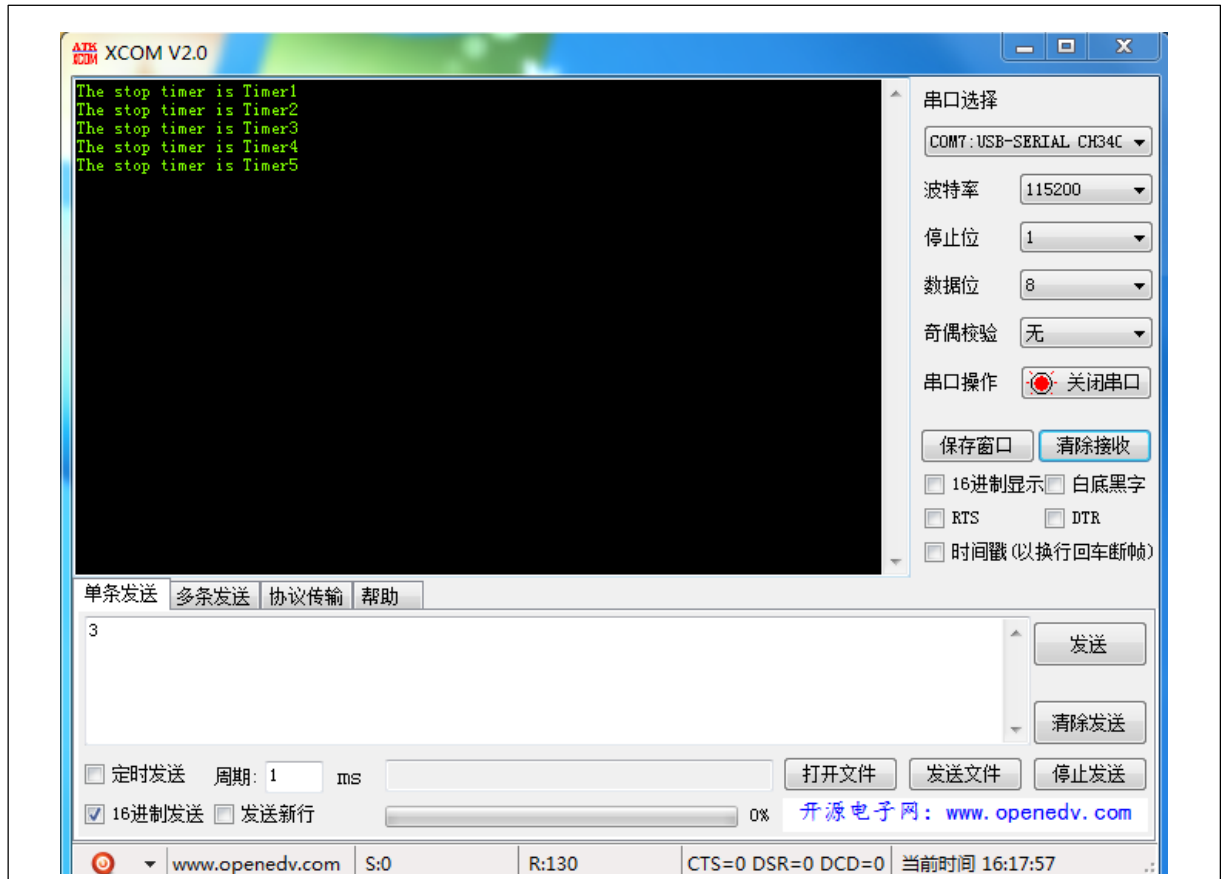
    if( ulCount >= ulMaxExpiryCountBeforeStopping )
    {
        /* 关闭软件定时器 */
        xTimerStop( xTimer, 0 );
        /* 获取软件定时器名字并打印 */
        printf("The stop timer is %s\r\n",pcTimerGetName(xTimer));
        at32_led_toggle(LED4);
    }
    else
    {
        /* 设置软件定时器 ID */
        vTimerSetTimerID( xTimer, ( void * ) ulCount );
        at32_led_toggle(LED2);
    }
}
}

```

以上是软件定时器相关的例程源程序，程序中在**SoftwareTimer_task**中创建了5个定时器，周期分别是1s、2s、3s、4s、5s。在回调函数中判断每个软件定时器是否溢出了10次，如果溢出10此则关闭定时器并打印出哪个定时器被关闭。所以每个10s就会有一个定时器被关闭。

编译并下载程序到目标板，运行效果如下：

图 36. 软件定时器例程演示



从上图打印信息看来，符合程序的设定。定时器1到5会依次关闭。

11 FreeRTOS 低功耗模式

本节来介绍FreeRTOS的低功耗模式使用，FreeRTOS提供了一种Tickless机制来管理低功耗，是当前小型RTOS所采用的通用低功耗方法，比如embOS，RTX和uCOS-III（类似方法）都有这种机制。

11.1 Tickless 机制介绍

FreeRTOS的低功耗采用Tickless这种方式，那么tickless又是怎样一种模式呢？仅从字母上看tick是滴答时钟的意思，less是tick的后缀，表示较少的，这里的含义可以表示为无滴答时钟。整体看这个字母就是表示滴答时钟节拍停止运行的情况。

反映在FreeRTOS上，tickless又是怎样一种情况呢？当用户任务都被挂起或者阻塞时，最低优先级的空闲任务会得到执行。那么AT32支持的睡眠模式就可以放在空闲任务里面实现。为了实现低功耗最优设计，我们把睡眠直接放在空闲任务就可以了。进入空闲任务后，首先要计算可以执行低功耗的最大时间，也就是求出下一个要执行的高优先级任务还剩多少时间。然后就是把低功耗的唤醒时间设置为这个求出的时间，到时间后系统会从低功耗模式被唤醒，继续执行多任务。这个就是所谓的tickless模式。从上面的讲解中可以看出，实现tickless模式最麻烦是低功耗可以执行的时间如何获取。关于这个问题，FreeRTOS已经为我们做好了。

对于M4内核来说，FreeRTOS已经提供了tickless低功耗代码的实现，通过调用指令WFI实现睡眠模式，具体代码的实现就在port.c文件中，用户只需在FreeRTOSConfig.h文件中配置宏定义configUSE_TICKLESS_IDLE为1即可。如果配置此参数为2，那么用户可以自定义tickless低功耗模式的实现。当用户将宏定义configUSE_TICKLESS_IDLE配置为1且系统运行满足以下两个条件时，系统内核会自动的调用低功耗宏定义函数portSUPPRESS_TICKS_AND_SLEEP()：

1. 当前空闲任务正在运行，所有其它的任务处在挂起状态或者阻塞状态。
2. 根据用户配置configEXPECTED_IDLE_TIME_BEFORE_SLEEP的大小，只有当系统可运行于低功耗模式的时钟节拍数大于等于这个参数时，系统才可以进入到低功耗模式。此参数默认已经在FreeRTOS.h文件进行定义了，下面是具体的定义内容，当然，用户也可以在FreeRTOSConfig.h文件中重新定义

```
#ifndef configEXPECTED_IDLE_TIME_BEFORE_SLEEP
    #define configEXPECTED_IDLE_TIME_BEFORE_SLEEP 2
#endif

#if configEXPECTED_IDLE_TIME_BEFORE_SLEEP < 2
    #error configEXPECTED_IDLE_TIME_BEFORE_SLEEP must not be less than 2
#endif
```

此处配套例程设置为：

```
#define configEXPECTED_IDLE_TIME_BEFORE_SLEEP 10
```

函数portSUPPRESS_TICKS_AND_SLEEP是FreeRTOS实现tickless模式的关键，此函数被空闲任务调用，其定义是在portmacro.h文件中：

```
/* Tickless idle/low power functionality. */
```

```
#ifndef portSUPPRESS_TICKS_AND_SLEEP
    extern void vPortSuppressTicksAndSleep( TickType_t xExpectedIdleTime );
    #define portSUPPRESS_TICKS_AND_SLEEP( xExpectedIdleTime )
vPortSuppressTicksAndSleep( xExpectedIdleTime )
#endif
```

portSUPPRESS_TICKS_AND_SLEEP这个函数最终会由空闲任务调用，从而进入低功耗模式。

FreeRTOS自带的程序是让系统进入睡眠模式，对于AT32 MCU来讲，拥有睡眠模式、停止模式、待机模式，为什么FreeRTOS只提供睡眠模式的低功耗呢？这和FreeRTOS的Tickless模式实现相关，Tickless是利用M4内核的Systick定时器来算出进入低功耗的时间，然而在停止模式和待机模式下，Systick的时钟是会变的，会导致计时不准，从而导致FreeRTOS系统心跳不准，所以是不能使用这两种低功耗模式的。

FreeRTOS的port.c文件中进入低功耗的代码段如下：

```
configPRE_SLEEP_PROCESSING( xModifiableIdleTime );
if( xModifiableIdleTime > 0 )
{
    __dsb( portSY_FULL_READ_WRITE );
    __wfi();
    __isb( portSY_FULL_READ_WRITE );
}
configPOST_SLEEP_PROCESSING( xExpectedIdleTime );
```

代码中调用wfi指令使其进入睡眠模式，上下还有两个宏定义，是提供给开发者扩展使用的。可以在里面增加一些进入低功耗时关闭时钟降低主频和退出低功耗开启时钟升高主频的程序。

在FreeRTOSConfig.h中重定向如下：

```
/* 低功耗相关处理 */
extern void Enter_Low_Power_Mode(uint32_t xExpectedIdleTime);
extern void Output_Low_Power_Mode(uint32_t xExpectedIdleTime);

#define configPRE_SLEEP_PROCESSING    Enter_Low_Power_Mode
#define configPOST_SLEEP_PROCESSING   Output_Low_Power_Mode
```

在main.c文件中定义函数如下：

```
/* 低功耗相关配置 */
void Enter_Low_Power_Mode(uint32_t xExpectedIdleTime);
void Output_Low_Power_Mode(uint32_t xExpectedIdleTime);

void Enter_Low_Power_Mode(uint32_t xExpectedIdleTime)
{
    printf("Enter low power mode.\r\n");
```

```

/* 用户可根据需求，关闭外设时钟，降低主频等操作 */
}
void Output_Low_Power_Mode(uint32_t xExpectedIdleTime)
{
    printf("Output low power mode.\r\n");
    /* 用户根据需求恢复到进低功耗之前的状态 */
}

```

注：如果用户想使用AT32低功耗的停机模式和待机模式，也可添加相应程序使其进入想要的低功耗模式，但是唤醒源需要自行配置并不能保证系统时钟是准确的。例如使用到了软件定时器组的话，如果进入了低功耗，则会导致定时不准。

11.2 例程介绍

工程名：13Low_Power_FreeRTOS

程序源码：

```

#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* 开始任务优先级 */
#define START_TASK_PRIO      1
/* 开始任务堆栈大小 */
#define START_STK_SIZE      128
/* 开始任务任务句柄 */
TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* 消息处理任务优先级 */
#define Process_Message_TASK_PRIO      1
/* 消息处理任务堆栈大小 */
#define Process_Message_STK_SIZE      256
/* 消息处理任务任务句柄 */
TaskHandle_t Process_MessageTask_Handler;
/* 消息处理任务入口函数 */
void Process_Message_task(void *pvParameters);

/* 消息接受任务优先级 */
#define Receive_Message_TASK_PRIO      3
/* 消息接受任务堆栈大小 */
#define Receive_Message_STK_SIZE      256
/* 消息接受任务任务句柄 */
TaskHandle_t Receive_MessageTask_Handler;

```

```

/* 消息接受任务入口函数 */
void Receive_Message_task(void *pvParameters);

/* 定义一个消息结构体 */
typedef struct A_Message
{
    char ucMessageID;
    u8 ucData;
} AMessage;

/* 定义一个消息队列 */
QueueHandle_t AT_xQueue;
/* 消息队列长度和消息大小 */
#define QUEUE_LENGTH 5
#define QUEUE_ITEM_SIZE sizeof( AMessage )

/* 低功耗相关配置 */
void Enter_Low_Power_Mode(uint32_t xExpectedIdleTime);
void Output_Low_Power_Mode(uint32_t xExpectedIdleTime);

void Enter_Low_Power_Mode(uint32_t xExpectedIdleTime)
{
    printf("Enter low power mode.\r\n");
    /* 用户可根据需求，关闭外设时钟，降低主频等操作 */
}

void Output_Low_Power_Mode(uint32_t xExpectedIdleTime)
{
    printf("Output low power mode.\r\n");
    /* 用户根据需求恢复到进低功耗之前的状态 */
}

int main(void)
{
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

    system_clock_config();

    at32_led_init(LED2);
    at32_led_init(LED3);
    at32_led_init(LED4);

    at32_button_init();

    /* init usart1 */
    uart_print_init(115200);
    
```

```

/* 创建开始任务 */
xTaskCreate((TaskFunction_t)start_task,
            (const char*   )"start_task",
            (uint16_t      )START_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )START_TASK_PRIO,
            (TaskHandle_t* )&StartTask_Handler);

/* 开启任务调度器 */
vTaskStartScheduler();
}

/* 开始任务函数 */
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();
    /* 创建消息队列 */
    AT_xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );
    if(AT_xQueue == NULL)
    {
        /* 消息队列创建失败 */
        while(1);
    }
    /* 必须在创建消息队列之后再初始化定时器 */
    TIMER_Init();
    /* 创建消息接受任务 */
    xTaskCreate((TaskFunction_t)Receive_Message_task,
                (const char*   )"Receive_Message_task",
                (uint16_t      )Receive_Message_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t   )Receive_Message_TASK_PRIO,
                (TaskHandle_t* )&Receive_MessageTask_Handler);

    /* 创建消息处理任务 */
    xTaskCreate((TaskFunction_t)Process_Message_task,
                (const char*   )"Process_Message_task",
                (uint16_t      )Process_Message_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t   )Process_Message_TASK_PRIO,
                (TaskHandle_t* )&Process_MessageTask_Handler);

    /* 创建调试任务 */
    xTaskCreate((TaskFunction_t)debug_task,
                (const char*   )"Debug_task",
                (uint16_t      )Debug_STK_SIZE,
                (void*         )NULL,
                (UBaseType_t   )Debug_TASK_PRIO,
                (TaskHandle_t* )&DebugTask_Handler);

```



```

        (TaskHandle_t*) &DebugTask_Handler);

/* 删除开始任务 */
vTaskDelete(StartTask_Handler);

/* 退出临界区 */
taskEXIT_CRITICAL();
}

/* 消息接受任务函数 */
void Receive_Message_task(void *pvParameters)
{
    AMessage Message1;
    while(1)
    {
        switch(GetUartData())
        {
            case NODATA:
                break;
            case 0x01:
                Message1.ucMessageID = 'a';
                Message1.ucData = 0x01;
                /* 接受 Toggle LED2 的消息 */
                xQueueSend(AT_xQueue, &Message1, 10);
                break;
            case 0x02:
                Message1.ucMessageID = 'b';
                Message1.ucData = 0x02;
                /* 接受 Toggle LED3 的消息 */
                xQueueSend(AT_xQueue, &Message1, 10);
                break;
            case 0x03:
                Message1.ucMessageID = 'c';
                Message1.ucData = 0x03;
                /* 接受 Toggle LED4 的消息 */
                xQueueSend(AT_xQueue, &Message1, 10);
                break;
            default:
                break;
        }
        vTaskDelay(100);
    }
}

/* 消息处理任务函数 */
void Process_Message_task(void *pvParameters)
{
    AMessage Message2;
    while(1)

```

```

{
    /* 接受消息 */
    if(xQueueReceive(AT_xQueue, &Message2, portMAX_DELAY) != pdPASS)
    {
        printf("no message\r\n");
    }
    else
    {
        /* 判断消息类型并做出相应动作 */
        if((Message2.ucData == 0x01)&&(Message2.ucMessageID == 'a'))
            at32_led_toggle(LED2);
        if((Message2.ucData == 0x02)&&(Message2.ucMessageID == 'b'))
            at32_led_toggle(LED3);
        if((Message2.ucData == 0x03)&&(Message2.ucMessageID == 'c'))
            at32_led_toggle(LED4);
        if((Message2.ucData == 0x04)&&(Message2.ucMessageID == 'd'))
            printf("Timer interrupt\r\n");
    }
}
}

/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次调试信息 */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/-----*\r\n");
            printf("Task          Status          priority      Remaining_Stack      Num\r\n");
            vTaskList((char *)&buff);
            printf("%s\r\n",buff);
            printf("/-----*\r\n");
            printf("Task          Runing_Num      Usage_Rate\r\n");
            vTaskGetRunTimeStats((char *)&buff);
            printf("%s\r\n",buff);
        }
        vTaskDelay(10);
    }
}

void TMR3_GLOBAL_IRQHandler(void)
{
    AMessage Message3;

```

```

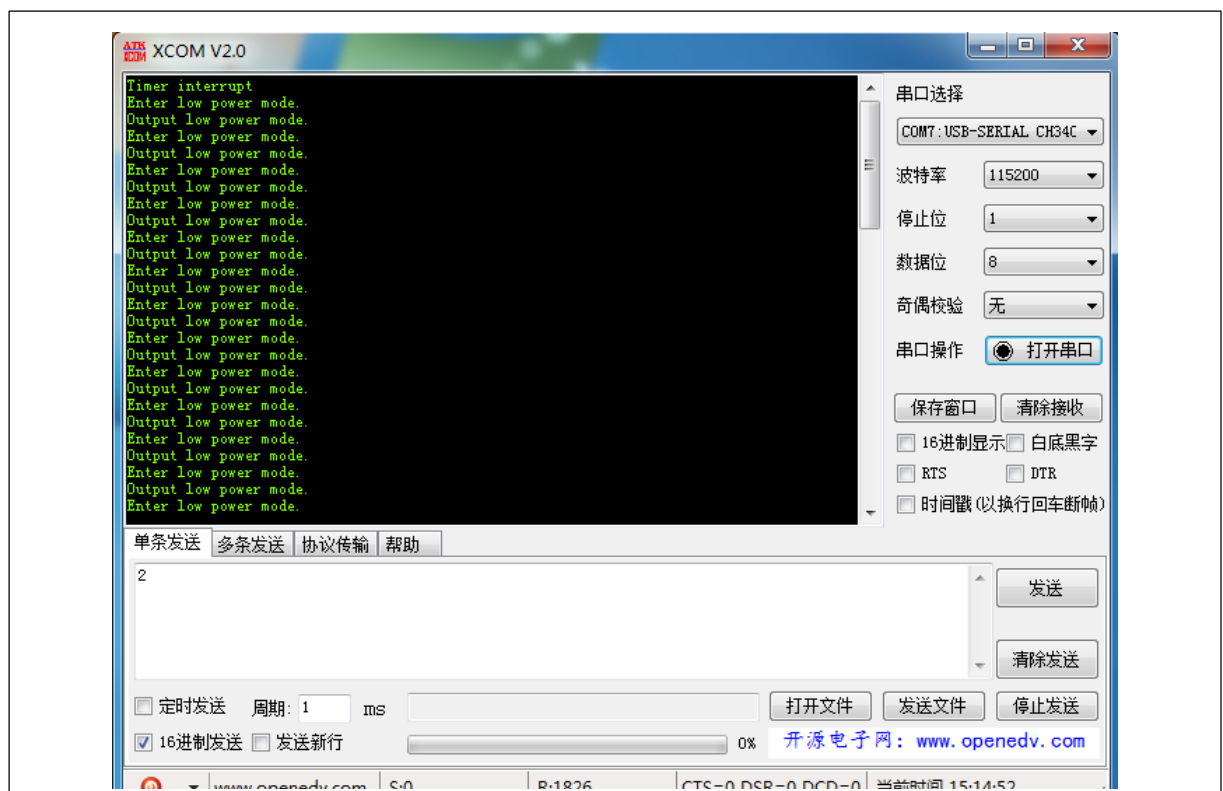
BaseType_t xHigherPriorityTaskWoken = pdFALSE;
if(tmr_flag_get(TMR3,TMR_OVF_FLAG)==SET)
{
    Message3.ucMessageID = 'd';
    Message3.ucData = 0x04;
    /* 发送定时器中断消息到消息队列 */
    xQueueSendFromISR(AT_xQueue,&Message3,&xHigherPriorityTaskWoken);
    /* 判断是否需要任务切换 */
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    tmr_flag_clear(TMR3,TMR_OVF_FLAG);
}
}

```

以上是FreeRTOS低功耗模式例程的源程序，此程序是在消息队列程序的基础上增加低功耗修改而来。修改点上面有讲述。程序具体细节可查看配套工程查看。

编译并下载程序到目标板，运行效果如下：

图 37. 低功耗模式例程演示



以上就是低功耗模式例程的运行结果，打印可看出程序会一直循环进入和退出低功耗模式，因为只要运行到空闲任务，判断进入空闲任务的时间大于10的系统时钟就会进入低功耗，之后达到时间又退出低功耗。

12 FreeRTOS 内存管理方式

本节介绍一下FreeRTOS的内存管理机制，FreeRTOS提供了5种内存管理机制，开发者可根据应用的具体需求选择合适的内存管理方式。其源程序在源码包的如下路径：

FreeRTOS\portable\MemMang。

12.1 内存管理方式一

这种方式是5种内存管理机制中最简单的一种，它简单到只能分配内存，不能释放内存。这种方式的最大优点就是安全，对于绝大多数嵌入式系统，特别是对于安全要求很高的系统，这种内存管理策略是很有用的。因为对于系统软件来说，逻辑越简单就意味着越安全。大多数系统都不需要动态删除信号量、消息队列、任务等，而是在系统初始化时一次性创建好后便一直使用，永远不会删除。所以这个内存管理策略实现简洁、安全可靠、使用度非常高。

这种内存分配策略就像是切面包，将整个内存去比作一长条面包，需要多少就切多少直到面包被切完。这种策略的另一个优点是不会产生内存碎片，内存使用率很高。还需要注意一点，FreeRTOS的内存管理策略都会使用字节对齐，在AT32上采用8字节对齐。所以实际申请的内存可能比需要的内存大，比如需要的内存为12字节，因为8字节对齐，所以实际得到内存为16字节。

上图就是在内存管理方式一下内存运行后的分布框图，开头和末尾两端会根据内存对齐舍弃一部分（如果分配的内存刚好内存对齐就不需要舍弃）。每次调用pvPortMalloc函数会返回获得的内存开始的地址。

下面看看内存管理方式一的源程序：

```
void *pvPortMalloc( size_t xWantedSize )
{
    void *pvReturn = NULL;
    static uint8_t *pucAlignedHeap = NULL;

    /* 确保申请的字节数是字节对齐的整数倍*/
    #if( portBYTE_ALIGNMENT != 1 )
    {
        if( xWantedSize & portBYTE_ALIGNMENT_MASK )
        {
            /* 计数字节对齐后的需申请的字节数 */
            xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
                portBYTE_ALIGNMENT_MASK ) );
        }
    }
    #endif

    vTaskSuspendAll();
    {
        if( pucAlignedHeap == NULL )
        {
            /* 确保申请的内存的起始字节正确对齐 */
            pucAlignedHeap = ( uint8_t * ) ( ( ( portPOINTER_SIZE_TYPE )
```

```

        &ucHeap[ portBYTE_ALIGNMENT ] ) & ( ~( ( portPOINTER_SIZE_TYPE )
        portBYTE_ALIGNMENT_MASK ) ) );
    }

    /* 确保有足够的空间够内存的申请 */
    if( ( ( xNextFreeByte + xWantedSize ) < configADJUSTED_HEAP_SIZE ) &&
        ( ( xNextFreeByte + xWantedSize ) > xNextFreeByte ) )/* Check for overflow. */
    {
        /* Return the next free byte then increment the index past this
        block. */
        pvReturn = pucAlignedHeap + xNextFreeByte;
        xNextFreeByte += xWantedSize;
    }

    traceMALLOC( pvReturn, xWantedSize );
}
( void ) xTaskResumeAll();

#if( configUSE_MALLOC_FAILED_HOOK == 1 )
{
    if( pvReturn == NULL )
    {
        extern void vApplicationMallocFailedHook( void );
        vApplicationMallocFailedHook();
    }
}
#endif

return pvReturn;
}

```

12.2 内存管理方式二

第二种内存管理方式要比第一种复杂一些，它使用一个最佳匹配算法，允许释放之前已分配的内存，但是它不会把相邻的内存块合并成一个大的内存块。

这种内存管理方式适用于一些重复分配和释放相同堆栈空间的任务、队列、信号量等等，并且不考虑内存碎片的产生。不使用与分配和释放随机大小内存的系统。与第一种内存分配方式相同，都是定义一个大的数组，定义为：

```
static uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
```

申请内存：

与第一种管理策略不同，第二种策略定义了一个数据结构来管理空闲的内存块，让空闲的内存块组成一个链表，其定义如下：

```
typedef struct A_BLOCK_LINK
```

```
{
    struct A_BLOCK_LINK *pNextFreeBlock;    /*<< 指向下一块空闲内存 */
    size_t xBlockSize;                      /*<< 空闲内存块的大小，包括链表结构 */
} BlockLink_t;

/* 创建一对链表结构体，xStart 表示链表头，xEnd 表示链表尾 */
static BlockLink_t xStart, xEnd;
```

下面看看申请内存的源程序：

```
void *pvPortMalloc( size_t xWantedSize )
{
    BlockLink_t *pxBlock, *pxPreviousBlock, *pxNewBlockLink;
    static BaseType_t xHeapHasBeenInitialised = pdFALSE;
    void *pvReturn = NULL;

    vTaskSuspendAll();
    {
        /* 判断是否为第一次调用，如果是需要初始化内存堆栈*/
        if( xHeapHasBeenInitialised == pdFALSE )
        {
            prvHeapInit();
            xHeapHasBeenInitialised = pdTRUE;
        }

        /*申请的字节数需要加上链表结构的大小*/
        if( xWantedSize > 0 )
        {
            xWantedSize += heapSTRUCT_SIZE;

            /* 确保申请的字节大小是字节对齐的 */
            if( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) != 0 )
            {
                /* 进行字节对齐 */
                xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
                    portBYTE_ALIGNMENT_MASK ) );
            }
        }

        if( ( xWantedSize > 0 ) && ( xWantedSize < configADJUSTED_HEAP_SIZE ) )
        {
            /*空闲内存块是按照从小到大链接的，找到合适的内存块*/
            pxPreviousBlock = &xStart;
            pxBlock = xStart.pNextFreeBlock;
            while( ( pxBlock->xBlockSize < xWantedSize ) && ( pxBlock->pNextFreeBlock !=
                NULL ) )
```

```

        {
            pxPreviousBlock = pxBlock;
            pxBlock = pxBlock->pxNextFreeBlock;
        }

        /* 如果没有找到合适的内存块的话将不会执行 */
        if( pxBlock != &xEnd )
        {
            /* 返回申请到内存地址，跳过链表结构体后的有效地址*/
            pvReturn = ( void * ) ( ( ( uint8_t * ) pxPreviousBlock->pxNextFreeBlock ) +
                                    heapSTRUCT_SIZE );

            /* 申请到的内存块需要从空闲链表中剔除*/
            pxPreviousBlock->pxNextFreeBlock = pxBlock->pxNextFreeBlock;

            /* 如果申请到的内存块足够大，可以将其一分为二*/
            if ( ( pxBlock->xBlockSize - xWantedSize ) > heapMINIMUM_BLOCK_SIZE )
            {
                /* 去除分配出去的内存，在剩余部分内存的开头插入一个链表*/
                pxNewBlockLink = ( void * ) ( ( ( uint8_t * ) pxBlock ) + xWantedSize );

                /* Calculate the sizes of two blocks split from the single
                block. */
                pxNewBlockLink->xBlockSize = pxBlock->xBlockSize - xWantedSize;
                pxBlock->xBlockSize = xWantedSize;

                /* 将其插入空闲链表中*/
                prvInsertBlockIntoFreeList( ( pxNewBlockLink ) );
            }

            xFreeBytesRemaining -= pxBlock->xBlockSize;
        }
    }

    traceMALLOC( pvReturn, xWantedSize );
}

( void ) xTaskResumeAll();
/* 如果分配失败可调用钩子函数 */
#if( configUSE_MALLOC_FAILED_HOOK == 1 )
{
    if( pvReturn == NULL )
    {
        extern void vApplicationMallocFailedHook( void );
        vApplicationMallocFailedHook();
    }
}

```

```

    }
    #endif

    return pvReturn;
}

```

由以上源程序可以看出，开始有整个内存对组成唯一一个空闲块，在空闲块的起始地址放置了一个链表结构，用于存储这个空闲块的大小和下一个空闲块的地址。在最初阶段，由于只有一个空闲块，所以空闲块的pxNextFreeBlock指向链表xEnd，而链表xStart的pxNextFreeBlock指向空闲块。这样xStart、空闲块和xEnd就组成了一个单链表，xStart表示链表头，xEnd表示链表尾。随着内存申请和释放，空闲块可能会越来越多，但他们任然是以xStart作为链表头，xEnd作为链表尾，根据空闲块大小排序，小的在前，大的在后。

当申请N字节内存时，实际上不仅需要分配N字节内存，还要分配一个BlockLink_t结构体空间，用于描述这个内存块，结构体空间位于空闲块的最开始处，需要注意的是，和第一种分配方式一样，申请的内存大小加上BlockLink_t结构体大小后都要进行字节对齐操作。

来看一下申请内存的过程：首先计算实际要分配的内存大小，判断申请内存是否合法。如果合法则从链表头xStart开始查找，如果某个空闲内存块大小大于或等于需要分配的大小，则从这块内存取出合适的大小返回给申请者，剩下的内存块组成一个新的内存块并按照空闲内存块的大小顺序插入到空闲链表。注意，返回的内存地址不包括链表结构。

内存释放：

第二种内存管理方式可以释放内存，下面来看看是如何实现的。因为不需要合并相邻的空闲块，所以释放内存也相对简单一些，基本思路就是根据传入的参数找到链表结构，然后将这个内存插入到空闲列表中并更新未分配的内存堆计数器大小。源程序如下：

```

void vPortFree( void *pv )
{
    uint8_t *puc = ( uint8_t * ) pv;
    BlockLink_t *pxLink;

    if( pv != NULL )
    {
        /*根据传入参数找到链表结构 */
        puc -= heapSTRUCT_SIZE;

        /*防止某些编译器警告*/
        pxLink = ( void * ) puc;

        vTaskSuspendAll();
        {
            /* 将这个空闲块加入到空闲链表*/
            prvInsertBlockIntoFreeList( ( ( BlockLink_t * ) pxLink ) );
            /* 更新未分配的内存堆大小 */
            xFreeBytesRemaining += pxLink->xBlockSize;

```



```

        traceFREE( pv,  pxLink->xBlockSize );
    }
    ( void ) xTaskResumeAll();
}
}

```

12.3 内存管理方式三

第三种内存管理策略只是简单的封装了标准库的malloc()和free()函数，采用的封装策略是操作内存前挂起调度器，完成后在恢复调度器。封装后的malloc()和free()函数具备线程保护机制。

第一种和第二种内存管理策略都是通过定义一个大数组作为内存堆，数组的大小有宏定义configTOTAL_HEAP_SIZE指定。第三种非内存管理策略与前两种不同，它不在需要通过数组定义内存堆，而是需要编译器设置内存堆空间，一般在启动程序中设定。

下面看看源程序：

内存申请：

```

void *pvPortMalloc( size_t xWantedSize )
{
    void *pvReturn;

    vTaskSuspendAll();
    {
        pvReturn = malloc( xWantedSize );
        traceMALLOC( pvReturn,  xWantedSize );
    }
    ( void ) xTaskResumeAll();

    #if( configUSE_MALLOC_FAILED_HOOK == 1 )
    {
        if( pvReturn == NULL )
        {
            extern void vApplicationMallocFailedHook( void );
            vApplicationMallocFailedHook();
        }
    }
    #endif

    return pvReturn;
}

```

内存释放：

```

void vPortFree( void *pv )
{
    if( pv )
    {

```

```

        vTaskSuspendAll();
        {
            free( pv );
            traceFREE( pv, 0 );
        }
        ( void ) xTaskResumeAll();
    }
}

```

12.4 内存管理方式四

第四种内存分配策略和第二种比较类似，只不过增加了一个合并算法，将相邻的空闲内存块合并成一个大块。和第一种和第二种内存管理策略一样，内存堆任然是一个大数组。定义为：

```
static uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
```

内存申请：

和第二种管理策略一样，使用一个链表结构来管理空闲内存块，结构体定义为：

```

typedef struct A_BLOCK_LINK
{
    struct A_BLOCK_LINK *pxNextFreeBlock;    /*<< 指向下一个空闲块 */
    size_t xBlockSize;                       /*<<空闲内存块的大小，包括链表结构*/
} BlockLink_t;

```

与第二种管理策略一样，空闲内存块也是以单链表的方式组织起来的，**BlockLink_t**类型的局部静态变量**xStart**表示链表头，但第四种内存管理策略的链表尾保存在空闲块的最后位置，并使用**BlockLink_t**类型的局部静态变量**pxEnd**指向这个区域（第二种内存管理策略使用静态变量**xEnd**表示链表尾）。

第四种内存管理策略和第二种内存管理策略还有一个很大的不同是：第四种内存管理策略的空闲内存块不是以内存块大小为存储顺序，而是以地址的大小为存储顺序，这也是为了适应合并算法。

来看看相应的源程序：

```

void *pvPortMalloc( size_t xWantedSize )
{
    BlockLink_t *pxBlock, *pxPreviousBlock, *pxNewBlockLink;
    void *pvReturn = NULL;

    vTaskSuspendAll();
    {
        /*如果是第一次调用，则需要初始化内存堆*/
        if( pxEnd == NULL )
        {
            prvHeapInit();
        }
        else

```

```

{
    mtCOVERAGE_TEST_MARKER();
}

/*申请内存大小合法性检查*/
if( ( xWantedSize & xBlockAllocatedBit ) == 0 )
{
    /* 实际申请内存大小需加上链表结构体的大小*/
    if( xWantedSize > 0 )
    {
        xWantedSize += xHeapStructSize;

        /* 字节对齐检查, 向上扩大到字节对齐的整数倍*/
        if( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) != 0x00 )
        {
            /* 字节对齐计算 */
            xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
                portBYTE_ALIGNMENT_MASK ) );
            configASSERT( ( xWantedSize & portBYTE_ALIGNMENT_MASK )
                == 0 );
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    if( ( xWantedSize > 0 ) && ( xWantedSize <= xFreeBytesRemaining ) )
    {
        /* 从链表开头查找合适的空闲内存块*/
        pxPreviousBlock = &xStart;
        pxBlock = xStart.pxNextFreeBlock;
        while( ( pxBlock->xBlockSize < xWantedSize ) &&
            ( pxBlock->pxNextFreeBlock != NULL ) )
        {
            pxPreviousBlock = pxBlock;
            pxBlock = pxBlock->pxNextFreeBlock;
        }

        /* 如果没有找到则不会执行以下内容*/
        if( pxBlock != pxEnd )

```

```

{
    /* 返回分配的内存指针要跳过 BlockLink_t 结构体*/
    pvReturn = ( void * ) ( ( ( uint8_t * )
        pxPreviousBlock->pxNextFreeBlock ) + xHeapStructSize );

    /* 从空闲内存链表中移除分配到的内存块*/
    pxPreviousBlock->pxNextFreeBlock = pxBlock->pxNextFreeBlock;

    /* 如果剩下的内存足够大则组成一个新的空闲内存块*/
    if( ( pxBlock->xBlockSize - xWantedSize ) >
        heapMINIMUM_BLOCK_SIZE )
    {
        /* 在剩余内存块的其实位置放置一个链表结构并初始化链表成员*/
        pxNewBlockLink = ( void * ) ( ( ( uint8_t * ) pxBlock ) +
            xWantedSize );
        configASSERT( ( ( ( size_t ) pxNewBlockLink ) &
            portBYTE_ALIGNMENT_MASK ) == 0 );

        /* 计算剩余空闲内存块的大小*/
        pxNewBlockLink->xBlockSize = pxBlock->xBlockSize -
            xWantedSize;
        pxBlock->xBlockSize = xWantedSize;

        /* 插入到空闲链表*/
        prvInsertBlockIntoFreeList( pxNewBlockLink );
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    xFreeBytesRemaining -= pxBlock->xBlockSize;

    if( xFreeBytesRemaining < xMinimumEverFreeBytesRemaining )
    {
        xMinimumEverFreeBytesRemaining = xFreeBytesRemaining;
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    /*将已经分配的内存块表示为已分配*/
    pxBlock->xBlockSize |= xBlockAllocatedBit;

```

```

        pxBlock->pxNextFreeBlock = NULL;
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
else
{
    mtCOVERAGE_TEST_MARKER();
}
}
else
{
    mtCOVERAGE_TEST_MARKER();
}

    traceMALLOC( pvReturn,  xWantedSize );
}
( void ) xTaskResumeAll();

#if( configUSE_MALLOC_FAILED_HOOK == 1 )
{
    if( pvReturn == NULL )
    {
        extern void vApplicationMallocFailedHook( void );
        vApplicationMallocFailedHook();
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
#endif

configASSERT( ( ( ( size_t ) pvReturn ) & ( size_t ) portBYTE_ALIGNMENT_MASK ) == 0 );
return pvReturn;
}

```

由以上源程序可以看出，开始有整个内存对组成唯一一个空闲块，在空闲块的起始地址放置了一个链表结构，用于存储这个空闲块的大小和下一个空闲块的地址。在最初阶段，由于只有一个空闲块，所以空闲块的pxNextFreeBlock指向链表xEnd，而链表xStart的pxNextFreeBlock指向空闲块。这样xStart、空闲块和xEnd就组成了一个单链表，xStart表示链表头，xEnd表示链表尾。随着内存申请和释放，空闲块可能会越来越多，但他们任然是以xStart作为链表头，xEnd作为链表尾，根据空闲块地址大小排序，地址小的在前，地址大的在后。

当申请N字节内存时，实际上不仅需要分配N字节内存，还要分配一个BlockLink_t结构体空间，用于描述这个内存块，结构体空间位于空闲块的最开始处，需要注意的是，和第一种、第二种分配方式一样，申请的内存大小加上BlockLink_t结构体大小后都要进行字节对齐操作。

来看一下申请内存的过程：首先计算实际要分配的内存大小，判断申请内存是否合法。如果合法则从链表头xStart开始查找，如果某个空闲内存块大小大于或等于需要分配的大小，则从这块内存取出合适的大小返回给申请者，剩下的内存块组成一个新的内存块并按照空闲内存块地址的大小顺序插入到空闲链表。在插入空闲列表的过程中，还会执行合并算法，首先判断这个空闲块是不是可以和上一个块合并成一个大块，如果可以则合并，然后再判断是否可以和下一个空闲块合并成一个大块，如果可以则合并。注意，返回的内存地址不包括链表结构。

内存释放：

第四种内存管理策略的内存释放也比较简单，根据传入的参数找到链表结构，然后将这个内存块插入进去，需要注意的是插入过程会执行合并算法。最后将这个内存块标志为空闲。

```
void vPortFree( void *pv )
{
    uint8_t *puc = ( uint8_t * ) pv;
    BlockLink_t *pxLink;

    if( pv != NULL )
    {
        /* 根据传入参数找到链表结构*/
        puc -= xHeapStructSize;

        /* 防止编译器报错*/
        pxLink = ( void * ) puc;

        /* 检查这个内存块确实被分配出去*/
        configASSERT( ( pxLink->xBlockSize & xBlockAllocatedBit ) != 0 );
        configASSERT( pxLink->pxNextFreeBlock == NULL );

        if ( ( pxLink->xBlockSize & xBlockAllocatedBit ) != 0 )
        {
            if( pxLink->pxNextFreeBlock == NULL )
            {
                /* 将内存块表示为空闲*/
                pxLink->xBlockSize &= ~xBlockAllocatedBit;

                vTaskSuspendAll();
                {
                    /* 更新未分配的内存堆大小*/
                    xFreeBytesRemaining += pxLink->xBlockSize;
                    traceFREE( pv, pxLink->xBlockSize );
                    prvInsertBlockIntoFreeList( ( ( BlockLink_t * ) pxLink ) );
                }
            }
        }
    }
}
```

```

    }
    ( void ) xTaskResumeAll();
}
else
{
    mtCOVERAGE_TEST_MARKER();
}
}
else
{
    mtCOVERAGE_TEST_MARKER();
}
}
}
}

```

12.5 内存管理方式五

第五种内存管理策略允许内存堆跨越多个连续的内存区。除此之外其他操作都和第四种内存管理方式及其相似。第一、第二和第四种的内存管理方式都是利用一个大的数组作为内存堆使用，并且只需要程序指定数组的大小（通过`static uint8_t ucHeap[configTOTAL_HEAP_SIZE];`），而第五种内存管理策略就不是这样，首先它允许跨内存区定义多个内存堆，比如在片内RAM中定义一个内存堆，还可以在片外RAM中再定义一个内存堆；用户只需要指定每个内存堆的起始地址和大小即可，FreeRTOS为了方便管理多个内存堆，专门定义了一个结构体`HeapRegion_t`用于多个内存堆的管理。其原型为：

```

/* Used by heap_5.c. */
typedef struct HeapRegion
{
    uint8_t *pucStartAddress; /* 起始地址 */
    size_t xSizeInBytes;      /* 空间大小 */
} HeapRegion_t;

```

FreeRTOS的`heap_5.c`文件中提供了一个案例，示范如何定义多个内存堆。实例程序如下：

```

HeapRegion_t xHeapRegions[] =
{
    { ( uint8_t * ) 0x80000000UL, 0x10000 }, << Defines a block of 0x10000 bytes starting
    at address
    0x80000000
    { ( uint8_t * ) 0x90000000UL, 0xa0000 }, << Defines a block of 0xa0000 bytes starting
    at address of
    0x90000000
    { NULL, 0 } << Terminates the array.
};

```

以上程序中定义了两段内存堆，当然可以定义更多的内存堆；需要注意的是`xHeapRegions`数组的最

后一个元素必须是{ NULL, 0 }, 这是为了告诉系统初始化内存堆什么时候结束。

分配内存和释放内存和第四种管理策略基本相同, 这里就不再解释了, 不理解可参考前面小节的内存管理方式四。这里主要讲解下多个内存堆的初始化过程, 源程序如下:

```
void vPortDefineHeapRegions( const HeapRegion_t * const pxHeapRegions )
{
    BlockLink_t *pxFirstFreeBlockInRegion = NULL, *pxPreviousFreeBlock;
    size_t xAlignedHeap;
    size_t xTotalRegionSize, xTotalHeapSize = 0;
    BaseType_t xDefinedRegions = 0;
    size_t xAddress;
    const HeapRegion_t *pxHeapRegion;

    /* Can only call once! */
    configASSERT( pxEnd == NULL );

    pxHeapRegion = &( pxHeapRegions[ xDefinedRegions ] );

    while( pxHeapRegion->xSizeInBytes > 0 )
    {
        xTotalRegionSize = pxHeapRegion->xSizeInBytes;

        /* 确保内存堆起始地址字节对齐 */
        xAddress = ( size_t ) pxHeapRegion->pucStartAddress;
        if( ( xAddress & portBYTE_ALIGNMENT_MASK ) != 0 )
        {
            xAddress += ( portBYTE_ALIGNMENT - 1 );
            xAddress &= ~portBYTE_ALIGNMENT_MASK;

            /* 调节字节对齐后的内存堆大小 */
            xTotalRegionSize -= xAddress - ( size_t ) pxHeapRegion->pucStartAddress;
        }

        xAlignedHeap = xAddress;

        /*如果没有初始化 xStart 的话就初始化 */
        if( xDefinedRegions == 0 )
        {
            /* xStart 用来指向第一块空闲内存*/
            xStart.pxNextFreeBlock = ( BlockLink_t * ) xAlignedHeap;
            xStart.xBlockSize = ( size_t ) 0;
        }
        else
        {
            /* 当已经初始化完一块内存堆后才会执行这里*/
            configASSERT( pxEnd != NULL );
        }
    }
}
```



```

        /* Check blocks are passed in with increasing start addresses. */
        configASSERT( xAddress > ( size_t ) pxEnd );
    }

    /*记录 pxEnd 在上一个区域中的位置（如果有） */
    pxPreviousFreeBlock = pxEnd;

    /* pxEnd 用来记录空闲链表的末尾，即 pxEnd 处在最后一个区域的尾部*/
    xAddress = xAlignedHeap + xTotalRegionSize;
    xAddress -= xHeapStructSize;
    xAddress &= ~portBYTE_ALIGNMENT_MASK;
    pxEnd = ( BlockLink_t * ) xAddress;
    pxEnd->xBlockSize = 0;
    pxEnd->pxNextFreeBlock = NULL;

    /*首先此区域内有一个空闲块，其大小为整个区域减去空闲块结构体的大小*/
    pxFirstFreeBlockInRegion = ( BlockLink_t * ) xAlignedHeap;
    pxFirstFreeBlockInRegion->xBlockSize = xAddress - ( size_t )
pxFirstFreeBlockInRegion;
    pxFirstFreeBlockInRegion->pxNextFreeBlock = pxEnd;

    /*如果这不是构成整个区域的第一块区域，则将前一块区域链接到此区域*/
    if( pxPreviousFreeBlock != NULL )
    {
        pxPreviousFreeBlock->pxNextFreeBlock = pxFirstFreeBlockInRegion;
    }

    xTotalHeapSize += pxFirstFreeBlockInRegion->xBlockSize;

    /* 跳转到下一块内存堆 */
    xDefinedRegions++;
    pxHeapRegion = &( pxHeapRegions[ xDefinedRegions ] );
}

xMinimumEverFreeBytesRemaining = xTotalHeapSize;
xFreeBytesRemaining = xTotalHeapSize;

/* Check something was actually defined before it is accessed. */
configASSERT( xTotalHeapSize );

/* Work out the position of the top bit in a size_t variable. */
xBlockAllocatedBit = ( ( size_t ) 1 ) << ( ( sizeof( size_t ) * heapBITS_PER_BYTE ) - 1 );
}

```

可以看到以上程序会根据用户给出的xHeapRegions结构体来按照顺序初始化内存堆，其主要功能就是将这些内存堆链接起来供后续系统分配和释放使用。

13 FreeRTOS 流缓存

本节介绍FreeRTOS的流缓存的使用，流缓存的作用主要是为了数据传递。

13.1 流缓存介绍

流缓存提供任务与任务间和中断和任务间的数据传递。不同于FreeRTOS的其他消息传递手段，流缓存最佳的使用方式是单个发送者和单个接受者，例如从中断处理函数中将数据发送给某个任务或者在多核芯片上的双核通信，数据从一个核心传递到另外一个核心。流缓存的数据流通靠的是复制的方式，即发送者将数据拷贝到缓存区，然后接受者从缓存将数据拷贝走。

流缓存区传输连续的字节流，消息缓存器可传输任意大小但不连续的消息，消息缓存区采用流缓存区进行传递数据。消息缓存区将在后续的章节介绍。

注：对于FreeRTOS对象，流缓存对象（消息缓存也是如此，应为消息缓存是在流缓存之上构建的）假设只有一个任务或中断对他有写入操作，而且只有一个任务或中断对他有读取操作，那么这样可以保证是安全的；但是不同于其他的FreeRTOS对象，当有多个任务或中断对流缓存有写入动作，有多个任务或中断对流缓存有读取动作的话是不安全的。所以在这种情况下调用相关的API函数的时候要用临界区进行保护而且等待超时时间设置为0。

13.2 流缓存 API

下面来看看流缓存相关的API函数：

表 47. 流缓存 API

流缓存 API 函数	
API	描述
<code>xStreamBufferBytesAvailable()</code>	获取流缓存中的数据量
<code>xStreamBufferCreate()</code>	创建一个流缓存
<code>xStreamBufferCreateStatic()</code>	静态方式创建一个流缓存
<code>vStreamBufferDelete()</code>	删除一个流缓存
<code>xStreamBufferIsEmpty()</code>	判断流缓存是否为空
<code>xStreamBufferIsFull()</code>	判断流缓存是否满了
<code>xStreamBufferReceive()</code>	从流缓存接收数据
<code>xStreamBufferReceiveFromISR()</code>	从流缓存接收数据（中断级）
<code>xStreamBufferReset()</code>	复位指定流缓存
<code>xStreamBufferSend()</code>	发送数据到流缓存
<code>xStreamBufferSendFromISR()</code>	发送数据到流缓存（中断级）
<code>xStreamBufferSetTriggerLevel()</code>	流缓存的触发等级
<code>xStreamBufferSpacesAvailable()</code>	查看流缓存的空闲空间大小

以上就是流缓存相关的API函数，下面来看看几个常用的API函数原型。

`xStreamBufferCreate();`

描述：

创建一个流缓存。

原型：

表 48. 创建流缓存函数原型

StreamBufferHandle_t xStreamBufferCreate(size_t xBufferSizeBytes, size_t xTriggerLevelBytes);	
参数	描述
xBufferSizeBytes	流缓存的空间大小
xTriggerLevelBytes	流缓存的触发等级

返回值：

创建的流缓存的句柄。

xStreamBufferReceive ();

描述：

从流缓存接收数据。

原型：

表 49. 从流缓存接收数据函数原型

size_t xStreamBufferReceive(StreamBufferHandle_t xStreamBuffer, void *pvRxData, size_t xBufferLengthBytes, TickType_t xTicksToWait);	
参数	描述
xStreamBuffer	流缓存的句柄
pvRxData	数据存放地址
xBufferLengthBytes	接收数据长度
xTicksToWait	超时等待时间

返回值：

接收到的数据量。

xStreamBufferSend();

描述：

发送数据到流缓存。

原型：

表 50. 发送数据到流缓存函数原型

size_t xStreamBufferSend(StreamBufferHandle_t xStreamBuffer, const void *pvTxData, size_t xDataLengthBytes, TickType_t xTicksToWait);	
参数	描述

xStreamBuffer	流缓存的句柄
pvTxData	要发送的数据地址
xBufferLengthBytes	发送数据长度
xTicksToWait	超时等待时间

返回值:

发送的数据量。

13.3 例程介绍

工程名: **14Stream_Buffers_FreeRTOS**

程序源码:

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "stream_buffer.h"

/* 开始任务优先级 */
#define START_TASK_PRIO      1
/* 开始任务堆栈大小 */
#define START_STK_SIZE       128
/* 开始任务任务句柄 */
TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* Stream Buffers 发送任务优先级 */
#define Stream_Buffers_Send_TASK_PRIO      3
/* Stream Buffers 发送任务堆栈大小 */
#define Stream_Buffers_Send_STK_SIZE       256
/* Stream Buffers 发送任务任务句柄 */
TaskHandle_t Stream_Buffers_SendTask_Handler;
/* Stream Buffers 发送任务入口函数 */
void Stream_Buffers_Send_task(void *pvParameters);

/* Stream Buffers 接收任务优先级 */
#define Stream_Buffers_Receive_TASK_PRIO    3
/* Stream Buffers 接收任务堆栈大小 */
#define Stream_Buffers_Receive_STK_SIZE     256
/* Stream Buffers 接收任务任务句柄 */
TaskHandle_t Stream_Buffers_ReceiveTask_Handler;
/* Stream Buffers 接收任务入口函数 */
void Stream_Buffers_Receive_task(void *pvParameters);
```

```

/* 调试任务优先级 */
#define Debug_TASK_PRIO      3
/* 调试任务堆栈大小 */
#define Debug_STK_SIZE      512
/* 调试任务任务句柄 */
TaskHandle_t DebugTask_Handler;
/* 调试任务入口函数 */
void debug_task(void *pvParameters);

/* 定义一个 Stream Buffer */
StreamBufferHandle_t AT_xStreamBuffer;
const size_t AT_xStreamBufferSizeBytes = 100, AT_xTriggerLevel = 1;

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);
    at32_button_init();
    /* init usart1 */
    uart_print_init(115200);
    /* 创建开始任务 */
    xTaskCreate((TaskFunction_t)start_task,
                (const char* )"start_task",
                (uint16_t )START_STK_SIZE,
                (void* )NULL,
                (UBaseType_t )START_TASK_PRIO,
                (TaskHandle_t* )&StartTask_Handler);
    /* 打开调度器 */
    vTaskStartScheduler();
}

/* 开始任务函数 */
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();

    /* 创建 Stream Buffer */
    AT_xStreamBuffer = xStreamBufferCreate( AT_xStreamBufferSizeBytes,
    AT_xTriggerLevel );
    if( AT_xStreamBuffer == NULL )
    {
        /* 创建 Stream Buffer 失败 */
        while(1);
    }

    /* 初始化定时器 */
}

```

```

TIMER_Init();
/* 创建 Stream Buffers 接收任务 */
xTaskCreate((TaskFunction_t)Stream_Buffers_Receive_task,
            (const char*   )"Receive_Stream_Buffers_task",
            (uint16_t      )Stream_Buffers_Receive_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )Stream_Buffers_Receive_TASK_PRIO,
            (TaskHandle_t* )&Stream_Buffers_ReceiveTask_Handler);
/* 创建 Stream Buffers 发送任务 */
xTaskCreate((TaskFunction_t)Stream_Buffers_Send_task,
            (const char*   )"Send_Stream_Buffers_task",
            (uint16_t      )Stream_Buffers_Send_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )Stream_Buffers_Send_TASK_PRIO,
            (TaskHandle_t* )&Stream_Buffers_SendTask_Handler);
/* 创建调试任务 */
xTaskCreate((TaskFunction_t)debug_task,
            (const char*   )"Debug_task",
            (uint16_t      )Debug_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )Debug_TASK_PRIO,
            (TaskHandle_t* )&DebugTask_Handler);
/* 删除开始任务 */
vTaskDelete(StartTask_Handler);
/* 退出临界区 */
taskEXIT_CRITICAL();
}
/* Stream Buffers 接收任务函数 */
void Stream_Buffers_Receive_task(void *pvParameters)
{
    uint8_t ucRxData[ 2 ] = { '0', '0' };
    size_t xReceivedBytes;
    const TickType_t xBlockTime = pdMS_TO_TICKS( 20 );
    while(1)
    {
        /* 从 AT_xStreamBuffer 接收数据 */
        xReceivedBytes = xStreamBufferReceive( AT_xStreamBuffer, ( void * ) ucRxData,
        sizeof( ucRxData ), xBlockTime );
        if( xReceivedBytes > 0 )
        {
            printf("%s\r\n", ucRxData);
        }
    }
}
/* Stream Buffers 发送任务函数 */

```

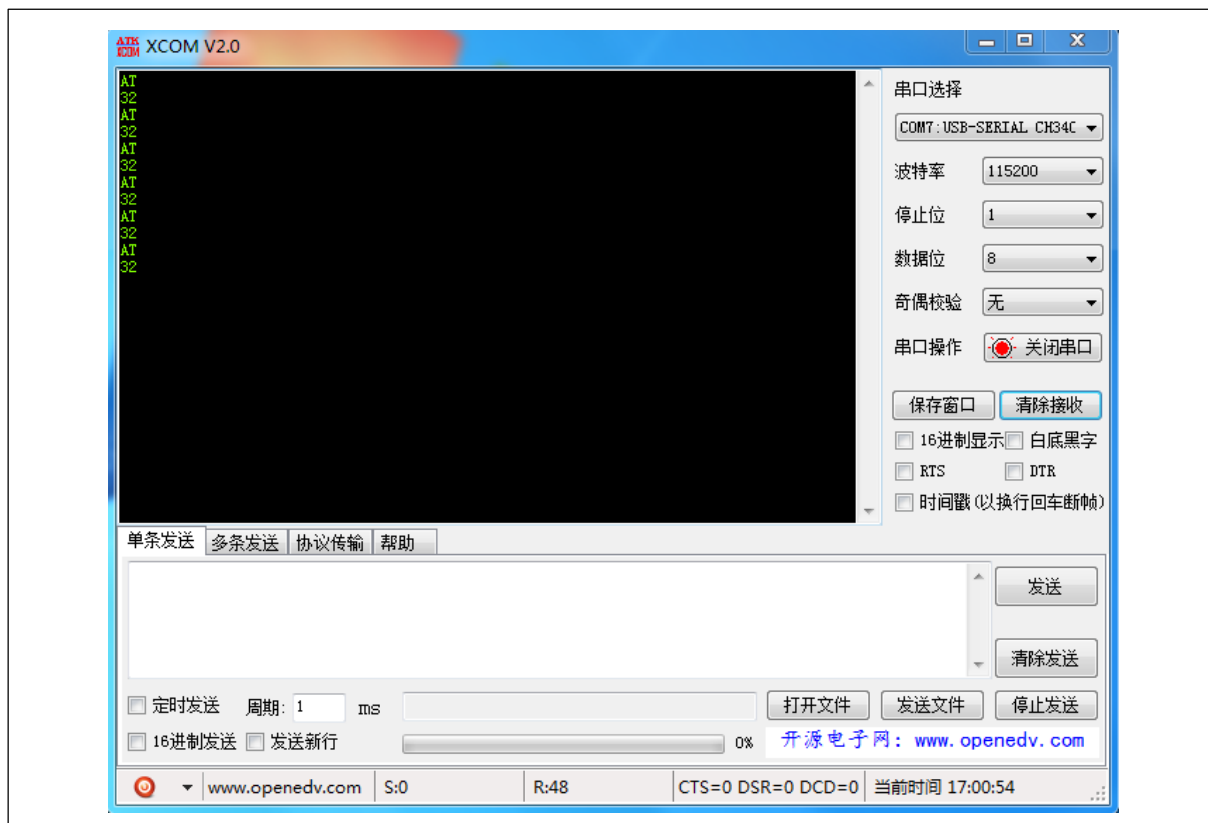
```
void Stream_Buffers_Send_task(void *pvParameters)
{
    size_t xBytesSent;
    uint8_t ucArrayToSend[] = { 'A', 'T', '3', '2' };
    const TickType_t x100ms = pdMS_TO_TICKS( 100 );
    while(1)
    {
        /* 发送 ucArrayToSend[]中的数据 */
        xBytesSent = xStreamBufferSend( AT_xStreamBuffer, ( void * ) ucArrayToSend,
            sizeof( ucArrayToSend ), x100ms );
        if( xBytesSent != sizeof( ucArrayToSend ) )
        {
            printf("#1:%d data written in.\r\n", xBytesSent);
        }
        vTaskDelay(3000);
    }
}

/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次任务信息 */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/*****\r\n");
            printf("Task      Status      priority      Remaining_Stack      Num\r\n");
            vTaskList((char *)&buff);
            printf("%s\r\n", buff);
            printf("/*****\r\n");
            printf("Task      Runing_Num      Usage_Rate\r\n");
            vTaskGetRunTimeStats((char *)&buff);
            printf("%s\r\n", buff);
        }
        vTaskDelay(10);
    }
}
```

以上就是流缓存例程的配套源码，程序中首先创建一个流缓存对象，然后有任务循环的向流缓存中写入数据，另外会有任务从流缓存中接受数据；当流缓存中没有数据的话，接受数据的任务就会进入阻塞态等待流缓存中存入数据。当接受任务接受到数据后，会将数据打印出来，本例程设置每次接收流缓存中的三个数据。

编译并下载程序到目标板，运行效果如下：

图 38. 流缓存例程演示



可以看到接收任务每次只接受到3个数据，然后将其打印出来。

14 FreeRTOS 消息缓存

本节介绍消息缓存的使用，消息缓存的主要作用也是数据的传递。

14.1 消息缓存介绍

消息缓存提供任务与任务间和中断和任务间的数据传递。不同于FreeRTOS的其他消息传递手段，消息缓存最佳的使用方式是单个发送者和单个接受者，例如从中断处理函数中将数据发送给某个任务或者在多核芯片上的双核通信，数据从一个核心传递到另外一个核心。消息缓存的数据流通靠的是复制的方式，即发送者将数据拷贝到缓存区，然后接受者从缓存将数据拷贝走。消息缓存是在流缓存的基础上实现的，但消息缓存只能传递和接受相同长度的数据，即发送10字节数据，那么每次接受也只能接受10字节。

注：对于FreeRTOS对象，消息缓存对象假设只有一个任务或中断对他有写入操作，而且只有一个任务或中断对他有读取操作，那么这样可以保证是安全的；但是不同于其他的FreeRTOS对象，当有多个任务或中断对消息缓存有写入动作，有多个任务或中断对消息缓存有读取动作的话是不安全的。所以在这种情况下调用相关的API函数的时候要用临界区进行保护而且等待超时时间设置为0。

14.2 消息缓存 API

下面来看看消息缓存的API函数：

表 51. 计数型信号量 API

消息缓存 API 函数	
API	描述
xMessageBufferCreate()	创建一个消息缓存
xMessageBufferCreateStatic()	静态方式创建一个消息缓存
vMessageBufferDelete()	删除一个消息缓存
xMessageBufferIsEmpty()	判断消息缓存是否为空
xMessageBufferIsFull()	判断消息缓存是否满了
xMessageBufferReceive()	从消息缓存接收数据
xMessageBufferReceiveFromISR()	从消息缓存接收数据（中断级）
xMessageBufferReset()	复位指定消息缓存
xMessageBufferSend()	发送数据到消息缓存
xMessageBufferSendFromISR()	发送数据到消息缓存（中断级）
xMessageBufferSpacesAvailable()	查看消息缓存的空闲空间大小

以上就是流缓存相关的API函数，下面来看看几个常用的API函数原型。

xMessageBufferCreate ();

描述：

创建一个消息缓存。

原型：

表 52. 创建消息缓存函数原型

MessageBufferHandle_t xMessageBufferCreate(size_t xBufferSizeBytes);	
参数	描述
xBufferSizeBytes	消息缓存的空间大小

返回值:

创建的消息缓存的句柄。

xMessageBufferReceive ();

描述:

从消息缓存接收数据。

原型:

表 53. 从消息缓存接收数据函数原型

size_t xMessageBufferReceive(MessageBufferHandle_t xMessageBuffer, void *pvRxData, size_t xBufferLengthBytes, TickType_t xTicksToWait);	
参数	描述
xMessageBuffer	消息缓存的句柄
pvRxData	数据存放地址
xBufferLengthBytes	接收数据长度
xTicksToWait	超时等待时间

返回值:

接收到的数据量。

xStreamBufferSend();

描述:

发送数据到消息缓存。

原型:

表 54. 发送数据到消息缓存函数原型

Size_t xMessageBufferSend(MessageBufferHandle_t xMessageBuffer, const void *pvTxData, size_t xDataLengthBytes, TickType_t xTicksToWait);	
参数	描述
xMessageBuffer	消息缓存的句柄
pvTxData	要发送的数据地址
xDataLengthBytes	发送数据长度
xTicksToWait	超时等待时间

返回值：
发送的数据量。

14.3 例程介绍

工程名：15Message_Buffers_FreeRTOS

程序源码：

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "stream_buffer.h"
#include "message_buffer.h"

/* 开始任务优先级*/
#define START_TASK_PRIO      1
/* 开始任务堆栈大小 */
#define START_STK_SIZE      128
/* 开始任务任务句柄 */
TaskHandle_t StartTask_Handler;
/* 开始任务入口函数 */
void start_task(void *pvParameters);

/* Message Buffers 发送任务优先级 */
#define Message_Buffers_Send_TASK_PRIO      4
/* Message Buffers 发送任务堆栈大小 */
#define Message_Buffers_Send_STK_SIZE      256
/* Message Buffers 发送任务任务句柄 */
TaskHandle_t Message_Buffers_SendTask_Handler;
/* Message Buffers 发送任务入口函数 */
void Message_Buffers_Send_task(void *pvParameters);

/* Message Buffers 接收任务优先级 */
#define Message_Buffers_Receive_TASK_PRIO      3
/* Message Buffers 接收任务堆栈大小 */
#define Message_Buffers_Receive_STK_SIZE      256
/* Message Buffers 接收任务任务句柄 */
TaskHandle_t Message_Buffers_ReceiveTask_Handler;
/* Message Buffers 接收任务入口函数 */
void Message_Buffers_Receive_task(void *pvParameters);

/* 调试任务优先级 */
#define Debug_TASK_PRIO      3
```

```

/* 调试任务堆栈大小 */
#define Debug_STK_SIZE      512
/* 调试任务任务句柄 */
TaskHandle_t DebugTask_Handler;
/* 调试任务入口函数 */
void debug_task(void *pvParameters);

/* 定义一个 Message Buffer */
MessageBufferHandle_t AT_xMessageBuffer;
const size_t AT_xMessageBufferSizeBytes = 100;

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);
    at32_button_init();
    uart_print_init(115200);
    /* 创建开始任务 */
    xTaskCreate((TaskFunction_t)start_task,
                (const char* )"start_task",
                (uint16_t )START_STK_SIZE,
                (void* )NULL,
                (UBaseType_t )START_TASK_PRIO,
                (TaskHandle_t* )&StartTask_Handler);
    /* 打开调度器 */
    vTaskStartScheduler();
}

/* 开始任务函数 */
void start_task(void *pvParameters)
{
    /* 进入临界区 */
    taskENTER_CRITICAL();

    /* 创建 Message Buffer */
    AT_xMessageBuffer = xMessageBufferCreate( AT_xMessageBufferSizeBytes );
    if( AT_xMessageBuffer == NULL )
    {
        /* 创建 Message Buffer 失败 */
        while(1);
    }
    /* 初始化定时器 */
    TIMER_Init();
    /* 创建 Message Buffers 接收任务 */
    xTaskCreate((TaskFunction_t)Message_Buffers_Receive_task,
                (const char* )"Receive_Message_Buffers_task",

```

```

        (uint16_t      )Message_Buffers_Receive_STK_SIZE,
        (void*         )NULL,
        (UBaseType_t   )Message_Buffers_Receive_TASK_PRIO,
        (TaskHandle_t* )&Message_Buffers_ReceiveTask_Handler);
/* 创建 Message Buffers 发送任务 */
xTaskCreate((TaskFunction_t)Message_Buffers_Send_task,
            (const char*   )"Send_Message_Buffers_task",
            (uint16_t      )Message_Buffers_Send_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )Message_Buffers_Send_TASK_PRIO,
            (TaskHandle_t* )&Message_Buffers_SendTask_Handler);
/* 创建调试任务 */
xTaskCreate((TaskFunction_t)debug_task,
            (const char*   )"Debug_task",
            (uint16_t      )Debug_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t   )Debug_TASK_PRIO,
            (TaskHandle_t* )&DebugTask_Handler);
/* 删除开始任务 */
vTaskDelete(StartTask_Handler);
/* 退出临界区 */
taskEXIT_CRITICAL();
}
/* Message Buffers 接收任务函数 */
void Message_Buffers_Receive_task(void *pvParameters)
{
    uint8_t ucRxData[ 5 ] = {'0', '0', '0', '0', '\0'}, ucRxData1[ 15 ] =
    {'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '\0'};
    size_t xReceivedBytes;
    const TickType_t xBlockTime = pdMS_TO_TICKS( 20 );
    while(1)
    {
        /* 进入临界区 */
        taskENTER_CRITICAL();
        /* 从 AT_xMessageBuffer 接收数据 */
        xReceivedBytes = xMessageBufferReceive( AT_xMessageBuffer, ( void * ) ucRxData,
            sizeof( ucRxData ), xBlockTime );
        if( xReceivedBytes > 0 )
        {
            printf("%s\r\n", ucRxData);
        }
        /* 退出临界区 */
        taskEXIT_CRITICAL();

        /* 进入临界区 */

```

```
taskENTER_CRITICAL();
/* 从 AT_xMessageBuffer 接收数据 */
xReceivedBytes = xMessageBufferReceive( AT_xMessageBuffer, ( void * ) ucRxData1,
sizeof( ucRxData1 ), xBlockTime );
if( xReceivedBytes > 0 )
{
    printf("%s\r\n", ucRxData1);
}
/* 退出临界区 */
taskEXIT_CRITICAL();

}
}
/* Message Buffers 发送任务函数 */
void Message_Buffers_Send_task(void *pvParameters)
{
    size_t xBytesSent;
    uint8_t ucArrayToSend[] = { 'A', 'T', '3', '2' };
    char *pcStringToSend = "Artery Tek MCU";
    while(1)
    {
        /* 进入临界区 */
        taskENTER_CRITICAL();
        /* 发送 ucArrayToSend[]中的数据 */
        xBytesSent = xMessageBufferSend( AT_xMessageBuffer, ( void * ) ucArrayToSend,
sizeof( ucArrayToSend ), 0 );
        if( xBytesSent != sizeof( ucArrayToSend ) )
        {
            printf("#1:%d data written in.\r\n", xBytesSent);
        }
        /* 退出临界区 */
        taskEXIT_CRITICAL();

        /* 进入临界区 */
        taskENTER_CRITICAL();
        /* 发送 pcStringToSend 字符串内容 */
        xBytesSent = xMessageBufferSend( AT_xMessageBuffer, ( void * ) pcStringToSend,
strlen( pcStringToSend ), 0 );
        if( xBytesSent != strlen( pcStringToSend ) )
        {
            printf("#2:%d data written in.\r\n", xBytesSent);
        }
        /* 退出临界区 */
        taskEXIT_CRITICAL();
        vTaskDelay(1000);
    }
}
```

```

    }
}

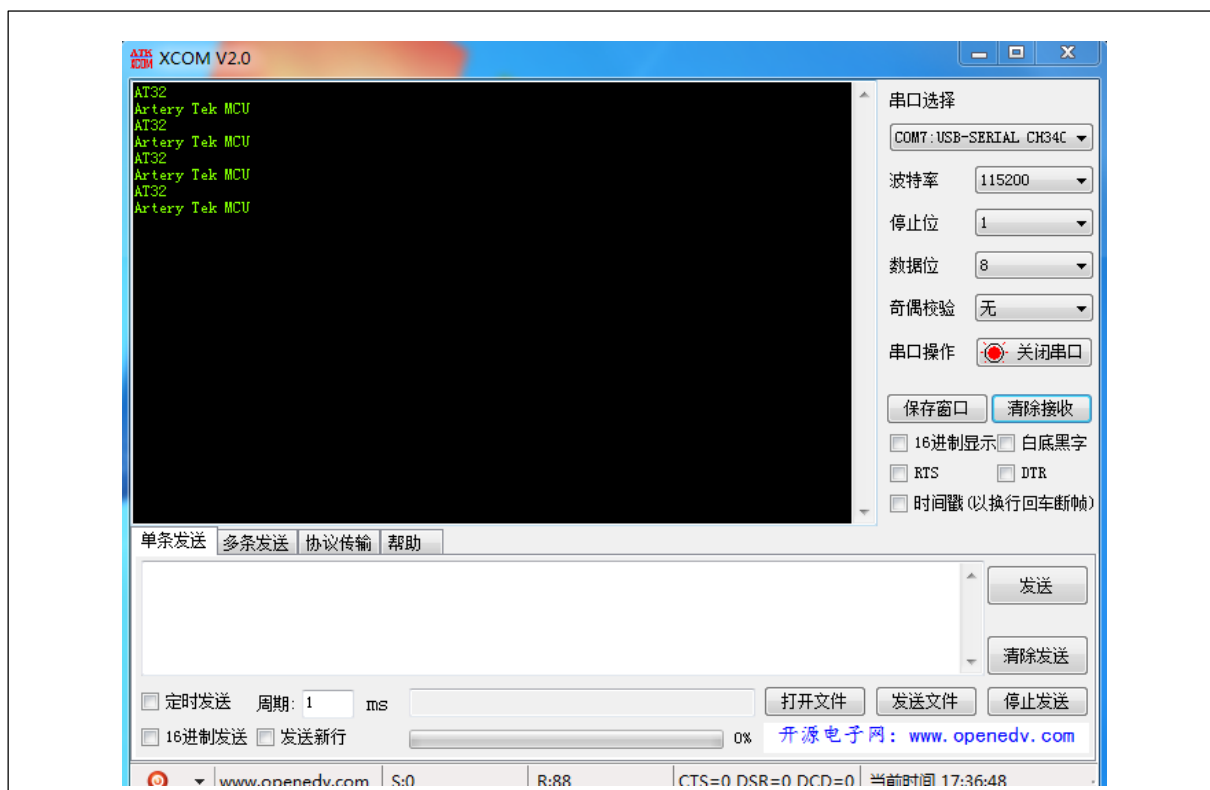
/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次任务信息 */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/-----*\r\n");
            printf("Task    Status    priority    Remaining_Stack    Num\r\n");
            vTaskList((char *)&buff);
            printf("%s\r\n", buff);
            printf("/-----*\r\n");
            printf("Task        Runing_Num        Usage_Rate\r\n");
            vTaskGetRunTimeStats((char *)&buff);
            printf("%s\r\n", buff);
        }
        vTaskDelay(10);
    }
}

```

以上就是消息缓存例程的源程序。程序开始创建一个消息缓存对象，之后会有一个任务向消息缓存中先后发送长度为4和长度为14的消息，然后会有任务从消息缓存中接受任务并打印出来。

编译并下载程序到目标板，运行效果如下：

图 39. 消息缓存例程演示



可以看到接受到的消息和程序设计相符。

15 FreeRTOS 任务通知

本节来看看FreeRTOS的任务通知功能。任务通知功能在某些特定情况下可以替代信号量、事件标志组等，并且效率更高。所以掌握任务通知对于提高系统效率是很有帮组的。

15.1 任务通知介绍

任务通知在FreeRTOS中是一个可选的功能，如果想使用这个功能的话要将宏configUSE_TASK_NOTIFICATIONS定义为1。FreeRTOS的每一个任务都有一个31bit的通知值，任务控制块中的ulNotifiedValue就是这个通知值，任务通知是一个事件，假如某个任务通知的接收任务因为等待任务通知而阻塞的话，向这个接收任务发送任务通知就会解除接收任务阻塞态。任务通知可以通过以下几种修改ulNotifiedValue的方式来实现相应的功能：

- ① 如果上次通知值还未被处理，不覆盖接受任务的通知值。
- ② 任何条件下都覆盖接受任务的通知值。
- ③ 更新接受任务的通知值的一个或多个BIT。
- ④ 增加接受任务的通知值。

合理、灵活的使用上面这些更改任务通知值可以在一些场合下替代队列、二值信号量、计数型信号量、事件标志组，并且在效率上也有提高。据FreeRTOS官方实测数据，在使用任务通知实现二值信号量功能的时候，解除任务阻塞的时间比直接使用二值信号量要快45%，并且使用的RAM空间更少。

任务通知虽然可以提高效率并减少RAM的使用，但是任务通知也有使用上的限制：

- ① 任务通知只能有一个接收任务，其实大多数应用都是这种情况。
- ② 接收任务可以因为还没接收到通知而进入阻塞态，但是发送任务不会因为发送失败而阻塞。

15.2 任务通知 API

下面来看看任务通知相关的API函数：

表 55. 任务通知 API

任务通知 API 函数	
API	描述
xTaskNotify()	发送通知，带有通知值但不保留接收任务原通知值
xTaskNotifyFromISR()	发送通知，带有通知值但不保留接收任务原通知值（中断级）
xTaskNotifyAndQuery()	发送通知，带有通知值并保留接收任务原通知值
xTaskNotifyAndQueryFromISR()	发送通知，带有通知值并保留接收任务原通知值（中断级）
xTaskNotifyGive()	发送通知，不带通知值并且不保留原通知值，会让接收任务的通知值加一
vTaskNotifyGiveFromISR()	发送通知，不带通知值并且不保留原通知值，会让接收任务的通知值加一（中断级）
xTaskNotifyStateClear()	清除任务的任务通知状态
ulTaskNotifyTake()	获取任务通知，可以设置在退出时将任务通知值减一或清零（分别用于二值信号量和计数型信号量）

xTaskNotifyWait()	获取任务通知，比 ulTaskNotifyTake()更为强大，可作为事件标志组时的任务接收函数
--------------------------	--

以上是FreeRTOS任务通知相关API函数，具体参数细节可自行查阅官方指导手册。

15.3 例程介绍

工程名: **16Task_Notify_FreeRTOS**

程序源码:

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* Task1 任务优先级*/
#define Task1_TASK_PRIO      2
/* Task1 任务堆栈大小 */
#define Task1_STK_SIZE      128
/* Task1 任务任务句柄 */
TaskHandle_t AT_Task1_Handler;
/* Task1 任务入口函数 */
void AT_Task1(void *pvParameters);

/* Task2 任务优先级*/
#define Task2_TASK_PRIO      2
/* Task2 任务堆栈大小 */
#define Task2_STK_SIZE      128
/* Task2 任务任务句柄 */
TaskHandle_t AT_Task2_Handler;
/* Task2 任务入口函数 */
void AT_Task2(void *pvParameters);

/* Task3 任务优先级*/
#define Task3_TASK_PRIO      2
/* Task3 任务堆栈大小 */
#define Task3_STK_SIZE      128
/* Task3 任务任务句柄 */
TaskHandle_t AT_Task3_Handler;
/* Task3 任务入口函数 */
void AT_Task3(void *pvParameters);

/* 调试任务优先级 */
#define Debug_TASK_PRIO      3
/* 调试任务堆栈大小 */
#define Debug_STK_SIZE      512
/* 调试任务任务句柄 */
```

```

TaskHandle_t DebugTask_Handler;
/* 调试任务入口函数 */
void debug_task(void *pvParameters);

#define TX_BIT 0x01
#define RX_BIT 0x02
static int Counter;
static uint32_t ulNotifiedValue;

int main(void)
{
    nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);

    system_clock_config();

    at32_led_init(LED2);
    at32_led_init(LED3);
    at32_led_init(LED4);

    at32_button_init();

    /* init usart1 */
    uart_print_init(115200);

    /* 进入临界区 */
    taskENTER_CRITICAL();

    /* 初始化定时器 */
    TIMER_Init();

    /* 创建 Task1 任务 */
    xTaskCreate((TaskFunction_t)AT_Task1,
                (const char* )"Task1",
                (uint16_t )Task1_STK_SIZE,
                (void* )NULL,
                (UBaseType_t )Task1_TASK_PRIO,
                (TaskHandle_t* )&AT_Task1_Handler);

    /* 创建 Task2 任务 */
    xTaskCreate((TaskFunction_t)AT_Task2,
                (const char* )"Task2",
                (uint16_t )Task2_STK_SIZE,
                (void* )NULL,
                (UBaseType_t )Task2_TASK_PRIO,
                (TaskHandle_t* )&AT_Task2_Handler);

    /* 创建 Task3 任务 */

```

```

xTaskCreate((TaskFunction_t)AT_Task3,
            (const char*   )"Task3",
            (uint16_t      )Task3_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t    )Task3_TASK_PRIO,
            (TaskHandle_t*  )&AT_Task3_Handler);
/* 创建调试任务 */
xTaskCreate((TaskFunction_t)debug_task,
            (const char*   )"Debug_task",
            (uint16_t      )Debug_STK_SIZE,
            (void*         )NULL,
            (UBaseType_t    )Debug_TASK_PRIO,
            (TaskHandle_t*  )&DebugTask_Handler);
/* 开启任务调度器 */
vTaskStartScheduler();
}

/* Task1 任务函数 */
void AT_Task1(void *pvParameters)
{
    while(1)
    {
        vTaskDelay(50);
        /* 给 AT_Task2_Handler 任务发送任务通知 */
        printf("Send a notification to AT_Task2.\r\n");
        xTaskNotifyGive( AT_Task2_Handler );
        at32_led_toggle(LED4);
        vTaskDelay(1000);
        /* 等待接受 AT_Task2_Handler 的任务通知 */
        ulTaskNotifyTake( pdTRUE, portMAX_DELAY );
        printf("Received the AT_Task2's task notification.\r\n");
    }
}

/* Task2 任务函数 */
void AT_Task2(void *pvParameters)
{
    while(1)
    {
        vTaskDelay(100);
        /* 等待接受 AT_Task1_Handler 的任务通知 */
        ulTaskNotifyTake( pdTRUE, portMAX_DELAY );
        printf("Received the AT_Task1's task notification.\r\n");
        vTaskDelay(2000);
        /* 给 AT_Task1_Handler 任务发送任务通知 */
        printf("Send a notification to AT_Task1.\r\n");
    }
}

```

```

    xTaskNotifyGive( AT_Task1_Handler );
    at32_led_toggle(LED4);
}
}
/* Task3 任务函数 */
void AT_Task3(void *pvParameters)
{
    BaseType_t xResult;
    while(1)
    {
        /* 等待接受任务通知 */
        xResult = xTaskNotifyWait( pdFALSE, 0xffffffff, &ulNotifiedValue, portMAX_DELAY );
        if( xResult == pdPASS )
        {
            /* 接受到了任务通知 */
            if( ( ulNotifiedValue & TX_BIT ) != 0 )
            {
                at32_led_toggle(LED2);
                printf("The TX ISR has set a bit.\r\n");
            }
            if( ( ulNotifiedValue & RX_BIT ) != 0 )
            {
                at32_led_toggle(LED3);
                printf("The RX ISR has set a bit.\r\n");
            }
        }
    }
}
/* 硬件定时器 3 中断函数 */
void TMR3_GLOBAL_IRQHandler(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    if(tmr_flag_get(TMR3,TMR_OVF_FLAG)==SET)
    {
        Counter++;
        if( (Counter % 2) == 0 )
        {
            printf("Notify the task by setting the RX_BIT in the task's notification value.\r\n");
            xTaskNotifyFromISR( AT_Task3_Handler, RX_BIT, eSetBits, &xHigherPriorityTaskWoken );
        }
        else
        {
            printf("Notify the task by setting the TX_BIT in the task's notification value.\r\n");
            xTaskNotifyFromISR( AT_Task3_Handler, TX_BIT, eSetBits, &xHigherPriorityTaskWoken );
        }
    }
}

```

```

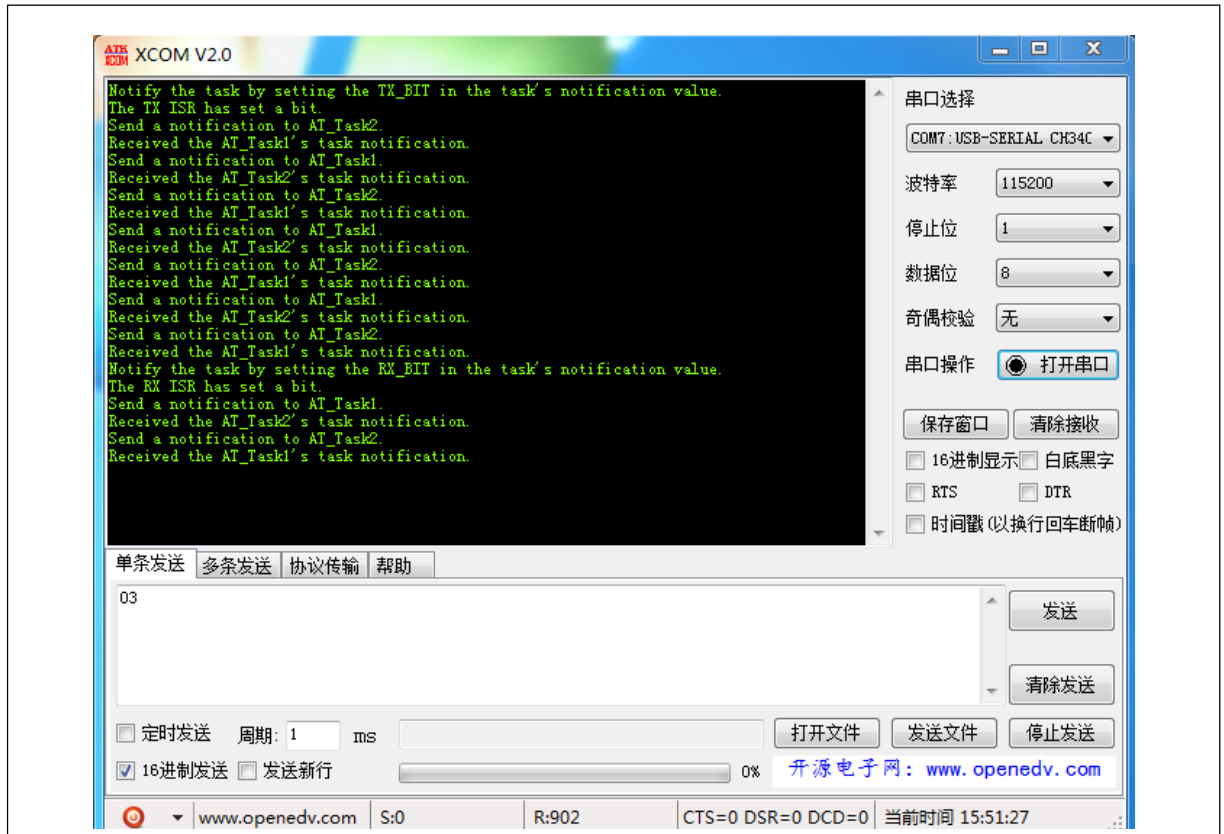
    tmr_flag_clear(TMR3,TMR_OVF_FLAG);
    /* 判断是否需要任务切换 */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
}
/* 调试任务函数 */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* 按下按键打印一次调试信息 */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/*-----*/\r\n");
            printf("Task          Status          priority      Remaining_Stack      Num\r\n");
            vTaskList((char *)&buff);
            printf("%s\r\n",buff);
            printf("/*-----*/\r\n");
            printf("Task      Runing_Num      Usage_Rate\r\n");
            vTaskGetRunTimeStats((char *)&buff);
            printf("%s\r\n",buff);
        }
        vTaskDelay(10);
    }
}

```

以上是任务通知例程的程序源码，程序中一共创建了三个任务，其中任务1和任务2相互发送通知和接受通知，任务3则负责接受硬件定时器3的中断处理函数中发送的通知。整个程序运行起来会有打印信息提示，可查看打印内容观察系统的运行流程。该例程展示了任务通知的使用，模拟了FreeRTOS的信号量和事件标志组的功能。具体实现细节请查看对应工程源码。

编译并下载程序到目标板，运行效果如下：

图 40. 任务通知例程演示



从打印信息看来，程序运行流程符合程序设计。本节就介绍到这里，具体细节可对照程序查看或翻阅FreeRTOS官方手册。

16 FreeRTOS 综合 Demo 演示

16.1 Demo 功能简介

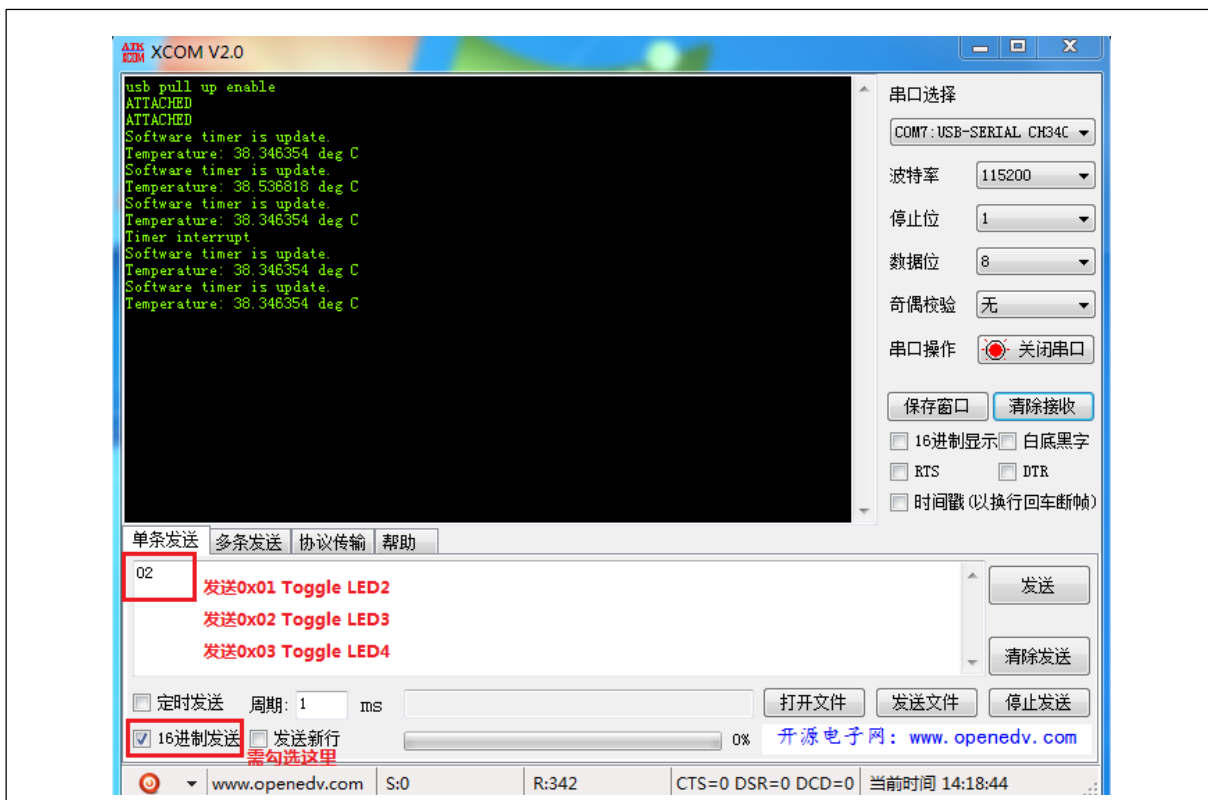
本节将介绍一个综合一点的例程，该例程使用FreeRTOS系统配合AT32 MCU实现了如下几个功能：

- ① AT32系列MCU上使用上位机串口工具发送特定数据控制板载的LED灯；
- ② 使用硬件定时器产生硬件中断发送消息给任务；
- ③ 使用FreeRTOS的软件定时器定时使用ADC采集MCU上的温度传感器数值；
- ④ AT32F403系列MCU上还实现了USB Keyboard功能，即按下USER按键程序通过USB传输按键值模拟键盘功能。

16.2 例程演示

1.首先看看如何通过上位机发送数据控制板载的LED灯：

图 41. 综合例程演示 1



如上图所示，当发送0x01时，板载的LED2会翻转；当发送0x02时，板载的LED3会翻转；当发送0x03时，板载的LED4会翻转。为了实现此功能，程序建立了两个任务，一个任务负责处理串口接收到的数据，然后根据相应的数据发送消息到消息队列；另外一个任务负责接受接受消息队列中的消息，然后判断消息后作出相应的动作，即翻转对应的LED灯。

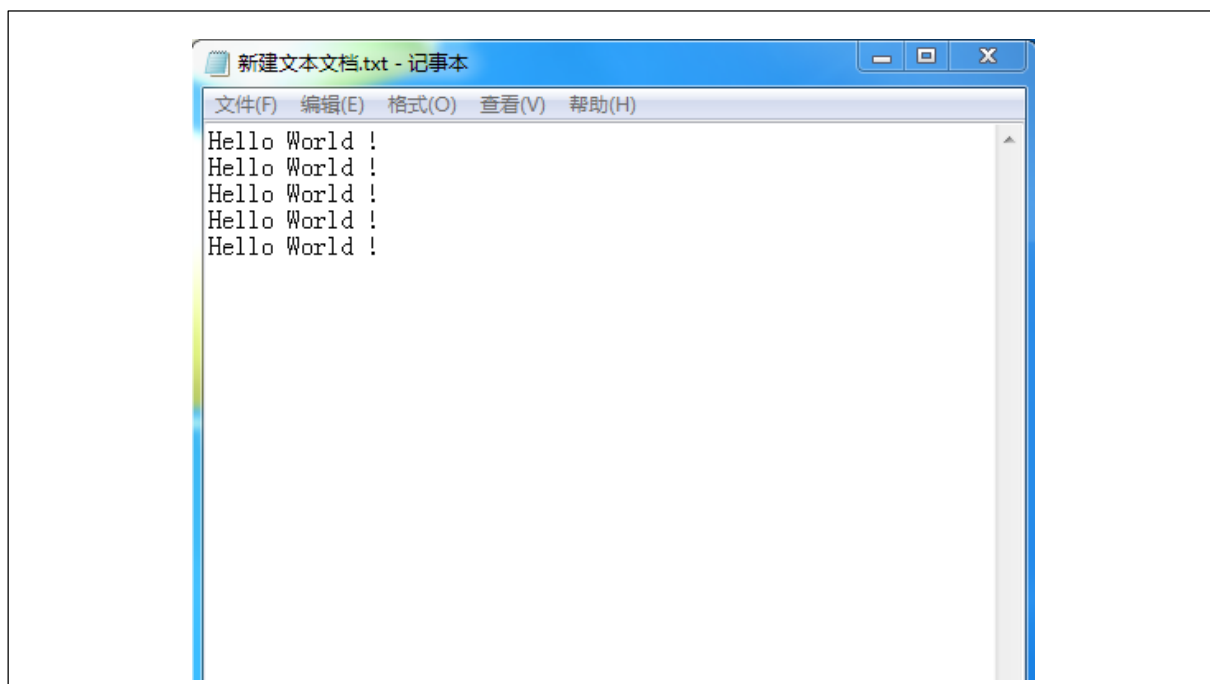
2.再来看看例程是如何实现采集MCU的温度传感器数值的：

为了实现定时采集温度传感器数值的功能，使用了FreeRTOS的软件定时器组功能。程序初始化了一个软件定时器，并且定义周期为2000个系统时钟节拍，即每2000个系统节拍后采集一次温度传感器的数值并打印出来。打印现象可在上图中看到。温度的采集使用AT32 MCU的ADC。

3.最后看看USB Keyboard功能是如何实现的：

USB Keyboard功能是基于403系列MCU的USB Device实现的，在413/415系列MCU上是不支持此功能的。在使用时需保证USB和PC之间的连线正常，此例程配套的板子上有USB接口，使用相应的线缆链接即可。链接后打开一个文本文档或者Word均可，然后按下USER按键，可以看到鼠标指向的文本文档上会出现Hello World！字样。如下图：

图 42. 综合例程演示 2



以上是综合Demo的相关内容，程序实现细节请参考对应的工程。：

17 版本历史

表 56. 文档版本历史

日期	版本	变更
2021.12.16	2.0.0	最初版本

重要通知 - 请仔细阅读

买方自行负责对本文所述雅特力产品和服务的选择和使用，雅特力概不承担与选择或使用本文所述雅特力产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为雅特力授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在雅特力的销售条款中另有说明，否则，雅特力对雅特力产品的使用和/或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途（及其依据任何司法管辖区的法律的对应情况），或侵犯任何专利、版权或其他知识产权的默示保证。

雅特力产品并非设计或专门用于下列用途的产品：（A）对安全性有特别要求的应用，例如：生命支持、主动植入设备或对产品功能安全有要求的系统；（B）航空应用；（C）航天应用或航天环境；（D）武器，且/或（E）其他可能导致人身伤害、死亡及财产损害的应用。如果采购商擅自将其用于前述应用，即使采购商向雅特力发出了书面通知，风险及法律责任仍将由采购商单独承担，且采购商应独立负责在前述应用中满足所有法律和法规要求。

经销的雅特力产品如有不同于本文档中提出的声明和/或技术特点的规定，将立即导致雅特力针对本文所述雅特力产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大雅特力的任何责任。

© 2021 雅特力科技 保留所有权利