

FreeRTOS on AT32 MCU

Introduction

This application note introduces how to use FreeRTOS on AT32Fxx MCUs. The FreeRTOS is an open-source real-time operating system, which is widely used in various embedded applications. This application note details the porting of FreeRTOS, FreeRTOS kernel introduction and related demos, and provides the corresponding source program to help beginners have a better understanding of the AT32Fxx MCUs and FreeRTOS.

This application note also introduces how to make use of FreeRTOS features and how to use the FreeRTOS together with AT32 MCU peripherals to achieve the desired functions.

For better understanding of the FreeRTOS, please refer to this application note together with the corresponding programs and FreeRTOS quick start guide from the official website.

Note: Example codes in this application note are developed on the basis of BSP_V2.x.x provided by Artery. For other versions of BSP, please pay attention to the differences in usage.

Applicable products:

Part number	All series
-------------	------------

Contents

1	FreeRTOS	9
2	How to port FreeRTOS to AT32 MCU	11
2.1	Port FreeRTOS.....	11
2.2	Routine	13
3	FreeRTOS debugging	17
3.1	System configuration	17
3.2	Routine	18
4	FreeRTOS interrupt priority management.....	20
4.1	AT32 MCU interrupt configuration	20
4.2	FreeRTOS interrupt configuration.....	21
4.3	Differences between interrupt priority and task priority	22
4.4	Critical code region.....	22
4.5	Routine	23
5	FreeRTOS task management.....	28
5.1	Bare metal vs. RTOS.....	28
5.2	FreeRTOS task states	29
5.3	FreeRTOS idle task	30
5.4	FreeRTOS task related functions	30
5.5	Routine	33
6	FreeRTOS task scheduling	38
6.1	Cooperative scheduling	38
6.2	Preemptive scheduling	38
6.3	Time-slicing scheduling	39
6.4	Routine	41
7	FreeRTOS queue.....	46

7.1	Introduction.....	46
7.2	Queue API	47
7.3	Routine	50
8	FreeRTOS semaphore	57
8.1	Introduction.....	57
8.2	Binary semaphore	57
8.2.1	Introduction.....	57
8.2.2	Binary semaphore API	58
8.2.3	Routine	60
8.3	Counting semaphore	65
8.3.1	Introduction.....	65
8.3.2	Counting semaphore API	65
8.3.3	Routine	67
8.4	Mutex	72
8.4.1	Priority inversion.....	72
8.4.2	Introduction.....	73
8.4.3	Mutex API	74
8.4.4	Routine	76
8.5	Recursive mutex.....	81
8.5.1	Introduction.....	81
8.5.2	Recursive mutex API	81
8.5.3	Routine	83
9	FreeRTOS event groups/flags	88
9.1	Introduction.....	88
9.2	Event group API	89
9.3	Routine	90
10	FreeRTOS software timers.....	97
10.1	Introduction.....	97
10.2	Software timer API.....	99
10.3	Routine	102

11	FreeRTOS low power support	107
	11.1 Tickless idle mode	107
	11.2 Routine	109
12	FreeRTOS memory management	115
	12.1 Heap_1	115
	12.2 Heap_2	116
	12.3 Heap_3	120
	12.4 Heap_4	121
	12.5 Heap_5	126
13	FreeRTOS stream buffer	130
	13.1 Introduction	130
	13.2 Stream buffer API	130
	13.3 Routine	132
14	FreeRTOS message buffer	137
	14.1 Introduction	137
	14.2 Message buffer API	137
	14.3 Routine	139
15	FreeRTOS task notifications	145
	15.1 Introduction	145
	15.2 Task notification API	145
	15.3 Routine	146
16	FreeRTOS Demos	152
	16.1 Introduction	152
	16.2 Demo routines	152
17	Revision history	154

List of tables

Table 1. Interrupt priority grouping.....	20
Table 2. Critical code region API.....	23
Table 3. xTaskCreate().....	31
Table 4. xTaskCreateStatic().....	31
Table 5. vTaskDelete()	32
Table 6. vTaskSuspend()	32
Table 7. vTaskResume()	32
Table 8. vTaskDelay().....	33
Table 9. Queue API reference.....	47
Table 10. xQueueCreate()	48
Table 11. xQueueSend()	48
Table 12. xQueueSendFromISR() (within interrupt service routine)	49
Table 13. xQueueReceive()	49
Table 14. xQueueReceiveFromISR().....	50
Table 15. Binary semaphore API	58
Table 16. xSemaphoreCreateBinary ()	59
Table 17. vSemaphoreDelete ().....	59
Table 18. xSemaphoreGive ()	59
Table 19. xSemaphoreTake ().....	60
Table 20. Counting semaphore API	65
Table 21. xSemaphoreCreateCounting ()	66
Table 22. vSemaphoreDelete ().....	66
Table 23. xSemaphoreGive ()	66
Table 24. xSemaphoreTake ().....	67
Table 25. uxSemaphoreGetCount ().....	67
Table 26. Mutex API	74
Table 27. xSemaphoreCreateMutex().....	74
Table 28. vSemaphoreDelete ().....	74
Table 29. xSemaphoreGive ()	75
Table 30. xSemaphoreTake ().....	75
Table 31. uxSemaphoreGetCount ().....	75
Table 32. Recursive mutex API.....	81
Table 33. xSemaphoreCreateRecursiveMutex()	82
Table 34. xSemaphoreGiveRecursive().....	82

Table 35. xSemaphoreTakeRecursive()	82
Table 36. Event group API	89
Table 37. xEventGroupCreate()	89
Table 38. xEventGroupSetBits()	89
Table 39. xEventGroupWaitBits()	90
Table 40. Software timer API	99
Table 41. xTimerCreate()	99
Table 42. xTimerStart()	100
Table 43. xTimerStop()	100
Table 44. pcTimerGetName()	101
Table 45. pvTimerGetTimerID()	101
Table 46. vTimerSetTimerID()	101
Table 47. Stream buffer API	130
Table 48. xStreamBufferCreate()	131
Table 49. xStreamBufferReceive ()	131
Table 50. xStreamBufferSend()	132
Table 51. Message buffer API	137
Table 52. xMessageBufferCreate ()	138
Table 53. xMessageBufferReceive ()	138
Table 54. xStreamBufferSend()	139
Table 55. Task notification API	145
Table 56. Document revision history	154

List of figures

Figure 1. FreeRTOS official website	11
Figure 2. FreeRTOS source directory.....	11
Figure 3. Project directory.....	12
Figure 4. Add header file.....	12
Figure 5. Porting routine demonstration	16
Figure 6. Configuration parameter debugging.....	17
Figure 7. Debugging routine demonstration	19
Figure 8. AT32 NVIC configuration	21
Figure 9. AT32 peripheral interrupt configuration	21
Figure 10. FreeRTOS interrupt configuraiton	21
Figure 11. FreeRTOS interrupt management routine.....	27
Figure 12. Bare-metal system operation process.....	28
Figure 13. RTOS operation process	29
Figure 14. Task state transitions	30
Figure 15. Task management routine	37
Figure 16. FreeRTOS Co-routines.....	38
Figure 17. Time-slicing scheduling	40
Figure 18. Time-slicing scheduling routine	45
Figure 19. Queue workflow	46
Figure 20. Queue routine.....	56
Figure 21. Queue routine (LED toggle)	56
Figure 22. Binary semaphore block diagram 1.....	58
Figure 23. Binary semaphore block diagram 2.....	58
Figure 24. Binary semaphore block diagram 3.....	58
Figure 25. Binary semaphore routine	64
Figure 26. Binary semaphore routine	64
Figure 27. Counting semaphore routine	71
Figure 28. Priority inversion	72
Figure 29. Solve priority inversion with mutexes	73
Figure 30. Mutex routine	81
Figure 31. Recursive mutex routine.....	87
Figure 32. Block diagram of event group.....	88
Figure 33. Event group routine	96
Figure 34. One-shot timer VS auto-reload timer	97

Figure 35. Software timer daemon task.....	98
Figure 36. Software timer routine	106
Figure 37. Low power mode routine	114
Figure 38. Stream buffer routine.....	136
Figure 39. Message buffer routine.....	144
Figure 40. Task notification routine	151
Figure 41. Demo 1	152
Figure 42. Demo 2	153

1 FreeRTOS

Introduction

The real-time operating system (RTOS) is widely used in embedded devices, which helps achieve more rational and efficient use of CPU, simplify application software, shorten system development time and improve system real-time efficiency and reliability.

FreeRTOS is a lightweight real-time operating system and designed with features including task management, time management, semaphore, message queue, memory management, recorder, software timer and co-routines to meet requirements of lightweight systems. Because RTOS requires a certain amount of system resources (especially RAM resources), only a few real-time operating systems such as UCOSII/III, emboss, RTT and FreeRTOS can run on the MCU with small RAM. Compared to commercial operating systems such as UCOSII/III and emboss, FreeRTOS is a free open-source operating system and featured with open source code, portability, scalability and flexible scheduling, so that it can be ported and easily used on MCUs.

Features

FreeRTOS has the following features:

- Kernel configurable by users
- Support cross platform
- High-level integrity of trusted code
- Small, simple and easy-to-use object code
- Conform to the MISRA-C coding standard guidelines
- Powerful execution tracking capabilities
- Stack overflow detection
- No limit to the number and priority of tasks
- Multiple tasks can be assigned the same priority
- Queue, binary semaphore, counting semaphore, recursive communication and synchronous tasks
- Priority inheritance
- Open-source and freely available source code

System function

As a lightweight operating system, the FreeRTOS is designed with features including task management, time management, semaphore, message queue, memory management, recorder, software timer and co-routines to meet requirements of small systems. The FreeRTOS kernel supports priority scheduling algorithm to assign tasks with different priorities according to their importance, and CPU always executes the task that is ready and with the highest priority. The FreeRTOS also supports round-robin scheduling algorithm, which allows different tasks of the same priority to share the system resources if no task of a higher priority is ready.

The FreeRTOS kernel can be configured as preemptive or non-preemptive as required. If a preemptive kernel is configured, the high-priority task in ready state always gets the CPU before a low-priority task, which improves the real-time efficiency of system. If a non-preemptive kernel is configured, the high-priority task in ready state can only be executed until the current task actively terminates, which improves the operating efficiency of system.

Summary

The FreeRTOS is one of the few embedded operating systems that are real-time efficient, open source, reliable, ease-of-use and cross-platform available. It supports more than 30 hardware platforms including X86, Xilinx and Altera, which makes it valuable to more applications.

Version

The latest version: FreeRTOS V10.2.1 2019-05-13

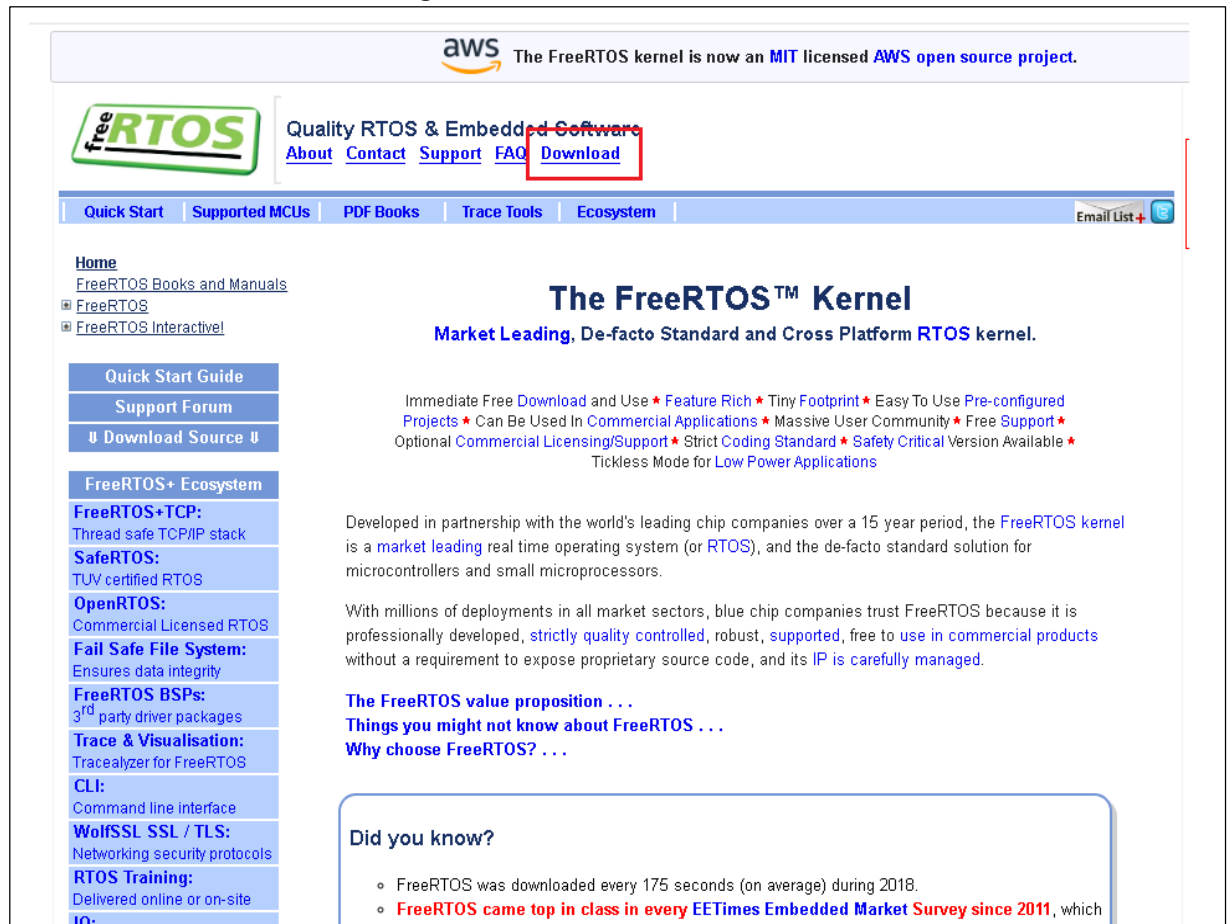
2 How to port FreeRTOS to AT32 MCU

This section introduces how to port the FreeRTOS package to AT32Fxx MCUs (AT-START Board).

2.1 Port FreeRTOS

Step 1: Download source package from FreeRTOS official website (www.freertos.org), as shown in Figure 1.

Figure 1. FreeRTOS official website



Click on “Download” to download the latest version of FreeRTOS source package (which is V10.2.1 in this application note).

Step 2: Copy the *middlewares* folder in FreeRTOS source package to the project, and add it to the project directory, as shown in Figure 2 and Figure 3.

Figure 2. FreeRTOS source directory

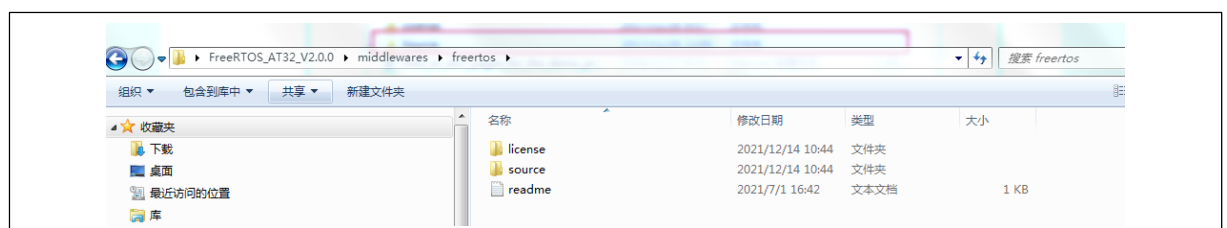
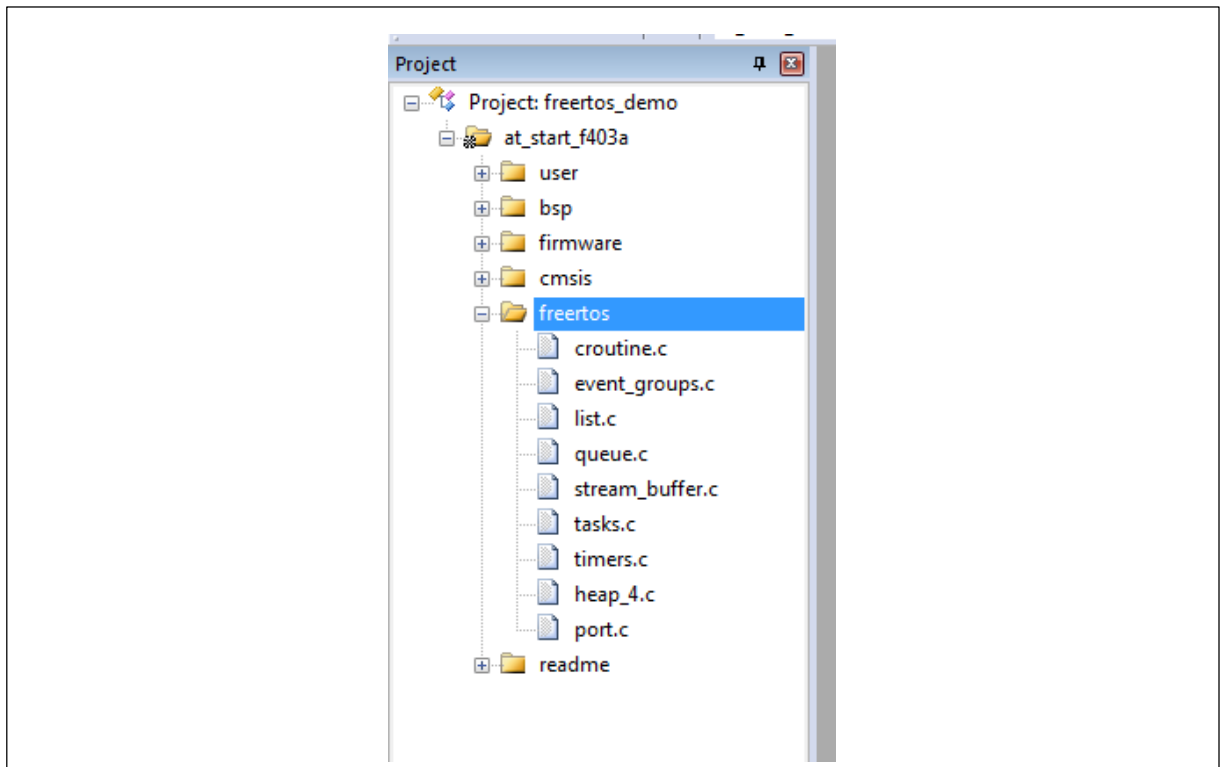


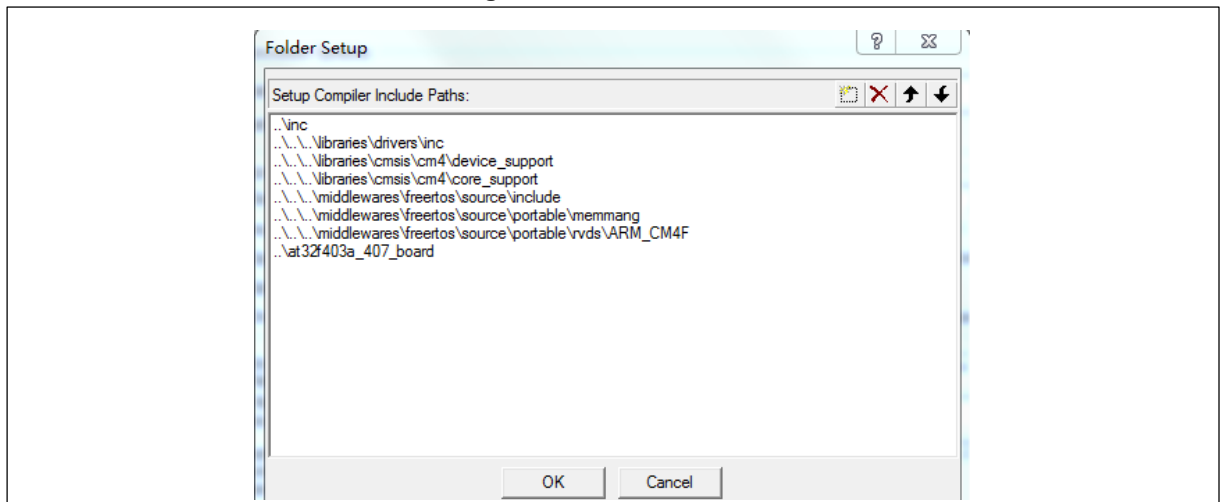
Figure 3. Project directory



Files in the *freertos* folder are under the source directory, except for *heap_4.c* in Source\portable\MemMang and *port.c* in Source\portable\RVDS\ARM_CM4F.

Step 3: Add the header file path the MDK, as shown in Figure 4.

Figure 4. Add header file



Step 4: Compile and then an error (*FreeRTOSConfig.h* not found) is reported. This header file is a FreeRTOS system configuration file, which can be created by users or found from the demo in the source package. In this example, a reference routine is used, or you can create a *FreeRTOSConfig.h* file according to the system requirements.

Go through the above 4 steps and then compile again, and no error is reported. Then, compile a task scheduling program to check whether the porting is successful.

2.2 Routine

Project name: Porting_FreeRTOS_AT32

Program source code:

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

/* start task priority */
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE       128
/* start task handle */
TaskHandle_t StartTask_Handler;
/* start task entry function */
void start_task(void *pvParameters);

/* LED0 task priority */
#define LED0_TASK_PRIO      4
/* LED0 task stack size */
#define LED0_STK_SIZE       128
/* LED0 task handle */
TaskHandle_t LED0Task_Handler;
/* LED0 task entry function */
void led0_task(void *pvParameters);

/* LED1 task priority */
#define LED1_TASK_PRIO      3
/* LED1 task stack size */
#define LED1_STK_SIZE       128
/* LED1 task handle */
TaskHandle_t LED1Task_Handler;
/* LED1 task entry function */
void led1_task(void *pvParameters);

/* floating-point calculation task priority */
#define FLOAT_TASK_PRIO      2
/* floating-point calculation task stack size */
#define FLOAT_STK_SIZE       256
/* floating-point calculation task handle */
TaskHandle_t FLOATTask_Handler;
/* floating-point calculation task entry function */
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);

/* create start task */
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);

/* enable task scheduler */
vTaskStartScheduler();
}

/* start task function */
void start_task(void *pvParameters)
{
    /* enter critical zone */
    taskENTER_CRITICAL();
    /* create LED0 task */
    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);
    /* create LED1 task */
    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);
    /* create floating-point calculation task */
    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);

/* delete start task */
vTaskDelete(StartTask_Handler);
/* exit critical zone */
taskEXIT_CRITICAL();
}

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

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

/* floating-point calculation task function */
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);
    }
}

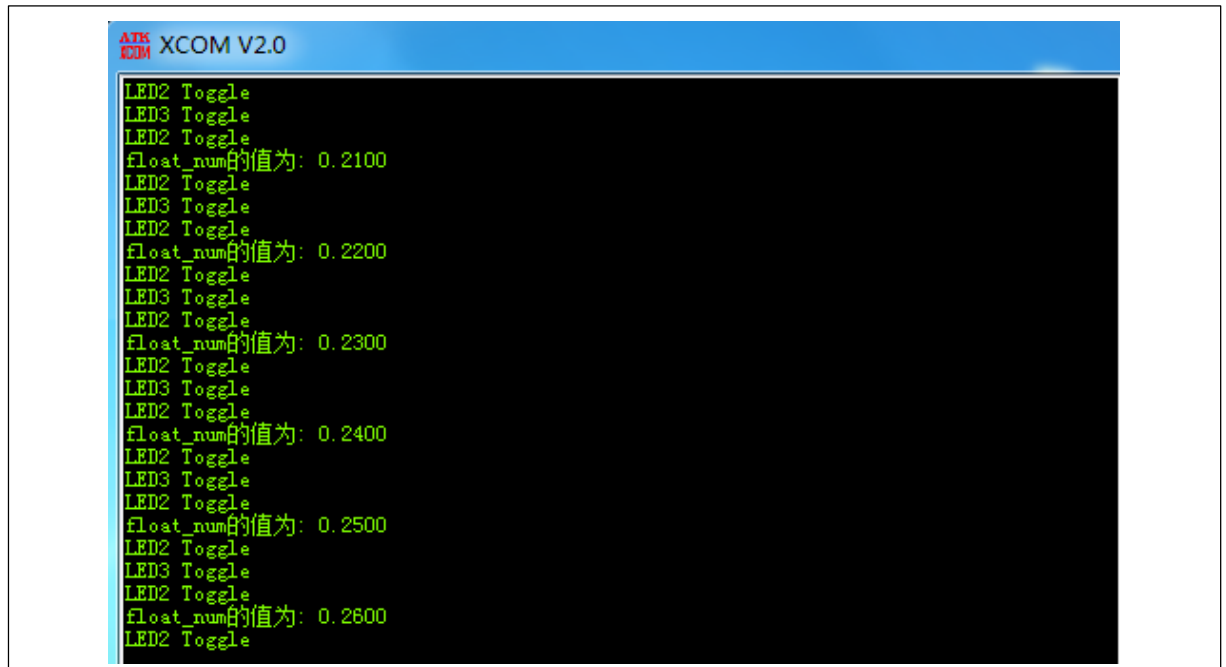
```

In the porting routine source program, three tasks are created, i.e., LED0, LED1 and floating-point calculation. Different from ordinary programs (without RTOS), each task function has an infinite

loop, and these three tasks are scheduled by the task scheduler.

Compile and download to the target board to view the execution. The LEDs on the target board toggles alternately, indicating that tasks are running alternately; or print out the result in the serial port assistant, as shown in Figure 5.

Figure 5. Porting routine demonstration



```
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
```

Users can port the FreeRTOS to AT32 MCUs by the above means, and the project source code in this section is also used in the following contents.

3 FreeRTOS debugging

This section introduces the debugging of FreeRTOS. The FreeRTOS kernel has certain API functions that can be used to get the task running status in the current system. This debugging method also helps find out problems in the program development stage.

3.1 System configuration

Configure the system before FreeRTOS debugging. Firstly, open the corresponding macro, which is as shown in Figure 6.

Figure 6. Configuration parameter debugging

```

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

```

Configuration parameters shown in Figure 6 are stored in the *FreeRTOSConfig.h* file. The “debug_timerTick” is the time base for the system running time statistics task, which is defined in the *timer.c* file. TIMER2 is enabled in the project, and overflow interrupt (every 1 ms) is enabled to allow the debug_timerTick to be incremented by 1.

The program code is as follows:

```

#include "timer.h"

volatile uint32_t debug_timerTick; //task running time statistics
extern uint32_t systemcoreclock;

void TIMER_Init(void)
{
    /* enable TMR2 clock */
    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);

    /* initialize 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);

    /* enable 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);
    }
}
```

3.2 Routine

Project name: 02Debug_FreeRTOS

Program source code:

```
/* debugging task priority */
#define Debug_TASK_PRIO      2
/* debugging task stack size */
#define Debug_STK_SIZE      512
/* debugging task handle */
TaskHandle_t DebugTask_Handler;
/* debugging task entry function */
void debug_task(void *pvParameters);

//create debugging task
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);

/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out the debugging information */
        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);
}
}

```

This routine is obtained by adding a task to the FreeRTOS routine code. Refer to the project for details.

The `vTaskList()` and `vTaskGetRunTimeStats()` functions are used in the routine. The `vTaskList()` is used to get the task name, task status, priority, remaining stack size and task serial number, store these contents into the array of incoming parameters, and then call the print function to output to the specified serial port. The `vTaskGetRunTimeStats()` is used to get the task name, running time statistics and usage rate, store these contents into the array of incoming parameters, and then call the print function to output to the specified serial port. Both functions provide the system status and promote development of program.

Compile and download the program to the target board, and then print out the result, as shown in Figure 7.

Figure 7. Debugging routine demonstration



The printed result displays all the information about the current system, including tasks, task status (X: running, R: ready, B: blocked, D: deleted, S: suspended), usage rate, task serial number (task creation sequence), and running time statistics (in terms of the hardware timer overflow).

4 FreeRTOS interrupt priority management

This section introduces FreeRTOS interrupt priority management, which helps users to have a better understanding of FreeRTOS and programming.

4.1 AT32 MCU interrupt configuration

Before configuring AT32 MCU interrupts, users need to have a brief understanding of NVIC (Nested vectored interrupt controller). AT32 MCUs incorporate a 32-bit ARM® Cortex®-M4 processor core, and the priority of each interrupt is set with 8 bits in the corresponding register, thereby giving up to $2^8 = 256$ priority levels, which can be adjusted according to the actual applications. For example, AT32Fxx MCUs use the first four bits [7:4] (MSB), and the other four bits (LSB) are set to zero, thereby giving up to $2^4 = 16$ priority levels. The priority group, preemption priority and subpriority of AT32Fxx MCUs are listed in Table 1.

Table 1. Interrupt priority grouping

Priority group	Preemption priority	Subpriority	Description of four bits (MSB)
NVIC_PriorityGroup_0	Level 0	Level 0-15	0 bit for preemption priority 4 bits for subpriority
NVIC_PriorityGroup_1	Level 0-1	Level 0-7	1 bit for preemption priority 3 bits for subpriority
NVIC_PriorityGroup_2	Level 0-3	Level 0-3	2 bits for preemption priority 2 bits for subpriority
NVIC_PriorityGroup_3	Level 0-7	Level 0-1	3 bits for preemption priority 1 bit for subpriority
NVIC_PriorityGroup_4	Level 0-15	Level 0	4 bits for preemption priority 0 bit for subpriority

AT32 MCUs support five priority groups, and the default priority group 0 after system power-on indicates that the four most significant bits are used for subpriority only.

Notes on the preemption priority and subpriority:

- 1) The interrupt of high preemption priority can be acknowledged during the execution of interrupt service program with low preemption priority, that is, interrupt nesting; in other words, the interrupt of high preemption priority preempts the execution of interrupt with low preemption priority.
- 2) If multiple interrupts have the same preemption priority, the one with the highest subpriority will be acknowledged.
- 3) On the premise of the same preemption priority, if the interrupt with low subpriority is being executed, the one with high subpriority has to wait until the low-subpriority interrupt execution is completed, which means that interrupt nesting is not supported.
- 4) Reset, NMI and Hard Fault have negative priority, which is higher than ordinary interrupt priority and cannot be configured.

It should be noted that system interrupts (such as PendSV, SVC and SysTick) may not definitely have priority over external interrupts (such as SPI and USART).

In this application note, the NVIC_PriorityGroup_4 is selected, which is also the officially recommended interrupt management method.

Call one library function to configure the NVIC, as shown in Figure 8.

Figure 8. AT32 NVIC configuration

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

This interrupt priority group has preemption priority only. Therefore, configure the subpriority as 0, as shown in Figure 9.

Figure 9. AT32 peripheral interrupt configuration

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

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

4.2 FreeRTOS interrupt configuration

FreeRTOS interrupt configuration is completed in *FreeRTOSConfig.h*, and the contents include:

Figure 10. FreeRTOS interrupt configuration

```
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使用*/
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

This macro defines the number of bits actually used in the AT32 MCU 8-bit interrupt priority register (4-bit priority field for all AT32 MCUs). Users can delete the conditional compilation and define “#define configPRIO_BITS 4” directly. The “__NVIC_PRIO_BITS” is defined in the AT32 standard library header file *at32f403a_407.h*. If this header file is included in the *FreeRTOSConfig.h* file, the conditional compilation will be executed.

2) #define configPRIO_BITS __NVIC_PRIO_BITS

```
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY 0x0f
```

This macro defines the priorities of SysTick and PendSV interrupts to be used for FreeRTOS. On the premise of priority group 4, this macro ranges from 0 to 15, that is, dedicated for preemption priority. It is configured as 0x0F, which means that both SysTick and PendSV

interrupts have the lowest priority (recommended in actual applications).

3) `#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 0x03`

This macro defines the highest priority interrupts managed by FreeRTOS, that is, allowing users to call the highest priority FreeRTOS API in the interrupt service routine. On the premise of priority group 4, configure the “`configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY`” as “0x03”, indicating that users can call FreeRTOS API functions in interrupts of preemption priority level 3-15 (not allowed in interrupts of preemption priority level 0-2).

4) `#define configKERNEL_INTERRUPT_PRIORITY`

An 8-bit priority level value (`configKERNEL_INTERRUPT_PRIORITY`) can be obtained after 4-bit offset of the “`configLIBRARY_LOWEST_INTERRUPT_PRIORITY`”. This 8-bit value can be actually assigned in the corresponding interrupt priority register. It is used for priority configuration of PendSV and SysTick interrupts. For example, the “`configLIBRARY_LOWEST_INTERRUPT_PRIORITY`” is configured as “0x0F” and becomes “0xF0” after 4-bit offset, which means that the priority level of SysTick and PendSV is 240.

5) `#define configMAX_SYSCALL_INTERRUPT_PRIORITY`

A 8-bit priority level value is obtained after 4-bit offset of the macro “`configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY`”, that is, the “`configMAX_SYSCALL_INTERRUPT_PRIORITY`”, which is assigned in the register `basepri`, and actually used in the corresponding interrupt priority register. The global switch interrupt can be realized after this macro definition value is assigned to the register `basepri`. For example, configure the “`configLIBRARY_LOWEST_INTERRUPT_PRIORITY`” as 0x01 and then get 0x10 (i.e., 16) after 4-bit offset. Call the FreeRTOS disable interrupt, and then all interrupts with priority level greater than 16 will be disabled, and those with priority level lower than 16 will not be disabled. Assign a priority “0” to the register `basepri`, and then those disabled interrupts will be enabled.

Complete the abovementioned procedures, and the FreeRTOS can manage peripherals (switch interrupts) with priority greater than 3, which can improve the system real-time performance to deal with emergency tasks.

4.3 Differences between interrupt priority and task priority

In short, there is no connection between the interrupt priority and task priority. Regardless of the task priority, interrupt priority is always higher than any task priority, which means that the interrupt preempts the execution of any task.

For AT32F403, AT32F413 and AT32F415 series, the lower the priority numbers, the higher interrupt priorities. In FreeRTOS, the lower the priority numbers, the lower task priorities.

4.4 Critical code region

The critical code region refers to code segments that need to be executed entirely and continuously without interruption. For example, the initialization of peripherals needs to be executed in specified timings and cannot be interrupted. In FreeRTOS, disable interrupts when entering the critical code region and enable when exiting. The FreeRTOS has multiple critical code regions to protect the program segments.

There are four functions related to critical code region protection.

Table 2. Critical code region API

Function	Description
taskENTER_CRITICAL()	Enter into critical code region, interrupt disabled
taskEXIT_CRITICAL()	Exit critical code region, interrupt enabled
taskENTER_CRITICAL_FROM_ISR()	Enter into critical code region, interrupt disabled (for ISR)
taskEXIT_CRITICAL_FROM_ISR(x)	Exit critical code region, interrupt enabled (for ISR)

As shown in Table 2, these four functions are in pairs, where the first two functions are used for ordinary tasks and the later two functions are used for interrupt routine.

Functions used for ordinary tasks and interrupt routine can realize the same results through different methods. The general entry critical code region function is implemented by C language, with an internal global variable to achieve the critical code region nesting; while the entry critical code region function used in interrupt is implemented by assembly language, and the nesting is realized by saving and restoring the register basepri value. The assembly language is more efficient than C language, which is suitable for interrupt routine that requires higher operating efficiency.

4.5 Routine

Project name: 03Interrupt_FreeRTOS

Program source code:

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

/* start task priority */
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE       128
/* start task handle */
TaskHandle_t StartTask_Handler;
/* start task entry function */
void start_task(void *pvParameters);

/* LED0 task priority */
#define LED0_TASK_PRIO      4
/* LED0 task stack size */
#define LED0_STK_SIZE       128
/* LED0 task handle */
TaskHandle_t LED0Task_Handler;
/* LED0 task entry function */
void led0_task(void *pvParameters);

/* LED1 task priority */
```

```
#define LED1_TASK_PRIO      3
/* LED1 task stack size */
#define LED1_STK_SIZE      128
/* LED1 task handle */
TaskHandle_t LED1Task_Handler;
/* LED1 task entry function */
void led1_task(void *pvParameters);

/* Debugging task priority */
#define Debug_TASK_PRIO    2
/* Debugging task stack size */
#define Debug_STK_SIZE     512
/* Debugging task handle */
TaskHandle_t DebugTask_Handler;
/* Debugging task entry function */
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();
    /* Create start task */
    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);
    /* Enable task scheduler */
    vTaskStartScheduler();
}

/* start task function */
```



```

void start_task(void *pvParameters)
{
    /* enter into critical code region */
    taskENTER_CRITICAL();
    /* create LED0 task */
    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);
    /* create LED1 task */
    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);
    /* create debugging task */
    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);

    /* delete start task */
    vTaskDelete(StartTask_Handler);
    /* exit critical code region*/
    taskEXIT_CRITICAL();
}

/* LED0 task function */
void led0_task(void *pvParameters)
{
    while(1)
    {
        /* enter critical code region */
        taskENTER_CRITICAL();
        at32_led_toggle(LED3);
        printf("enter critical code region\r\n");
        /* when TMR5 hardware interrupt is used to simulate ordinary delay, vTaskDelay() can be used to generate
        task scheduling */
        while(tmr_flag_get(TMR5,TMR_OVF_FLAG)==RESET);
        tmr_flag_clear(TMR5,TMR_OVF_FLAG);
        printf("exit critical code region\r\n");
    }
}

```

```

/* exit critical code region */
taskEXIT_CRITICAL();
vTaskDelay(1000);
}
}

/* LED1 task function */
void led1_task(void *pvParameters)
{
    while(1)
    {
        at32_led_toggle(LED2);
        vTaskDelay(500);
    }
}

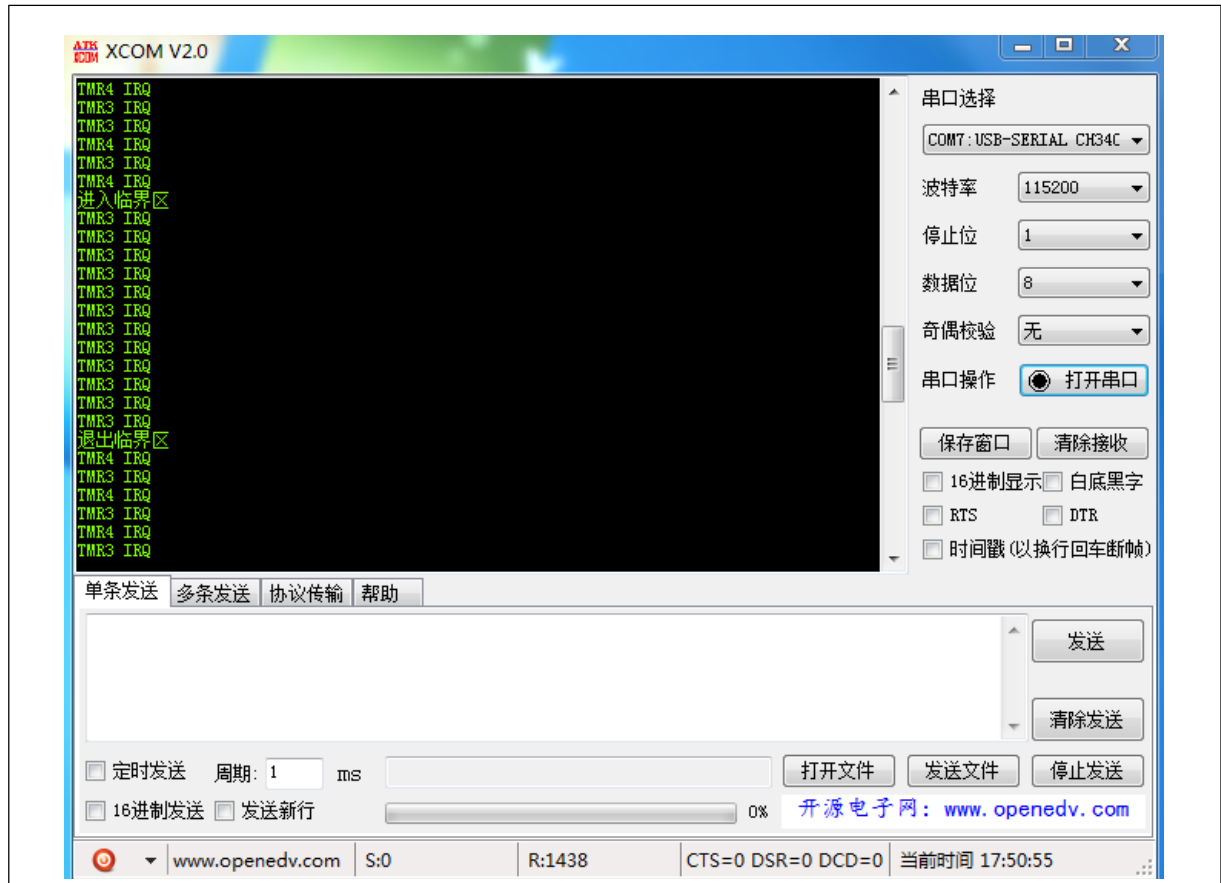
/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out the task information */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/*-----*/\r\n");
            printf("Task          Status          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);
    }
}

```

Two hardware timers (TMR3 and TMR4) are created in this program to periodically generate interrupts. The priority of interrupt generated by TMR4 is “7” and it is managed by the FreeRTOS entry critical code region program; the priority of interrupt generated by TMR3 is “2” and it cannot be managed by the FreeRTOS entry critical code region program.

Compile and download the program to the target board, and the execution is as follows:

Figure 11. FreeRTOS interrupt management routine



The print result shows that no interrupt is generated by TMR4 after entering the critical code region, which indicates that the interrupt is prohibited by the FreeRTOS kernel; TMR4 generates interrupts normally after exiting the critical code region, which indicates that the interrupt is enabled by FreeRTOS.

5 FreeRTOS task management

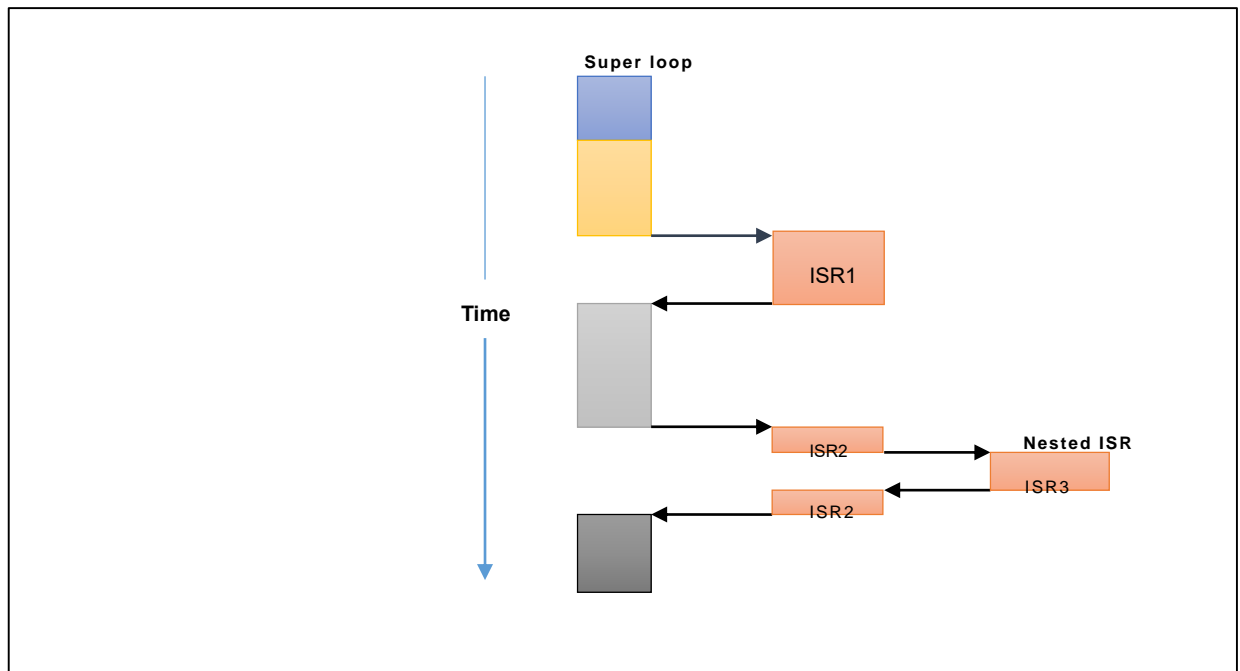
This section introduces task management in FreeRTOS, which is critical for the management of operating system.

5.1 Bare metal vs. RTOS

Bare-metal system

A bare-metal system is generally executed as a super loop, in which each operation is not real-time. To ensure real-time performance, interrupts are used. For example, AT32 MCU has multiple internal interrupt, and emergency events can be executed in these interrupts. This is the foreground-background system, where the background refers to the super loop and the foreground refers to interrupt handling. Figure 12 shows the bare-metal system operation process.

Figure 12. Bare-metal system operation process



As shown in Figure 12, the program is mainly executed in the super loop, but it may be interrupted and continue being executed in the interrupt routine. The program in interrupt is high real-time, but the real-time performance of other parts becomes low; therefore, an operating system is required.

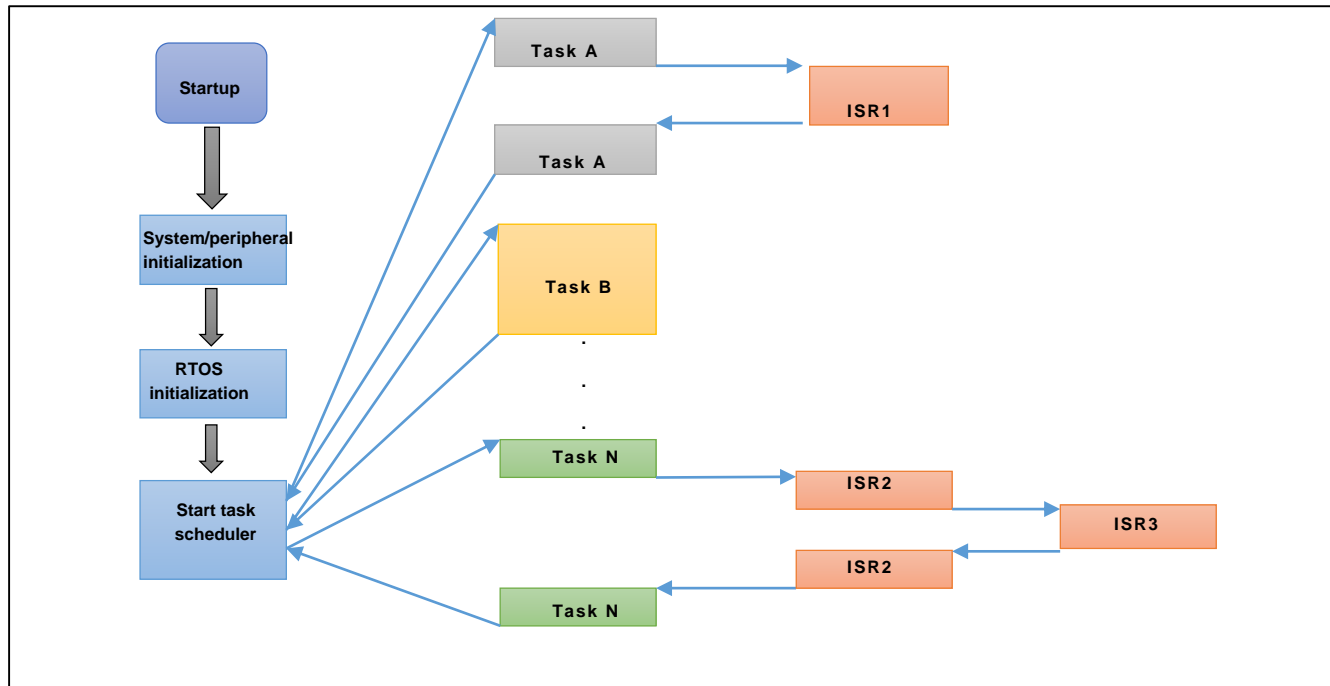
RTOS operation process

In the RTOS, each task is a loop, which can be referred to in the aforementioned codes.

The task scheduler is critical for the implementation of multi-task system or RTOS, which decides the task to be executed by using related scheduling algorithm. Create a task and initialize OS, and then the task scheduler can decide the execution sequence of task A, task B and task C, thus to implement a multi-task system. It should be noted that only one task can be executed at a time, and tasks just look like being executed at the same time based on decisions of task scheduler. For example, the system has tasks A, B and C and each task runs for 10MS; which seems like these tasks are running simultaneously.

The RTOS operation process is simplified as follows:

Figure 13. RTOS operation process



As shown in Figure 13, after the task scheduler is enabled, tasks in the system start polling operation. The task scheduler allows each task to run for a period of time, instead of completing the current task before starting the next one (in bare-metal system), which has better real-time performance. In addition, the RTOS is designed with features such as semaphore and queue, which makes the operating system more flexible.

5.2 FreeRTOS task states

This section introduces the FreeRTOS task states and task state transitions.

FreeRTOS task states include:

1) Running

When a task is actually executing it is said to be in the Running state. It is currently utilising the processor.

2) Ready

Ready tasks are those that are able to execute (they are not in the Blocked or Suspended state) but are not currently executing because a different task of equal or higher priority is already in the Running state.

3) Blocked

A task is said to be in the Blocked state if it is currently waiting for semaphore, queue or event group. In addition, if a task calls a delay function, it will block (be placed into the Blocked state) until the delay period has expired.

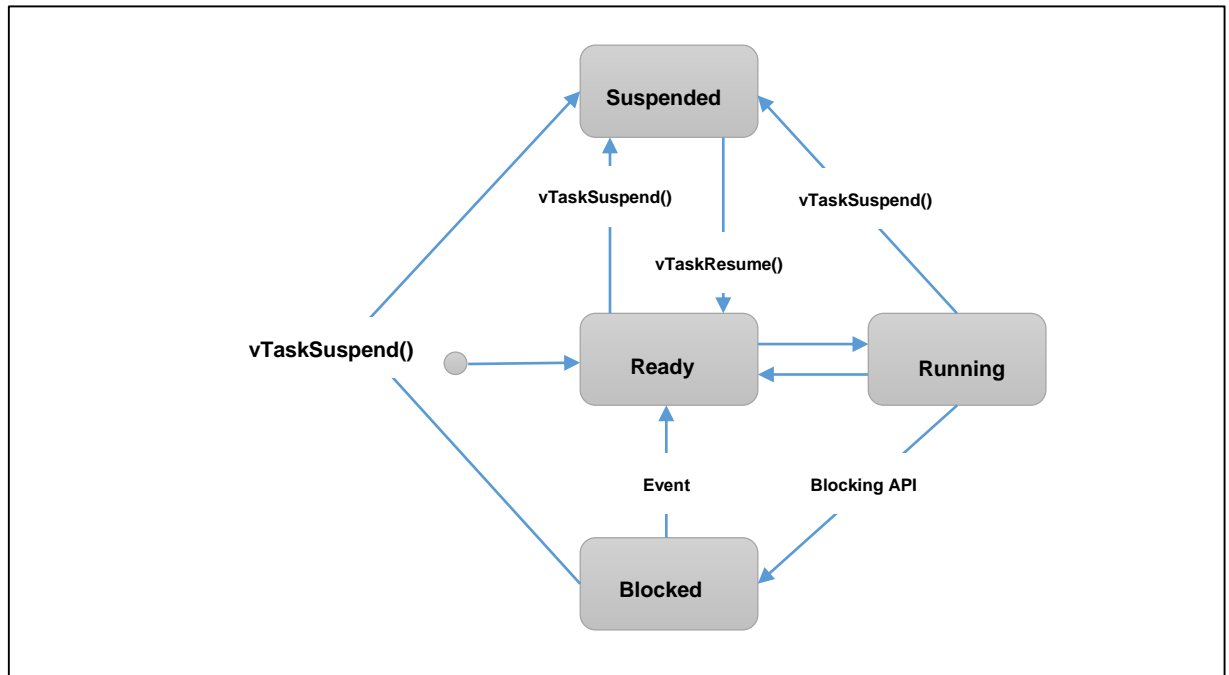
4) Suspended

Like tasks that are in the Blocked state, tasks in the Suspended state cannot be selected to enter the Running state. Tasks only enter or exit the Suspended state when explicitly commanded to do

so through the `vTaskSuspend()` and `xTaskResume()` API calls respectively.

Figure 14 shows the transition among these task states.

Figure 14. Task state transitions



As shown in Figure 14, a task enters Ready state once it is created, and then enters Running state. The task in Running state enters the Blocked state when a Blocked state entry function is called, or it enters the Suspended state when a Suspended state entry function is called. The task in Blocked state will enter Ready state after an event is received, and the task in Suspended state can be transitioned between the Ready state and Running state by calling the corresponding API. The task in Blocked state also can enter Suspended state by calling the corresponding API.

5.3 FreeRTOS idle task

Almost all small RTOS systems have an idle task, which is a system task and must be executed and cannot be closed by the user program. The idle task also can be found in large systems such as Windows.

The idle task is created automatically when the RTOS scheduler is started to ensure there is always at least one task that is able to run. It is created at the lowest possible priority to ensure it does not use any CPU time if there are higher priority application tasks in the ready state. In addition, this idle task can be used to enter low-power mode (there is no task to execute when run the idle task, indicating the best time to enter low-power mode).

5.4 FreeRTOS task related functions

This section introduces commonly used FreeRTOS API functions. For more functions, please refer to the FreeRTOS user manual in the official website.

Task create

It includes task create dynamic and task create static, with different API function name and parameters.

xTaskCreate();

Description:

Create a task with dynamic method.

Prototype:

Table 3. xTaskCreate()

BaseType_t xTaskCreate(TaskFunction_t pvTaskCode, const char * const pcName, unsigned short usStackDepth, void *pvParameters, UBaseType_t uxPriority, TaskHandle_t *pxCreatedTask);	
Parameter	Description
pvTaskCode	Task function
pcName	Task name
usStackDepth	Task stack size
pvParameters	Task parameters
uxPriority	Task priority
pxCreatedTask	Task handle

Return value:

Task create success flag.

xTaskCreateStatic();

Description:

Create a task with static method.

Prototype:

Table 4. xTaskCreateStatic()

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);	
Parameter	Description
pvTaskCode	Task function
pcName	Task name
ulStackDepth	Task stack size
pvParameters	Task parameters
uxPriority	Task priority
puxStackBuffer	Task stack memory
pxTaskBuffer	Task control block memory

Return value:

Task create success flag.

The difference between task create dynamic and task create static is whether the task control block memory and task stack memory are provided manually. For task create dynamic, the required RAM is automatically allocated, which is manually allocated for task create static.

Task delete

vTaskDelete();

Description:

Delete the specified task.

Prototype:

Table 5. vTaskDelete()

void vTaskDelete(TaskHandle_t pxTask);	
Parameter	Description
pxTask	The handle of the task to be deleted.

Return value:

None.

Task suspend

vTaskSuspend();

Description:

Suspend any task.

Prototype:

Table 6. vTaskSuspend()

void vTaskSuspend(TaskHandle_t pxTaskToSuspend);	
Parameter	Description
pxTaskToSuspend	Handle to the task being suspended.

Return value:

None.

Task resume

vTaskResume();

Description:

Resume a suspended task.

Prototype:

Table 7. vTaskResume()

void vTaskResume(TaskHandle_t pxTaskToResume);	
Parameter	Description
pxTaskToResume	Handle to the task being readied.

Return value:

None.

Task delay

vTaskDelay();

Description:

Transition the task in Running state to Blocked state.

Prototype:

Table 8. vTaskDelay()

void vTaskDelay(TickType_t xTicksToDelay);	
Parameter	Description
xTicksToDelay	The amount of time, in tick periods.

Return value:

None.

For more task related APIs, please refer to the FreeRTOS API Reference in the official website.

5.5 Routine

Project name: 04TaskManagement_FreeRTOS

Program source code:

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

/* start task priority */
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE       128
/* start task handle */
TaskHandle_t StartTask_Handler;
/* start task entry function */
void start_task(void *pvParameters);

/* LED0 task priority */
#define LED0_TASK_PRIO       3
/* LED0 task stack size */
#define LED0_STK_SIZE        128
/* LED0 task handle */
TaskHandle_t LED0Task_Handler;
/* LED0 task entry function */
void led0_task(void *pvParameters);

/* LED1 task priority */
#define LED1_TASK_PRIO       3
```

```

/* LED1 task stack size */
#define LED1_STK_SIZE      128
/* LED1 task handle */
TaskHandle_t LED1Task_Handler;
/* LED1 task entry function */
void led1_task(void *pvParameters);

/* print task priority */
#define Printf_TASK_PRIO    2
/* print task stack size */
#define Printf_STK_SIZE     128
/* print task handle */
TaskHandle_t PrintfTask_Handler;
/* print task entry function */
void Printf_task(void *pvParameters);

/* debugging task priority */
#define Debug_TASK_PRIO     2
/* debugging task stack size */
#define Debug_STK_SIZE     512
/* debugging task handle */
TaskHandle_t DebugTask_Handler;
/* debugging task entry function */
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();
    /* create start task */
    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);
/* enable task scheduler */
vTaskStartScheduler();
}

/* start task function */
void start_task(void *pvParameters)
{
    /* enter critical code region */
    taskENTER_CRITICAL();
    /* create LED0 task */
    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);
    /* create LED1 task */
    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);
    /* create debugging task */
    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);
    /* create print task */
    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);
    /* delete start task */
    vTaskDelete(StartTask_Handler);
    /* exit critical code region */
    taskEXIT_CRITICAL();
}

```

```

/* LED0 task function */
void led0_task(void *pvParameters)
{
    while(1)
    {
        at32_led_toggle(LED3);
        printf("LED3 Toggle\r\n");
        /* suspend print task */
        vTaskSuspend(PrintfTask_Handler);
        printf("Suspend printf task!\r\n");
        vTaskDelay(1000);
        printf("Resume printf task!\r\n");
        /* resume suspended print task */
        vTaskResume(PrintfTask_Handler);
        vTaskDelay(1000);
    }
}

/* LED1 task function */
void led1_task(void *pvParameters)
{
    while(1)
    {
        at32_led_toggle(LED2);
        printf("LED2 Toggle\r\n");
        vTaskDelay(100);
    }
}

/* print task function */
void Printf_task(void *pvParameters)
{
    while(1)
    {
        printf("printf task!\r\n");
    }
}

/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out debugging information */

```

```

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);
}
}

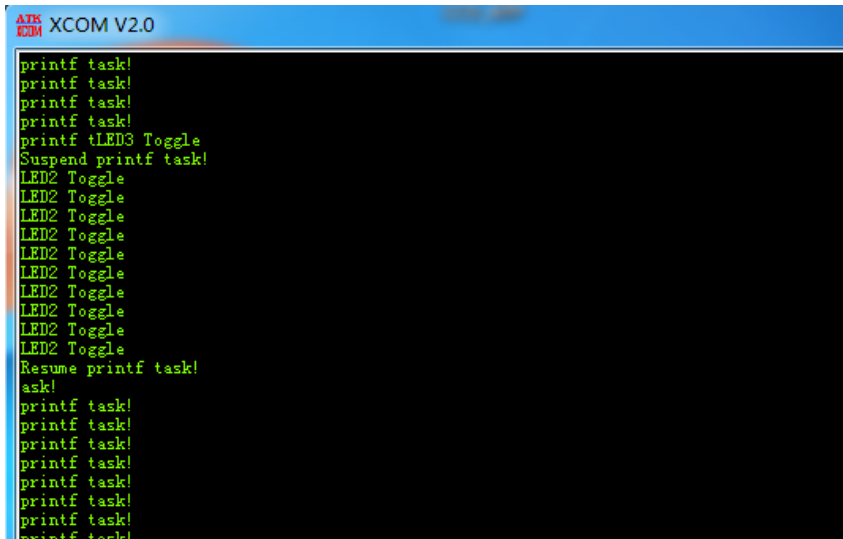
```

This is the FreeRTOS task management routine source code, in which four tasks (two LED tasks, one print task and one task running information output task) are created. Call the “vTaskSuspend()” function in the LED0 task to suspend the print task, and then call “vTaskDelay()” to delay a period of time to call the “vTaskResume()” to resume the suspended print task to ready state.

A start task is also created, in which the above four tasks are created. The start task runs once and then it is deleted by calling the “vTaskDelete()” function. Therefore, the start task is used to create these four tasks.

Compile and download the program to the target board, and the execution is as follows:

Figure 15. Task management routine



```

AT32
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!

```

The print result shows that once the “LED3 Toggle” is output, the “Suspend printf task” is output immediately; at this point, the print task is suspended. It resumes ready state after the “Resume printf task” is output, and starts to run and print out “printf task”. This output process conforms to the program and verifies the abovementioned API functions.

6 FreeRTOS task scheduling

This section introduces three task scheduling methods supported by FreeRTOS, including cooperative, preemptive and time slicing. For more details, visit the FreeRTOS official website.

6.1 Cooperative scheduling

Figure 16. FreeRTOS Co-routines

FreeRTOS BSPs:
3rd party driver packages

Trace & Visualisation:
Tracealyzer for FreeRTOS

CLI:
Command line interface

WolfSSL SSL / TLS:
Networking security protocols

RTOS Training:
Delivered online or on-site

IO:
read(), write(), ioctl() interface

FreeRTOS+ Lab Projects

IoT MQTT:
PubSub messaging protocol

IoT Task Pool:
A shared collection of tasks

FreeRTOS+POSIX:
POSIX threading API

FreeRTOS+FAT:
Thread aware file system

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**

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.

6.2 Preemptive scheduling

RTOS is an operating system with excellent real-time performance that is realized by preemptive scheduling.

Before learning more about the preemptive scheduling, users need to have a comprehensive understanding about the scheduler. In short, a scheduler decides the task to execute by using related scheduling algorithms. All schedulers have the following characteristics:

1. A scheduler can distinguish between the ready state and suspended state (task is suspended due to delay, semaphore waits, mailbox waits and event group waits).
2. A scheduler can select a task in ready state and activate it (execute the task). The task currently being executed is in running state.

3. The biggest difference between different schedulers is how to allocate the time to complete tasks in ready state.

The core of an embedded real-time operating system is the scheduler and task switch. The implementation of task switch is slightly different in different embedded real-time operating system, and task switch is the similar for basically the same hardware kernel architectures. The core of scheduler is scheduling algorithm, which may be different.

The preemptive scheduling is one of scheduler algorithms. In actual applications, different tasks require different response times. For example, if motor, keyboard and LCD display are required in an application, the motor requires faster response than the keyboard and LCD display. In this case, if cooperative or time slicing method is used, the motor cannot be responded in time; therefore, the preemptive scheduling method is required. If this method is used, a high priority task always gets the CPU once it is readied. For example, if a running task is preempted by a high priority task, it is suspended and the high priority task gets the CPU to run. If an interrupt service routine decides a high priority task to enter Ready state, when the interrupt is complete, the interrupted low priority task is suspended, and the high priority task starts to run.

The preemptive scheduling algorithm selects tasks according to their priorities and helps optimize the task response time. In general, all tasks have different priorities, and the preemptive scheduler selects the highest priority task to run.

The operation process of preemptive scheduler is as follows:

1. Initialize system and enable task scheduler. The highest priority Task1 is executed and enter suspended state at the time when the blocked API functions (such as delay, event flag waits and semaphore waits) are called, which means that Task1 releases CPU to run the low priority tasks.
2. FreeRTOS continues to run the next high priority Task2 in the list of readied tasks, which may be one of the following two situations:
 - a) Task1 resumes from suspended state to ready state due to delay or reception of semaphore, and Task2 execution is preempted by Task1 with the preemptive scheduler.
 - b) Task2 continues to run and enter suspended state at the time when the blocked API functions (such as delay, event flag waits and semaphore waits) are called, and then the next high priority task in the list of readied tasks will be executed.
3. If the user creates multiple tasks and uses preemptive scheduler, the current task is preempted by a high priority task or CPU is released by calling blocked API functions to execute low priority task. An idle task is executed if no other task is running.

6.3 Time-slicing scheduling

The round-robin algorithm is commonly used in small embedded RTOS systems, which can be used for preemptive or cooperative multi-tasks. In addition, the time-slicing scheduling is suitable for situations where real-time response is not required. In FreeRTOS, we usually use scheduling policy by mixing both the time-slicing scheduling algorithm with preemptive scheduling algorithm. Tasks with low real-time requirements can be set with the same priority level (low priority is recommended), and tasks with high real-time requirements can be set with high priority level and

used together with queue and semaphore.

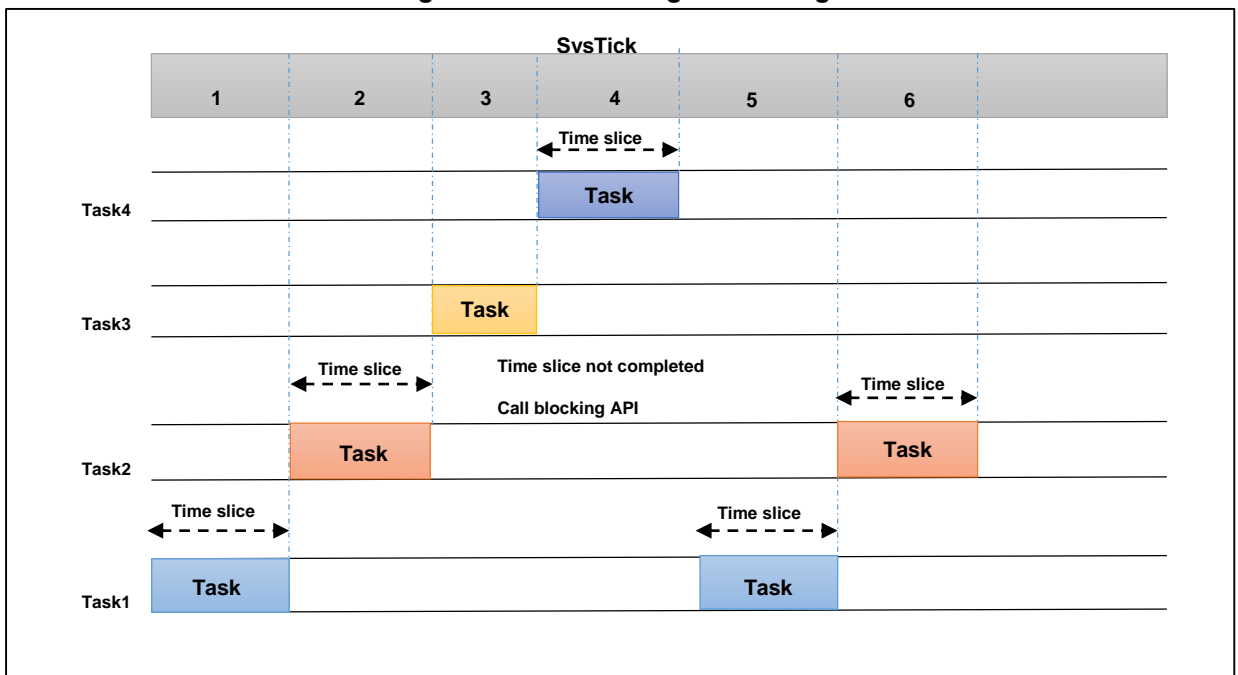
To implement round-robin algorithm, a specified list is allocated to tasks with the same priority level to record the current ready tasks and assign a time slice (i.e., execution time period; switch tasks when the time slice is completed) to each task. In FreeRTOS, only tasks with the same priority use the time-slicing scheduling method. In addition, users need to enable the macro “#define configUSE_TIME_SLICING 1” in *FreeRTOSConfig.h* file. By default, this macro is enabled in *FreeRTOS.h* file.

The operation process of the time-slicing scheduling is as follows:

Operation conditions:

1. It is for time-slicing scheduling only.
2. Create four tasks (Task1, Task2, Task3 and Task4) with the same priority.
3. Each task is allocated with a specified time slice (1 system clock cycle).

Figure 17. Time-slicing scheduling



As shown in Figure 17,

1. Execute Task1 for one system clock cycle, and then switch to Task2 through time-slicing scheduling.
2. Execute Task2 for one system clock cycle, and then switch to Task3 through time-slicing scheduling.
3. Call blocked API functions when executing Task3; although Task3 is not executed for a complete system clock cycle, it switches to Task4 through time-slicing scheduling (note: the uncompleted time slice will not be used, and the next time Task3 is to be executed for one system clock cycle).
4. Execute Task4 for one system clock cycle, and then switch to Task1 through time-slicing scheduling.

The next section introduces a routine that uses time-slicing scheduling method.

6.4 Routine

Project name: 05Time_Slicing_FreeRTOS

Program source code:

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

/* start task priority*/
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE      128
/* start task handle */
TaskHandle_t StartTask_Handler;
/* start task entry function */
void start_task(void *pvParameters);

/* Task1 priority */
#define TASK1_PRIO      5
/* Task1 stack size */
#define TASK1_STK_SIZE      128
/* Task1 handle */
TaskHandle_t TASK1_Handler;
/* Task1 entry function */
void task1(void *pvParameters);

/* Task2 priority */
#define TASK2_PRIO      5
/* Task2 stack size */
#define TASK2_STK_SIZE      128
/* Task2 handle */
TaskHandle_t TASK2_Handler;
/* Task2 entry function */
void task2(void *pvParameters);

/* Task3 priority */
#define TASK3_PRIO      5
/* Task3 stack size */
#define TASK3_STK_SIZE      128
/* Task3 handle */
TaskHandle_t TASK3_Handler;
/* Task3 entry function */
void task3(void *pvParameters);
```

```

/* debugging task priority */
#define Debug_TASK_PRIO      6
/* debugging task stack size */
#define Debug_STK_SIZE      512
/* debugging task handle */
TaskHandle_t DebugTask_Handler;
/* debugging task entry function */
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();
    /* create start task */
    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);

    /* enable task scheduler */
    vTaskStartScheduler();
}

/* start task function */
void start_task(void *pvParameters)
{
    /* enter critical code region */
    taskENTER_CRITICAL();
    /* create task1 */
    xTaskCreate((TaskFunction_t)task1,
                (const char* )"task1",
                (uint16_t )TASK1_STK_SIZE,

```

```

        (void*      )NULL,
        (UBaseType_t )TASK1_PRIO,
        (TaskHandle_t*)&TASK1_Handler);

/* create task2 */
xTaskCreate((TaskFunction_t)task2,
            (const char*    )"task2",
            (uint16_t       )TASK2_STK_SIZE,
            (void*          )NULL,
            (UBaseType_t    )TASK2_PRIO,
            (TaskHandle_t*)&TASK2_Handler);

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

/* create debugging task */
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);

/* delete start task */
vTaskDelete(StartTask_Handler);

/* exit critical code region */
taskEXIT_CRITICAL();
}

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

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

```

```

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

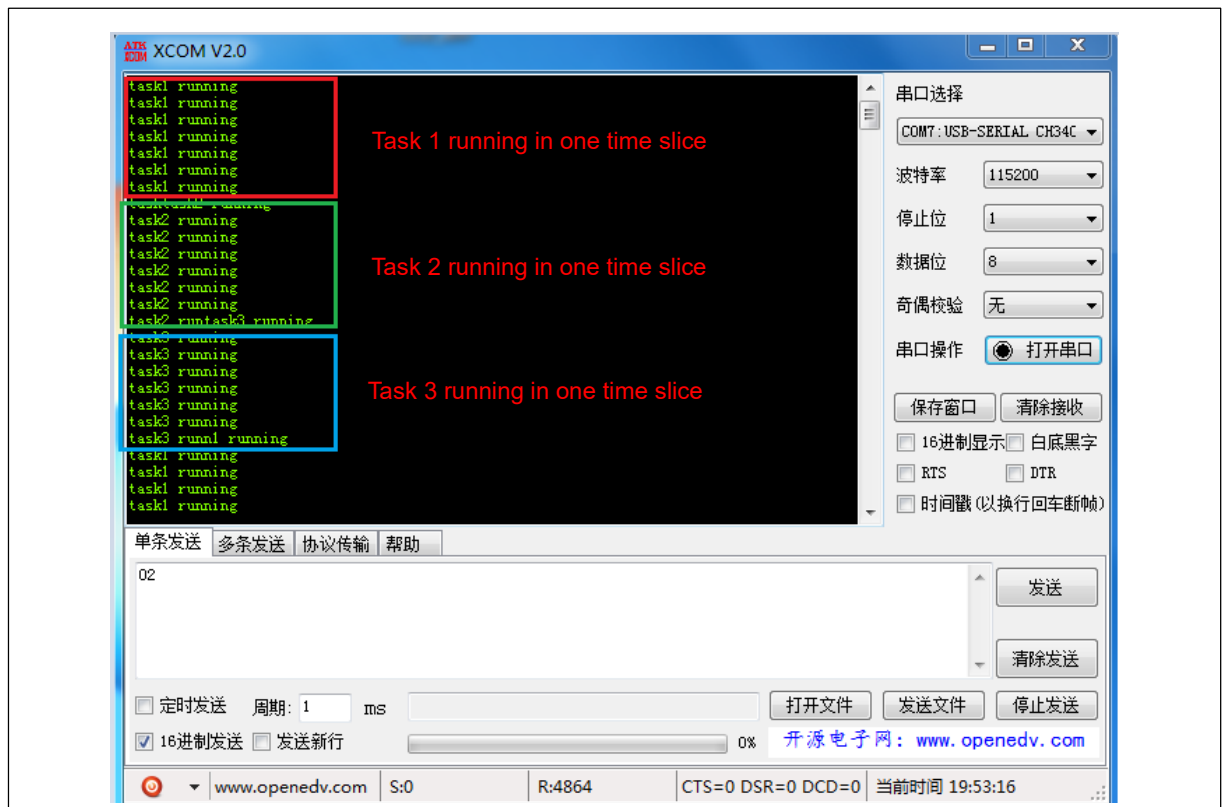
/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out the debugging information */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/*-----*/\r\n");
            printf("Task          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);
    }
}

```

In this routine, four tasks are created, including a print task (press the USER button to print out the current system information) and three tasks (print out a string to indicate the current running state). Note that these three tasks do not include any task switch function and are executed with the time-slicing scheduling method.

Compile and download the program to the target board, and the execution is as follows:

Figure 18. Time-slicing scheduling routine



The print result shows that these three tasks are executed according to the time slice, and one task switches to the next task automatically after a time slice is completed.

7 FreeRTOS queue

This section introduces FreeRTOS queue, which is widely used in actual applications and serves the basis of implementing other kernel services.

7.1 Introduction

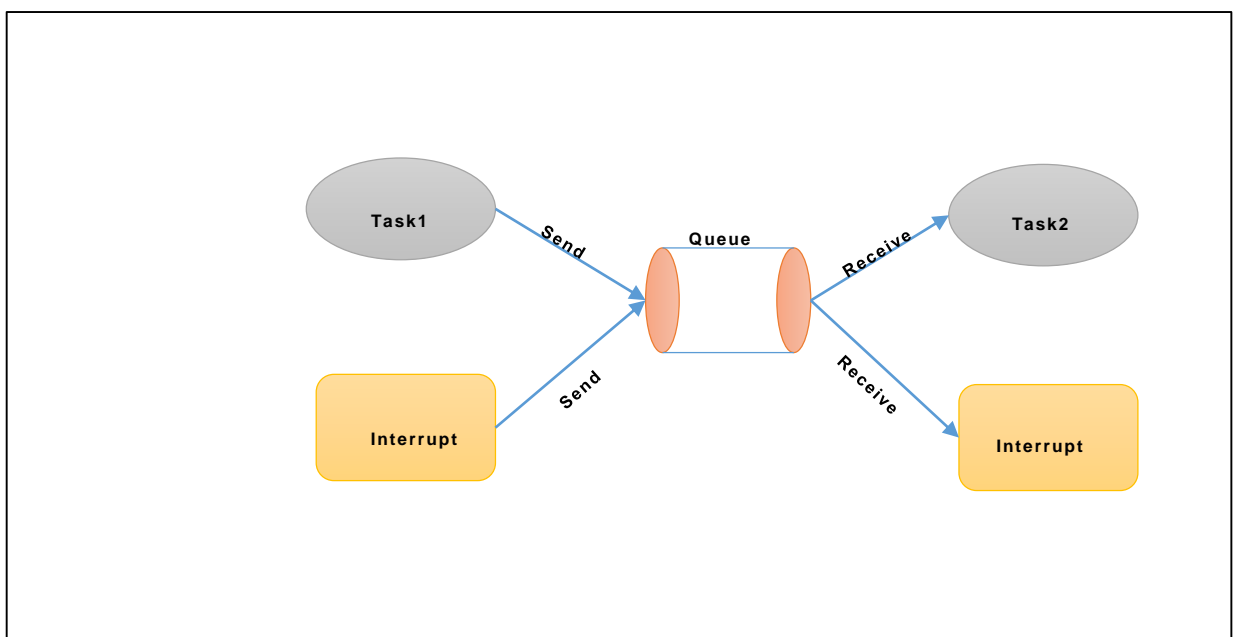
Queues are the primary form of intertask communications. Tasks or interrupt service subroutines can place a message into a queue; similarly, one or more tasks can obtain message from the queue through the RTOS kernel. Generally, the message that enters the queue firstly passes to the task, which means that the task gets the message firstly entering the queue, namely, FIFO (First In First Out). The FreeRTOS queue supports FIFO and LIFO.

In bare-metal systems, the global array is easy-to-use and preferred; while in RTOS systems, the following aspects should be considered when using global array:

1. Using queues allows RTOS kernel to manage tasks efficiently, which cannot be realized by using global array (task timeout also needs to be implemented by users).
2. Using global array may cause access conflict of multiple tasks.
3. Using queues can deal with the message transfer between the interrupt service routine and tasks.
4. FIFO is more applicable for data processing.

FreeRTOS queues can be used to send messages between tasks, and between interrupts and tasks. A queue should be created before message transfer. This queue is managed by the FreeRTOS kernel, and messages to be transferred are put into this queue. Task or interrupt message can be obtained from this queue, thus realizing message transfer. The process is as follows:

Figure 19. Queue workflow



As shown in Figure 19, the queue workflow is simple. In actual applications, the following aspects about interrupts should be noted:

1. The execution time of interrupt functions should be short as far as possible so that other abnormalities with lower priorities can be responded in time.
2. In actual applications, message processing in interrupts is not recommended. Users can send message notification task in the interrupt service routine and implement message processing in the task, which can effectively ensure real-time response of the interrupt service routine. Meanwhile, this task is allocated with high priority, so that it can be executed after exiting the interrupt function.
3. Interrupt specific queue functions (ending with FromISR) must be called in the interrupt service routine.

It should be noted that the implementation of interrupt service routine is different in operating system and bare-metal system.

1. If the FreeRTOS queue API function is not called, the programming is the same in operating system and bare-metal system.
2. If the FreeRTOS queue API function is called, check whether the high priority task is ready when exiting the interrupt; if it is ready, switch the task after exiting the interrupt (different from that in the bare-metal system).
3. It is recommended to set interrupt priority group 4 (NVIC_PriorityGroup_4) for AT32 MCUs.
4. Users needs to configure the priority group before enabling FreeRTOS multi-tasks, which should not be modified once configured.

7.2 Queue API

This section introduces FreeRTOS queue API functions.

Table 9. Queue API reference

Queue API functions	
API	Description
vQueueAddToRegistry()	Assigns a name to a queue and adds the queue to the registry
xQueueAddToSet()	Add the queue to the previously created queue set
xQueueCreate()	Create a queue
xQueueCreateSet()	Create a queue set
xQueueCreateStatic()	Create a queue with static method
vQueueDelete()	Delete a queue
char *pcQueueGetName()	Look up a queue name from the queue's handle
xQueueIsQueueEmptyFromISR()	Query a queue to determine if the queue is empty
xQueueIsQueueFullFromISR()	Query a queue to determine if the queue is full
uxQueueMessagesWaiting()	Query the number of messages in a queue
uxQueueMessagesWaitingFromISR()	Query the number of messages in a queue in interrupt
xQueueOverwrite()	Write to the queue even if the queue is full
xQueueOverwriteFromISR()	Write to the queue even if the queue is full (called from an interrupt)
xQueuePeek()	Receive an item from a queue without removing the item from the queue
xQueuePeekFromISR()	Receive an item from a queue without removing the item from the queue (called from an interrupt)

xQueueReceive()	Receive an item from a queue.
xQueueReceiveFromISR()	Receive an item from a queue (called from an interrupt)
xQueueRemoveFromSet()	Remove a queue from a queue set
xQueueReset()	Reset a queue to its original empty state
xQueueSelectFromSet()	Select a queue from a queue set
xQueueSelectFromSetFromISR()	Select a queue from a queue set (called from an interrupt)
xQueueSend() xQueueSendToBack()	Post an item to the back of a queue
xQueueSendToFront()	Post an item to the front of a queue
xQueueSendFromISR() xQueueSendToBackFromISR()	Post an item to the back of a queue (called from an interrupt)
xQueueSendToFrontFromISR()	Post an item to the front of a queue (called from an interrupt)
uxQueueSpacesAvailable()	Query the number of free spaces in a queue

Table 9 lists all queue API functions, and some are detailed in the following sections. For more information, please refer to the FreeRTOS API Reference in the official website.

xQueueCreate();

Description:

Create a queue.

Prototype:

Table 10. xQueueCreate()

QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);	
Parameter	Description
uxQueueLength	The maximum number of items the queue can hold at any one time
uxItemSize	The size required to hold each item in the queue

Return value:

A handle to the created queue

xQueueSend();

Description:

Post an item on a queue

Prototype:

Table 11. xQueueSend()

BaseType_t xQueueSend(QueueHandle_t xQueue, const void * pvltemToQueue, TickType_t xTicksToWait);	
Parameter	Description
xQueue	The handle to the queue on which the item is to be posted

pvItemToQueue	A pointer to the item that is to be placed on the queue
xTicksToWait	The maximum amount of time the task should block waiting for space to become available on the queue

Return value:

Item post success flag

xQueueSendFromISR();

Description:

Post an item to the back of a queue within an interrupt service routine.

Prototype:

Table 12. xQueueSendFromISR() (within interrupt service routine)

BaseType_t xQueueSendFromISR(QueueHandle_t xQueue, const void *pvItemToQueue, BaseType_t *pxHigherPriorityTaskWoken);	
Parameter	Description
xQueue	The handle to the queue on which the item is to be posted
pvItemToQueue	A pointer to the item that is to be placed on the queue
pxHigherPriorityTaskWoken	Flag indicating task switch before exiting the interrupt

Return value:

Data send success flag

xQueueReceive();

Description:

Receive an item from a queue

Prototype:

Table 13. xQueueReceive()

BaseType_t xQueueReceive(QueueHandle_t xQueue, void *pvBuffer, TickType_t xTicksToWait);	
Parameter	Description
xQueue	The handle to the queue from which the item is to be received
pvBuffer	Pointer to the buffer into which the received item will be copied
xTicksToWait	The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call

Return value:

Item receive success flag

xQueueReceiveFromISR();

Description:

Receive an item from a queue. It is safe to use this function from within an interrupt service routine.

Prototype:

Table 14. xQueueReceiveFromISR()

BaseType_t xQueueReceiveFromISR(QueueHandle_t xQueue, void *pvBuffer, BaseType_t *pxHigherPriorityTaskWoken);	
Parameter	Description
xQueue	The handle to the queue from which the item is to be received
pvBuffer	Pointer to the buffer into which the received item will be copied
pxHigherPriorityTaskWoken	A task may be blocked waiting for space to become available on the queue

Return value:

Item reception success flag

These functions are used in the following routine to help users have a better understanding.

7.3 Routine

Project name: 06Message_Queue_FreeRTOS

Program source code:

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

/* start task priority*/
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE      128
/* start task handle */
TaskHandle_t StartTask_Handler;
/* start task entry function */
void start_task(void *pvParameters);

/* message processing task priority */
#define Process_Message_TASK_PRIO      1
/* message processing task stack size */
#define Process_Message_STK_SIZE      256
/* message processing task handle*/
TaskHandle_t Process_MessageTask_Handler;
/* message processing task entry function*/
```

```

void Process_Message_task(void *pvParameters);

/* message receive task priority */
#define Receive_Message_TASK_PRIO      3
/* message receive task stack size */
#define Receive_Message_STK_SIZE      256
/* message receive task handle */
TaskHandle_t Receive_MessageTask_Handler;
/* message receive task entry function */
void Receive_Message_task(void *pvParameters);

/* debugging task priority */
#define Debug_TASK_PRIO      3
/* debugging task stack size */
#define Debug_STK_SIZE      512
/* debugging task handle */
TaskHandle_t DebugTask_Handler;
/* debugging task entry function */
void debug_task(void *pvParameters);

/* define a message structure */
typedef struct A_Message
{
    char ucMessageID;
    u8 ucData;
} AMessage;

/* define a queue */
QueueHandle_t AT_xQueue;
/* queue length and message size */
#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);

/* create start task */
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);

/* enable task scheduler */
vTaskStartScheduler();
}

/* start task function */
void start_task(void *pvParameters)
{
    /* enter critical code region */
    taskENTER_CRITICAL();
    /* create a queue */
    AT_xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );
    if(AT_xQueue == NULL)
    {
        /* queue creation failure */
        while(1);
    }
    /* initialize the timer after a queue is created */
    TIMER_Init();
    /* create message receive task */
    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);
    /* create message processing task */
    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);
    /* create debugging task */
    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);

/* delete start task */
vTaskDelete(StartTask_Handler);
/* exit critical code region */
taskEXIT_CRITICAL();
}
/* message receive task function */
void Receive_Message_task(void *pvParameters)
{
    AMessage  Message1;
    while(1)
    {
        switch(GetUartData())
        {
            case NODATA:
                break;
            case 0x01:
                Message1.ucMessageID = 'a';
                Message1.ucData = 0x01;
                /* receive Toggle LED2 message */
                xQueueSend(AT_xQueue,&Message1,10);
                break;
            case 0x02:
                Message1.ucMessageID = 'b';
                Message1.ucData = 0x02;
                /* receive Toggle LED3 message */
                xQueueSend(AT_xQueue,&Message1,10);
                break;
            case 0x03:
                Message1.ucMessageID = 'c';
                Message1.ucData = 0x03;
                /* receive Toggle LED4 message */
                xQueueSend(AT_xQueue,&Message1,10);
                break;
            default:
                break;
        }
        vTaskDelay(100);
    }
}
/* message processing task function */

```

```

void Process_Message_task(void *pvParameters)
{
    AMessage Message2;
    while(1)
    {
        /* receive message */
        if(xQueueReceive(AT_xQueue, &Message2, portMAX_DELAY) != pdPASS)
        {
            printf("no message\r\n");
        }
        else
        {
            /* determine message type and respond correspondingly */
            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");
        }
    }
}

/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out the debugging information */
        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;
        /* send timer interrupt message to the queue */
        xQueueSendFromISR(AT_xQueue,&Message3,&xHigherPriorityTaskWoken);
        /* check whether or not to switch task */
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
        tmr_flag_clear(TMR3,TMR_OVF_FLAG);
    }
}
```

In this routine, three tasks are created (start task not included). The first is debugging information print task, which prints the task information once it detects that the USER button on the target board is pressed, including the task name, task state, remaining task stack size, task serial number and the time period of CPU being occupied by this task. The second is message receive task, which periodically reads data from the serial port buffer, and then analyze the data (if any) and send the analysis result to the specified queue. The third is message processing task, which runs when there is data in queue and receives message from the queue, and then toggles LED2/3/4 according to the message. A hardware timer is enabled in this program. A message is sent to the queue when an overflow interrupt is generated by the timer, and then the message processing task receives this message and prints out "Timer interrupt".

Note: Ensure that the hardware is configured correctly before running this routine, and connect the serial port 1 on the target board to PC (using USB-to-serial tools).

Compile and download the program to the target board, and the execution is as follows:

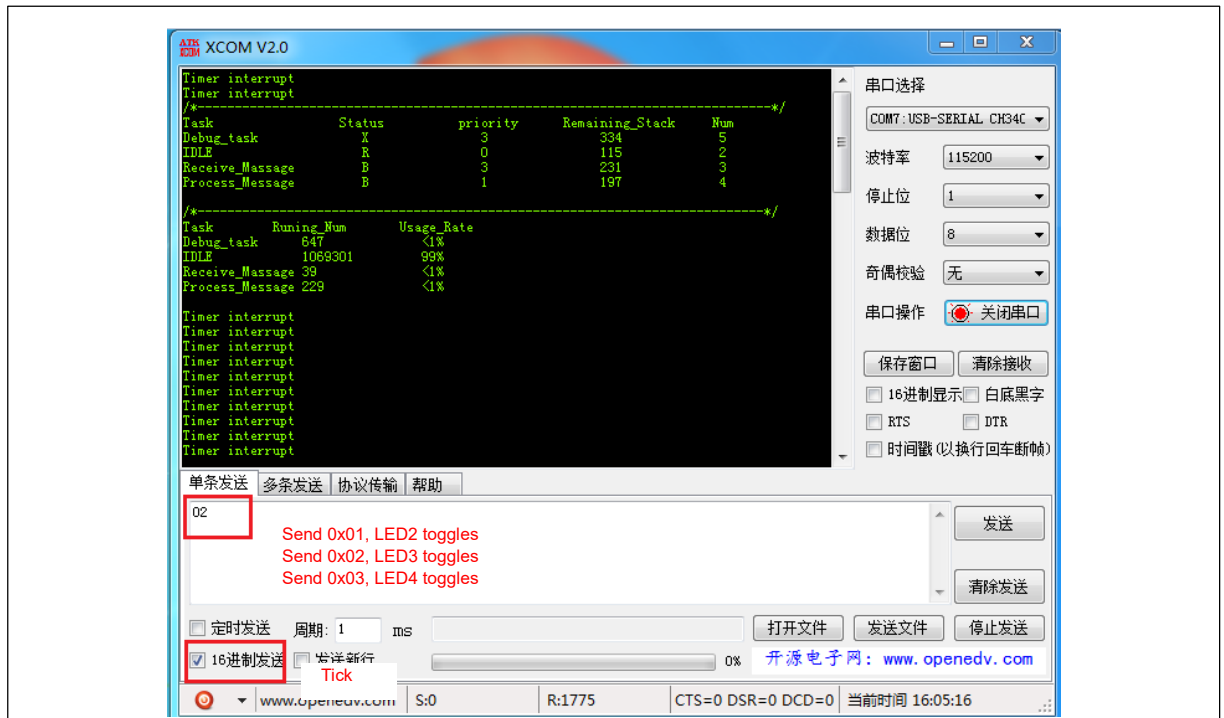
Figure 20. Queue routine



The print result shows that timer interrupt is sent to the queue and the task information is output once the USER button is pressed.

The host sends a message to control LEDs on the target board, as shown below:

Figure 21. Queue routine (LED toggle)



Send 0x01, 0x02 and 0x03 to control LED2/3/4 respectively (tick "Send hexadecimal").

8 FreeRTOS semaphore

This section introduces FreeRTOS semaphores, including binary semaphores, counting semaphores, mutexes and recursive mutexes, which is critical to inter-task communication and resource sharing.

8.1 Introduction

Invented in mid-1960s, semaphores were initially used to set a flag to indicate the usage of common resources. This flag is queried before a task accesses the common resources, and the subsequent operation is determined based on the query result.

In actual applications, for example, a park has 100 parking spots, and the semaphore is 100 if all parking spots are not occupied. The semaphore is decremented or incremented by 1 every time one car drives to or leaves the parking spot. It becomes 0 when there are 100 cars in the parking spots, and no car can get a semaphore to drive into the park. The semaphore is not released until a car leaves the parking lot, at which point the car that is going to drive into the park can get this released semaphore to occupy a parking lot.

In actual application, semaphores are mainly used to implement the following functions:

1. Synchronization between two tasks or between the interrupt function and task (common resource=1)
2. Management of multiple common resources

FreeRTOS supports binary semaphores and counting semaphores to implement the above functions. The binary semaphore can be regarded as a special counting semaphore, which is initialized to have one available resource only. FreeRTOS also provides API functions to those semaphores, while RTX and uCOS-II/III only support semaphore function and set different initial values to implement binary semaphores and counting semaphores. FreeRTOS can use counting semaphores to achieve the same effect

8.2 Binary semaphore

8.2.1 Introduction

Binary semaphores are used for both mutual exclusion and synchronization purposes. Binary semaphores and mutexes are very similar but have some subtle differences: mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

Semaphore API functions permit a block time to be specified. The block time indicates the maximum number of 'ticks' that a task should enter the Blocked state when attempting to 'take' a semaphore, should the semaphore not be immediately available. If more than one task blocks on the same semaphore, then the task with the highest priority will be the task that is unblocked the next time the semaphore becomes available.

Think of a binary semaphore as a queue that can only hold one item. The queue can therefore only be empty or full (hence binary). Tasks and interrupts using the queue don't care what the queue holds - they only want to know if the queue is empty or full. This mechanism can be exploited to

synchronize two tasks or a task with an interrupt.

Here is an example of binary semaphores available from FreeRTOS official website.

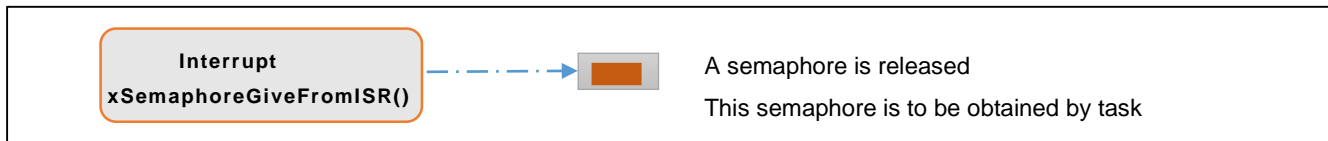
1. The task enters Blocked state because there is no semaphore.

Figure 22. Binary semaphore block diagram 1



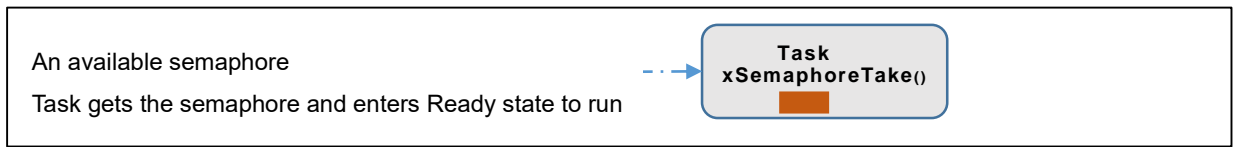
2. A semaphore is released within an interrupt program.

Figure 23. Binary semaphore block diagram 2



3. The task gets the semaphore and switches from Blocked state to Running state.

Figure 24. Binary semaphore block diagram 3



These figures show the binary semaphore operation process. The solid red rectangle represents a binary semaphore, which should be released before being obtained by the task. If the task does not get a binary semaphore, it enters Blocked state until this binary semaphore is released and obtained.

8.2.2 Binary semaphore API

As shown in Table 15, FreeRTOS provides API functions of binary semaphore to be called by users in programs.

Table 15. Binary semaphore API

Binary semaphore API functions	
API	Description
<code>xSemaphoreCreateBinary()</code>	Create a binary semaphore
<code>xSemaphoreCreateBinaryStatic()</code>	Create a binary semaphore with static method
<code>vSemaphoreDelete()</code>	Delete a semaphore
<code>xSemaphoreGive()</code>	Release a semaphore
<code>xSemaphoreGiveFromISR()</code>	Release a semaphore (called from an interrupt)
<code>xSemaphoreTake()</code>	Obtain a semaphore
<code>xSemaphoreTakeFromISR()</code>	Obtain a semaphore (called from an interrupt)

These API functions are called to create, delete, release and obtain binary semaphores.

Note: These release, obtain and delete API functions are commonly used by binary semaphores, counting semaphores and recursive semaphores.

xSemaphoreCreateBinary ();

Description:

Create a binary semaphore.

Prototype:

Table 16. xSemaphoreCreateBinary ()

SemaphoreHandle_t xSemaphoreCreateBinary(void);	
Parameter	Description
Void	None

Return value:

Handle of the created binary semaphore

vSemaphoreDelete ();

Description:

Delete a semaphore.

Prototype:

Table 17. vSemaphoreDelete ()

void vSemaphoreDelete(SemaphoreHandle_t xSemaphore);	
Parameter	Description
xSemaphore	Handle of the semaphore being deleted

Return value:

None.

xSemaphoreGive ();

Description:

Release a semaphore.

Prototype:

Table 18. xSemaphoreGive ()

BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore);	
Parameter	Description
xSemaphore	Handle of the semaphore being released

Return value:

Semaphore release success flag

Note: xSemaphoreGiveFromISR() is called to release a semaphore by an interrupt.

xSemaphoreTake ();

Description:

Obtain a semaphore.

Prototype:

Table 19. xSemaphoreTake ()

BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);	
Parameter	Description
xSemaphore	Handle of the semaphore being taken
xTicksToWait	The time in ticks to wait for the semaphore to become available.

Return value:

Semaphore take success flag

Note: xSemaphoreTakeFromISR () is called to obtain a semaphore by an interrupt.

8.2.3 Routine

Project name: 07Binary_Semaphore_FreeRTOS

Program source code:

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

/* start task priority */
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE       128
/* start task handle */
TaskHandle_t StartTask_Handler;
/* start task entry function */
void start_task(void *pvParameters);

/* message processing task priority */
#define Process_Binary_Semaphore_TASK_PRIO      1
/* message processing task stack size */
#define Process_Binary_Semaphore_STK_SIZE       256
/* message processing task handle */
TaskHandle_t Process_Binary_Semaphore_Task_Handler;
/* message processing task entry function */
void Process_Binary_Semaphore_task(void *pvParameters);

/* debugging task priority */
#define Debug_TASK_PRIO      3
```

```

/* debugging task stack size*/
#define Debug_STK_SIZE      512
/* debugging task handle */
TaskHandle_t DebugTask_Handler;
/* debugging task entry function */
void debug_task(void *pvParameters);

/* define semaphore */
SemaphoreHandle_t AT_xSemaphore;

/* define a semaphore and increment by 1 after the binary semaphore is received */
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);

    /* create start task */
    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);

    /* enable task scheduler */
    vTaskStartScheduler();
}

/* start task function*/
void start_task(void *pvParameters)
{
    /* enter critical code region */
    taskENTER_CRITICAL();

```

```

/* create a binary semaphore */
AT_xSemaphore = xSemaphoreCreateBinary();

if( AT_xSemaphore == NULL )
{
    /* binary semaphore creation failure */
    while(1);
}

/* initialize the timer after the queue is created */
TIMER_Init();

/* create message processing task */
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);

/* create debugging task */
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);

/* delete start task */
vTaskDelete(StartTask_Handler);

/* exit critical code region */
taskEXIT_CRITICAL();
}

/* message processing task function */
void Process_Binary_Semaphore_task(void *pvParameters)
{
    while(1)
    {
        /* wait to receive binary semaphore */
        if( xSemaphoreTake( AT_xSemaphore, portMAX_DELAY ) == pdTRUE )
        {
            /* increment by 1 */
            Take_Semaphore_Counter++;
            printf("The %dth semaphore is received.\r\n",Take_Semaphore_Counter);
        }
    }
}

```

```

    }
}
/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out debugging information */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/*-----*/\n");
            printf("Task          Status          priority      Remaining_Stack      Num\n");
            vTaskList((char *)&buff);
            printf("%s\n",buff);
            printf("/*-----*/\n");
            printf("Task          Runing_Num      Usage_Rate\n");
            vTaskGetRunTimeStats((char *)&buff);
            printf("%s\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.\n",Take_Semaphore_Counter+1);
        /* send a semaphore */
        xSemaphoreGiveFromISR( AT_xSemaphore, &xHigherPriorityTaskWoken );

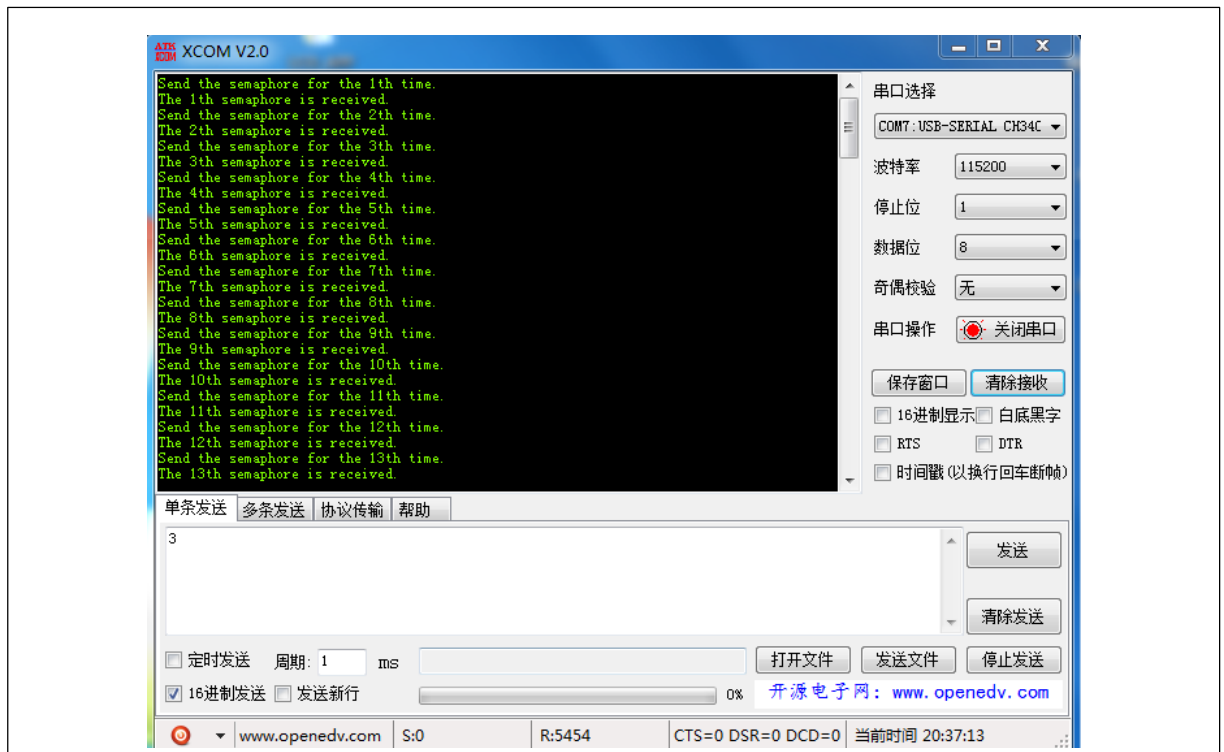
        /* check whether or not to switch the task */
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
        tmr_flag_clear(TMR3,TMR_OVF_FLAG);
    }
}

```

In this routine, the synchronization between a task and an interrupt is implemented. When a timer interrupt is generated, a binary semaphore is released in the interrupt handler, and the blocked task waiting for this binary semaphore enters Ready state to run. After the task runs and consumes this binary semaphore, it enters Blocked state and wait for a binary semaphore. This program also has a debugging information print task, and users can press the on-board USER button to print out all task information through serial port.

Compile and download the program to the target board, and the execution is as follows:

Figure 25. Binary semaphore routine



The print result shows that each time a binary semaphore is released in the timer interrupt function, the waiting task runs once.

Figure 26. Binary semaphore routine



Press the USER button to print out the information of all tasks in the current system, as shown in Figure 26.

8.3 Counting semaphore

8.3.1 Introduction

This section introduces counting semaphores. Just as binary semaphores can be thought of as queues of length zero or one, counting semaphores can be thought of as queues of length greater than one. Again, users of the semaphore are not interested in the data that is stored in the queue - just whether or not the queue is empty or not.

Counting semaphores are typically used for two things:

1. Counting events

In this usage scenario, an event handler will give a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will take a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the count value to be zero when the semaphore is created.

2. Resource management

In this usage scenario, the counting semaphore indicates the number of resources available (for example, the remaining available parking spots). The semaphore is decremented by 1 each time a task takes the resource (the car drives to the park) or incremented by 1 each time the task releases the resource (the car leaves the park). The initial semaphore represents the total resources. For example, if there are 100 parking spots, the initial value is semaphore is 100.

8.3.2 Counting semaphore API

Table 20 lists the counting semaphore API functions.

Table 20. Counting semaphore API

Counting semaphore API functions	
API	Description
xSemaphoreCreateCounting()	Create a counting semaphore
xSemaphoreCreateCountingStatic()	Create a counting semaphore with static method
vSemaphoreDelete()	Delete a semaphore
xSemaphoreGive()	Release a semaphore
xSemaphoreGiveFromISR()	Release a semaphore (called from an interrupt)
xSemaphoreTake()	Obtain a semaphore
xSemaphoreTakeFromISR()	Obtain a semaphore (called from an interrupt)
uxSemaphoreGetCount()	Get the current semaphore count

These API functions are called to create, delete, release and obtain counting semaphores.

Note: These release, obtain and delete API functions are commonly used by binary semaphores, counting semaphores and recursive semaphores.

xSemaphoreCreateCounting ();

Description:

Create a counting semaphore.

Prototype:

Table 21. xSemaphoreCreateCounting ()

SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount, UBaseType_t uxInitialCount);	
Parameter	Description
uxMaxCount	The maximum count value that can be reached.
uxInitialCount	The count value assigned to the semaphore when it is created.

Return value:

Handle of the created counting semaphore

vSemaphoreDelete ();

Description:

Delete a semaphore.

Prototype:

Table 22. vSemaphoreDelete ()

void vSemaphoreDelete(SemaphoreHandle_t xSemaphore);	
Parameter	Description
xSemaphore	Handle of the semaphore being deleted

Return value:

None.

xSemaphoreGive ();

Description:

Release a semaphore.

Prototype:

Table 23. xSemaphoreGive ()

BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore);	
Parameter	Description
xSemaphore	Handle of the semaphore being released

Return value:

Semaphore release success flag

Note: xSemaphoreGiveFromISR() is called to release a semaphore by an interrupt.

xSemaphoreTake ();

Description:

Obtain a semaphore.

Prototype:

Table 24. xSemaphoreTake ()

BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);	
Parameter	Description
xSemaphore	Handle of the semaphore being taken
xTicksToWait	The time in ticks to wait for the semaphore to become available.

Return value:

Semaphore take success flag

Note: xSemaphoreTakeFromISR () is called to obtain a semaphore by an interrupt.

uxSemaphoreGetCount ();

Description:

Get the count of a semaphore.

Prototype:

Table 25. uxSemaphoreGetCount ()

UBaseType_t uxSemaphoreGetCount(SemaphoreHandle_t xSemaphore);	
Parameter	Description
xSemaphore	Handle of the semaphore being queried

Return value:

The current count value of semaphores

8.3.3 Routine

Project name: 08Counting_Semaphore_FreeRTOS

Program source code:

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

/* start task priority */
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE       128
/* start task handle */
TaskHandle_t StartTask_Handler;
/* start task entry function */
```

```

void start_task(void *pvParameters);

/* message processing task priority */
#define Process_Counting_Semaphore_TASK_PRIO      1
/* message processing task stack size */
#define Process_Counting_Semaphore_STK_SIZE        256
/* message processing task handle */
TaskHandle_t Process_Counting_Semaphore_Task_Handler;
/* message processing task entry function */
void Process_Counting_Semaphore_task(void *pvParameters);

/* debugging task priority */
#define Debug_TASK_PRIO      3
/* debugging task stack size */
#define Debug_STK_SIZE      512
/* debugging task handle */
TaskHandle_t DebugTask_Handler;
/* debugging task entry function */
void debug_task(void *pvParameters);

/* define semaphore */
SemaphoreHandle_t AT_xSemaphore;
/* semaphore initial value */
uint8_t Semaphore_Initail_Vaule = 10;
/* semaphore maximum value */
uint8_t Semaphore_Max_Vaule = 10;

/* remaining semaphore count */
int Remaining_Semaphore = 0;

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);
    at32_button_init();
    /* init usart1 */
    uart_print_init(115200);
    /* create start task */
    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);
    /* enable task scheduler */
    vTaskStartScheduler();
}

```

```

}

/* start task function */
void start_task(void *pvParameters)
{
    /* enter critical code region */
    taskENTER_CRITICAL();

    /* create counting semaphore */
    AT_xSemaphore =
    xSemaphoreCreateCounting(Semaphore_Max_Vaule, Semaphore_Initail_Vaule);

    if( AT_xSemaphore == NULL )
    {
        /* counting semaphore creation failure */
        while(1);
    }

    /* initialize the timer after the queue is created */
    TIMER_Init();

    /* create message processing task */
    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);

    /* create debugging task */
    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);

    /* delete start task */
    vTaskDelete(StartTask_Handler);

    /* exit critical code region */
    taskEXIT_CRITICAL();
}

/* message processing task function */
void Process_Counting_Semaphore_task(void *pvParameters)
{

```

```

while(1)
{
    /* wait to receive counting semaphore */
    if( xSemaphoreTake( AT_xSemaphore, portMAX_DELAY ) == pdTRUE )
    {
        printf("Semaphore is received.\r\n");
        /* remaining counting semaphores */
        Remaining_Semaphore = uxSemaphoreGetCount(AT_xSemaphore);

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

/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out task information */
        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 interrupt function */
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");
        /* send a semaphore */
    }
}

```

```

xSemaphoreGiveFromISR( AT_xSemaphore, &xHigherPriorityTaskWoken );

/* check whether or not to switch task */
portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

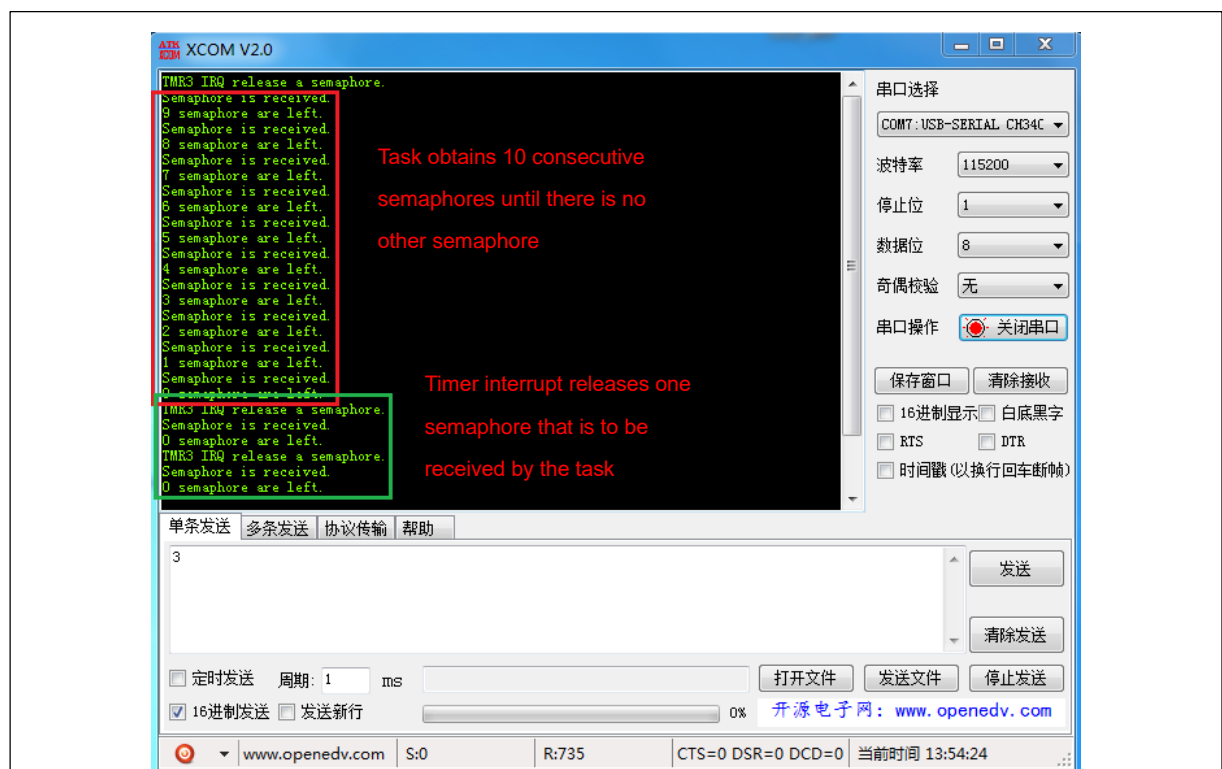
/* clear TMR3 interrupt flag */
TMR_ClearFlag(TMR3, TMR_FLAG_Update);
}
}

```

In this program, a counting semaphore is created, and both the maximum and initial semaphore counts are set to 10; then a semaphore obtain task is created to obtain semaphores continuously. The initial semaphore count is 10; therefore, the task obtains 10 semaphores continuously. After all semaphores are obtained, the task enters Blocked state to wait for available semaphores. Initialize the hardware timer in this program, and a semaphore is released in the timer interrupt handler. After this semaphore is obtained, the blocked task enters Ready state to run. If there is no other semaphore to be obtained by the task, the task enters Blocked state to wait for the semaphore. The operation result can be print out through serial port.

Compile and download the program to the target board, and the execution is as follows:

Figure 27. Counting semaphore routine



The print information shows that the operation consistent with the program logic, and the task consumes all semaphores and then enters Blocked state to wait for available semaphores.

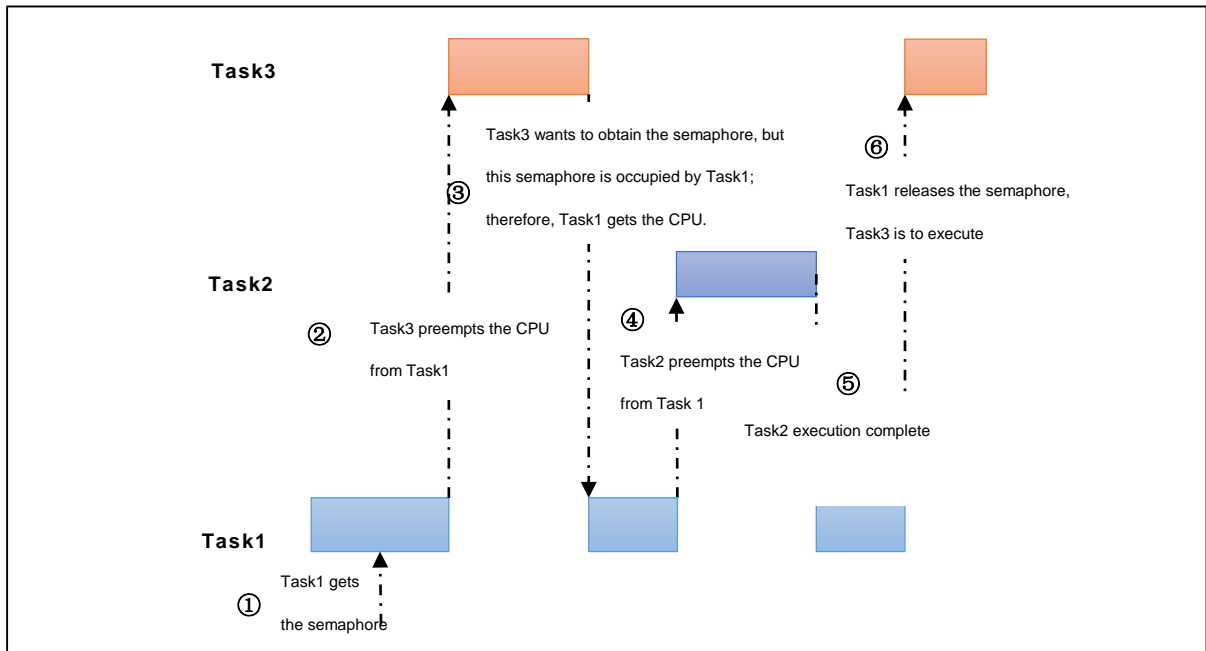
8.4 Mutex

8.4.1 Priority inversion

Priority inversion is a common pitfall in preemptive kernel, which is not allowed in real-time operating systems. Priority inversion disrupts the intended execution sequence, causing low priority tasks to run before high priority tasks.

Priority inversion may occur in the following situations:

Figure 28. Priority inversion



As shown in Figure 28,

- 1) Task1 is running and it gets the binary semaphore.
- 2) Task3 enters Ready state and has the highest priority; therefore, it preempts the CPU from Task1, while the binary semaphore is occupied by Task1.
- 3) Task3 wants to obtain the binary semaphore, but Task1 does not release the semaphore; therefore, Task1 gets the CPU, and Task3 enters Blocked state and wait for the release of semaphore.
- 4) Task1 is running, while Task2 enters Ready state and preempts the CPU because it has a higher priority over Task1.
- 5) Task2 is completed and then releases CPU. At this point, Task3 is waiting for a binary semaphore, so that it cannot run; therefore, Task1 gets the CPU, causing priority inversion. Although Task3 has the highest priority, it is waiting for a binary semaphore; therefore, its priority lowers to be the same as the priority of Task1; meanwhile, Task2 is running, which leads to a worse situation.

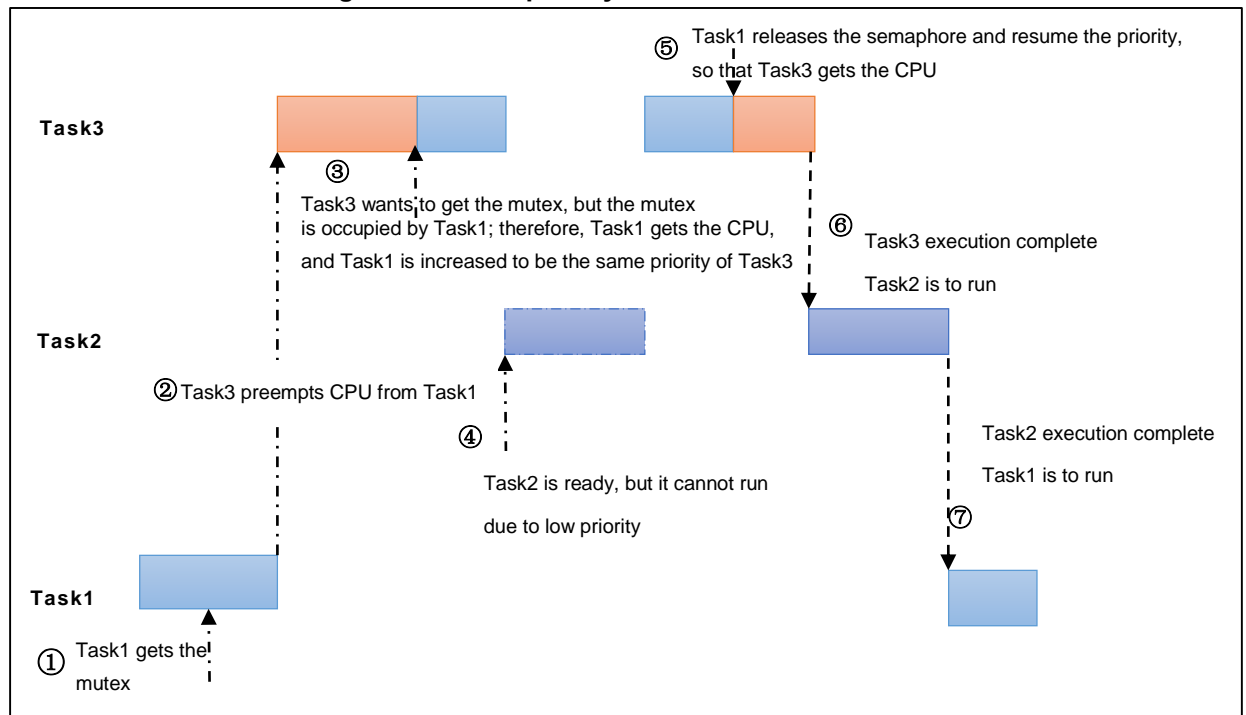
Such priority inversion is often fatal and irreversible in some applications with high real-time requirements. Therefore, mutexes are used to avoid priority inversion.

8.4.2 Introduction

Mutexes are designed to implement mutual exclusive access to resources. Both mutexes and binary semaphores support mutual exclusive access to resources, while priority inversion may occur when binary semaphores are used. If mutexes are, when one high priority task wants to get the mutex occupied by a low priority task, this high priority task enters Blocked state (before that, the low priority task that occupies the mutex is increased to the same priority level of the high priority task).

In this way, the low priority task occupying the mutex is not preempted by other medium priority tasks, which reduces the time to respond to high priority task as far as possible. Although the high priority task does not get the CPU and obtain the mutex from low priority task immediately, the latency in response is as low as possible, which basically meets real-time requirements.

Figure 29. Solve priority inversion with mutexes



As shown in Figure 29,

- 1) Task1 gets the CPU and obtains mutex.
- 2) Task3 preempts the CPU from Task1 (Task3 has the highest priority), and then starts to run.
- 3) Task3 wants to get the mutex during running, but this mutex is occupied by Task1. At this time, Task3 increases the Task1 to its own high priority level, which means that Task1 starts to run as soon as Task3 releases the CPU;
- 4) Task2 enters Ready state from Blocked state; at this point, Task1 has a higher priority than Task2; therefore, Task2 cannot get the CPU but to wait, and Task1 continues running.
- 5) Task1 releases the mutex during running, and then the kernel resumes Task1 low priority (lower than Task2); then Task2 gets the mutex and starts to run.
- 6) Task3 is completed and releases the CPU; then, Task2 starts to run.
- 7) After Task2 is completed, there is no task with priority higher than Task1; therefore, Task1

continues to run.

Mutexes are used to minimize latency in response and improves the system real-time performance.

Note: Mutexes cannot be used by interrupt handler.

8.4.3 Mutex API

Table 26 lists the mutex API functions.

Table 26. Mutex API

Mutex API functions	
API	Description
xSemaphoreCreateMutex()	Create a mutex
xSemaphoreCreateMutexStatic()	Create a mutex with static method
vSemaphoreDelete()	Delete a mutex
xSemaphoreGive()	Release a mutex
xSemaphoreTake()	Obtain a mutex
xSemaphoreGetMutexHolder()	Obtain the handle of the task that holds the mutex

These API functions are called to create, delete, release and obtain mutexes.

Note: These release, obtain and delete API functions are commonly used by binary semaphores, counting semaphores and recursive semaphores.

xSemaphoreCreateMutex();

Description:

Create a mutex.

Prototype:

Table 27. xSemaphoreCreateMutex()

SemaphoreHandle_t xSemaphoreCreateMutex(void);	
Parameter	Description
Null	None

Return value:

Handle of the created mutex

vSemaphoreDelete ();

Description:

Delete a mutex.

Prototype:

Table 28. vSemaphoreDelete ()

void vSemaphoreDelete(SemaphoreHandle_t xSemaphore);	
Parameter	Description
xSemaphore	Handle of the mutex being deleted

Return value:

None.

xSemaphoreGive ();

Description:

Release a mutex.

Prototype:

Table 29. xSemaphoreGive ()

BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore);	
Parameter	Description
xSemaphore	Handle of the mutex being released

Return value:

Mutex release success flag

xSemaphoreTake ();

Description:

Obtain a mutex.

Prototype:

Table 30. xSemaphoreTake ()

BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);	
Parameter	Description
xSemaphore	Handle of the mutex being obtained
xTicksToWait	The time in ticks to wait for the mutex to become available.

Return value:

Mutex take success flag

uxSemaphoreGetCount ();

Description:

Obtain the count of a mutex.

Prototype:

Table 31. uxSemaphoreGetCount ()

UBaseType_t uxSemaphoreGetCount(SemaphoreHandle_t xSemaphore);	
Parameter	Description
xSemaphore	Handle of the mutex being queried

Return value:

The mutex current count value

8.4.4 Routine

Project name: 09Mutex_Semaphore_FreeRTOS

Program source code:

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

/* start task priority */
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE      128
/* start task handle */
TaskHandle_t StartTask_Handler;
/* start task entry function */
void start_task(void *pvParameters);

/* high-priority task priority */
#define High_Priority_TASK_PRIO      5
/* high-priority task stack size */
#define High_Priority_STK_SIZE      256
/* high-priority task handle */
TaskHandle_t High_Priority_Task_Handler;
/* high-priority task entry function */
void High_Priority_task(void *pvParameters);

/* medium-priority task priority */
#define Middle_Priority_TASK_PRIO      4
/* medium-priority task stack size */
#define Middle_Priority_STK_SIZE      256
/* medium-priority task handle */
TaskHandle_t Middle_Priority_Task_Handler;
/* medium-priority task entry function */
void Middle_Priority_task(void *pvParameters);

/* low-priority task priority */
#define Low_Priority_TASK_PRIO      3
/* low-priority task stack size */
#define Low_Priority_STK_SIZE      256
/* low-priority task handle */
TaskHandle_t Low_Priority_Task_Handler;
/* low-priority task entry function */
void Low_Priority_task(void *pvParameters);

/* debugging task priority */
```

```

#define Debug_TASK_PRIO      3
/* debugging task stack size*/
#define Debug_STK_SIZE      512
/* debugging task handle */
TaskHandle_t DebugTask_Handler;
/* debugging task entry function */
void debug_task(void *pvParameters);

/* define semaphore */
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);

    /* create start task */
    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);

    /* enable task scheduler */
    vTaskStartScheduler();
}

/* start task function */
void start_task(void *pvParameters)
{
    /* enter critical code region */
    taskENTER_CRITICAL();

    /* create mutex */
    AT_xSemaphore = xSemaphoreCreateMutex();

```

```

if( AT_xSemaphore == NULL )
{
    /* mutex creation failure */
    while(1);
}

/* initialize the timer after the mutex is created */
TIMER_Init();

/* create high priority task */
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);

/* create medium priority task */
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);

/* create low priority task */
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);

/* create debugging task */
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);

/* delete start task */
vTaskDelete(StartTask_Handler);

/* exit critical code region */
taskEXIT_CRITICAL();
}

/* high priority task function */
void High_Priority_task(void *pvParameters)

```

```
{

while(1)
{
    /* delay occurs; high priority task cannot get the mutex */
    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)
    {
        /* low priority task gets the mutex */
        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");
            /* medium priority task enters Ready state */
            vTaskResume(Middle_Priority_Task_Handler);
            /* set timer interrupt flag to simulate ordinary delay */
            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 );
        }
    }
}
```

```

    }
}

}
/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out the debugging information */
        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);
    }
}

```

This program simulates mutexes to avoid priority inversion. There are three tasks with different priorities. The program creates mutexes and these three tasks in sequence. In order to avoid the high priority task from getting the mutex at the very beginning, a necessary delay is used to switch the task, which means that medium priority task is to run. The medium priority task suspends itself in its task function to enter Blocked state, and then the low priority task is to run. The low priority task gets the mutex and then release the medium priority task to enter Ready state. At this point, the medium priority task will run immediately if no mutex is used. But in this scenario, a mutex is used and therefore the medium priority task cannot run. Once the low priority task delays and releases the mutex, the previously blocked high priority task will run immediately. The high priority task will get the mutex, and then the medium priority task runs once and then enters Blocked state. After that, neither the low nor high priority task cannot get the mutex; therefore, none of these three tasks will run again.

Compile and download the program to the target board, and the execution is as follows:

Figure 30. Mutex routine



The print information shows that the operation consistent with the program logic, and the routine demonstrates advantages of using mutexes in systems with high real-time requirements.

8.5 Recursive mutex

8.5.1 Introduction

Recursive mutexes are regarded as special mutexes. A mutex obtained by a task cannot be taken repeatedly, while a mutex used recursively can be taken repeatedly in one task. It should be noted that once a recursive mutex is obtained by a task, it cannot be taken by any other tasks.

When a task gets a mutex to access a common resource, this resource is owned by this task. To access this common resource in the task function, users need to get this mutex again. However, this mutex cannot be obtained, causing deadlock of task. In this case, a recursive mutex can be used to solve this problem.

Note: Recursive mutexes cannot be used in interrupt handler.

8.5.2 Recursive mutex API

Table 32 lists the recursive mutex API functions.

Table 32. Recursive mutex API

Recursive mutex API functions	
API	Description
xSemaphoreCreateRecursiveMutex()	Create a recursive mutex
xSemaphoreCreateRecursiveMutexStatic()	Create a recursive mutex with static method
vSemaphoreDelete()	Delete a recursive mutex

xSemaphoreGiveRecursive()	Release a a recursive mutex
xSemaphoreTakeRecursive()	Obtain a a recursive mutex

These API functions are called to create, delete, release and obtain recursive mutexes.

xSemaphoreCreateRecursiveMutex();

Description:

Create a recursive mutex.

Prototype:

Table 33. xSemaphoreCreateRecursiveMutex()

SemaphoreHandle_t xSemaphoreCreateRecursiveMutex(void);	
Parameter	Description
Null	Note

Return value:

Handle of the created recursive mutex

xSemaphoreGiveRecursive();

Description:

Release a recursive mutex.

Prototype:

Table 34. xSemaphoreGiveRecursive()

BaseType_t xSemaphoreGiveRecursive(SemaphoreHandle_t xMutex);	
Parameter	Description
xMutex	Handle of the recursive mutex being released

Return value:

Recursive mutex release success flag

xSemaphoreTakeRecursive();

Description:

Obtain a recursive mutex.

Prototype:

Table 35. xSemaphoreTakeRecursive()

BaseType_t xSemaphoreTakeRecursive(SemaphoreHandle_t xMutex, TickType_t xTicksToWait);	
Parameter	Description
xMutex	Handle of the mutex being obtained
xTicksToWait	The time in ticks to wait for the semaphore to become available.

Return value:

Semaphore take success flag

8.5.3 Routine

Project name: 10Recursive_Mutex_Semaphore_FreeRTOS

Program source code:

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

/* start task priority */
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE       128
/* start task handle */
TaskHandle_t StartTask_Handler;
/* start task entry function */
void start_task(void *pvParameters);

/* ordinary task priority */
#define Ordinary_TASK_PRIO    3
/* ordinary task stack size */
#define Ordinary_STK_SIZE     256
/* ordinary task handle */
TaskHandle_t Ordinary_Task_Handler;
/* ordinary task entry function */
void Ordinary_task(void *pvParameters);

/* recursive task priority */
#define Recursive_TASK_PRIO   3
/* recursive task stack size */
#define Recursive_STK_SIZE    256
/* recursive task handle */
TaskHandle_t Recursive_Task_Handler;
/* recursive task entry function */
void Recursive_task(void *pvParameters);

/* debugging task priority */
#define Debug_TASK_PRIO       4
/* debugging task stack size */
#define Debug_STK_SIZE        512
/* debugging task handle */
TaskHandle_t DebugTask_Handler;
/* debugging task entry function */
void debug_task(void *pvParameters);

/* define a semaphore */
```

```

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);

    /* create start task */
    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);

    /* enable task scheduler */
    vTaskStartScheduler();
}

/* start task function */
void start_task(void *pvParameters)
{
    /* enter critical code region */
    taskENTER_CRITICAL();

    /* create a recursive mutex */
    AT_xSemaphore = xSemaphoreCreateRecursiveMutex();

    if( AT_xSemaphore == NULL )
    {
        /* recursive mutex creation failure */
        while(1);
    }

    /* initialize the timer after the recursive mutex is created */
    TIMER_Init();
}

```

```

/* create recursive task */
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);

/* create ordinary task */
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);

/* create debugging task */
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);

/* delete start task */
vTaskDelete(StartTask_Handler);

/* exit critical code region */
taskEXIT_CRITICAL();
}

/* ordinary task function */
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");
        }
    }
}

```

```

}
/* recursive task function */
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);
    }
}
/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out the debugging information */
        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");
        }
    }
}

```

```

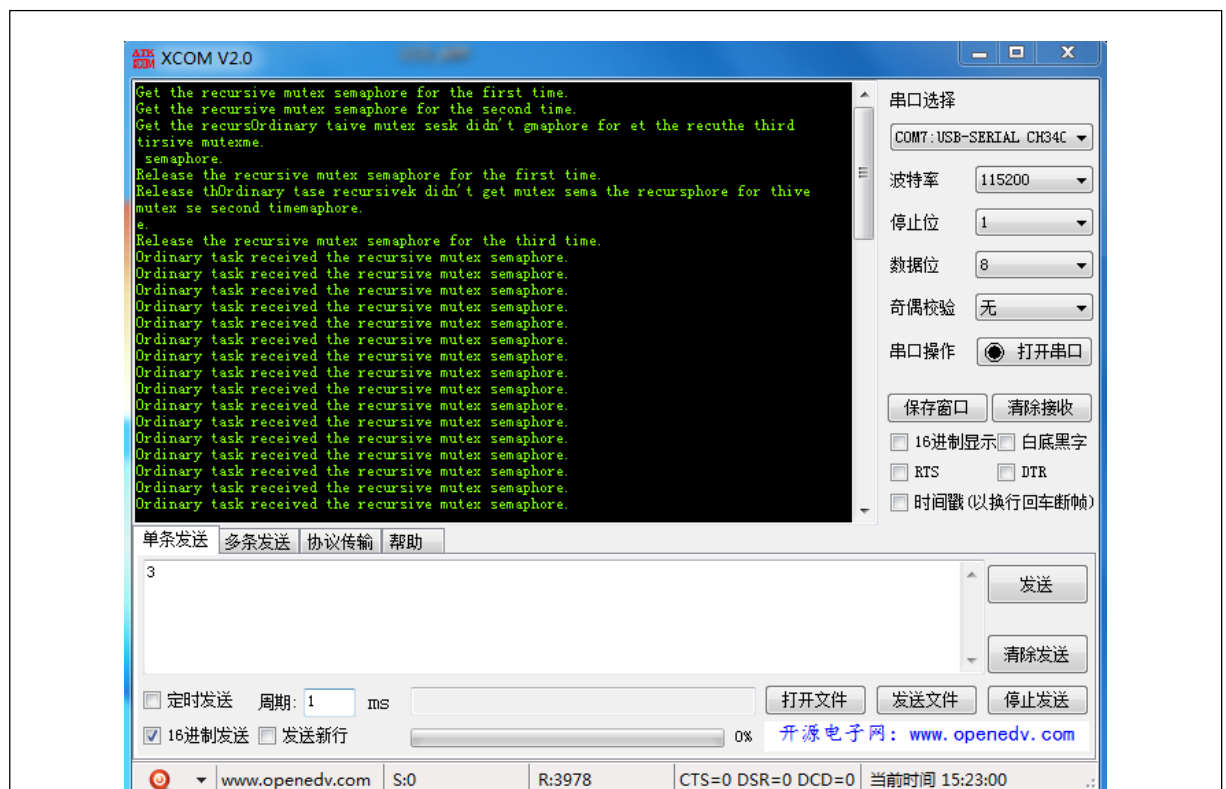
printf("Task      Runing_Num      Usage_Rate\r\n");
vTaskGetRunTimeStats((char *)&buff);
printf("%s\r\n",buff);
}
vTaskDelay(10);
}
}

```

In this program, a recursive mutex and two tasks are created. The mutex is obtained for three times recursively in the Recursive_task, and the Ordinary_task can only obtain the recursive mutex after this mutex is released by Recursive_task for three times.

Compile and download the program to the target board, and the execution is as follows:

Figure 31. Recursive mutex routine



The print result shows that the Recursive_task firstly obtains a mutex recursively for three times, and this mutex cannot be obtained by Ordinary_task only after it is released by Recursive_task for three times continuously. This result conforms to the program design and characteristics of recursive mutexes.

Note: The disorder in printing is a normal phenomenon caused by the scheduling of Recursive_task and Ordinary_task.

9 FreeRTOS event groups/flags

This section introduces FreeRTOS event groups (flags), which is a kernel service widely used for task synchronization.

9.1 Introduction

Provided by FreeRTOS kernel, the event group is one of the effective mechanisms to realize multi-task synchronization. An event group is a set of event bits. The kernel manages a variable in which different bits can be configured by tasks to realize synchronization. For example, if a task only can execute the subsequent task codes after both BIT0 and BIT1 are set to 1, a special task is required to set these two bits. The waiting task runs once both bits are set, thus realizing inter-task synchronization.

In bare-metal systems, the global variable is easy-to-use and preferred; while in RTOS systems, the following aspects should be considered when using global variable:

- 1) Using event groups allows RTOS kernel to manage tasks efficiently, which cannot be realized by using global variable (task timeout needs to be implemented by users).
- 2) Using global variable may cause access conflict of multiple tasks.
- 3) Using event groups can deal with the synchronization between the interrupt service routine and tasks.

Figure 32. Block diagram of event group

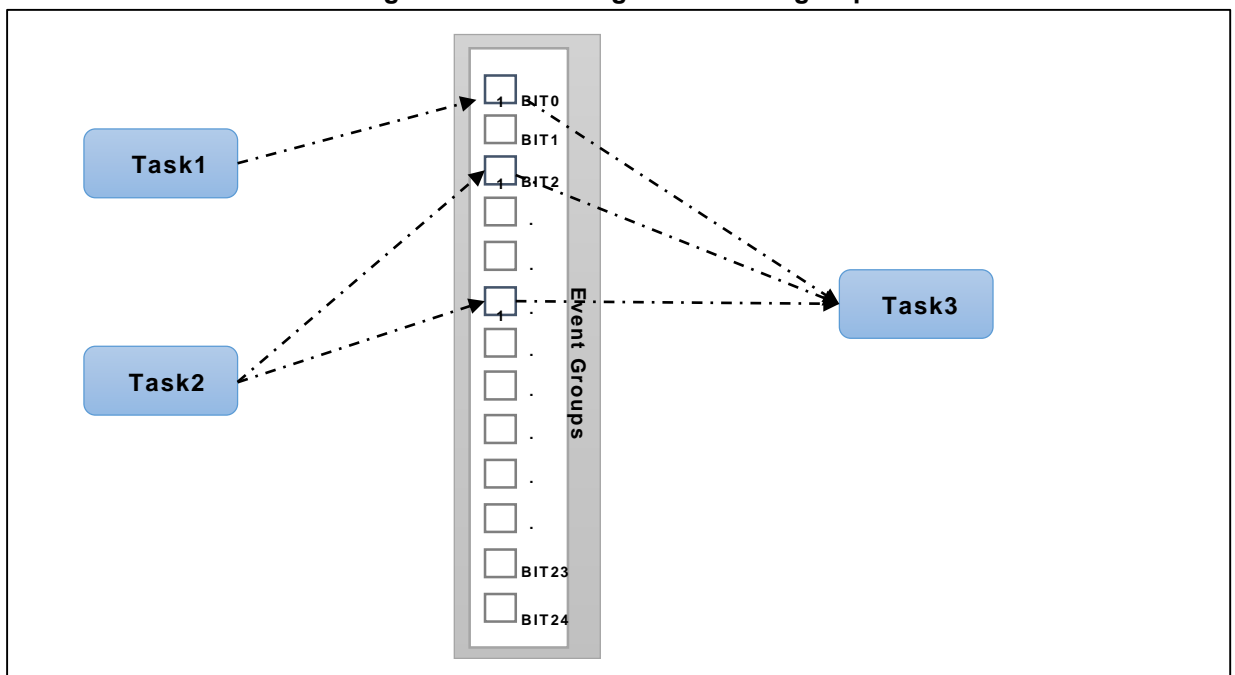


Figure 32 shows the operation process of event groups. Task1 and Task2 set the corresponding bits in the event group. After bits are set as required by Task3, Task3 gets the event group and clear the corresponding bits.

9.2 Event group API

Table 36 lists the event group API functions.

Table 36. Event group API

Event group API functions	
API	Description
xEventGroupClearBits()	Clear the bit within an event group
xEventGroupClearBitsFromISR()	Clear the bit within an event group (called from an interrupt)
xEventGroupCreate()	Create an event group
xEventGroupCreateStatic()	Create an event group with static method
vEventGroupDelete()	Delete an event group
xEventGroupGetBits()	Get the current value of the event bits
xEventGroupGetBitsFromISR()	Get the current value of the event bits (called from an interrupt)
xEventGroupSetBits()	Set bits within an event group
xEventGroupSetBitsFromISR()	Set bits within an event group (called from an interrupt)
xEventGroupSync()	Set bits within an event group and then wait for a combination of bits to be set within the same event group (synchronization)
xEventGroupWaitBits()	Wait for a bit of an event group to be set

xEventGroupCreate();

Description:

Create a new event group.

Prototype:

Table 37. xEventGroupCreate()

EventGroupHandle_t xEventGroupCreate(void);	
Parameter	Description
Null	None

Return value:

Handle of the created event group

xEventGroupSetBits();

Description:

Set bits within an event group.

Prototype:

Table 38. xEventGroupSetBits()

EventBits_t xEventGroupSetBits(EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet);	
--	--

Parameter	Description
xEventGroup	Handle of the event group
uxBitsToSet	Bits to be set

Return value:

Value of the event group

xEventGroupWaitBits();

Description:

Wait for a bit of an event group to be set

Prototype:

Table 39. xEventGroupWaitBits()

EventBits_t xEventGroupWaitBits(const EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToWaitFor, const BaseType_t xClearOnExit, const BaseType_t xWaitForAllBits, TickType_t xTicksToWait);	
Parameter	Description
xEventGroup	Handle of an event group
uxBitsToWaitFor	A bitwise value that indicates the bit or bits to test inside the event group
xClearOnExit	Whether or not all waiting bits on return
xWaitForAllBits	Create either a logical AND test (where all bits must be set) or a logical OR test (where one or more bits must be set)
xWaitForAllBits	The maximum amount of time (specified in 'ticks') to wait for one/all of the bits to become set

Return value:

Value of the event group

9.3 Routine

Project name: 11Event_Groups_FreeRTOS

Program source code:

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

/* start task priority */
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE       128
/* start task handle */
```

```

TaskHandle_t StartTask_Handler;
/* start task entry function */
void start_task(void *pvParameters);

/* event group task1 priority */
#define EventGroup1_TASK_PRIO      3
/* event group task1 stack size */
#define EventGroup1_STK_SIZE      256
/* event group task1 handle */
TaskHandle_t EventGroup1Task_Handler;
/* event group task1 entry function */
void EventGroup1_task(void *pvParameters);

/* event group task2 priority */
#define EventGroup2_TASK_PRIO      3
/* event group task2 stack size */
#define EventGroup2_STK_SIZE      256
/* event group task2 handle */
TaskHandle_t EventGroup2Task_Handler;
/* event group task2 entry function */
void EventGroup2_task(void *pvParameters);

/* event group receive task priority */
#define EventGroup_Receive_TASK_PRIO      4
/* event group receive task stack size */
#define EventGroup_Receive_STK_SIZE      256
/* event group receive task handle */
TaskHandle_t EventGroup_ReceiveTask_Handler;
/* event group receive task entry function */
void EventGroup_Receive_task(void *pvParameters);

/* debugging task priority */
#define Debug_TASK_PRIO      5
/* debugging task stack size */
#define Debug_STK_SIZE      512
/* debugging task handle */
TaskHandle_t DebugTask_Handler;
/* debugging task entry function */
void debug_task(void *pvParameters);

/* BIT sent by task*/
#define TASK_0_BIT ( 1 << 0 )
#define TASK_1_BIT ( 1 << 1 )
#define TASK_2_BIT ( 1 << 2 )

```

```

/* task synchronization flag */
#define ALL_SYNC_BITS ( TASK_0_BIT | TASK_1_BIT | TASK_2_BIT )

/* define an event group */
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);

    /* create start task */
    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);

    /* enable task scheduler */
    vTaskStartScheduler();
}

/* start task function */
void start_task(void *pvParameters)
{
    /* enter critical code region */
    taskENTER_CRITICAL();

    /* create an event group */
    AT_xEventBits = xEventGroupCreate();

    if(AT_xEventBits == NULL)
    {
        /* event group creation failure */
        while(1);
    }
}

```

```

/* initialize the timer after an event group is created */
TIMER_Init();
/* create event group task1 */
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);
/* create event group task2 */
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);
/* create event group receive task */
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);
/* create debugging task */
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);

/* delete start task */
vTaskDelete(StartTask_Handler);
/* exit critical code region */
taskEXIT_CRITICAL();
}
/* event group task1 function */
void EventGroup1_task(void *pvParameters)
{
    EventBits_t uxBits;

    while(1)
    {
        /* set BIT0&BIT2 within the event group */
        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");
}

}
}
/* event group task2 function */
void EventGroup2_task(void *pvParameters)
{
    EventBits_t uxBits;

    while(1)
    {
        vTaskDelay(1500);
        /* set BIT1 within the event group */
        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");
        }
    }
}

```

```

/* event group receive task function */
void EventGroup_Receive_task(void *pvParameters)
{
    EventBits_t uxBits;

    while(1)
    {
        /* receive ALL_SYNC_BITS to synchronize tasks */
        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");
    }
}

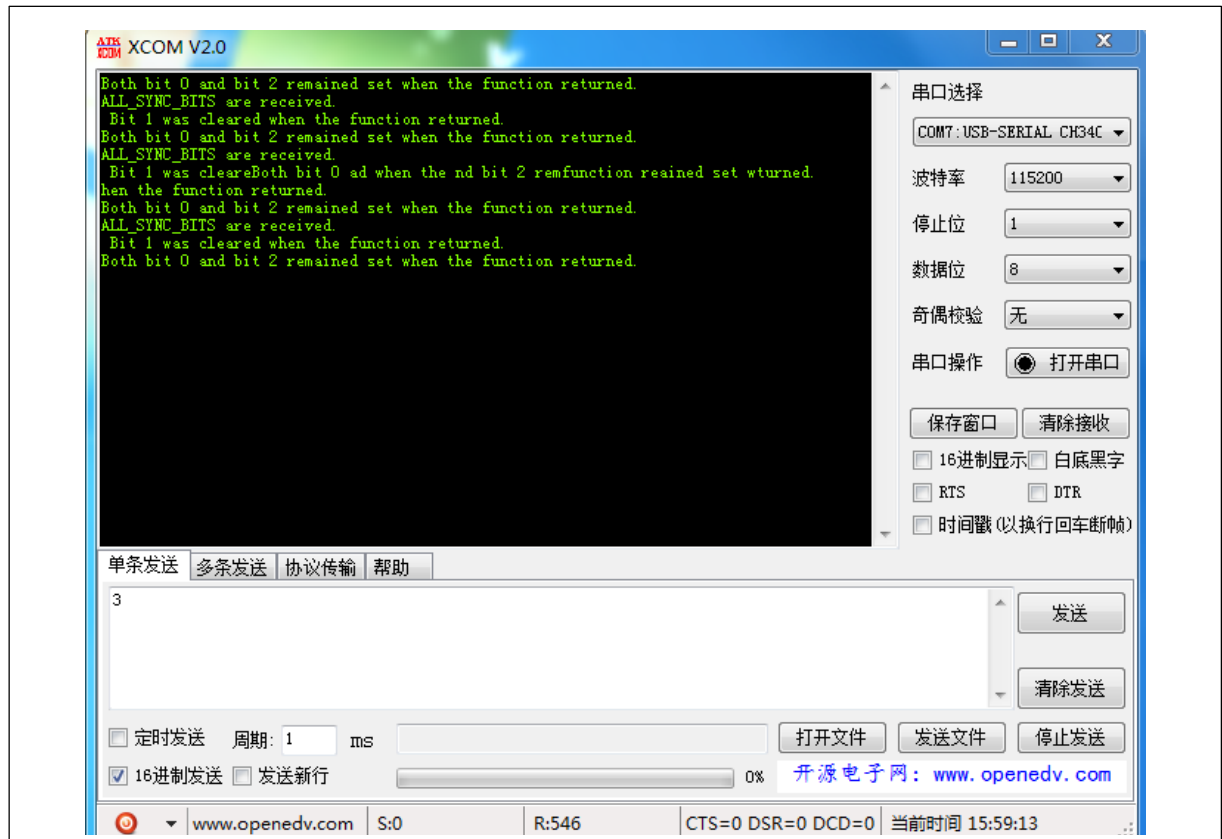
/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out debugging information */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/*-----*/\r\n");
            printf("Task          Status          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);
    }
}

```

In this program, an event group is defined. EventGroup1_task and EventGroup2_task set the corresponding bits by calling the xEventGroupSetBits, and EventReceive_task waits for the corresponding bit to become set. After all bits are set as required, EventReceive_task enters Ready state to run. This routine realize multi-task synchronization. In addition, users can press the USER button to print out the system task information.

Compile and download the program to the target board, and the execution is as follows:

Figure 33. Event group routine



As shown in the print result, EventGroup1_task sets BIT0 and BIT2, and then the EventReceive_task runs immediately (in fact, EventGroup2_task has run and set BIT2 already, and the kernel switches to the EventReceive_task as soon as BIT2 is set). EventGroup2_task runs after EventReceive_task. Therefore, BIT1 is cleared (by EventReceive_task) when the function returned.

10 FreeRTOS software timers

This section introduces FreeRTOS software timer group, which is a kernel service, and multiple software timers can be configured.

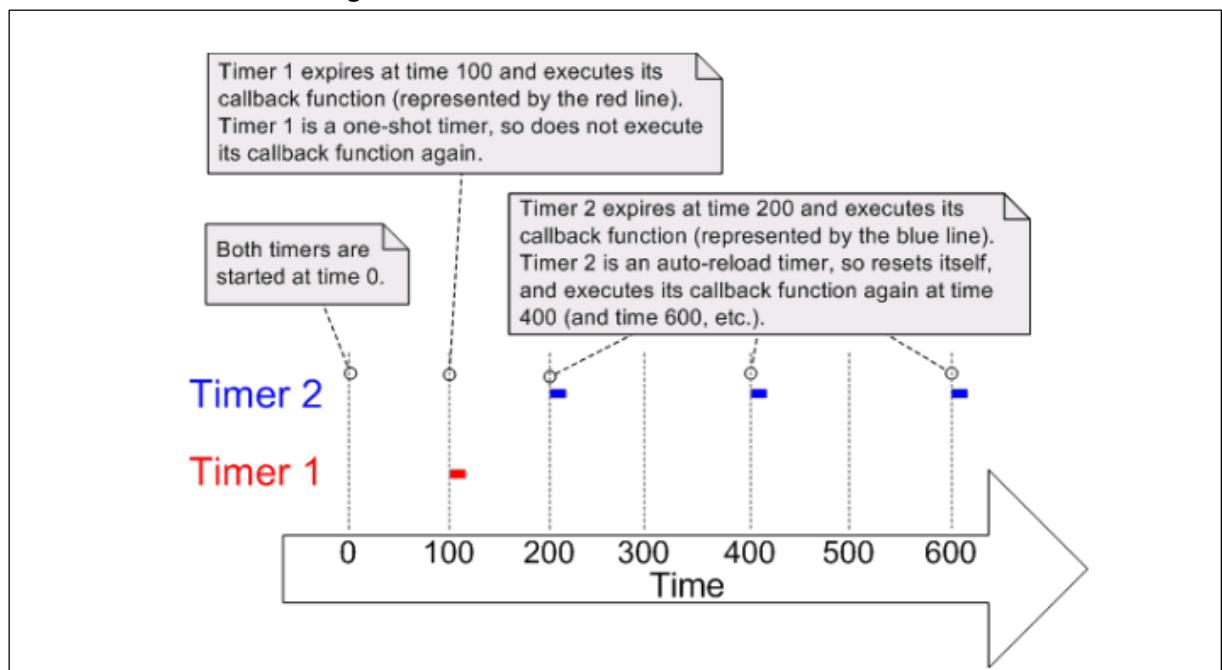
10.1 Introduction

The time base of FreeRTOS software timer is realized based on the system clock tick. The implementation of software timers does not require any hardware timer, and multiple software timers can be created.

For hardware timers, features are implemented within the timer interrupt; while for software timers, features are implemented within the callback function that is determined when creating the software timer.

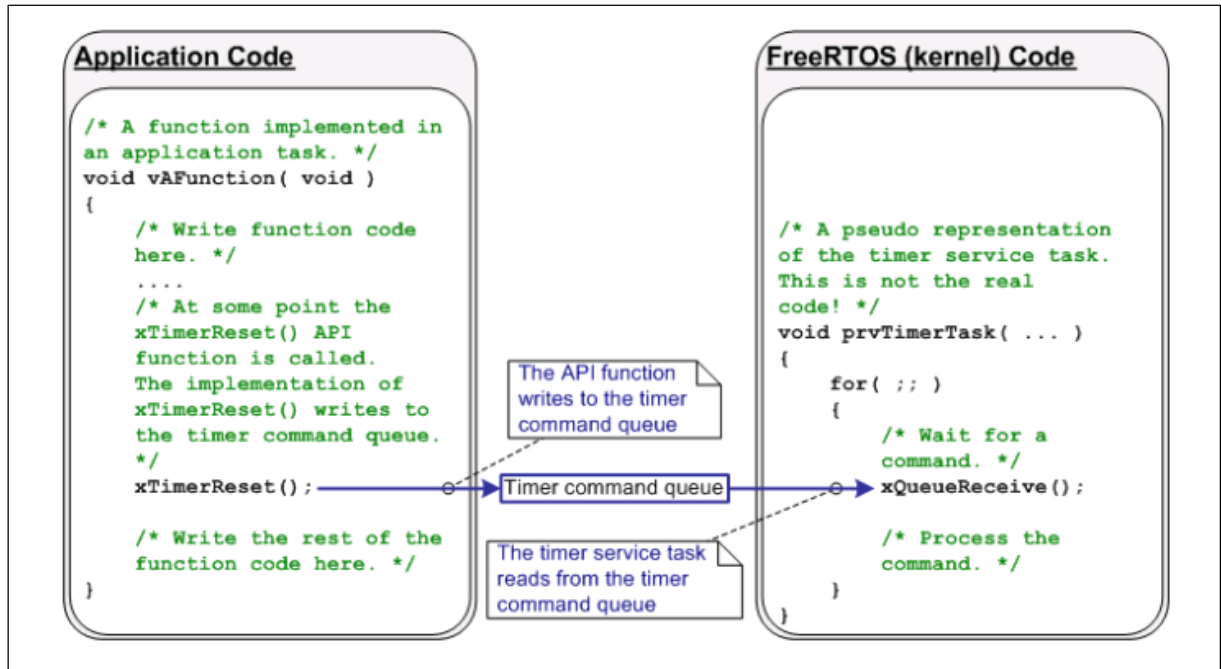
There are two types of timer, one-shot timers, and auto-reload timers. Once started, a one-shot timer will execute its callback function only once. It can be manually re-started, but will not automatically re-start itself. Conversely, once started, an auto-reload timer will automatically re-start itself after each execution of its callback function, resulting in periodic callback execution. After the period expires, the one-shot timer or auto-reload timer will call its callback function, to which users can add the project code to be executed. The difference in behavior between a one-shot timer and an auto-reload timer is demonstrated by the timeline in the diagram below.

Figure 34. One-shot timer VS auto-reload timer



FreeRTOS creates a task for software timers, which is called the timer service/daemon task. This task is automatically created by the system after the software timer is enabled. The diagram below demonstrates this scenario.

Figure 35. Software timer daemon task



As shown in Figure 35, the left shows the user application code and the right shows the simplified code of kernel software timer, and their communication is realized by queues. The application program calls the corresponding software timer API function and then send messages into the software timer queue; then the timer service task reads from the queue and operates accordingly.

Define the below constants before using the FreeRTOS software timer.

#define configUSE_TIMERS 1

Set to 1 to enable the software timer.

#define configTIMER_TASK_PRIORITY 10

Set the priority of the timer service task. The higher the priority, the better real-time response of the callback function and the more accurate of the timer. It is set by users as needed.

#define configTIMER_QUEUE_LENGTH 15

This sets the maximum number of unprocessed commands that the timer command queue can hold at any one time. It is set by users according to the number of software timers.

#define configTIMER_TASK_STACK_DEPTH 128

This sets the size of the stack allocated to the timer service task. It is set by users as needed.

Configure these constants to make the FreeRTOS software timer available in an application.

10.2 Software timer API

Table 40 lists the software timer API functions.

Table 40. Software timer API

Software timer API functions	
API	Description
xTimerChangePeriod()	Change the period of a timer
xTimerChangePeriodFromISR()	Change the period of a timer (called from an interrupt)
xTimerCreate()	Create a timer
xTimerCreateStatic()	Create a timer with static method
xTimerDelete()	Delete a timer
xTimerGetExpiryTime()	Get the time at which the software timer will expire
pcTimerGetName()	Get the name of a timer
xTimerGetPeriod()	Get the period of a timer
xTimerGetTimerDaemonTaskHandle()	Get the task handle associated with the software timer daemon/service task
pvTimerGetTimerID()	Get the ID assigned to a timer
xTimerIsTimerActive()	Query a software timer to see if it is active or dormant
xTimerPendFunctionCall()	Pend the execution of a function to the RTOS daemon task
xTimerPendFunctionCallFromISR()	Pend the execution of a function to the RTOS daemon task (called from an interrupt)
xTimerReset()	Reset a timer
xTimerResetFromISR()	Reset a timer (called from an interrupt)
vTimerSetTimerID()	Set the ID assigned to a timer
xTimerStart()	Start a timer
xTimerStartFromISR()	Start a timer (called from an interrupt)
xTimerStop()	Stop a timer
xTimerStopFromISR()	Stop a timer (called from an interrupt)

xTimerCreate();

Description:

Create a timer.

Prototype:

Table 41. xTimerCreate()

TimerHandle_t xTimerCreate(const char *pcTimerName, const TickType_t xTimerPeriod, const UBaseType_t uxAutoReload, void * const pvTimerID, TimerCallbackFunction_t pxCallbackFunction);	
Parameter	Description
pcTimerName	Name of the timer
xTimerPeriod	Period of the timer
uxAutoReload	Mode of the timer

pvTimerID	Timer ID
pxCallbackFunction	The function to call when the timer expires.

Return value:

Handle of the created software timer

xTimerStart();

Description:

Start a timer.

Prototype:

Table 42. xTimerStart()

BaseType_t xTimerStart(TimerHandle_t xTimer, TickType_t xTicksToWait);	
Parameter	Description
xTimer	Handle of the timer being started/restarted
xTicksToWait	The time, in ticks, that the calling task should be held in the Blocked state to wait for the start command to be successfully sent to the timer command queue

Return value:

Timer start success flag

xTimerStop();

Description:

Stop a timer.

Prototype:

Table 43. xTimerStop()

BaseType_t xTimerStop(TimerHandle_t xTimer, TickType_t xTicksToWait);	
Parameter	Description
xTimer	Handle of the timer being stopped
xTicksToWait	The time, in ticks, that the calling task should be held in the Blocked state to wait for the stop command to be successfully sent to the timer command queue

Return value:

Timer stop success flag

pcTimerGetName();

Description:

Get the name of a timer.

Prototype:

Table 44. pcTimerGetName()

const char * pcTimerGetName(TimerHandle_t xTimer);	
Parameter	Description
xTimer	The timer being queried

Return value:

A pointer to the name of a timer

pvTimerGetTimerID();

Description:

Get the timer ID.

Prototype:

Table 45. pvTimerGetTimerID()

void *pvTimerGetTimerID(TimerHandle_t xTimer);	
Parameter	Description
xTimer	The timer being queried

Return value:

Timer ID

vTimerSetTimerID();

Description:

Set the timer ID.

Prototype:

Table 46. vTimerSetTimerID()

void vTimerSetTimerID(TimerHandle_t xTimer, void *pvNewID);	
Parameter	Description
xTimer	The timer being updated
pvNewID	The handle to which the timer's identifier will be set

Return value:

None.

For details about other software timer API functions, refer to the API Reference in FreeRTOS official website.

10.3 Routine

Project name: 12Software_Timer_FreeRTOS

Program source code:

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

/* start task priority */
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE       128
/* start task handle */
TaskHandle_t StartTask_Handler;
/* start task entry function */
void start_task(void *pvParameters);

/* software timer create task priority */
#define SoftwareTimer_Create_TASK_PRIO      4
/* software timer create task stack size */
#define SoftwareTimer_Create_STK_SIZE       256
/* software timer create task handle */
TaskHandle_t SoftwareTimer_CreateTask_Handler;
/* software timer create task entry function */
void SoftwareTimer_Create_task(void *pvParameters);

/* debugging task priority */
#define Debug_TASK_PRIO      5
/* debugging task stack size */
#define Debug_STK_SIZE       512
/* debugging task handle */
TaskHandle_t DebugTask_Handler;
/* debugging task entry function */
void debug_task(void *pvParameters);

/* the number of software timers to be created */
#define NUM_TIMERS 5
/* define an array of software timers */
TimerHandle_t AT_xTimers[ NUM_TIMERS ];
/* name of a software timer */
char *Timer_Name[] = {"Timer1", "Timer2", "Timer3", "Timer4", "Timer5"};
/* timer callback function */
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);

/* create start task */
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);

/* enable task scheduler */
vTaskStartScheduler();
}

/* start task function */
void start_task(void *pvParameters)
{
    /* enter critical code region */
    taskENTER_CRITICAL();

    /* initialize the timer after a queue is created */
    TIMER_Init();

    /* software timer create task */
    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);

    /* create debugging task */
    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);

/* delete start task */
vTaskDelete(StartTask_Handler);
/* exit critical code region */
taskEXIT_CRITICAL();
}

/* software timer create task function */
void SoftwareTimer_Create_task(void *pvParameters)
{
    int i;
    /* enter critical code region */
    taskENTER_CRITICAL();
    for( i=0; i<NUM_TIMERS; i++ )
    {
        /* create 5 software timers, loop mode */
        AT_xTimers[ i ] = xTimerCreate( Timer_Name[i],
                                        ( 1000 * i ) + 1000,
                                        pdTRUE,
                                        ( void * ) 0,
                                        AT_vTimerCallback );

        if( AT_xTimers[ i ] == NULL )
        {
            /* fail to create */
            while(1);
        }
        else
        {
            if( xTimerStart( AT_xTimers[ i ], portMAX_DELAY ) != pdPASS )
            {
                /* timer start fails*/
                while(1);
            }
        }
    }
}

/* delete software timer create task */
vTaskDelete(SoftwareTimer_CreateTask_Handler);
/* exit critical code region */
taskEXIT_CRITICAL();
}

/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];

```



```

while(1)
{
    /* press the button to print out the debugging information */
    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);
}

/* software timer callback function */
void AT_vTimerCallback( TimerHandle_t xTimer )
{
    const uint32_t ulMaxExpiryCountBeforeStopping = 10;
    uint32_t ulCount;
    /* get software timer ID */
    ulCount = ( uint32_t ) pvTimerGetTimerID( xTimer );

    ulCount++;

    if( ulCount >= ulMaxExpiryCountBeforeStopping )
    {
        /* stop software timer */
        xTimerStop( xTimer, 0 );
        /* get the software timer name and print it out */
        printf("The stop timer is %s\r\n",pcTimerGetName(xTimer));
        at32_led_toggle(LED4);
    }
    else
    {
        /* set software timer ID */
        vTimerSetTimerID( xTimer, ( void * ) ulCount );
        at32_led_toggle(LED2);
    }
}

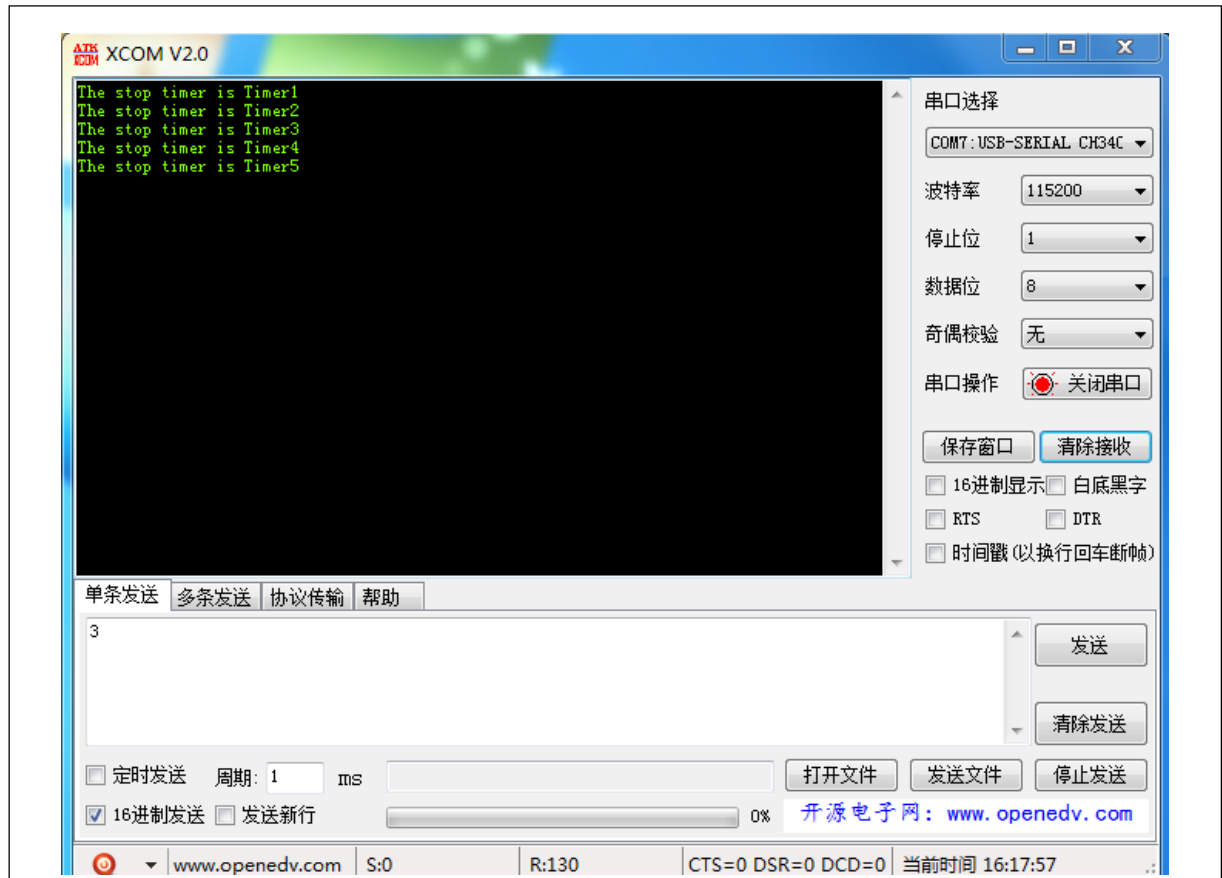
```

In this program, five timers are created in SoftwareTimer_task, with the period of 1s, 2s, 3s, 4s and

5s, respectively. Determine whether the software timer overflow occurs for 10 times in the callback function. If yes, disable the timer and print out the disabled timer. In this scenario, one timer is disabled every 10s.

Compile and download the program to the target board, and the execution is as follows:

Figure 36. Software timer routine



As shown in Figure 36, this process conforms to the program, and timers 1-5 are disabled in sequence.

11 FreeRTOS low power support

This section introduces low power support of FreeRTOS. FreeRTOS supports a Tickless mechanism to reduce power consumption, which is a common method used by small RTOS systems, such as emboss, RTX and uCOS-III (similar method).

11.1 Tickless idle mode

The tick refers to clock tick, and tickless means that the periodic tick is stopped.

In FreeRTOS, when the application tasks are suspended or blocked, the lowest priority idle task will run. Therefore, the AT32 MCU sleep mode can be realized in this idle task (to minimize power consumption, the sleep mode is set in the idle task). Enter the idle task and then calculate the maximum time of low power execution (i.e., the remaining time to execute the next high priority task). Set this calculated time as the low power mode wakeup time, and once the wakeup time expires, the system is woken up from low power mode to continue to run multiple tasks. This is the tickless idle functionality. The key to implement tickless idle mode is the low power execution time.

For the Cortex®-M4 processor core, FreeRTOS provides implementation of tickless idle mode code by calling WFI instruction to enter sleep mode (the implementation of specific code is in *port.c* file). Built in tickless idle functionality is enabled by defining `configUSE_TICKLESS_IDLE` as 1 in *FreeRTOSConfig.h*, and User defined tickless idle functionality can be provided by defining `configUSE_TICKLESS_IDLE` as 2. When the tickless idle functionality is enabled, the kernel will call the `portSUPPRESS_TICKS_AND_SLEEP()` macro when the following two conditions are both true:

1. Idle task is the only task able to run because all the application tasks are either in the Blocked state or in the Suspended state.
2. At least n further complete tick periods will pass before the kernel is due to transition an application task out of the Blocked state, where n is set by the `configEXPECTED_IDLE_TIME_BEFORE_SLEEP` definition in *FreeRTOSConfig.h*.

```
#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
```

In this case, set:

```
#define configEXPECTED_IDLE_TIME_BEFORE_SLEEP 10
```

The `portSUPPRESS_TICKS_AND_SLEEP` is critical for FreeRTOS tickless idle mode, which is called by the idle task and defined in *portmacro.h*:

```
/* 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
```

The portSUPPRESS_TICKS_AND_SLEEP is called by the idle task to enter low power mode. AT32 MCUs support sleep mode, standby mode and Deepsleep mode, while FreeRTOS supports sleep mode only. For the FreeRTOS tickles idle mode, the SysTick timer of M4 core is used to calculate the time to enter low power mode, which is changeable in Deepsleep mode and standby mode, causing inaccurate timing and further resulting in inaccurate FreeRTOS system tick.

The code segment used to enter low power mode in FreeRTOS *port.c* file is as below:

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

In this code segment, WFI instruction is called to enter sleep mode, and two macros are provided to developers for extension. Developers can add the related program to disable the clock to lower frequency (entering low power mode) or enable the clock to increase frequency (exiting low power mode).

Redirection in FreeRTOSConfig.h is as below:

```
/* low power mode related operations */
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
```

Define functions in the *main.c* file as below:

```
/* low power mode related configuration */
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");
```

```

    /* users can disable peripheral clock and lower the frequency as needed */
}
void Output_Low_Power_Mode(uint32_t xExpectedIdleTime)
{
    printf("Output low power mode.\r\n");
    /* resume the state before low power mode as required */
}

```

Note: To enter the AT32 MCU standby mode and Deepsleep mode, users can add the corresponding program, while the wakeup source also needs to be configured by users and the system clock may not be accurate definitely. For example, if software timers are used, entering low power mode may lead to inaccurate timing.

11.2 Routine

Project name: 13Low_Power_FreeRTOS

Program source code:

```

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

/* start task priority */
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE      128
/* start task handle */
TaskHandle_t StartTask_Handler;
/* start task entry function */
void start_task(void *pvParameters);

/* message processing task priority */
#define Process_Message_TASK_PRIO      1
/* message processing task stack size */
#define Process_Message_STK_SIZE      256
/* message processing task handle */
TaskHandle_t Process_MessageTask_Handler;
/* message processing task entry function */
void Process_Message_task(void *pvParameters);

/* message receive task priority */
#define Receive_Message_TASK_PRIO      3
/* message receive task stack size */
#define Receive_Message_STK_SIZE      256
/* message receive task handle */

```

```

TaskHandle_t Receive_MessageTask_Handler;
/* message receive task entry function */
void Receive_Message_task(void *pvParameters);

/* define a message structure */
typedef struct A_Message
{
    char ucMessageID;
    u8 ucData;
} AMessage;

/* define a queue */
QueueHandle_t AT_xQueue;
/* queue length and item size */
#define QUEUE_LENGTH 5
#define QUEUE_ITEM_SIZE sizeof( AMessage )

/* low power mode related configuration */
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");
    /* users can disable peripheral clock and lower the frequenct as needed */
}

void Output_Low_Power_Mode(uint32_t xExpectedIdleTime)
{
    printf("Output low power mode.\r\n");
    /* resume the state before low power mode as needed */
}

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);

```

```

/* create start task */
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);

/* enable task scheduler */
vTaskStartScheduler();
}

/* start task function */
void start_task(void *pvParameters)
{
    /* enter critical code region */
    taskENTER_CRITICAL();
    /* create a queue */
    AT_xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );
    if(AT_xQueue == NULL)
    {
        /* queue creation failure */
        while(1);
    }
    /* initialize the timer after a queue is created */
    TIMER_Init();
    /* create message receive task */
    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);
    /* create message processing task */
    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);
    /* create debugging task */
    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);

/* delete start task */
vTaskDelete(StartTask_Handler);

/* exit critical code region */
taskEXIT_CRITICAL();
}

/* message receive task function */
void Receive_Message_task(void *pvParameters)
{
    AMessage  Message1;
    while(1)
    {
        switch(GetUartData())
        {
            case NODATA:
                break;
            case 0x01:
                Message1.ucMessageID = 'a';
                Message1.ucData = 0x01;
                /* receive Toggle LED2 */
                xQueueSend(AT_xQueue,&Message1,10);
                break;
            case 0x02:
                Message1.ucMessageID = 'b';
                Message1.ucData = 0x02;
                /* receive Toggle LED3 */
                xQueueSend(AT_xQueue,&Message1,10);
                break;
            case 0x03:
                Message1.ucMessageID = 'c';
                Message1.ucData = 0x03;
                /* receive Toggle LED4 */
                xQueueSend(AT_xQueue,&Message1,10);
                break;
            default:
                break;
        }
        vTaskDelay(100);
    }
}

/* message processing task function */
void Process_Message_task(void *pvParameters)
{
    AMessage  Message2;

```



```

while(1)
{
    /* receive a message */
    if(xQueueReceive(AT_xQueue, &Message2, portMAX_DELAY) != pdPASS)
    {
        printf("no message\r\n");
    }
    else
    {
        /* determine the message type and operate accordingly */
        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");
    }
}
}
/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out the debugging information */
        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)
{

```

```

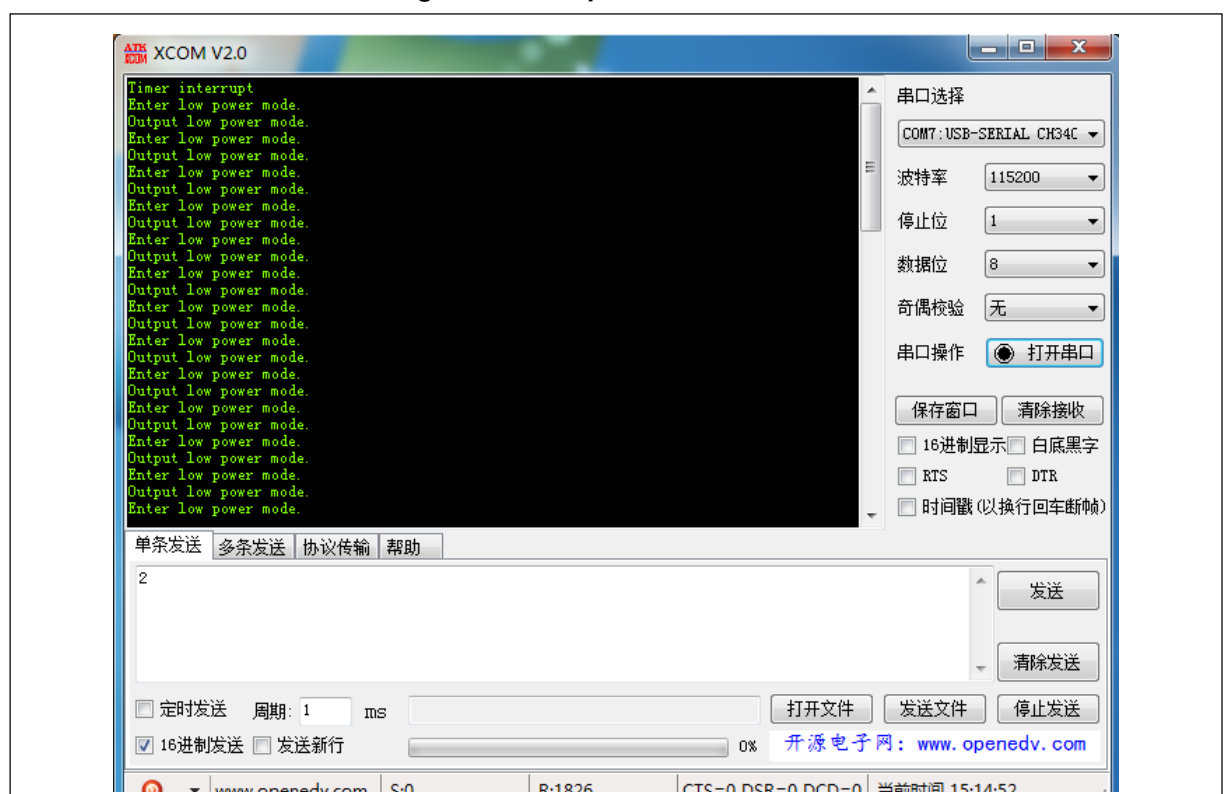
AMessage Message3;
BaseType_t xHigherPriorityTaskWoken = pdFALSE;
if(tmr_flag_get(TMR3,TMR_OVF_FLAG)==SET)
{
    Message3.ucMessageID = 'd';
    Message3.ucData = 0x04;
    /* send timer interrupt to a queue */
    xQueueSendFromISR(AT_xQueue,&Message3,&xHigherPriorityTaskWoken);
    /* determine whether or not to switch the task */
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    tmr_flag_clear(TMR3,TMR_OVF_FLAG);
}
}

```

This is the source program of FreeRTOS low power mode routine, where the low power mode related modifications are added based on the queue routine.

Compile and download the program to the target board, and the execution is as follows:

Figure 37. Low power mode routine



The result shows that the program loops in and out of low power mode. When the idle task runs, the system enters low power mode if the time to enter idle task is greater than 10, and exits low power mode when this time expires.

12 FreeRTOS memory management

This section introduces five sample memory allocation implementations, each of which are described in the following subsections. Each provided implementation is contained in a separate source file (heap_1.c, heap_2.c, heap_3.c, heap_4.c and heap_5.c respectively) which are located in the */portable/MemMang* directory of the main RTOS source code download.

12.1 Heap_1

Heap_1 is the simplest implementation of all. It does not permit memory to be freed once it has been allocated. Despite this, heap_1.c is appropriate for a large number of embedded applications. This is because many small and deeply embedded applications create all the tasks, queues, semaphores, etc. required when the system boots, and then use all of these objects for the lifetime of program (until the application is switched off again, or is rebooted). Nothing ever gets deleted.

The implementation simply subdivides a single array into smaller blocks as RAM is requested, without generating memory fragments. In addition, FreeRTOS supports byte alignment (8-byte alignment for AT32); therefore, the actual memory may be greater than the required memory (for example, the required memory is 12 bytes and the actual memory is 16 bytes due to 8-byte alignment).

Both ends will be discarded properly based on memory alignment (not if the allocated memory is exactly aligned). Calling the pvPortMalloc function returns the memory start address.

Source program of heam_1:

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

    /* ensure the requested number of bytes is an integer multiple of byte alignment */
    #if( portBYTE_ALIGNMENT != 1 )
    {
        if( xWantedSize & portBYTE_ALIGNMENT_MASK )
        {
            /* count the number of bytes requested after byte alignment */
            xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
                portBYTE_ALIGNMENT_MASK ) );
        }
    }
    #endif

    vTaskSuspendAll();
    {
        if( pucAlignedHeap == NULL )
        {
            /* ensure that the requested memory start byte is properly aligned */
            pucAlignedHeap = ( uint8_t * ) ( ( ( portPOINTER_SIZE_TYPE )
```

```

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

    /* ensure enough space for the requested memory */
    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 Heap_2

Heap_2 is more complex than heap_1. It uses an optimal matching algorithm and allows previously allocated blocks to be freed. It does not combine adjacent free blocks into a single large block.

Heap_2 is suitable for tasks, queues and semaphores that repeatedly allocate and freed the same heap space, regardless of the memory fragmentation. It should not be used if the memory being allocated and freed is of a random size. Similar to heap_1, it defines a large array as below:

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

request memory:

Heap_2 defines a data structure to manage free memory blocks and forms a block link as below:

```

typedef struct A_BLOCK_LINK
{

```

```

    struct A_BLOCK_LINK *pxNextFreeBlock;    /*<< pointer to the next free block */
    size_t xBlockSize;                      /*<< free block size, including the block link structure
*/
} BlockLink_t;

/* create a pair of link structures (xStart: link start; xEnd: link end) */
static BlockLink_t xStart, xEnd;

```

Memory request source program:

```

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

    vTaskSuspendAll();
    {
        /* determine whether it is called for the first time; if yes, initialize the block heap */
        if( xHeapHasBeenInitialised == pdFALSE )
        {
            prvHeapInit();
            xHeapHasBeenInitialised = pdTRUE;
        }

        /* add the link structure size to the requested number of bytes */
        if( xWantedSize > 0 )
        {
            xWantedSize += heapSTRUCT_SIZE;

            /* ensure byte alignment of the requested bytes */
            if( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) != 0 )
            {
                /* perform byte alignment */
                xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize &
                    portBYTE_ALIGNMENT_MASK ) );
            }
        }

        if( ( xWantedSize > 0 ) && ( xWantedSize < configADJUSTED_HEAP_SIZE ) )
        {
            /* free blocks are linked from the smallest to largest; find appropriate blocks */
            pxPreviousBlock = &xStart;
            pxBlock = xStart.pxNextFreeBlock;
            while( ( pxBlock->xBlockSize < xWantedSize ) && ( pxBlock->pxNextFreeBlock !=
                NULL ) )

```

```

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

        /* not execute if appropriate blocks are not found */
        if( pxBlock != &xEnd )
        {
            /* return to the requested memory address, and skip the valid address after
the link structure */
            pvReturn = ( void * ) ( ( ( uint8_t * ) pxPreviousBlock->pxNextFreeBlock ) +
                heapSTRUCT_SIZE );

            /* remove the requested block from free block link */
            pxPreviousBlock->pxNextFreeBlock = pxBlock->pxNextFreeBlock;

            /* the requested block can be halved if it is large enough */
            if( ( pxBlock->xBlockSize - xWantedSize ) > heapMINIMUM_BLOCK_SIZE )
            {
                /* remove the allocated blocks, and insert a link at the beginning of the
remaining blocks */
                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;

                /* insert to the free block link */
                prvInsertBlockIntoFreeList( ( pxNewBlockLink ) );
            }

            xFreeBytesRemaining -= pxBlock->xBlockSize;
        }
    }

    traceMALLOC( pvReturn, xWantedSize );
}
( void ) xTaskResumeAll();
/* call hook function if allocation fails */
#if( configUSE_MALLOC_FAILED_HOOK == 1 )
{
    if( pvReturn == NULL )
    {
        extern void vApplicationMallocFailedHook( void );

```

```

        vApplicationMallocFailedHook();
    }
}
#endif

return pvReturn;
}

```

In this source program, there is unique single free block, and a link structure is placed at its starting address to store this free block and the address of the next free block. In the initial phase, the `pxNextFreeBlock` of the only free block points to link `xEnd`, and the `pxNextFreeBlock` of `xStart` points to the free block. In this way, `xStart`, free block and `xEnd` form a single link, where `xStart` represents the start of link and `xEnd` represents the end of link. As the memory is requested and freed, more and more free blocks are generated, and these free blocks still have `xStart` (link start) and `xEnd` (link end) and are sorted by size, with the small ones first and the large ones last.

When a N-byte memory is requested, actually not only allocate a N-byte memory, but also a `BlockLink_t` structure space at the beginning of the free block for block description. It should be noted that byte alignment is required after the `BlockLink_t` structure size is added to the requested memory size.

The memory request process is as follows: calculate the actual size of memory to be allocated and check whether the memory request is feasible; if yes, query from the `xStart`; then, if the free block size is greater than or equal to the size to be allocated, take an appropriate size from the free block to the requester, and the remaining blocks form a new single block and are then inserted into the free block link in order of the free block size. Note that the returned memory address does not include the link structure.

Free memory:

`Heap_2` permits memory to be freed, and the implementation is not complex because it does not combine adjacent free blocks. Users need to find the link structure according to the input parameters, insert the memory block to free block link list and update the value of unallocated heap counter.

The source program is as below:

```

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

    if( pv != NULL )
    {
        /* find link structure according to input parameters */
        puc -= heapSTRUCT_SIZE;
    }
}

```

```

/* prevent compiler warnings */
pxLink = ( void * ) puc;

vTaskSuspendAll();
{
    /* add the free block into the link list*/
    prvInsertBlockIntoFreeList( ( ( BlockLink_t * ) pxLink ) );
    /* update unallocated heap size */
    xFreeBytesRemaining += pxLink->xBlockSize;
    traceFREE( pv, pxLink->xBlockSize );
}
( void ) xTaskResumeAll();
}
}

```

12.3 Heap_3

Heap_3 simply wraps the standard malloc() and free() for thread safety. It suspends the scheduler before operating the memory and resume the scheduler after completion.

Heap_1 and heap_2 define an array as a heap, and the array size is set by configTOTAL_HEAP_SIZE. Different from that, heap_3 does not define an array as a heap but uses a compiler to set the heap (typically in boots).

Related source programs are as below:

Request memory:

```

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;
}
```

Free memory:

```
void vPortFree( void *pv )
{
    if( pv )
    {
        vTaskSuspendAll();
        {
            free( pv );
            traceFREE( pv, 0 );
        }
        ( void ) xTaskResumeAll();
    }
}
```

12.4 Heap_4

Heap_4 is similar to heap_2, and it includes a coalescence algorithm to combine adjacent free blocks into a single large block. Similar to heap_1 and heap_2, it defines the available heap space as below:

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

Request memory:

It uses a link structure to manage free blocks, which is defined as below:

```
typedef struct A_BLOCK_LINK
{
    struct A_BLOCK_LINK *pxNextFreeBlock;    /*<< pointer to the next free block */
    size_t xBlockSize;                       /*<< free block size, including the block link structure */
} BlockLink_t;
```

Free blocks are organized into a single link list, and the BlockLink_t type local static variable xStart represents the link start. For heap_4, the link end is stored in the last place of the free block, and a BlockLink_t type local static variable pxEnd points to this place (for heap_2, xEnd represents the link end).

The biggest difference between heap_4 and heap_2 is that free blocks in heap_4 are not sorted in size but according to the address size.

Related source programs are as below:

```
void *pvPortMalloc( size_t xWantedSize )
{
    BlockLink_t *pxBlock, *pxPreviousBlock, *pxNewBlockLink;
```

```

void *pvReturn = NULL;

vTaskSuspendAll();
{
    /* if it is called for the first time, initialize the heap */
    if( pxEnd == NULL )
    {
        prvHeapInit();
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    /*check whether the requested memory size is feasible */
    if( ( xWantedSize & xBlockAllocatedBit ) == 0 )
    {
        /* add the link structure size to the actual requested memory size */
        if( xWantedSize > 0 )
        {
            xWantedSize += xHeapStructSize;

            /* check for byte alignment; scale up to integer multiples of byte alignment */
            if( ( xWantedSize & portBYTE_ALIGNMENT_MASK ) != 0x00 )
            {
                /* byte alignment calculation */
                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 ) )
        {
            /* query the appropriate free block from the link start */
            pxPreviousBlock = &xStart;

```

```

pxBlock = xStart.pxNextFreeBlock;
while( ( pxBlock->xBlockSize < xWantedSize ) &&
      ( pxBlock->pxNextFreeBlock != NULL ) )
{
    pxPreviousBlock = pxBlock;
    pxBlock = pxBlock->pxNextFreeBlock;
}

/* do not execute the following if appropriate free blocks are not found */
if( pxBlock != pxEnd )
{
    /* return the allocated memory pointer to skip BlockLink_t structure */
    pvReturn = ( void * ) ( ( ( uint8_t * )
                             pxPreviousBlock->pxNextFreeBlock ) + xHeapStructSize );

    /* remove allocated blocks from the free block link list */
    pxPreviousBlock->pxNextFreeBlock = pxBlock->pxNextFreeBlock;

    /* remaining blocks (large enough) form a new free block */
    if( ( pxBlock->xBlockSize - xWantedSize ) >
        heapMINIMUM_BLOCK_SIZE )
    {
        /* place a link structure at the start of the remaining blocks, and
        initialize contents in the link list */
        pxNewBlockLink = ( void * ) ( ( ( uint8_t * ) pxBlock ) +
                                       xWantedSize );
        configASSERT( ( ( ( size_t ) pxNewBlockLink ) &
                       portBYTE_ALIGNMENT_MASK ) == 0 );

        /* calculate the size of remaining free blocks */
        pxNewBlockLink->xBlockSize = pxBlock->xBlockSize -
                                     xWantedSize;
        pxBlock->xBlockSize = xWantedSize;

        /* insert into the free block link list */
        prvInsertBlockIntoFreeList( pxNewBlockLink );
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    xFreeBytesRemaining -= pxBlock->xBlockSize;

    if( xFreeBytesRemaining < xMinimumEverFreeBytesRemaining )

```

```

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

        /* mark the allocated blocks as "allocated" */
        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;
}

```

In this source program, there is unique single free block, and a link structure is placed at its starting address to store this free block and the address of the next free block. In the initial phase, the `pxNextFreeBlock` of the only free block points to `link xEnd`, and the `pxNextFreeBlock` of `xStart` points to the free block. In this way, `xStart`, free block and `xEnd` form a single link, where `xStart` represents the start of link and `xEnd` represents the end of link. As the memory is requested and freed, more and more free blocks are generated, and these free blocks still have `xStart` (link start) and `xEnd` (link end) and are sorted by size, with the small ones first and the large ones last.

When a N-byte memory is requested, actually not only allocate a N-byte memory, but also a `BlockLink_t` structure space at the beginning of the free block for block description. It should be noted that byte alignment is required after the `BlockLink_t` structure size is added to the requested memory size.

The memory request process is as follows: calculate the actual size of memory to be allocated and check whether the memory request is feasible; if yes, query from the `xStart`; then, if the free block size is greater than or equal to the size to be allocated, take an appropriate size from the free block to the requester, and the remaining blocks form a new single block and are then inserted into the free block link in order of the free block size. During the process of inserting blocks to the link, the coalescence algorithm is executed to check whether a free block can be combined with the previous block into a large block (if yes, combine them) and then check whether it can be combined with the next free block into a large block (if yes, combine them). Note that the returned memory address does not include the link structure.

Free memory:

Users need to find the link structure according to the input parameters and insert the memory block into free block link list. Note that the coalescence algorithm is executed during the inserting process, and finally this block is marked as “free”.

```

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

    if( pv != NULL )
    {
        /* find link structure according to input parameters */
        puc -= xHeapStructSize;

        /* prevent compiler warnings */
        pxLink = ( void * ) puc;
    }
}

```

```

/* check to confirm that this block is allocated */
configASSERT( ( pxLink->xBlockSize & xBlockAllocatedBit ) != 0 );
configASSERT( pxLink->pxNextFreeBlock == NULL );

if( ( pxLink->xBlockSize & xBlockAllocatedBit ) != 0 )
{
    if( pxLink->pxNextFreeBlock == NULL )
    {
        /* mark the block as "free" */
        pxLink->xBlockSize &= ~xBlockAllocatedBit;

        vTaskSuspendAll();
        {
            /* update the size of unallocated heap */
            xFreeBytesRemaining += pxLink->xBlockSize;
            traceFREE( pv, pxLink->xBlockSize );
            prvInsertBlockIntoFreeList( ( ( BlockLink_t * ) pxLink ) );
        }
        ( void ) xTaskResumeAll();
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
else
{
    mtCOVERAGE_TEST_MARKER();
}
}
}

```

12.5 Heap_5

Heap_5 is similar to heap_4, except that it allows the heap to span multiple non-adjacent (non-contiguous) memory regions. Heap_1, heap_2 and heap_4 use an array as a heap and define the array size in program (by static uint8_t ucHeap[configTOTAL_HEAP_SIZE];), while heap_5 defines multiple heaps to span multiple non-adjacent memory regions. For example, it can define a heap in on-chip RAM and also a heap in external RAM. Users only need to define the start address and size of each heap.

To facilitate the management of multiple heaps, FreeRTOS defines a specific structure "HeapRegion_t" as below:

```

/* Used by heap_5.c. */
typedef struct HeapRegion
{

```

```
uint8_t *pucStartAddress; /* start address */
size_t xSizeInBytes;      /* heap size */
} HeapRegion_t;
```

FreeRTOS heap_5.c file provides an example to demonstrate how to define multiple heaps, as shown below:

```
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.
};
```

In this program, two heap regions are defined. Note that the last element in xHeapRegions array must be { NULL, 0 } to tell the system when heap initialization is completed.

The memory allocation and memory release of heap_5 are basically the same as that of heap_4.

The following section introduces initialization of multiple heaps, and the source program is as below:

```
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 = &(amp; pxHeapRegions[ xDefinedRegions ] );

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

        /* ensure heap start address is byte-aligned */
        xAddress = ( size_t ) pxHeapRegion->pucStartAddress;
        if( ( xAddress & portBYTE_ALIGNMENT_MASK ) != 0 )
        {
```

```

        xAddress += ( portBYTE_ALIGNMENT - 1 );
        xAddress &= ~portBYTE_ALIGNMENT_MASK;

        /* adjust the heap size after byte alignment */
        xTotalRegionSize -= xAddress - ( size_t ) pxHeapRegion->pucStartAddress;
    }

    xAlignedHeap = xAddress;

    /* initialize xStart */
    if( xDefinedRegions == 0 )
    {
        /* xStart points to the first free block */
        xStart.pxNextFreeBlock = ( BlockLink_t * ) xAlignedHeap;
        xStart.xBlockSize = ( size_t ) 0;
    }
    else
    {
        /* execute after one heap is initialized */
        configASSERT( pxEnd != NULL );

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

    /* record the position of pxEnd in the previous block (if any) */
    pxPreviousFreeBlock = pxEnd;

    /* pxEnd records the end of free block link, which means that pxEnd is in the end of the last
    region */
    xAddress = xAlignedHeap + xTotalRegionSize;
    xAddress -= xHeapStructSize;
    xAddress &= ~portBYTE_ALIGNMENT_MASK;
    pxEnd = ( BlockLink_t * ) xAddress;
    pxEnd->xBlockSize = 0;
    pxEnd->pxNextFreeBlock = NULL;

    /* this region includes a free block, and the block size equals to the entire region size
    minus the size of the free block structure */
    pxFirstFreeBlockInRegion = ( BlockLink_t * ) xAlignedHeap;
    pxFirstFreeBlockInRegion->xBlockSize = xAddress - ( size_t )
pxFirstFreeBlockInRegion;
    pxFirstFreeBlockInRegion->pxNextFreeBlock = pxEnd;

    /* if it is not the first block in this region, link the previous region to this region */

```



```

        if( pxPreviousFreeBlock != NULL )
        {
            pxPreviousFreeBlock->pxNextFreeBlock = pxFirstFreeBlockInRegion;
        }

        xTotalHeapSize += pxFirstFreeBlockInRegion->xBlockSize;

        /* skip to the next heap */
        xDefinedRegions++;
        pxHeapRegion = &(amp; 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 );
}

```

This program initializes heaps in sequence according to the xHeapRegions structure in user application, and its main function is to link these heaps for subsequent allocation and release.

13 FreeRTOS stream buffer

This section introduces FreeRTOS stream buffer, which is mainly used for data transfer.

13.1 Introduction

Stream buffers are an RTOS task to RTOS task, and interrupt to task communication primitives. Unlike most other FreeRTOS communications primitives, they are optimised for single reader single writer scenarios, such as passing data from an interrupt service routine to a task, or from one microcontroller core to another on dual core CPUs. Data is passed by copy - the data is copied into the buffer by the sender and out of the buffer by the reader.

Stream buffers pass a continuous stream of bytes. Message buffers pass variable sized but discrete messages. Message buffers use stream buffers for data transfer.

Note: Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. In this case, if related API functions are called, a critical section should be used for protection and the send/receive block time is set to 0.

13.2 Stream buffer API

Table 47 lists stream buffer API functions.

Table 47. Stream buffer API

Stream buffer API functions	
API	Description
xStreamBufferBytesAvailable()	Query a stream buffer to see how much data it contains
xStreamBufferCreate()	Create a stream buffer
xStreamBufferCreateStatic()	Create a stream buffer with static method
vStreamBufferDelete()	Delete a stream buffer
xStreamBufferIsEmpty()	Query a stream buffer to see if it is empty
xStreamBufferIsFull()	Query a stream buffer to see if it is full
xStreamBufferReceive()	Receive bytes from a stream buffer
xStreamBufferReceiveFromISR()	Receive bytes from a stream buffer (called from an interrupt)
xStreamBufferReset()	Reset a stream buffer
xStreamBufferSend()	Send bytes to a stream buffer
xStreamBufferSendFromISR()	Send bytes to a stream buffer (called from an interrupt)
xStreamBufferSetTriggerLevel()	Stream buffer trigger level
xStreamBufferSpacesAvailable()	Query a stream buffer to see how much free space it contains

xStreamBufferCreate();

Description:

Create a stream buffer.

Prototype:

Table 48. xStreamBufferCreate()

StreamBufferHandle_t xStreamBufferCreate(size_t xBufferSizeBytes, size_t xTriggerLevelBytes);	
Parameter	Description
xBufferSizeBytes	The total number of bytes the stream buffer will be able to hold at any one time.
xTriggerLevelBytes	Stream buffer trigger level

Return value:

Handle of the created stream buffer

xStreamBufferReceive ();

Description:

Receive bytes from a stream buffer

Prototype:

Table 49. xStreamBufferReceive ()

size_t xStreamBufferReceive(StreamBufferHandle_t xStreamBuffer, void *pvRxData, size_t xBufferLengthBytes, TickType_t xTicksToWait);	
Parameter	Description
xStreamBuffer	Handle of the stream buffer from which bytes are to be received
pvRxData	A pointer to the buffer into which the received bytes will be copied
xBufferLengthBytes	The maximum number of bytes to receive in one call
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for data to become available if the stream buffer is empty.

Return value:

The number of bytes read from the stream buffer.

xStreamBufferSend();

Description:

Send bytes to a stream buffer.

Prototype:

Table 50. xStreamBufferSend()

size_t xStreamBufferSend(StreamBufferHandle_t xStreamBuffer, const void *pvTxData, size_t xDataLengthBytes, TickType_t xTicksToWait);	
Parameter	Description
xStreamBuffer	Handle of the stream buffer to which a stream is being sent
pvTxData	A pointer to the buffer that holds the bytes to be copied into the stream buffer
xBufferLengthBytes	The maximum number of bytes to copy into the stream buffer.
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for enough space to become available in the stream buffer

Return value:

The number of bytes written to the stream buffer.

13.3 Routine

Project name: 14Stream_Buffers_FreeRTOS

Program source code:

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

/* start task priority */
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE       128
/* start task handle */
TaskHandle_t StartTask_Handler;
/* start task entry function */
void start_task(void *pvParameters);

/* Stream Buffers send task priority */
#define Stream_Buffers_Send_TASK_PRIO      3
/* Stream Buffers send task stack size */
#define Stream_Buffers_Send_STK_SIZE       256
/* Stream Buffers send task handle */
TaskHandle_t Stream_Buffers_SendTask_Handler;
```

```

/* Stream Buffers send task entry function */
void Stream_Buffers_Send_task(void *pvParameters);

/* Stream Buffers receive task priority */
#define Stream_Buffers_Receive_TASK_PRIO      3
/* Stream Buffers receive task stack size*/
#define Stream_Buffers_Receive_STK_SIZE      256
/* Stream Buffers receive task handle */
TaskHandle_t Stream_Buffers_ReceiveTask_Handler;
/* Stream Buffers receive task entry function */
void Stream_Buffers_Receive_task(void *pvParameters);

/* debugging task priority */
#define Debug_TASK_PRIO      3
/* debugging task stack size */
#define Debug_STK_SIZE      512
/* debugging task handle */
TaskHandle_t DebugTask_Handler;
/* debugging task entry function */
void debug_task(void *pvParameters);

/* define a 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);
    /* create start task */
    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);
    /* enable task scheduler */
    vTaskStartScheduler();
}

/* start task function */
void start_task(void *pvParameters)

```

```

{
    /* enter critical code region */
    taskENTER_CRITICAL();

    /* create a Stream Buffer */
    AT_xStreamBuffer = xStreamBufferCreate( AT_xStreamBufferSizeBytes,
    AT_xTriggerLevel );
    if( AT_xStreamBuffer == NULL )
    {
        /* Stream Buffer creation fails */
        while(1);
    }
    /* initialize the timer */
    TIMER_Init();
    /* create Stream Buffers receive task */
    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);
    /* create Stream Buffers send task */
    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);
    /* create debugging task */
    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);
    /* delete start task */
    vTaskDelete(StartTask_Handler);
    /* exit critical code region */
    taskEXIT_CRITICAL();
}
/* Stream Buffers receive task function */
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)
{
    /* receive data from AT_xStreamBuffer */
    xReceivedBytes = xStreamBufferReceive( AT_xStreamBuffer, ( void * ) ucRxData,
        sizeof( ucRxData ), xBlockTime );
    if( xReceivedBytes > 0 )
    {
        printf("%s\r\n", ucRxData);
    }
}

/* Stream Buffers send task function */
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)
    {
        /* send data from ucArrayToSend */
        xBytesSent = xStreamBufferSend( AT_xStreamBuffer, ( void * ) ucArrayToSend,
            sizeof( ucArrayToSend ), x100ms );
        if( xBytesSent != sizeof( ucArrayToSend ) )
        {
            printf("#1:%d data written in.\r\n", xBytesSent);
        }
        vTaskDelay(3000);
    }
}

/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out the task information */
        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");

```

```

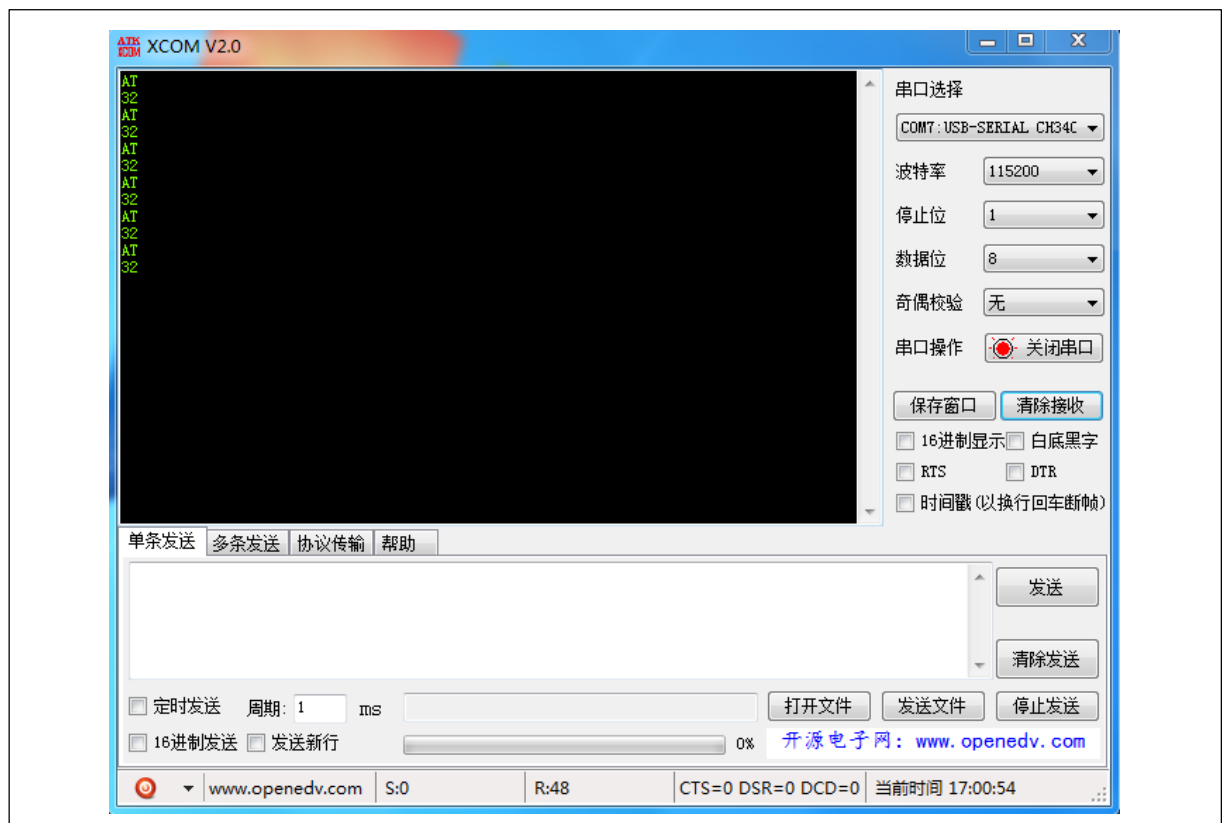
vTaskGetRunTimeStats((char *)&buff);
printf("%s\r\n", buff);
}
vTaskDelay(10);
}
}

```

In this program, a stream buffer object is created, and then a task writes data to and receives data from the stream buffer in a loop manner. If the stream buffer is empty, the data receive task enters Blocked state and wait for data to be sent to the stream buffer. The data is print out after it is received by the task. This program allows the task to receives three data from stream buffer each time.

Compile and download the program to the target board, and the execution is as follows:

Figure 38. Stream buffer routine



The result shows that the task receives three data each time.

14 FreeRTOS message buffer

This section introduces FreeRTOS message buffer, which is mainly used for data transfer.

14.1 Introduction

message buffers are an RTOS task to RTOS task, and interrupt to task communication primitives. Unlike most other FreeRTOS communications primitives, they are optimised for single reader single writer scenarios, such as passing data from an interrupt service routine to a task, or from one microcontroller core to another on dual core CPUs. Data is passed through a message buffer by copy - the data is copied into the buffer by the sender and out of the buffer by the reader. Message buffers are built on top of stream buffers (that is, they use the stream buffer implementation). Unlike when using a stream buffer, a 10 byte message can only be read out as a 10 byte message, not as individual bytes.

Note: Uniquely among FreeRTOS objects, the message buffer implementation assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. In this case, if related API functions are called, a critical section should be used for protection and the send/receive block time is set to 0.

14.2 Message buffer API

Table 51 lists message buffer API functions.

Table 51. Message buffer API

Message buffer API functions	
API	Description
xMessageBufferCreate()	Create a message buffer
xMessageBufferCreateStatic()	Create a message buffer with static method
vMessageBufferDelete()	Delete a message buffer
xMessageBufferIsEmpty()	Query a message buffer to see if it is empty
xMessageBufferIsFull()	Query a message buffer to see if it is full
xMessageBufferReceive()	Receive a discrete message from a message buffer
xMessageBufferReceiveFromISR()	Receive a discrete message from a message buffer (called from an interrupt)
xMessageBufferReset()	Reset a message buffer
xMessageBufferSend()	Send a discrete message to a message buffer
xMessageBufferSendFromISR()	Send a discrete message to a message buffer (called from an interrupt)
xMessageBufferSpacesAvailable()	Query a message buffer to see how much free space it contains

xMessageBufferCreate ();

Description:

Create a message buffer.

Prototype:

Table 52. xMessageBufferCreate ()

MessageBufferHandle_t xMessageBufferCreate(size_t xBufferSizeBytes);	
Parameter	Description
xBufferSizeBytes	The total number of bytes (not messages) the message buffer will be able to hold at any one time.

Return value:

Handle of the created message buffer

xMessageBufferReceive ();

Description:

Receives a discrete message from a message buffer

Prototype:

Table 53. xMessageBufferReceive ()

size_t xMessageBufferReceive(MessageBufferHandle_t xMessageBuffer, void *pvRxData, size_t xBufferLengthBytes, TickType_t xTicksToWait);	
Parameter	Description
xMessageBuffer	Handle of the message buffer from which a message is being received
pvRxData	A pointer to the buffer into which the received message is to be copied
xBufferLengthBytes	The maximum length of the message that can be received
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for a message

Return value:

The length, in bytes, of the message read from the message buffer, if any.

xStreamBufferSend();

Description:

Send a discrete message to a message buffer

Prototype:

Table 54. xStreamBufferSend()

Size_t xMessageBufferSend(MessageBufferHandle_t xMessageBuffer, const void *pvTxData, size_t xDataLengthBytes, TickType_t xTicksToWait);	
Parameter	Description
xMessageBuffer	Handle of the message buffer to which a message is being sent
pvTxData	A pointer to the message that is to be copied into the message buffer
xBufferLengthBytes	The length of the message
xTicksToWait	The maximum amount of time the calling task should remain in the Blocked state to wait for enough space to become available in the message buffer

Return value:

The number of bytes written to the message buffer.

14.3 Routine

Project name: 15Message_Buffers_FreeRTOS

Program source code:

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

/* start task priority */
#define START_TASK_PRIO      1
/* start task stack size */
#define START_STK_SIZE       128
/* start task handle */
TaskHandle_t StartTask_Handler;
/* start task entry function */
void start_task(void *pvParameters);

/* Message Buffers send task priority */
#define Message_Buffers_Send_TASK_PRIO      4
/* Message Buffers send task stack size */
#define Message_Buffers_Send_STK_SIZE       256
/* Message Buffers send task handle */
TaskHandle_t Message_Buffers_SendTask_Handler;
/* Message Buffers send task entry function */
```

```

void Message_Buffers_Send_task(void *pvParameters);

/* Message Buffers receive task priority */
#define Message_Buffers_Receive_TASK_PRIO      3
/* Message Buffers receive task stack size */
#define Message_Buffers_Receive_STK_SIZE        256
/* Message Buffers receive task handle*/
TaskHandle_t Message_Buffers_ReceiveTask_Handler;
/* Message Buffers receive task entry function */
void Message_Buffers_Receive_task(void *pvParameters);

/* debugging task priority */
#define Debug_TASK_PRIO      3
/* debugging task stack size */
#define Debug_STK_SIZE        512
/* debugging task handle */
TaskHandle_t DebugTask_Handler;
/* debugging task entry function */
void debug_task(void *pvParameters);

/* define a 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);
    /* create start task */
    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);
    /* enable task scheduler */
    vTaskStartScheduler();
}

/* start task function */
void start_task(void *pvParameters)
{
    /* enter critical code region */

```

```

taskENTER_CRITICAL();

/* create a Message Buffer */
AT_xMessageBuffer = xMessageBufferCreate( AT_xMessageBufferSizeBytes );
if( AT_xMessageBuffer == NULL )
{
    /* Message Buffer creation failure */
    while(1);
}
/* initialize the timer */
TIMER_Init();
/* create Message Buffers receive task */
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);
/* create Message Buffers send task */
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);
/* create debugging task */
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);

/* delete start task */
vTaskDelete(StartTask_Handler);
/* exit critical code region */
taskEXIT_CRITICAL();
}

/* Message Buffers receive task function */
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)

```

```

{
    /* enter critical code region */
    taskENTER_CRITICAL();
    /* receive data from AT_xMessageBuffer */
    xReceivedBytes = xMessageBufferReceive( AT_xMessageBuffer, ( void * ) ucRxData,
        sizeof( ucRxData ), xBlockTime );
    if( xReceivedBytes > 0 )
    {
        printf("%s\r\n", ucRxData);
    }
    /* exit critical code region */
    taskEXIT_CRITICAL();

    /* enter critical code region */
    taskENTER_CRITICAL();
    /* receive data from AT_xMessageBuffer */
    xReceivedBytes = xMessageBufferReceive( AT_xMessageBuffer, ( void * ) ucRxData1,
        sizeof( ucRxData1 ), xBlockTime );
    if( xReceivedBytes > 0 )
    {
        printf("%s\r\n", ucRxData1);
    }
    /* exit critical code region */
    taskEXIT_CRITICAL();

}
}
/* Message Buffers send task function */
void Message_Buffers_Send_task(void *pvParameters)
{
    size_t xBytesSent;
    uint8_t ucArrayToSend[] = { 'A', 'T', '3', '2' };
    char *pcStringToSend = "Artery Tek MCU";
    while(1)
    {
        /* enter critical code region */
        taskENTER_CRITICAL();
        /* send data from ucArrayToSend */
        xBytesSent = xMessageBufferSend( AT_xMessageBuffer, ( void * ) ucArrayToSend,
            sizeof( ucArrayToSend ), 0 );
        if( xBytesSent != sizeof( ucArrayToSend ) )
        {
            printf("#1:%d data written in.\r\n", xBytesSent);
        }
        /* exit critical code region */
    }
}

```

```

taskEXIT_CRITICAL();

/* enter critical code region */
taskENTER_CRITICAL();
/* send pcStringToSend string */
xBytesSent = xMessageBufferSend( AT_xMessageBuffer, ( void * ) pcStringToSend,
strlen( pcStringToSend ), 0 );
if( xBytesSent != strlen( pcStringToSend ) )
{
    printf("#2:%d data written in.\r\n", xBytesSent);
}
/* exit critical code region */
taskEXIT_CRITICAL();
vTaskDelay(1000);
}
}

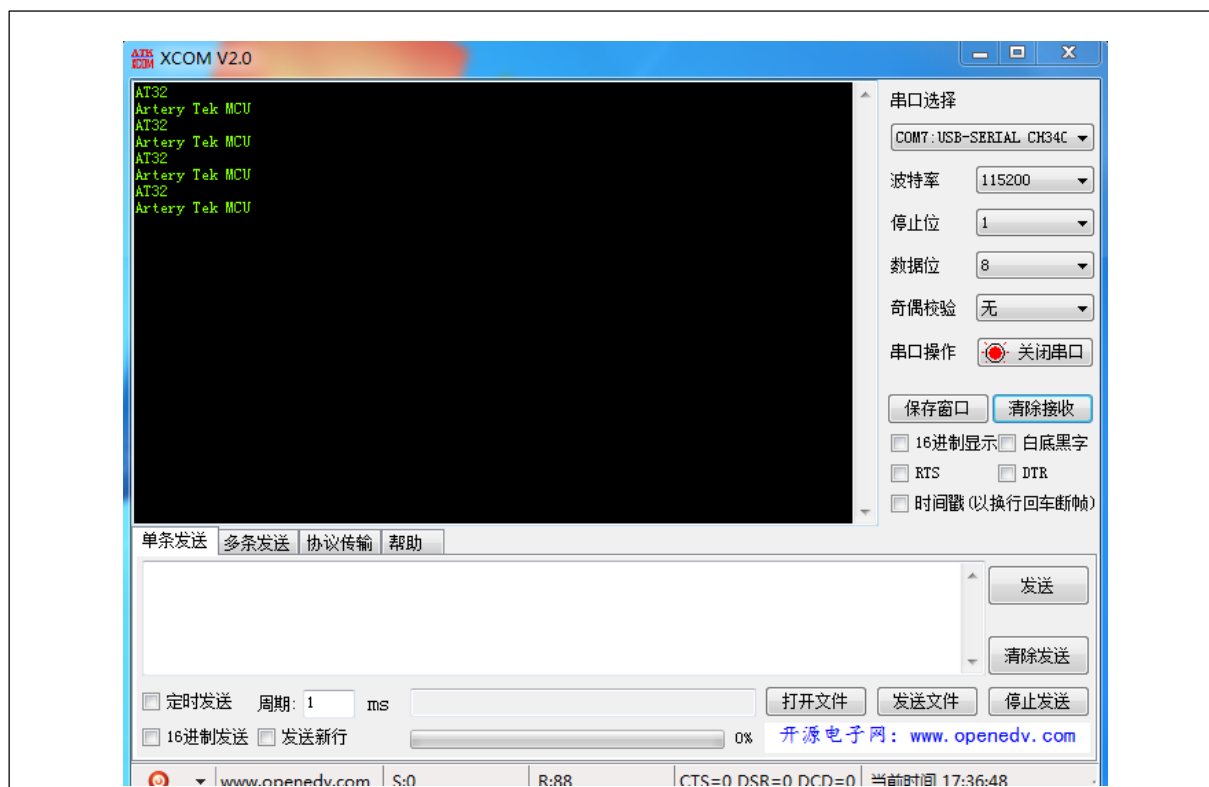
/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out task information */
        if(at32_button_press() == USER_BUTTON)
        {
            printf("/*-----*/\r\n");
            printf("Task      Status      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);
    }
}

```

In this program, a message buffer object is created; a task sends one 4-byte and one 14-byte message in sequence to the message buffer, and then a task receives messages from the message buffer. Finally, the task information is print out.

Compile and download the program to the target board, and the execution is as follows:

Figure 39. Message buffer routine



The result shows that the received messages are as desired.

15 FreeRTOS task notifications

This section introduces FreeRTOS task notifications. In certain scenarios, the task notification can replace semaphore and event group with higher efficiency.

15.1 Introduction

The task notification is optional. To use this feature, set the `configUSE_TASK_NOTIFICATIONS` to 1. In FreeRTOS, each task has a 32-bit notification value (*ulNotifiedValue* in the task control block). The task notification is an event. If the receiving task is blocked to wait for task notification, sending task notification to the receiving task can remove its blocked state. The task notification can be used to implement functions by modifying *ulNotifiedValue* as below:

- ① Do not overwrite the receiving task notification value if the previous notification value is not read;
- ② Overwrite the receiving task notification value under any condition;
- ③ Update the one or more bits in the receiving task notification value;
- ④ Increment the receiving task notification value.

The flexibility of task notifications allows them to be used where otherwise it would have been necessary to create a separate queue, binary semaphore, counting semaphore or event group. Unblocking an RTOS task with a direct notification is 45% faster and uses less RAM than unblocking a task using an intermediary object such as a binary semaphore.

As would be expected, these performance benefits require some use case limitations:

- ① RTOS task notifications can only be used when there is only one task that can be the recipient of the event. This condition is however met in the majority of real world use cases.
- ② Only in the case where an RTOS task notification is used in place of a queue: While a receiving task can wait for a notification in the Blocked state (so not consuming any CPU time), a sending task cannot wait in the Blocked state for a send to complete if the send cannot complete immediately.

15.2 Task notification API

Table 55 lists task notification API functions.

Table 55. Task notification API

Task notification API functions	
API	Description
xTaskNotify()	Send notification, with notification value but not retain the original notification value of receiving task
xTaskNotifyFromISR()	Send notification, with notification value but not retain the original notification value of receiving task (called from an interrupt)
xTaskNotifyAndQuery()	Send notification, with notification value and retaining the original notification value of receiving task

xTaskNotifyAndQueryFromISR()	Send notification, with notification value and retaining the original notification value of receiving task (called from an interrupt)
xTaskNotifyGive()	Send notification, without notification value nor retaining the original notification value but increment (add one) the receiving task notification value
vTaskNotifyGiveFromISR()	Send notification, without notification value nor retaining the original notification value but increment (add one) the receiving task notification value (called from an interrupt)
xTaskNotifyStateClear()	Clear the task notification state
ulTaskNotifyTake()	Obtain task notification; decrement the task's notification value by one or clear to zero on exit (for binary semaphores and counting semaphores, respectively)
xTaskNotifyWait()	Obtain task notification; more powerful than ulTaskNotifyTake(), and can be used as receiving task function of event group

For details about these functions, please refer to the FreeRTOS API Reference in the official website.

15.3 Routine

Project name: 16Task_Notify_FreeRTOS

Program source code:

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

/* Task1 priority */
#define Task1_TASK_PRIO      2
/* Task1 stack size */
#define Task1_STK_SIZE      128
/* Task1 handle */
TaskHandle_t AT_Task1_Handler;
/* Task1 entry function */
void AT_Task1(void *pvParameters);

/* Task2 priority */
#define Task2_TASK_PRIO      2
/* Task2 stack size */
#define Task2_STK_SIZE      128
/* Task2 handle */
TaskHandle_t AT_Task2_Handler;
/* Task2 entry function */
void AT_Task2(void *pvParameters);
```

```

/* Task3 priority */
#define Task3_TASK_PRIO      2
/* Task3 stack size */
#define Task3_STK_SIZE      128
/* Task3 handle */
TaskHandle_t AT_Task3_Handler;
/* Task3 entry function */
void AT_Task3(void *pvParameters);

/* debugging task priority */
#define Debug_TASK_PRIO      3
/* debugging task stack size */
#define Debug_STK_SIZE      512
/* debugging task handle */
TaskHandle_t DebugTask_Handler;
/* debugging task entry function */
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);

    /* enter critical code region */
    taskENTER_CRITICAL();

    /* initialize the timer */
    TIMER_Init();

```

```

/* create Task1 */
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);

/* create Task2 */
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);

/* create Task3 */
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);

/* create debugging task */
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);

/* enable task scheduler */
vTaskStartScheduler();
}

/* Task1 function */
void AT_Task1(void *pvParameters)
{
    while(1)
    {
        vTaskDelay(50);
        /* send task notification to AT_Task2_Handler */
        printf("Send a notification to AT_Task2.\r\n");
        xTaskNotifyGive( AT_Task2_Handler );
        at32_led_toggle(LED4);
        vTaskDelay(1000);
        /* wait to receive task notification from AT_Task2_Handler */
        ulTaskNotifyTake( pdTRUE, portMAX_DELAY );
    }
}

```

```

    printf("Received the AT_Task2's task notification.\r\n");
}
}
/* Task2 function */
void AT_Task2(void *pvParameters)
{
    while(1)
    {
        vTaskDelay(100);
        /* wait to receive task notification from AT_Task1_Handler */
        ulTaskNotifyTake( pdTRUE, portMAX_DELAY );
        printf("Received the AT_Task1's task notification.\r\n");
        vTaskDelay(2000);
        /* send task notification to AT_Task1_Handler */
        printf("Send a notification to AT_Task1.\r\n");
        xTaskNotifyGive( AT_Task1_Handler );
        at32_led_toggle(LED4);
    }
}
/* Task3 function */
void AT_Task3(void *pvParameters)
{
    BaseType_t xResult;
    while(1)
    {
        /* wait to receive task notification */
        xResult = xTaskNotifyWait( pdFALSE, 0xffffffff, &ulNotifiedValue, portMAX_DELAY );
        if( xResult == pdPASS )
        {
            /* receive a task notification */
            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");
            }
        }
    }
}
/* hardware timer 3 interrupt function */
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);
        /* check whether or not to switch task */
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
    }
}
/* debugging task function */
void debug_task(void *pvParameters)
{
    u8 buff[500];
    while(1)
    {
        /* press the button to print out the debugging information */
        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);
    }
}

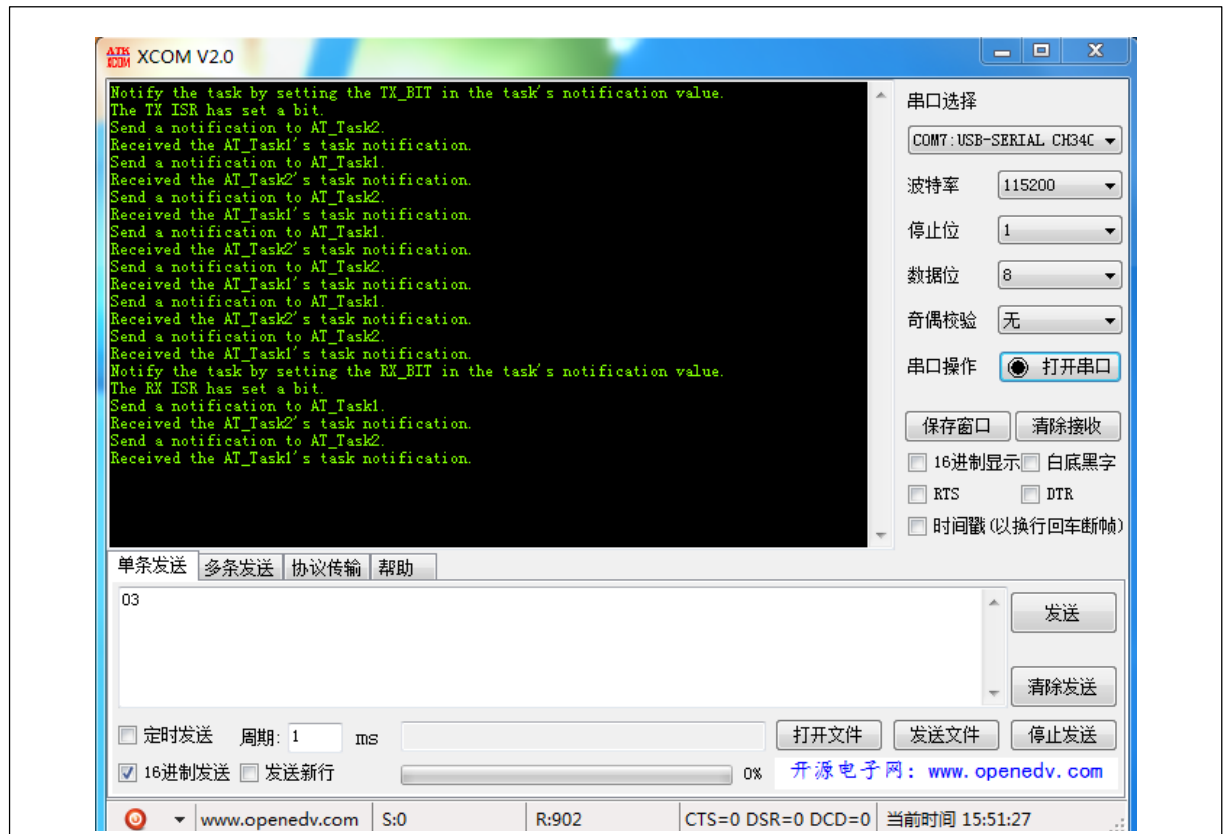
```

In this program, three tasks are created, where Task1 and Task2 sends/receives task notification to/from each other, and Task3 receives the notification sent from the hardware timer3 interrupt

handler. The operation process is print out. This routine demonstrates how to use task notifications, and simulates FreeRTOS semaphores and event group. For details, refer to the corresponding project source code.

Compile and download the program to the target board, and the execution is as follows:

Figure 40. Task notification routine



This result conforms to the program design. For details, please refer to the routine and documents in FreeRTOS official website.

16 FreeRTOS Demos

16.1 Introduction

This section introduces a routine that uses FreeRTOS on AT32 MCU to implement the following functions:

- ① AT32 MCU uses host serial port tool to send specific data to control the on-board LEDs;
- ② Generate an interrupt by using a hardware timer to send message to the task;
- ③ Use FreeRTOS software timer to collect the MCU temperature sensor value by ADC;
- ④ Implement USB Keyboard function for AT32F403 series MCUs, that is, press the USER button and then the program transfers the button value through USB to simulate keyboard .

16.2 Demo routines

1. Send data through the host serial port tool to control on-board LEDs.

Figure 41. Demo 1



As shown in Figure 41, send 0x01 to toggle LED2; send 0x02 to toggle LED3 and send 0x03 to toggle LED4. To implement this function, the program creates two tasks, in which one task is used to process the received data and then send message to a queue according to the data; the other one is used to receive message from a queue and then check whether or not to toggle the corresponding LED.

2. Collect MCU temperature sensor value.

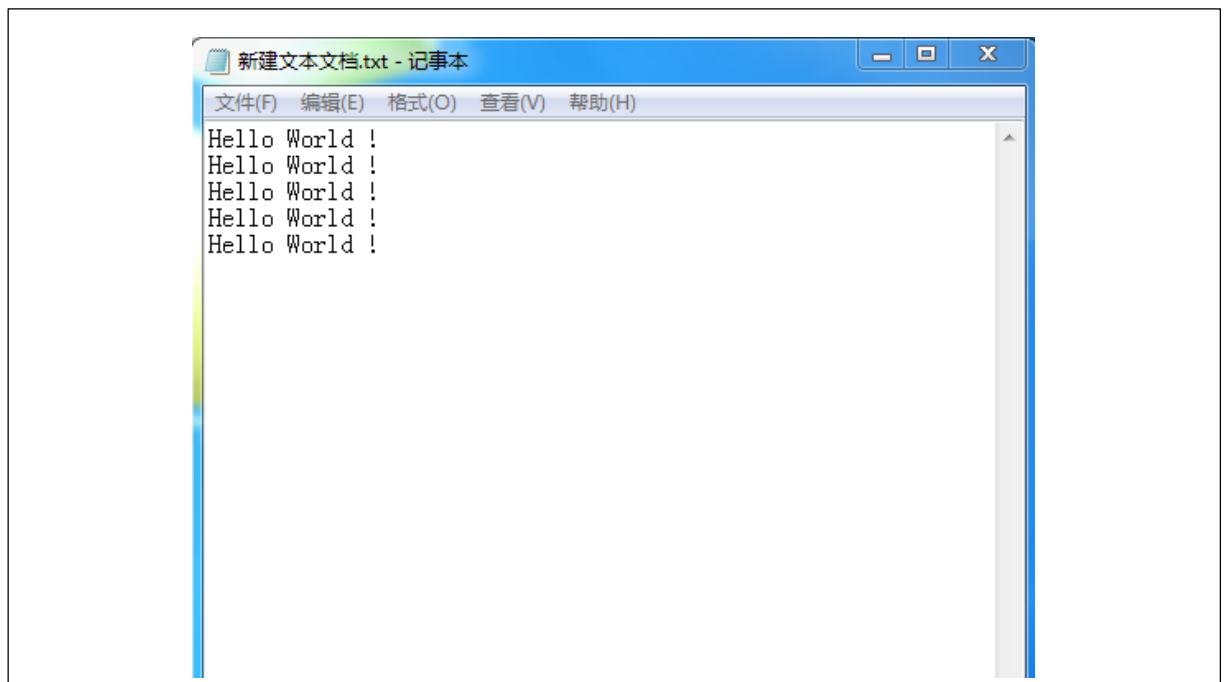
The FreeRTOS software timer is used to collect MCU temperature sensor value. The program initializes a software timer and defines the period of 2000 system clock ticks, which means that the

temperature sensor value is collected and print out every 2000 system clock ticks, as shown in the figure above. The ADC of AT32 MCU is also used for temperature sensor value collection.

3. Implement USB Keyboard function.

The USB Keyboard function is implemented on top of AT32F403 series MCU USB Device (not supported by AT32F413/415 series). Make sure that USB and PC are connected properly; then open a text or Word file and press the USER button, and "Hello World!" appears in the file, as shown below.

Figure 42. Demo 2



For details, please refer to the corresponding project.

17 Revision history

Table 56. Document revision history

Date	Version	Revision note
2021.12.16	2.0.0	Initial release.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

Purchasers are solely responsible for the selection and use of ARTERY's products and services; ARTERY assumes no liability for purchasers' selection or use of the products and the relevant services.

No license, express or implied, to any intellectual property right is granted by ARTERY herein regardless of the existence of any previous representation in any forms. If any part of this document involves third party's products or services, it does NOT imply that ARTERY authorizes the use of the third party's products or services, or permits any of the intellectual property, or guarantees any uses of the third party's products or services or intellectual property in any way.

Except as provided in ARTERY's terms and conditions of sale for such products, ARTERY disclaims any express or implied warranty, relating to use and/or sale of the products, including but not restricted to liability or warranties relating to merchantability, fitness for a particular purpose (based on the corresponding legal situation in any unjudicial districts), or infringement of any patent, copyright, or other intellectual property right.

ARTERY's products are not designed for the following purposes, and thus not intended for the following uses: (A) Applications that have specific requirements on safety, for example: life-support applications, active implant devices, or systems that have specific requirements on product function safety; (B) Aviation applications; (C) Aerospace applications or environment; (D) Weapons, and/or (E) Other applications that may cause injuries, deaths or property damages. Since ARTERY products are not intended for the above-mentioned purposes, if purchasers apply ARTERY products to these purposes, purchasers are solely responsible for any consequences or risks caused, even if any written notice is sent to ARTERY by purchasers; in addition, purchasers are solely responsible for the compliance with all statutory and regulatory requirements regarding these uses.

Any inconsistency of the sold ARTERY products with the statement and/or technical features specification described in this document will immediately cause the invalidity of any warranty granted by ARTERY products or services stated in this document by ARTERY, and ARTERY disclaims any responsibility in any form.

© 2021 Artery Technology -All Rights Reserved