STM32Cube高效开发教程（高级篇）

第7章 事件组

王维波
中国石油大学（华东）控制科学与工程学院

STM32Cube高效开发教程（高级篇）

作者：王维波，鄢志丹，王钊

人民邮电出版社

2022年2月出版

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

邮箱：wangwb@upc. edu. cn

第7章 事件组

7.1 事件组的原理和功能

7.2 事件组相关函数

7.3 事件组使用示例

7.4 通过事件组进行多任务同步

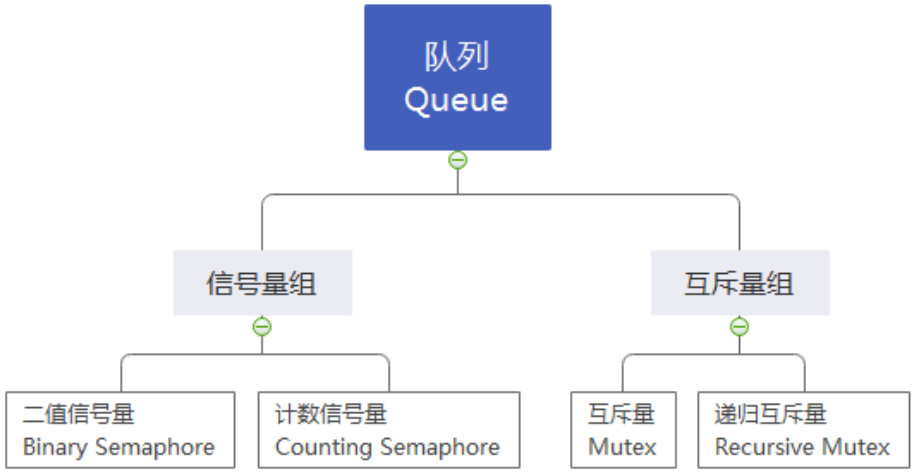
7.1 事件组的原理和功能

7.1.1 事件组的功能特点

7.1.2 事件组工作原理

7.1.1 事件组的功能特点

队列、信号量等进程间通信技术有如下的特点：



- 一个任务通常只处理一个事件，不能等待多个事件的发生，例如检测KeyLeft和KeyRight先后按下。如果需要处理多个事件，可能需要分解为多个任务，设置多个信号量。
- 可以有多个任务等待一个事件的发生，但是在事件发生时，只有最高优先级的任务解除阻塞状态，而不能同时解除多个任务的阻塞状态。

事件组（Event group）是FreeRTOS中另外一种进程间通讯方法，它与队列和信号量不同，具有自己的一些特点。

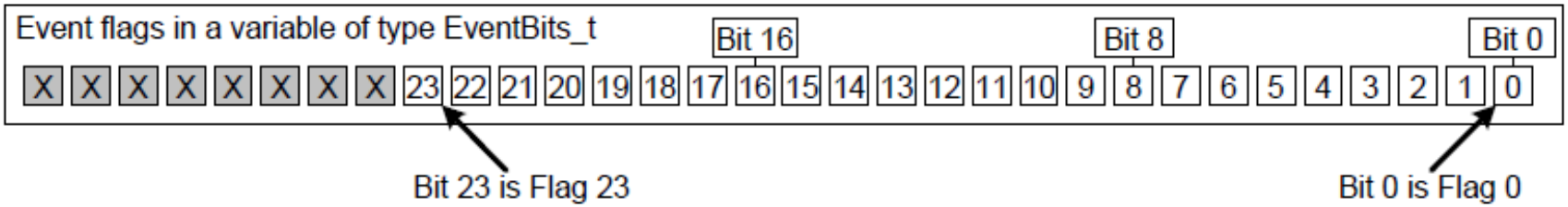
- 事件组允许任务等待一个或多个事件的组合，例如KeyLeft和KeyRight先后按下，或只有其中一个按键按下
- 事件组会解除所有等待同一事件的任务的阻塞状态，例如TaskA使用LED闪烁报警，TaskB使用蜂鸣器报警，当报警事件发生时，两个任务同时解除阻塞状态，两个任务都开始运行

事件组的这些特性使其适用于以下场景：

- 任务等待一组事件中的某个事件发生后做出响应（或运算关系），或一组事件都完成后做出响应（与运算关系）
- 将事件广播给多个任务，同时解锁多个任务
- 多个任务之间的同步

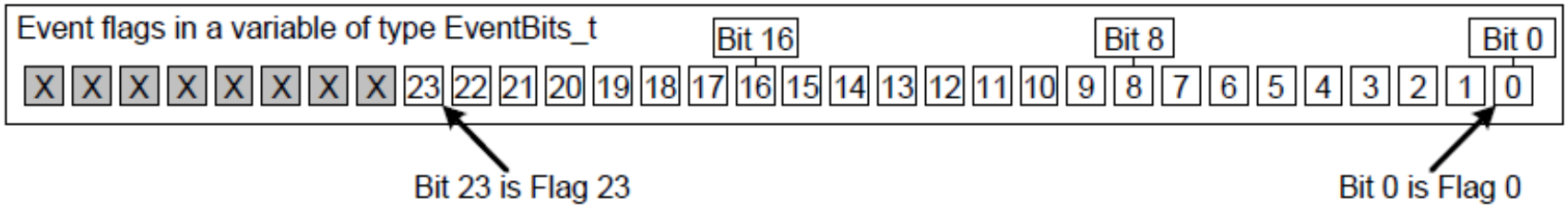
7.1.2 事件组工作原理

需要创建事件组对象。事件组有一个内部变量存储事件标志，当configUSE_16_BIT_TICKS为0时，这个变量是32位的，否则是16位的。在STM32上处理器上是32位的



事件标志只能是0或1，用单独的一个位来存储。一个事件组中的所有事件位保存在一个EventBits_t类型的变量里，所以一个事件又称为一个“事件位”

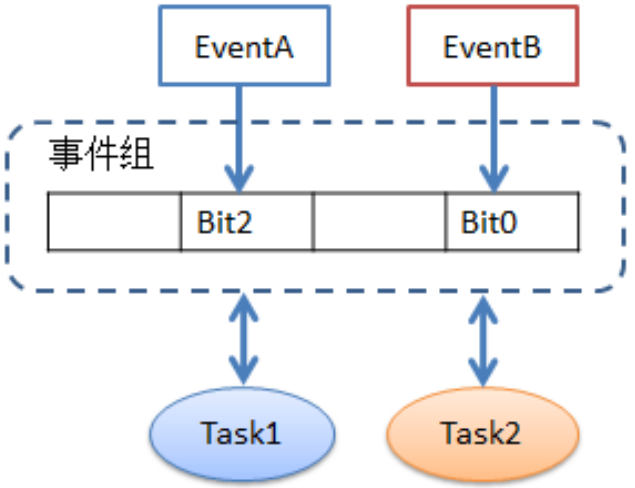
32位的事件组存储结构



- 31至24位是保留的，23至0位是事件位（Event Bits）
- 每一个位是一个事件标志（Event Flag），事件发生时将相应的位置1
- 所以32位的事件组可以最多可以处理24个事件

事件组运行原理

- 设置事件组中的位与某个事件对应，检测到事件发生时将相应的位置1，表示事件发生了。



- 可以有1个或多个任务等待事件组中的事件发生，可以是各个事件都发生（事件位的与运算），或某个事件发生（事件位的或运算）。
- 假设图中的Task1和Task2都以阻塞状态等待两个事件都发生，当Bit2和Bit0都被置为1后（不分先后顺序），两个任务都会被解除阻塞状态。所以，事件组具有广播功能。

7.2 事件组相关函数

7.2.1 相关函数概述

7.2.2 部分函数详解

7.2.1 相关函数概述

不能在CubeMX里可视化地创建事件组，需要自己编程

分组	函数	功能
事件组操作	xEventGroupCreate()	以动态分配内存方式创建事件组
	xEventGroupCreateStatic()	以静态分配内存方式创建事件组
	vEventGroupDelete()	删除已经创建的事件组
	vEventGroupSetNumber()	给事件组设置编号，编号的作用由用户定义
	uxEventGroupGetNumber()	读取事件组编号
事件位操作	xEventGroupSetBits()	将1个或多个事件位设置为1，设置的事件位用掩码表示
	xEventGroupSetBitsFromISR()	xEventGroupSetBits()的ISR版本
	xEventGroupClearBits()	清零某些事件位，清零的事件位用掩码表示
	xEventGroupClearBitsFromISR()	xEventGroupClearBits()的ISR版本
	xEventGroupGetBits()	返回事件组当前的值
	xEventGroupGetBitsFromISR()	xEventGroupGetBits()的ISR版本
等待事件	xEventGroupWaitBits()	进入阻塞状态，等待事件组合条件成立后解除阻塞状态
	xEventGroupSync()	用于多任务同步

1. 创建事件组

函数xEventGroupCreate()以动态分配内存方式创建事件组

```
EventGroupHandle_t  xEventGroupCreate( void );
```

创建事件组无需传递任何参数，函数返回的是所创建事件组的句柄变量，是一个指针变量，其他函数在操作事件组时，都需要使用事件组句柄变量作为输入参数。

类型EventGroupHandle_t的定义是

```
typedef void * EventGroupHandle_t;
```

2. 事件位置位

在某些事件发生时，用函数xEventGroupSetBits()在任务函数中将事件组的某些事件位置位

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

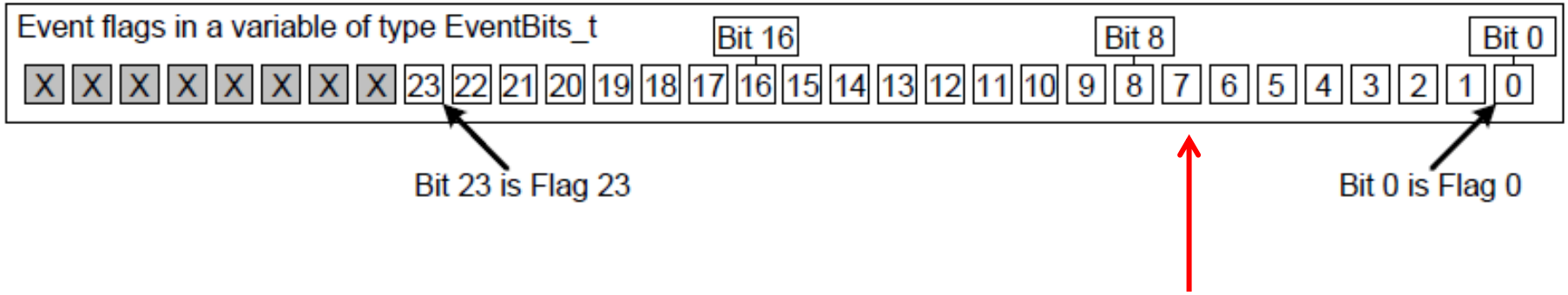
- xEventGroup 是所操作的事件组句柄
- uxBitsToSet 是所需要置位的事件位掩码
- 函数返回值类型是EventBits_t，是置位成功后事件组当前值

类型EventBits_t的定义如下，也就是TickType_t，在STM32处理器上，等同于类型uint32_t

```
typedef TickType_t  EventBits_t;
```

这个函数的使用关键是掩码uxBitsToSet的设置，需要置位的事件位在掩码中用1表示，其他位用0表示。

事件位的编号如图所示，如果需要置位事件组中的Bit7，则掩码是0x80；如果需要同时置位Bit7和Bit0，则掩码是0x81。一般情况下，一个事件只对应事件组中的一个事件位，一个事件发生时只需设置事件组中的一个位。



函数xEventGroupSetBitsFromISR()是在ISR中将事件组的某些事件位置位的函数，根据参数configUSE_TRACE_FACILITY的值是1或0，这个函数有两种不同的参数形式。默认的函数原型是

```
BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t xEventGroup,
    const EventBits_t uxBitsToSet, BaseType_t *pxHigherPriorityTaskWoken );
```

其中，pxHigherPriorityTaskWoken是指针BaseType_t*，是一个返回值（pdTRUE或pdFALSE），表示在退出ISR函数前是否需要申请进行一次任务调度。示意代码如下：

```
BaseType_t  highTaskWoken =pdFALSE;
xEventGroupSetBitsFromISR(xEventGroup, uxBitsToSet, &highTaskWoken);
portYIELD_FROM_ISR(highTaskWoken);   //申请进行一次任务调度
```

3. 事件位清零

函数xEventGroupClearBits()用于在任务函数中将事件组的某些事件位清零，其函数原型是

```
EventBits_t  xEventGroupClearBits( EventGroupHandle_t xEventGroup,
    const EventBits_t uxBitsToClear );
```

其中，uxBitsToClear是需要清零的事件位的掩码，需要清零的位设置为1。

函数的返回值是事件位被清零之前的事件组的值。

函数xEventGroupGetBitsFromISR()是ISR版本

```
BaseType_t xEventGroupClearBitsFromISR( EventGroupHandle_t
    xEventGroup, const EventBits_t uxBitsToSet);
```

在ISR函数中的事件位清零操作会被延后到定时器守护任务（timer daemon task）中去处理，函数的返回值为pdTRUE或pdFALSE，如果返回值为pdTRUE表示延后处理的消息成功发送给了定时器守护任务，否则就是没有成功发送。

4. 读取事件组当前值

函数xEventGroupGetBits()读取事件组当前的值

```
#define xEventGroupGetBits( xEventGroup )
    xEventGroupClearBits( xEventGroup, 0 )
```

它实际上就是执行了函数xEventGroupClearBits()，只是传递的事件位掩码是0，也就是不清除任何事件位，而返回事件组当前的值。

5. 等待事件组条件成立

函数xEventGroupWaitBits()使当前任务进入阻塞状态，以等待事件组中多个事件位表示的事件成立时再退出阻塞状态。事件组成立的条件可以是多个事件位都被置位（逻辑与运算），或其中某个事件位被置位（逻辑或运算）。

```
EventBits_t xEventGroupWaitBits( EventGroupHandle_t xEventGroup, const
    EventBits_t uxBitsToWaitFor, const BaseType_t xClearOnExit, const
    BaseType_t xWaitForAllBits, TickType_t xTicksToWait );
```

- xEventGroup，是所操作的事件组的句柄
- uxBitsToWaitFor，是所等待事件位的掩码。如果需要等待某个事件位置1，掩码中相应的位就设置为1

```
EventBits_t xEventGroupWaitBits( EventGroupHandle_t xEventGroup, const
    EventBits_t uxBitsToWaitFor, const BaseType_t xClearOnExit, const
    BaseType_t xWaitForAllBits, TickType_t xTicksToWait );
```

- xClearOnExit，设定值为pdTRUE或pdFALSE，退出时是否清楚掩码位
  - 如果设置为pdTRUE，则当函数在事件组条件成立而退出阻塞状态时，将清零掩码uxBitsToWaitFor中指定的所有位
  - 如果函数是因为超时而退出阻塞状态，即使xClearOnExit设置为pdTRUE也不会清零事件位

```
EventBits_t xEventGroupWaitBits( EventGroupHandle_t xEventGroup, const
    EventBits_t uxBitsToWaitFor, const BaseType_t xClearOnExit, const
    BaseType_t xWaitForAllBits, TickType_t xTicksToWait );
```

- xWaitForAllBits，设定值为pdTRUE或pdFALSE。是否等待所有位置位
  - 如果设置为pdTRUE，表示需要掩码中所有事件位都被置1才算条件成立（逻辑与运算）
  - 如果设置为pdFALSE，表示掩码中的某个事件位被置1条件就成立（逻辑或运算）
  - 当事件条件成立时函数就会退出，任务退出阻塞状态。
- xTicksToWait，在阻塞状态等待事件成立的超时节拍数

事件组区别于队列和信号量的主要特点

- 事件组可以等待多个事件发生后发生响应，而队列或信号量只能对一个事件做出响应。
- 可以有多个任务执行函数xEventGroupWaitBits()等待同一个事件组的同一个条件成立。当事件组条件成立时，多个任务都被解除阻塞状态，起到事件广播的作用。而使用队列或信号量时，当事件发生时只能有最高优先级的一个任务解除阻塞状态。

7.3 事件组使用示例

7.3.1 示例功能和CubeMX项目设置

7.3.2 程序功能实现

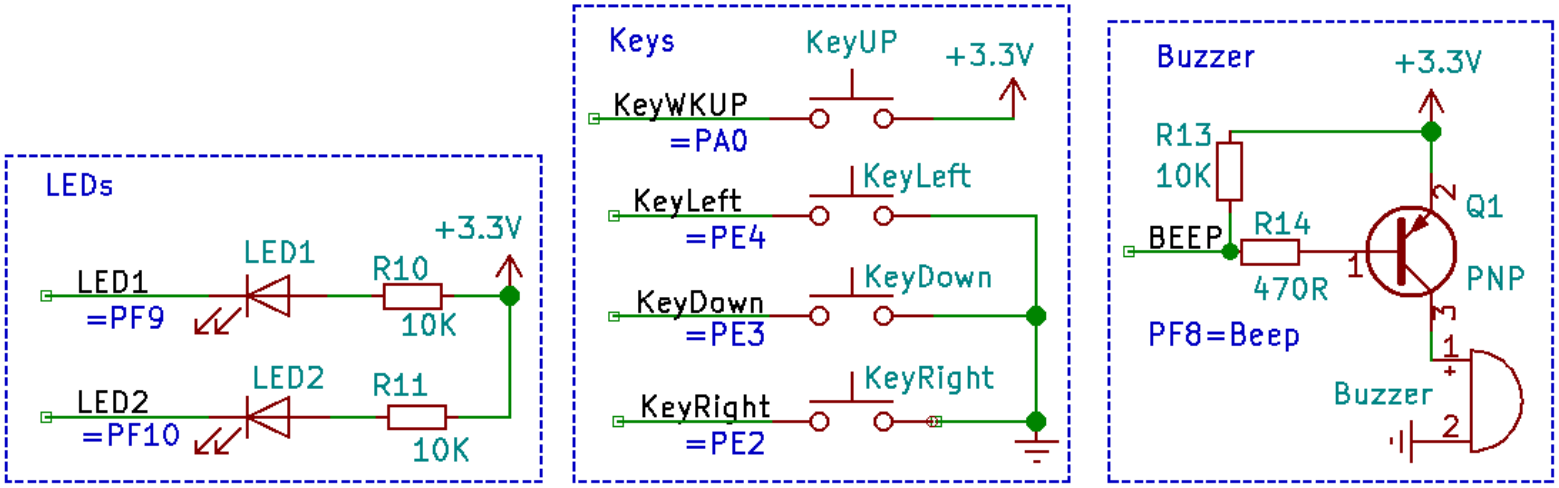
7.3.1 示例功能和CubeMX项目设置

示例Demo7_1EventGroup演示事件组的使用：

- 创建1个事件组和3个任务。
- 在任务Task_ScanKeys中检测KeyLeft和KeyRight两个按键是否按下，按下时将事件组中对应的事件位置位。检测到KeyDown按下时清零事件组。
- 任务Task_LED和Task_Buzzer都等待事件组中两个按键都按下的事件，条件成立时，任务Task_LED使LED1闪烁几次，Task_Buzzer使蜂鸣器响几次。

（1）按键、LED1和蜂鸣器的GPIO设置

本示例用到3个按键、LED1和蜂鸣器



Pin Name	GPIO mode	GPIO Pull-up/Pull-down	GPIO output level	User Label
PE2	Input mode	Pull-up	n/a	KeyRight
PE3	Input mode	Pull-up	n/a	KeyDown
PE4	Input mode	Pull-up	n/a	KeyLeft
PF8	Output Push Pull	No pull-up and no pull-down	High	Buzzer
PF9	Output Push Pull	No pull-up and no pull-down	Low	LED1

（2）FreeRTOS的设置

设计3个不同优先级的任务。任务Task_ScanKeys用于检测按键的状态，所以其优先级最高。其他两个任务是对事件组的响应，设置为两个不同的优先级是为了测试它们是否被同时解除阻塞状态。

Tasks

Task Name	Priority	Stack Size (Words)	Entry Function	Allocation
Task_Buzzer	osPriorityNormal	128	AppTask_Buzzer	Dynamic
Task_LED	osPriorityBelowNormal	128	AppTask_LED	Dynamic
Task_ScanKeys	osPriorityAboveNormal	128	AppTask_ScanKeys	Dynamic

在CubeMX中没有用于设计事件组的界面，不能在CubeMX里可视化地设计事件组，需要在导出的初始代码基础上编写代码创建事件组。

7.3.2 程序功能实现

1.主程序

```
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();                    //GPIO引脚初始化
  MX_FSMC_Init();

  /* USER CODE BEGIN 2 */
  TFTLCD_Init();                     //LCD 初始化
  LCD_ShowString(10, 10, (uint8_t *)"Demo7_1:Using Events Group");
  LCD_ShowString(10, 30, (uint8_t *)"1.Press KeyLeft and KeyRight");
  LCD_ShowString(10, 50, (uint8_t *)"  to activate buzzer and LED1");
  LCD_ShowString(10, 70, (uint8_t *)"2.Press KeyDown to clear events");
  /* USER CODE END 2 */

   osKernelInitialize();
   MX_FREERTOS_Init();
   osKernelStart();
while (1)
  { }
}
```

2. FreeRTOS初始化和对象的创建

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "event_groups.h"                    //事件组相关头文件
/* USER CODE END Includes */

/* Private variables ---------------------------------------------------------*/
/* USER CODE BEGIN Variables */
EventGroupHandle_t  eventGroupHandle;          //事件组对象句柄
/* USER CODE END Variables */

void MX_FREERTOS_Init(void)
{
      /* USER CODE BEGIN RTOS_QUEUES */
      eventGroupHandle= xEventGroupCreate();    //创建事件组
      /* USER CODE END RTOS_QUEUES */

      /* 创建任务 Task_Buzzer */
      Task_BuzzerHandle = osThreadNew(AppTask_Buzzer, NULL, &Task_Buzzer_attributes);

.....//创建其余2个任务的代码略
}
```

3. 三个任务的功能实现

使用了KeyLeft和KeyRight两个事件，对应于事件组中的两个事件位。定义了两个事件位掩码，即

```
#define  BITMASK_KEY_LEFT      0x04  //KeyLeft的事件位掩码，使用Bit2位
#define  BITMASK_KEY_RIGHT     0x01  //KeyRight的事件位掩码，使用Bit0位
```

任务Task_ScanKeys的功能就是扫描按键，当KeyLeft或KeyRight按下时置位相应的事件位， KeyDown按下时将两个事件位都清零。

```
for(;;)
{
        EventBits_t curBits=xEventGroupGetBits(eventGroupHandle);  //读取事件组当前值
        LCD_ShowUintHex(LcdX, 150, curBits, 1); //16进制显示
        keyCode= ScanPressedKey(50);     //最多等待50ms，不能使用参数KEY_WAIT_ALWAYS
        switch (keyCode)
        {
        case   KEY_LEFT:      //事件位Bit2置位
           xEventGroupSetBits(eventGroupHandle, BITMASK_KEY_LEFT);
           break;

        case   KEY_RIGHT:    //事件位Bit0置位
           xEventGroupSetBits(eventGroupHandle, BITMASK_KEY_RIGHT);
           break;

        case   KEY_DOWN:   //清除两个事件位
           xEventGroupClearBits(eventGroupHandle,
                 BITMASK_KEY_LEFT | BITMASK_KEY_RIGHT);
        }

        if (keyCode==KEY_NONE)
           vTaskDelay(50);      //未按下任何按键，延时不能太长，否则按键响应慢
        else
           vTaskDelay(200); //消除按键抖动影响，也用于事件调度
}
```

代码较长，看源程序

任务Task_LED的中使用函数xEventGroupWaitBits()等待事件组中的条件成立，事件成立后使LED1闪烁几次。

```
void AppTask_LED(void *argument)   //任务Task_LED，事件条件成立时闪烁LED1
{
  /* USER CODE BEGIN AppTask_LED */
        BaseType_t clearOnExit=pdTRUE;    // pdTRUE=退出时清除事件位
        BaseType_t waitForAllBits=pdTRUE; //pdTRUE=逻辑与, pdFALSE=逻辑或
        EventBits_t bitsToWait=BITMASK_KEY_LEFT | BITMASK_KEY_RIGHT;  //等待的事件位
        for(;;)
        {
           EventBits_t result= xEventGroupWaitBits(eventGroupHandle, bitsToWait,
                       clearOnExit, waitForAllBits, portMAX_DELAY );
         for(uint8_t i=0; i<10; i++)
         {     //使LED1闪烁几次
           HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
           vTaskDelay(pdMS_TO_TICKS(500));
         }
        }
  /* USER CODE END AppTask_LED */
}
```

任务Task_Buzzer等待同样的事件组事件，事件成立后使蜂鸣器响几次

```
void AppTask_Buzzer(void *argument)   //任务Task_Buzzer，事件条件成立时蜂鸣器发声几次
{
  /* USER CODE BEGIN AppTask_Buzzer */
        BaseType_t clearOnExit=pdTRUE;   // pdTRUE=退出时清除事件位
        BaseType_t waitForAllBits=pdTRUE;//pdTRUE=逻辑与, pdFALSE=逻辑或
        EventBits_t bitsToWait=BITMASK_KEY_LEFT |BITMASK_KEY_RIGHT;  //等待的事件位
        for(;;)
        {
         EventBits_t result= xEventGroupWaitBits(eventGroupHandle, bitsToWait,
                        clearOnExit,waitForAllBits,portMAX_DELAY );
         for(uint8_t i=0; i<10; i++)
         {
            HAL_GPIO_TogglePin(Buzzer_GPIO_Port, Buzzer_Pin);
            vTaskDelay(pdMS_TO_TICKS(500));
         }
        }
  /* USER CODE END AppTask_Buzzer */
}
```

4. 程序运行测试

运行时可以发现：先后按下KeyLeft和KeyRight后，LED1闪烁，蜂鸣器发声，说明两个任务都被解除了阻塞状态，虽然两个任务的xEventGroupWaitBits()函数中的参数clearOnExit都设置为pdTRUE，且两个任务的优先级不同。

7.4 通过事件组进行多任务同步

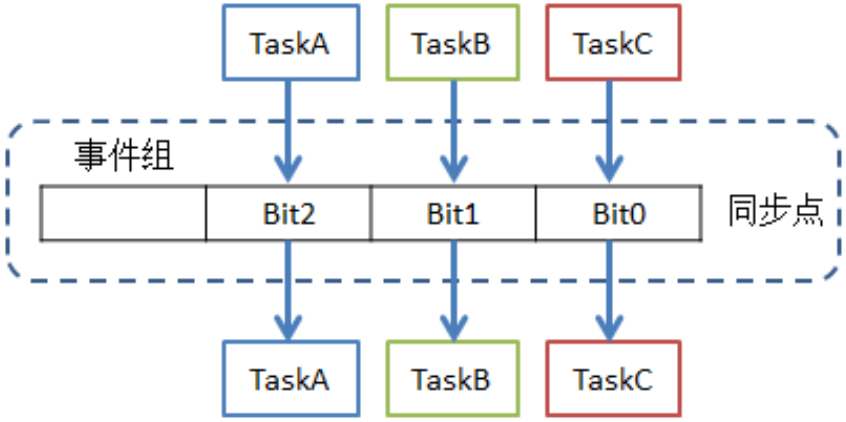
7.4.1 多任务同步原理

7.4.2 示例功能和CubeMX项目设置

7.4.3 程序功能实现

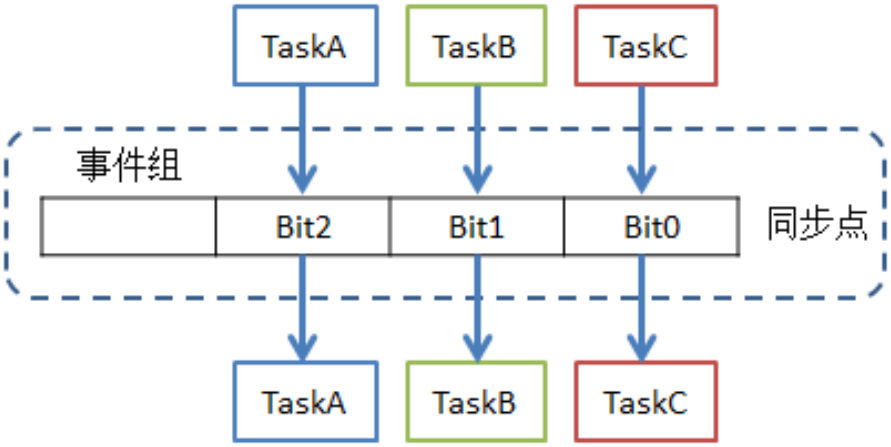
7.4.1 多任务同步原理

3个任务分别对应事件组中的3个事件位。这3个任务需要在等待某个条件成立之后再开始同步执行各自后面的程序，这个位置称为同步点（synchronization point）



- TaskA检测到KeyLeft按下了后到达了同步点，置位Bit2，它还需要等待Bit1和Bit0被置位后再运行后面的程序。
- TaskB检测到KeyDown按下了后到达了同步点，置位Bit1，它还需要等待Bit2和Bit0被置位后再运行后面的程序。
- TaskC检测到KeyRight按下了后到达了同步点，置位Bit0，它还需要等待Bit2和Bit1被置位后再运行后面的程序。

这3个任务在同步点将各自的事件位置1之后，再等待其他事件位置1后才开始运行，从而达到多个任务在某个同步点同步运行的目的。



任务的这个操作过程可以先后执行xEventGroupSetBits()和xEventGroupWaitBits()完成，但是这样就不是一步操作。函数xEventGroupSync()可以替代这两个函数实现一步操作，用于实现多任务之间的同步。

函数xEventGroupSync()的函数原型是：

```
EventBits_t xEventGroupSync( EventGroupHandle_t xEventGroup, const
    EventBits_t uxBitsToSet, const EventBits_t uxBitsToWaitFor, TickType_t
    xTicksToWait )
```

- xEventGroup，是所操作的事件组对象
- uxBitsToSet，是任务要置位的事件位的掩码，例如任务TaskA置位Bit2，那么uxBitsToSet就设置为0x04
- uxBitsToWaitFor，是需要等待的同步条件，是事件组中需要被置1的事件位的掩码。例如，在Bit2、Bit1、Bit0位都被置1，那么uxBitsToWaitFor就设置为0x07
- xTicksToWait，是任务在阻塞状态下等待的节拍数

7.4.2 示例功能和CubeMX项目设置

示例Demo7_2EventsSync演示通过事件组进行3个任务同步的功能。示例的主要功能和工作流程是：

- 创建3个任务，这3个任务分别检测按键KeyLeft、KeyDown和KeyRight的按下状态。
- 任务检测到所管理的按键按下后，调用xEventGroupSync()置位所对应的事件位，并开始等待同步条件成立。
- 3个任务的同步条件是3个按键都被按下，同步后任务1使LED1闪烁，任务2使LED2闪烁，任务3使蜂鸣器发声。

（1）按键、LED和蜂鸣器的GPIO设置

本示例中用到KeyLeft、KeyDown、KeyRight，LED1和LED2，蜂鸣器。

在CubeMX中设置相关GPIO引脚的工作模式、上拉或下拉、初始输出电平等参数，并设置了引脚的用户标签

Pin Name	GPIO mode	GPIO Pull-up/Pull-down	GPIO output level	User Label
PF10	Output Push Pull	No pull-up and no pull-down	Low	LED2
PF9	Output Push Pull	No pull-up and no pull-down	Low	LED1
PF8	Output Push Pull	No pull-up and no pull-down	High	Buzzer
PE4	Input mode	Pull-up	n/a	KeyLeft
PE3	Input mode	Pull-up	n/a	KeyDown
PE2	Input mode	Pull-up	n/a	KeyRight

（2）FreeRTOS的设置

设计3个任务，3个任务的主要属性如图所示。

3个任务使用了相同的优先级，使用不同的优先级也是没有问题的。

Tasks

Task Name	Priority	Stack Size (Words)	Entry Function	Allocation
Task_Buzzer	osPriorityNormal	128	AppTask_Buzzer	Dynamic
Task_LED1	osPriorityNormal	128	AppTask_LED1	Dynamic
Task_LED2	osPriorityNormal	128	AppTask_LED2	Dynamic

7.4.3 程序功能实现

1. 主程序

```
int main(void)
{
  HAL_Init();                       //外设复位， 系统初始化
  SystemClock_Config();             //系统时钟设置
  /* Initialize all configured peripherals */
  MX_GPIO_Init();                   //GPIO引脚初始化
  MX_FSMC_Init();

  /* USER CODE BEGIN 2 */
  TFTLCD_Init();                    //LCD 初始化
  LCD_ShowString(10, 10, (uint8_t *)"Demo7_2:Events Synchronization");
  LCD_ShowString(10, 30, (uint8_t *)"Press KeyLeft, KeyRight, KeyDown");
  LCD_ShowString(10, 50, (uint8_t *)"      to start test");
  LCD_ShowString(10, 70, (uint8_t *)"Press Reset to restart");
  /* USER CODE END 2 */

   osKernelInitialize();
   MX_FREERTOS_Init();
   osKernelStart();
while (1)
  {  }
}
```

2. FreeRTOS初始化和对象的创建

函数 MX_FREERTOS_Init()里创建事件组和3个任务。

```
/* Private variables -----------------------------------------------------------*/
/* USER CODE BEGIN Variables */
EventGroupHandle_t eventGroupHandle;                    //事件组句柄
/* USER CODE END Variables */

void MX_FREERTOS_Init(void)
{
    /* USER CODE BEGIN RTOS_QUEUES */
    eventGroupHandle= xEventGroupCreate();    //创建事件组
    /* USER CODE END RTOS_QUEUES */

    /* 创建任务 Task_Buzzer */
    Task_BuzzerHandle = osThreadNew(AppTask_Buzzer, NULL,
    &Task_Buzzer_attributes);

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

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

3. 三个任务的功能实现

程序中使用了事件组中的3个位，定义了4个事件位掩码，其中BITMASK_SYNC是其他3个事件位掩码的按位或运算，所以BITMASK_SYNC是0x07。

```
/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
#define  BITMASK_KEY_LEFT                0x04        //事件位掩码定义
#define  BITMASK_KEY_DOWN                0x02
#define  BITMASK_KEY_RIGHT               0x01
#define  BITMASK_SYNC   BITMASK_KEY_LEFT | BITMASK_KEY_DOWN | BITMASK_KEY_RIGHT
/* USER CODE END PD */
```

任务Task_LED1

检测到KeyLeft按下后置位0x04，等待事件组变为0x07时再同步运行。

```
void AppTask_LED1(void *argument)       //任务Task_LED1
{
  /* USER CODE BEGIN AppTask_LED1 */
        for(;;)
        {
          KEYS curKey=ScanPressedKey(50);   //检测KeyLeft键
          if (curKey != KEY_LEFT)    //KeyLeft键没有被按下
          {
             vTaskDelay(pdMS_TO_TICKS(50));
             continue;
          }
          LCD_ShowStr(10, LCD_CurY+LCD_SP15, (uint8_t *)"Task_LED1 reaches sync point");
          xEventGroupSync(eventGroupHandle, BITMASK_KEY_LEFT,
                          BITMASK_SYNC, portMAX_DELAY);    //同步点，等待同步
          while(1)
          {
             LED1_Toggle();
             vTaskDelay(pdMS_TO_TICKS(500));
          }
        }
/* USER CODE END AppTask_LED1 */
}
```

任务Task_LED2

检测到KeyRight按下后置位0x01，并等待事件组变为0x07时再同步运行。

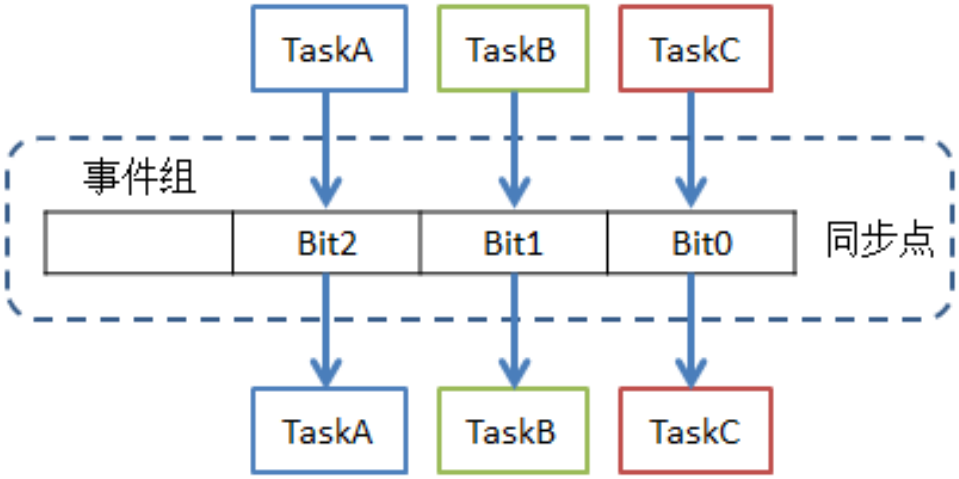
```
void AppTask_LED2(void *argument)       //任务Task_LED2
{
  /* USER CODE BEGIN AppTask_LED2 */
        for(;;)
        {
          KEYS curKey=ScanPressedKey(50);       //检测KeyRight键
          if (curKey != KEY_RIGHT)          //KeyRight键没有被按下
          {
             vTaskDelay(pdMS_TO_TICKS(50));
             continue;
          }
          LCD_ShowStr(10, LCD_CurY+LCD_SP15, (uint8_t *)"Task_LED2 reaches sync point");
          xEventGroupSync(eventGroupHandle, BITMASK_KEY_RIGHT,
                    BITMASK_SYNC, portMAX_DELAY);   //同步点，等待同步
          while(1)
          {
             LED2_Toggle();
             vTaskDelay(pdMS_TO_TICKS(500));
          }
        }
/* USER CODE END AppTask_LED2 */
}
```

任务Task_Buzzer

检测到KeyDown按下后置位0x02，并等待事件组变为0x07时再同步运行。

```
void AppTask_Buzzer(void *argument)     //任务Task_Buzzer
{
  /* USER CODE BEGIN AppTask_Buzzer */
        for(;;)
        {
          KEYS curKey=ScanPressedKey(50);       //检测KeyDown键
          if (curKey != KEY_DOWN)               //KeyDown键没有被按下
          {
             vTaskDelay(pdMS_TO_TICKS(50));
             continue;
          }
          LCD_ShowStr(10, LCD_CurY+LCD_SP15, (uint8_t *)"Task_Buzzer reaches sync point");
          xEventGroupSync(eventGroupHandle, BITMASK_KEY_DOWN,
                    BITMASK_SYNC, portMAX_DELAY);  //同步点，等待同步
          while(1)
          {
             Buzzer_Toggle();
             vTaskDelay(pdMS_TO_TICKS(500));
          }
        }
/* USER CODE END AppTask_Buzzer */
}
```

当事件组中掩码BITMASK_SYNC表示的3个位都被置1后，3个任务都同时被解除阻塞状态，继续执行各自后面的程序，这样就实现了3个任务的同步。



练习任务

1. 看教材，练习本章的示例。