5.事件组

2025年10月6日 11:42

一、事件组的引入

- 假设我们有一个智能家居的“空调控制任务”，它需要满足以下条件才能启动：
  - 1.室内传感器检测到室内温度高于26°（事件A）
  - 2.用户按下了“开关”按钮（事件B）
  - 3.电网电压稳定（事件C）
- 这三个事件分别是由三个独立的任务（传感器检测任务，按键任务，电压检测任务）负责监测，现在有一个问题是：空调控制任务如何同时等待三个事件都满足？
- 此时有人可能会说，我们可以借助二值信号量来实现，我们创建一个二值信号量，让三个任务依次获取二值信号量，不就可以实现三个事件同时满足了吗，我们来看一下二值信号量的实现代码

```c
// 错误示例：用信号量依次等待多个事件
void vAirConditionTask(void *pvParam) {
    while(1) {
        // 先等事件A（温度达标）
        xSemaphoreTake(xSemaphoreA, portMAX_DELAY);
        // 再等事件B（按键按下）
        xSemaphoreTake(xSemaphoreB, portMAX_DELAY);
        // 最后等事件C（电压稳定）
        xSemaphoreTake(xSemaphoreC, portMAX_DELAY);

        // 三个事件都满足，启动空调
        start_air_condition();
    }
}
```

- 这个逻辑看似可行，但是它存在一个**致命的缺陷：事件发生的顺序被强行绑定了，必须按照A，B，C去执行**
- 比如，如果用户先按下按键，释放了B事件的信号量，但是此时温度还未达标（事件A还未发生），那么B事件的信号量就会被浪费--因为空调任务此时还在等待事件A，等它轮到事件B时，事件B的信号量已经被提前消耗了（信号量时一次性的），导致事件B即使后续发生，任务也会一直阻塞等待事件B的步骤
- **更深层次的问题：信号量无法记住事件状态**
  - **信号量本质是一次性通知**：当一个**事件发生时，信号量被释放**，但如果此时等待任务还在处理其他任务，这个通知就会丢失（**信号量值从1变成0后，无法表示这个事件曾经发生过**），而**在多任务场合，任务需要直到所有任务是否已经发生过（不管顺序），这就需要一个机制能“记住每个事件的状态”**，并允许任务一次性检查多个任务的状态

二、事件组的诞生

- 事件组是 FreeRTOS 中用于**多任务间同步 / 通信**的核心机制，本质是一个**带位操作的整数变量**（**通常为 8/16/32 位，取决于 MCU 架构**），每个位代表一个 “事件”，**任务可通过 “等待指定事件位” 或 “设置 / 清除事件位” 实现灵活协作。**
- 它**用一个整数变量的每个bit位代表一个事件的状态（bit=1表示事件已经发生过，bit=0表示事件未发生）**，并提供API让任务可以：
  - 1.**记住所有任务的状态**：即使事件发生时任务正在处理其他事件，事件位也会置1，**不会丢失**
  - 2.**一次性等待多个事件的组合**：可以指定“**等待所有的事件都完成（AND）**”或“**等待其中一个事件完成（OR）**”，且**与事件发生的顺序无关**
  - 3.**灵活控制事件的生命周期**：任务被唤醒后，可以**选择自动清楚以满足事件的标志位，或者选择在合适的时机手动删除**

三、事件组的核心概念：3个关键属性

- **1.事件位（Event Bits）：事件组的最小单位，一个bit位代表一个事件**
  - 注意：不同 FreeRTOS 版本支持的最大事件位数不同，常见为 **configUSE_16_BIT_TICKS 使能时是 16 位，禁用时是 32 位**，实际使用需参考配置文件。
- **2.任务等待条件：任务等到事件时，需要指定两个核心参数：**

- **事件掩码（Event Mark）**：明确要等待那些事件位
- **等待方式（Wait Mode）**：决定“满足多少事件才算完成等待”，分两种：
  - 逻辑与（AND）：等待掩码所有位都置1才唤醒
  - 逻辑或（OR）：等待掩码中任意一位置1就唤醒

- **3.事件清除策略：任务被唤醒后，事件位是否保留，分两种：**
  - 自动清除：任务被唤醒的同时，自动清除掩码中被置1的位
  - 手动清除：任务被唤醒后事件位保持不动，后续需要手动调用函数清除

四、事件组的创建

- 创建事件组有专门的API函数，FreeRTOS 提供的事件组 API 均以 xEventGroup 开头，需包含头文件 **event_groups.h**，且需在 FreeRTOSConfig.h 中开启 **configUSE_EVENT_GROUPS = 1。**
- **xEventGroupCreate ()** 这是创建事件组用到的API函数，我们来深入底层代码看一下

```
#if ( configSUPPORT_DYNAMIC_ALLOCATION == 1 )

    Trae: 解释代码 | 注释代码 | ×
    EventGroupHandle_t xEventGroupCreate( void )
    {
        EventGroup_t * pxEventBits;
        traceENTER_xEventGroupCreate();
        pxEventBits = ( EventGroup_t * ) pvPortMalloc( sizeof( EventGroup_t ) );

        if( pxEventBits != NULL )
        {
            pxEventBits->uxEventBits = 0;
            vListInitialise( &( pxEventBits->xTasksWaitingForBits ) );
            traceEVENT_GROUP_CREATE( pxEventBits );
        }
        else
        {
            traceEVENT_GROUP_CREATE_FAILED();
        }

        traceRETURN_xEventGroupCreate( pxEventBits );

        return pxEventBits;
    }

#endif /* configSUPPORT_DYNAMIC_ALLOCATION */
```

- 这是一个**无参有返回值**的函数，创建成功后**返回一个事件组句柄**，句柄的类型时**EventGroupHandle_t类型**
- 首先函数一开始就会初始化一个**事件组结构体（EventGroup_t）**，下面**从栈中分配内存空间**，也是根据事件组结构体去分配的，那么我们需要深入观察一下事件组结构体（EventGroup_t）的底层代码时如何实现的、

```
typedef struct EventGroupDef_t
{
    EventBits_t uxEventBits;
    List_t xTasksWaitingForBits; /**< List of tasks waiting for a bit to be set. */

    #if ( configUSE_TRACE_FACILITY == 1 )
        UBaseType_t uxEventGroupNumber;
    #endif

    #if ( ( configSUPPORT_STATIC_ALLOCATION == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) )
        uint8_t ucStaticallyAllocated; /**< Set to pdTRUE if the event group is statically allocated
    #endif
} EventGroup_t;
```

- 我们可以看到事件组结构体的底层就是**初始化了事件组中的每一个事件位，以及还初始化了一个事件组等待任务置位的链表。**

- 这两个就是事件组结构体中的关键，也是事件组实现的关键。看完事件组结构体的底层的代码实现，我们在来看看事件组创建部分的代码。
- 在成功分配了事件组结构体内存空间之后，我们**要将事件组中的所有事件组标志位全都初始为0**，并且**初始化事件组的等待任务置位链表，最后返回一个事件组结构体。**

五、事件组的关键API函数：事件组的使用

1.事件组设置事件位API函数：xEventGroupSetBits（）

- 这个函数的功能是：**将事件组中一个或多个事件位置1**
- 函数的原型：**EventBits_t xEventGroupSetBits(EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet)**

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

- 这是一个**有参有返回值**的函数
- **返回值**：在设置完成后，函数会**返回事件组的当前值（就是被置1的位转换成十六进制的值）**
- **参数**：**xEventGroup**：目标事件组句柄

  uxBitsToSet：要置1的事件位

```
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup,
                                const EventBits_t uxBitsToSet )
{
    EventGroup_t * pxEventBits = xEventGroup;
    BaseType_t xMatchFound = pdFALSE;
    traceENTER_xEventGroupSetBits( xEventGroup, uxBitsToSet );
    configASSERT( xEventGroup );
    configASSERT( ( uxBitsToSet & eventEVENT_BITS_CONTROL_BYTES ) == 0 );
    pxList = &( pxEventBits->xTasksWaitingForBits );
    pxListEnd = listGET_END_MARKER( pxList );
    vTaskSuspendAll();
    {
        traceEVENT_GROUP_SET_BITS( xEventGroup, uxBitsToSet );
        pxListItem = listGET_HEAD_ENTRY( pxList );
        pxEventBits->uxEventBits |= uxBitsToSet;
        while( pxListItem != pxListEnd )
        {
            pxNext = listGET_NEXT( pxListItem );
            uxBitsWaitedFor = listGET_LIST_ITEM_VALUE( pxListItem );
            xMatchFound = pdFALSE;
            uxControlBits = uxBitsWaitedFor & eventEVENT_BITS_CONTROL_BYTES;
            uxBitsWaitedFor &= ~eventEVENT_BITS_CONTROL_BYTES;
            if( ( uxControlBits & eventWAIT_FOR_ALL_BITS ) == ( EventBits_t ) 0 )
            {
                if( ( uxBitsWaitedFor & pxEventBits->uxEventBits ) != ( EventBits_t ) 0 )
                {
                    xMatchFound = pdTRUE;
                }
```

```
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
else if( ( uxBitsWaitedFor & pxEventBits->uxEventBits ) == uxBitsWaitedFor )
{
    xMatchFound = pdTRUE;
}
else
{
    /* Need all bits to be set, but not all the bits were set. */
}

if( xMatchFound != pdFALSE )
{
    /* The bits match.  Should the bits be cleared on exit? */
    if( ( uxControlBits & eventCLEAR_EVENTS_ON_EXIT_BIT ) != ( EventBits_t ) 0 )
    {
        uxBitsToClear |= uxBitsWaitedFor;
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
```

-

```
            /* Store the actual event flag value in the task's event list
             * item before removing the task from the event list.  The
             * eventUNBLOCKED_DUE_TO_BIT_SET bit is set so the task knows
             * that is was unblocked due to its required bits matching, rather
             * than because it timed out. */
            vTaskRemoveFromUnorderedEventList( pxListItem, pxEventBits->uxEventBits | eventUNBLOCKED_DUE_TO_BIT_SET );
        }

        /* Move onto the next list item.  Note pxListItem->pxNext is not
         * used here as the list item may have been removed from the event list
         * and inserted into the ready/pending reading list. */
        pxListItem = pxNext;
    }

    /* Clear any bits that matched when the eventCLEAR_EVENTS_ON_EXIT_BIT
     * bit was set in the control word. */
    pxEventBits->uxEventBits &= ~uxBitsToClear;
}
( void ) xTaskResumeAll();

traceRETURN_xEventGroupSetBits( pxEventBits->uxEventBits );

return pxEventBits->uxEventBits;
```

唤醒是满足条件的所有等待任务

- **实现原理：**
  - 函数的实现大致可以分为这几步：
  - **1.参数验证：**

```c
configASSERT( xEventGroup );
configASSERT( ( uxBitsToSet & eventEVENT_BITS_CONTROL_BYTES ) == 0 );
```

  - 确保事件标志组句柄有效且没有尝试设置内核保留的控制位。
  - **2.暂停任务调度**

```c
vTaskSuspendAll();
```

  - 为**确保操作的原子性，保证能够完整的执行完事件位的设置，以免数据错乱，暂停所有任务的调度，防止被其他任务抢占。**

- **3.设置指定的位**

```c
pxEventBits->uxEventBits |= uxBitsToSet;
```

- 使用**位或操作**将传进来要**置位的事件组标志位置1**
- **4.检查并唤醒所有满足条件的等待任务**

- 如果任务设置为等待任意位（`eventWAIT_FOR_ALL_BITS` 未设置），只要有一个等待的位被设置，就唤醒该任务
- 如果任务设置为等待所有位（`eventWAIT_FOR_ALL_BITS` 已设置），则需要所有等待的位都被设置才唤醒该任务

- **5.清除需要清除的位**
- 如果任务在等待时设置了 **eventCLEAR_EVENTS_ON_EXIT_BIT** 标志，则在唤醒任务前清除相应的位。
- **6.恢复任务调度**

```c
( void ) xTaskResumeAll();
```

- 完成所有操作后，**恢复任务调度**

- **注意事项：**
  - 此函数不能在中断服务程序中调用，中断中应使用 xEventGroupSetBitsFromISR
  - 事件标志组中的**高位部分（由 eventEVENT_BITS_CONTROL_BYTES 定义）是内核保留的**，应用程序不应尝试设置这些位
  - 当多个任务等待同一事件标志组时，所有满足条件的任务都会被唤醒
- 这就是设置标志位函数的实现，接下来我们来看一下等待标志位的实现函数

- **EventBits_t xEventGroupWaitBits（EventGroupHandle_t xEventGroup,　//事件组句柄**
  const EventBits_t uxBitsToWaitFor,　//要等待那些位被设置
  const BaseType_t xClearOnExit,　//自动清除标志位，PdTRUE-自动清除，pdFALSE-不自动清除
  const BaseType_t xWaitForAllBits,　//等待方式，pdTRUE-全满足，pdFALSE-任一满足
  TickType_t xTicksToWait）　//超时等待时间
  - 这是一个**有参有返回值**的函数
  - 其中**返回值**是：如果在超时前返回则**返回值为此时的掩码值（就是设置位的十六进制值），超时返回0**
  - 各个参数的含义在每个参数后面都有注释

参数说明

1. xEventGroup：事件标志组句柄
2. uxBitsToWaitFor：要等待的位掩码（例如：等待位0和位4，传入 `(1 << 0) | (1 << 4)`）
3. xClearOnExit：退出时是否清除等待的位
   - `pdTRUE`：退出时清除已设置的等待位
   - `pdFALSE`：退出时保持位不变
4. xWaitForAllBits：等待条件
   - `pdTRUE`：等待所有指定的位都被设置
   - `pdFALSE`：等待任意一个指定的位被设置
5. xTicksToWait：最大等待时间（以系统节拍为单位）

  - 接下来我们详细看一下代码的底层实现逻辑

```c
EventBits_t xEventGroupWaitBits( EventGroupHandle_t xEventGroup,
                                 const EventBits_t uxBitsToWaitFor,
                                 const BaseType_t xClearOnExit,
                                 const BaseType_t xWaitForAllBits,
                                 TickType_t xTicksToWait )
{
    EventGroup_t * pxEventBits = xEventGroup;
    EventBits_t uxReturn, uxControlBits = 0;
    BaseType_t xWaitConditionMet, xAlreadyYielded;
    BaseType_t xTimeoutOccurred = pdFALSE;

    traceENTER_xEventGroupWaitBits( xEventGroup, uxBitsToWaitFor, xClearOnExit, xWaitForAllBits, xTicksToWait );

    /* Check the user is not attempting to wait on the bits used by the kernel
     * itself, and that at least one bit is being requested. */
    configASSERT( xEventGroup );
    configASSERT( ( uxBitsToWaitFor & eventEVENT_BITS_CONTROL_BYTES ) == 0 );
    configASSERT( uxBitsToWaitFor != 0 );
    #if ( ( INCLUDE_xTaskGetSchedulerState == 1 ) || ( configUSE_TIMERS == 1 ) )
    {
        configASSERT( !( ( xTaskGetSchedulerState() == taskSCHEDULER_SUSPENDED ) && ( xTicksToWait != 0 ) ) );
    }
    #endif

    vTaskSuspendAll();
    {
        const EventBits_t uxCurrentEventBits = pxEventBits->uxEventBits;

        /* Check to see if the wait condition is already met or not. */
        xWaitConditionMet = prvTestWaitCondition( uxCurrentEventBits, uxBitsToWaitFor, xWaitForAllBits );

        if( xWaitConditionMet != pdFALSE )
        {
            /* The wait condition has already been met so there is no need to
             * block. */
            uxReturn = uxCurrentEventBits;
            xTicksToWait = ( TickType_t ) 0;

            /* Clear the wait bits if requested to do so. */
            if( xClearOnExit != pdFALSE )
            {
                pxEventBits->uxEventBits &= ~uxBitsToWaitFor;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else if( xTicksToWait == ( TickType_t ) 0 )
        {
            /* The wait condition has not been met, but no block time was
             * specified, so just return the current value. */
            uxReturn = uxCurrentEventBits;
            xTimeoutOccurred = pdTRUE;
        }
        else
        {
            /* The task is going to block to wait for its required bits to be
             * set.  uxControlBits are used to remember the specified behaviour of
             * this call to xEventGroupWaitBits() - for use when the event bits
             * unblock the task. */
            if( xClearOnExit != pdFALSE )
            {
                uxControlBits |= eventCLEAR_EVENTS_ON_EXIT_BIT;
            }
```

检查等待条件是否满足

满足

返回事件值

判断是否清除

不满足且不等待

事件值

不满足且等待

```
    {
        uxControlBits |= eventCLEAR_EVENTS_ON_EXIT_BIT;
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    if( xWaitForAllBits != pdFALSE )
    {
        uxControlBits |= eventWAIT_FOR_ALL_BITS;
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    /* Store the bits that the calling task is waiting for in the
     * task's event list item so the kernel knows when a match is
     * found.  Then enter the blocked state. */
    vTaskPlaceOnUnorderedEventList( &( pxEventBits->xTasksWaitingForBits ), ( uxBitsToWaitFor | uxControlBits ), xTicksToWait );

    /* This is obsolete as it will get set after the task unblocks, but
     * some compilers mistakenly generate a warning about the variable
     * being returned without being set if it is not done. */
    uxReturn = 0;

    traceEVENT_GROUP_WAIT_BITS_BLOCK( xEventGroup, uxBitsToWaitFor );
}
xAlreadyYielded = xTaskResumeAll();

if( xTicksToWait != ( TickType_t ) 0 )
{
    if( xAlreadyYielded == pdFALSE )
    {
        taskYIELD_WITHIN_API();
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }

    /* The task blocked to wait for its required bits to be set - at this
     * point either the required bits were set or the block time expired.  If
     * the required bits were set they will have been stored in the task's
     * event list item, and they should now be retrieved then cleared. */
    uxReturn = uxTaskResetEventItemValue();

    if( ( uxReturn & eventUNBLOCKED_DUE_TO_BIT_SET ) == ( EventBits_t ) 0 )
    {
        taskENTER_CRITICAL();
        {
            /* The task timed out, just return the current event bit value. */
            uxReturn = pxEventBits->uxEventBits;

            /* It is possible that the event bits were updated between this
             * task leaving the Blocked state and running again. */
            if( prvTestWaitCondition( uxReturn, uxBitsToWaitFor, xWaitForAllBits ) != pdFALSE )
            {
                if( xClearOnExit != pdFALSE )
                {
                    pxEventBits->uxEventBits &= ~uxBitsToWaitFor;
                }
                else
                {
                    mtCOVERAGE_TEST_MARKER();
                }
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }

            xTimeoutOccurred = pdTRUE;
        }
        taskEXIT_CRITICAL();
```

```
            xTimeoutOccurred = pdTRUE;
        }
        taskEXIT_CRITICAL();
    }
    else
    {
        /* The task unblocked because the bits were set. */
    }

    /* The task blocked so control bits may have been set. */
    uxReturn &= ~eventEVENT_BITS_CONTROL_BYTES;
}

traceEVENT_GROUP_WAIT_BITS_END( xEventGroup, uxBitsToWaitFor, xTimeoutOccurred );

/* Prevent compiler warnings when trace macros are not used. */
( void ) xTimeoutOccurred;

traceRETURN_xEventGroupWaitBits( uxReturn );

return uxReturn;
```

- 这就是xEventGroupWaitBits的底层代码，可以看到代码很长，我们来分析一下，函数的运行过程
- **1.首先刚进入函数先进行参数验证和初始化**

```c
configASSERT( xEventGroup );
configASSERT( ( uxBitsToWaitFor & eventEVENT_BITS_CONTROL_BYTES ) == 0 );
configASSERT( uxBitsToWaitFor != 0 );
```

  - 确保数据的有效性，包括检查事件标志组句柄、不尝试等待内核保留的控制位、至少等待一个位
- **2.关闭任务调度器**

```
vTaskSuspendAll();
```

  - 调用API函数将系统的任务调度关闭，保证任务不会被抢占和打断，保证任务的原子性操作
- **3.立即检查等待条件**

```c
xWaitConditionMet = prvTestWaitCondition( uxCurrentEventBits, uxBitsToWaitFor, xWaitForAllBits );
```

  - 首先检查条件是否满足，**避免不必要的阻塞**
- **4.根据条件是否满足分为三种情况**
  - **4.1 情况一：条件已满足**

```c
if( xWaitConditionMet != pdFALSE )
{
    uxReturn = uxCurrentEventBits;
    if( xClearOnExit != pdFALSE )
    {
        pxEventBits->uxEventBits &= ~uxBitsToWaitFor;
    }
}
```

    - 如果等待条件已满，则直接返回当前事件值
    - 根据 xClearOnExit 参数决定是否清除相应的位
  - **4.2 情况二：阻塞但不等待**

```c
else if( xTicksToWait == ( TickType_t ) 0 )
{
    uxReturn = uxCurrentEventBits;
    xTimeoutOccurred = pdTRUE;
}
```

- 如果等待条件不满足且等待时间为0，则直接返回当前事件值
- 标记超时已发送

4.3 情况三：阻塞但等待

```c
else
{
    vTaskPlaceOnUnorderedEventList( &( pxEventBits->xTasksWaitingForBits ),
                                    ( uxBitsToWaitFor | uxControlBits ), xTicksToWait );
}
```

- **将任务放入事件标志组的等待列表**
- **任务由就绪态进入阻塞等待状态，等待事件超时或发生**
- **设置控制位（清除标志和等待所有位标志）**

5.开启任务调度

```c
xAlreadyYielded = xTaskResumeAll();
```

- 函数完成操作后，**调用API函数开启任务调度器**，恢复任务的正常调度，**保证FreeRTOS系统的正常运转**

6.阻塞后的处理

```c
uxReturn = uxTaskResetEventItemValue();
if( ( uxReturn & eventUNBLOCKED_DUE_TO_BIT_SET ) == ( EventBits_t ) 0 )
{
    // 超时处理
    uxReturn = pxEventBits->uxEventBits;
    // 检查条件是否在阻塞期间满足
    if( prvTestWaitCondition( uxReturn, uxBitsToWaitFor, xWaitForAllBits ) != pdFALSE )
    {
        if( xClearOnExit != pdFALSE )
        {
            pxEventBits->uxEventBits &= ~uxBitsToWaitFor;
        }
    }
    xTimeoutOccurred = pdTRUE;
}
else
{
    // 因位被设置而解除阻塞
}
```

- 任务阻塞处理会根据条件是否满足以及等待时间是否超时，来进行对应的操作，在最后会返回一个值
- **返回值说明**
  - 1.**因事件发生而解除阻塞**：返回事件发生时的事件值
  - 2.**因超时而解除阻塞**：返回当前标志组的值

讲完等待事件标志位设置函数之后，这里面有一个参数是是否选择清除标志位，可以选择不清除，那么说明后面是可以手动给它清除的，那么就要讲接下来的这个函数了，叫清除事件标志位函数

EventBits_t xEventGroupClearBits(EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToClear)

- 这是一个**有参有返回值**的函数
- **功能**：手动清除事件组中指定的事件位（用于 “手动清除策略” 的场景）
- **参数**：
  - **xEventGroup**:事件组句柄
  - **uxBitsToClear**：要清除的事件位
- **返回值**：清除标志前，事件组的值
- **注意**：仅任务中调用，不可在ISR中调用，ISR中需要加上FromISR后缀的API函数
- 接下来我们来看一下函数的底层代码实现

```
EventBits_t xEventGroupClearBits( EventGroupHandle_t xEventGroup,
                                  const EventBits_t uxBitsToClear )
{
    EventGroup_t * pxEventBits = xEventGroup;
    EventBits_t uxReturn;
    traceENTER_xEventGroupClearBits( xEventGroup, uxBitsToClear );
    /* Check the user is not attempting to clear the bits used by the kernel
     * itself. */
    configASSERT( xEventGroup );
    configASSERT( ( uxBitsToClear & eventEVENT_BITS_CONTROL_BYTES ) == 0 );

    taskENTER_CRITICAL();
    {
        traceEVENT_GROUP_CLEAR_BITS( xEventGroup, uxBitsToClear );

        /* The value returned is the event group value prior to the bits being
         * cleared. */
        uxReturn = pxEventBits->uxEventBits;

        /* Clear the bits. */
        pxEventBits->uxEventBits &= ~uxBitsToClear;
    }
    taskEXIT_CRITICAL();

    traceRETURN_xEventGroupClearBits( uxReturn );

    return uxReturn;
}
```

- 看了底层的代码实现，可以看到底层代码实现非常简单，首先就是参数检验
- **然后就是调用进入临界区函数，关中断**
- 紧接着就是**清除事件组中传入的要清除的标志位**
- 最后调用**退出临界区函数，开启中断，恢复系统运行**
- 最后返回一个**返回值**

- 这就是清除标志位函数，使用起来非常简单。
- 上面那些全都是在任务中调用的API函数，那么在中断服务程序中要调用那些API函数呢
- **ISR函数：**
  - 设置事件位：**BaseType_t xEventGroupSetBitsFromISR(EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet, BaseType_t *pxHigherPriorityTaskWoken);**
  - 清除事件位：**BaseType_t xEventGroupClearBitsFromISR(EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToClear);**
  - **关键参数**：pxHigherPriorityTaskWoken：输出参数，若设置事件位后唤醒了更高优先级任务，此变量会被置为 pdTRUE，需在 ISR 结尾调用 **portYIELD_FROM_ISR(pxHigherPriorityTaskWoken)** 触发任务切换。

六、避坑指南：5个常见错误及解决方法

1.错误一：未开启事件组配置

- **现象**：编译报错 “xEventGroupCreate未定义”
- **解决**：在 FreeRTOSConfig.h 中添加 **#define configUSE_EVENT_GROUPS 1**

- 2.错误二：等待超出支持的事件位数
  - **现象**：等待32位事件组的bit31时无响应（实际16位）
  - **解决**：查看 **portmacro.h** 中 configUSE_16_BIT_TICKS 的配置，16 位环境下最大支持 bit15，32 位环境下支持 bit31
- 3.错误三：ISR中调用普通API函数
  - **现象**：MCU死机或任务调度混乱
  - **解决**：**调用ISR中的API函数（带FromISR后缀的），并正确处理pxHigherPriorityTaskWoken**
- 4.错误四：自动清除 VS 手动清除混淆
  - **现象**：事件**触发一次后，后续任务无法再响应**
  - **解决**：若多个任务需响应同一事件，xClearOnExit 需设为 pdFALSE，并在**合适时机手动清除**；若仅单个任务响应，**用自动清除更高效**
- 5.错误五：永久等待（portMAX_DELAY）导致程序卡死
  - **现象**：等待的事件用不触发，任务一直阻塞
  - **解决**：**非必要不使用 portMAX_DELAY**，建议设置合理超时时间，超时后做错误处理（如重新初始化外设）

七、总结

- 事件组是FreeRTOS中**“多事件同步”的最优解**，核心是通过**位操作**实现灵活任务协作，关键掌握：
  - 1.事件位的置1和清除逻辑
  - 2.**等待方式（AND或OR）**和**清除策略（自动或手动）**
  - 3.**普通API和中断安全API的区分使用**

实战用法：事件组的典型用法

- **场景1：任务等待多个外设事件**
  - 需求:MCU需要采集温湿度数据，只有等“I2C总线空闲（BIT0）”和“温湿度传感器就绪（BIT1）”时，采集任务才能执行
    - 实现：
      - 1.I2C中断服务程序中，总线空闲时调用xEventGroupSetBitsFromISR() 置 1 BIT0
      - 2.传感器就绪引脚中断中，置1BIT1
      - 3.采集任务调用xEventGroupWaitBits() 等待 BIT0 和 BIT1（AND 方式），满足后执行采集，采集完成后手动清除 BIT0 和 BIT1
- **场景2：多任务响应同一事件**
  - 需求：按键按下（BIT0）时，任务 A 执行 “点亮 LED”，任务 B 执行 “发送串口消息”
    - 实现：
      - 1.按键中断置1BIT0
      - 2.任务 A 和任务 B 均调用 xEventGroupWaitBits() 等待 BIT0（OR 方式，xClearOnExit = pdFALSE）
      - 3.两个任务被唤醒后分别执行操作，最后由 “事件管理任务” 统一调用 xEventGroupClearBits() 清除 BIT0